CHAPTER 2

PERFORMANCE ISSUES

Learning Objectives

After studying this chapter, you should be able to:

- Understand the key performance issues that relate to computer design.
- Explain the reasons for the move to multicore organization, and understand the trade-off between cache and processor resources on a single chip.
- Distinguish among multicore, MIC, and GPGPU organizations.
- Summarize some of the issues in computer performance assessment.
- Discuss the SPEC benchmarks.
- Explain the differences among arithmetic, harmonic, and geometric means.

This chapter addresses the issue of computer system performance. We begin with a consideration of the need for balanced utilization of computer resources, which provides a perspective that is useful throughout the book. Next we look at contemporary computer organization designs intended to provide performance to meet current and projected demand. Finally, we look at tools and models that have been developed to provide a means of assessing comparative computer system performance.

2.1 DESIGNING FOR PERFORMANCE

Year by year, the cost of computer systems continues to drop dramatically, while the performance and capacity of those systems continue to rise equally dramatically. Today's laptops have the computing power of an IBM mainframe from 10 or 15 years ago. Thus, we have virtually "free" computer power. Processors are so inexpensive that we now have microprocessors we throw away. The digital pregnancy test is an example (used once and then thrown away). And this continuing technological revolution has enabled the development of applications of astounding complexity and power. For example, desktop applications that require the great power of today's microprocessor-based systems include

- Image processing
- Three-dimensional rendering
- Speech recognition
- Videoconferencing
- Multimedia authoring
- Voice and video annotation of files
- Simulation modeling

Workstation systems now support highly sophisticated engineering and scientific applications and have the capacity to support image and video applications. In addition, businesses are relying on increasingly powerful servers to handle transaction and database processing and to support massive client/server networks that have replaced the huge mainframe computer centers of yesteryear. As well, cloud service

providers use massive high-performance banks of servers to satisfy high-volume, high-transaction-rate applications for a broad spectrum of clients.

What is fascinating about all this from the perspective of computer organization and architecture is that, on the one hand, the basic building blocks for today's computer miracles are virtually the same as those of the IAS computer from over 50 years ago, while on the other hand, the techniques for squeezing the maximum performance out of the materials at hand have become increasingly sophisticated.

This observation serves as a guiding principle for the presentation in this book. As we progress through the various elements and components of a computer, two objectives are pursued. First, the book explains the fundamental functionality in each area under consideration, and second, the book explores those techniques required to achieve maximum performance. In the remainder of this section, we highlight some of the driving factors behind the need to design for performance.

Microprocessor Speed

What gives Intel x86 processors or IBM mainframe computers such mind-boggling power is the relentless pursuit of speed by processor chip manufacturers. The evolution of these machines continues to bear out Moore's law, described in Chapter 1. So long as this law holds, chipmakers can unleash a new generation of chips every three years—with four times as many transistors. In memory chips, this has quadrupled the capacity of **dynamic random-access memory (DRAM)**, still the basic technology for computer main memory, every three years. In microprocessors, the addition of new circuits, and the speed boost that comes from reducing the distances between them, has improved performance four- or fivefold every three years or so since Intel launched its x86 family in 1978.

But the raw speed of the microprocessor will not achieve its potential unless it is fed a constant stream of work to do in the form of computer instructions. Anything that gets in the way of that smooth flow undermines the power of the processor. Accordingly, while the chipmakers have been busy learning how to fabricate chips of greater and greater density, the processor designers must come up with ever more elaborate techniques for feeding the monster. Among the techniques built into contemporary processors are the following:

- **Pipelining:** The execution of an instruction involves multiple stages of operation, including fetching the instruction, decoding the opcode, fetching operands, performing a calculation, and so on. Pipelining enables a processor to work simultaneously on multiple instructions by performing a different phase for each of the multiple instructions at the same time. The processor overlaps operations by moving data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously. For example, while one instruction is being executed, the computer is decoding the next instruction. This is the same principle as seen in an assembly line.
- **Branch prediction:** The processor looks ahead in the instruction code fetched from memory and predicts which branches, or groups of instructions, are likely to be processed next. If the processor guesses right most of the time, it can prefetch the correct instructions and buffer them so that the processor is kept busy. The more sophisticated examples of this strategy predict not just

the next branch but multiple branches ahead. Thus, branch prediction potentially increases the amount of work available for the processor to execute.

- **Superscalar execution:** This is the ability to issue more than one instruction in every processor clock cycle. In effect, multiple parallel pipelines are used.
- **Data flow analysis:** The processor analyzes which instructions are dependent on each other's results, or data, to create an optimized schedule of instructions. In fact, instructions are scheduled to be executed when ready, independent of the original program order. This prevents unnecessary delay.
- **Speculative execution:** Using branch prediction and data flow analysis, some processors speculatively execute instructions ahead of their actual appearance in the program execution, holding the results in temporary locations. This enables the processor to keep its execution engines as busy as possible by executing instructions that are likely to be needed.

These and other sophisticated techniques are made necessary by the sheer power of the processor. Collectively they make it possible to execute many instructions per processor cycle, rather than to take many cycles per instruction.

Performance Balance

While processor power has raced ahead at breakneck speed, other critical components of the computer have not kept up. The result is a need to look for performance balance: an adjustment/tuning of the organization and architecture to compensate for the mismatch among the capabilities of the various components.

The problem created by such mismatches is particularly critical at the interface between processor and main memory. While processor speed has grown rapidly, the speed with which data can be transferred between main memory and the processor has lagged badly. The interface between processor and main memory is the most crucial pathway in the entire computer because it is responsible for carrying a constant flow of program instructions and data between memory chips and the processor. If memory or the pathway fails to keep pace with the processor's insistent demands, the processor stalls in a wait state, and valuable processing time is lost.

A system architect can attack this problem in a number of ways, all of which are reflected in contemporary computer designs. Consider the following examples:

- Increase the number of bits that are retrieved at one time by making DRAMs "wider" rather than "deeper" and by using wide bus data paths.
- Change the DRAM interface to make it more efficient by including a cache[1] or other buffering scheme on the DRAM chip.
- Reduce the frequency of memory access by incorporating increasingly complex and efficient cache structures between the processor and main memory. This includes the incorporation of one or more caches on the processor chip as well as on an off-chip cache close to the processor chip.

[1]A cache is a relatively small fast memory interposed between a larger, slower memory and the logic that accesses the larger memory. The cache holds recently accessed data and is designed to speed up subsequent access to the same data. Caches are discussed in Chapter 4.

- Increase the interconnect bandwidth between processors and memory by using higher-speed buses and a hierarchy of buses to buffer and structure data flow.

Another area of design focus is the handling of I/O devices. As computers become faster and more capable, more sophisticated applications are developed that support the use of peripherals with intensive I/O demands. Figure 2.1 gives some examples of typical peripheral devices in use on personal computers and workstations. These devices create tremendous data throughput demands. While the current generation of processors can handle the data pumped out by these devices, there remains the problem of getting that data moved between processor and peripheral. Strategies here include caching and buffering schemes plus the use of higher-speed interconnection buses and more elaborate interconnection structures. In addition, the use of multiple-processor configurations can aid in satisfying I/O demands.

The key in all this is balance. Designers constantly strive to balance the throughput and processing demands of the processor components, main memory, I/O devices, and the interconnection structures. This design must constantly be rethought to cope with two constantly evolving factors:

- The rate at which performance is changing in the various technology areas (processor, buses, memory, peripherals) differs greatly from one type of element to another.
- New applications and new peripheral devices constantly change the nature of the demand on the system in terms of typical instruction profile and the data access patterns.

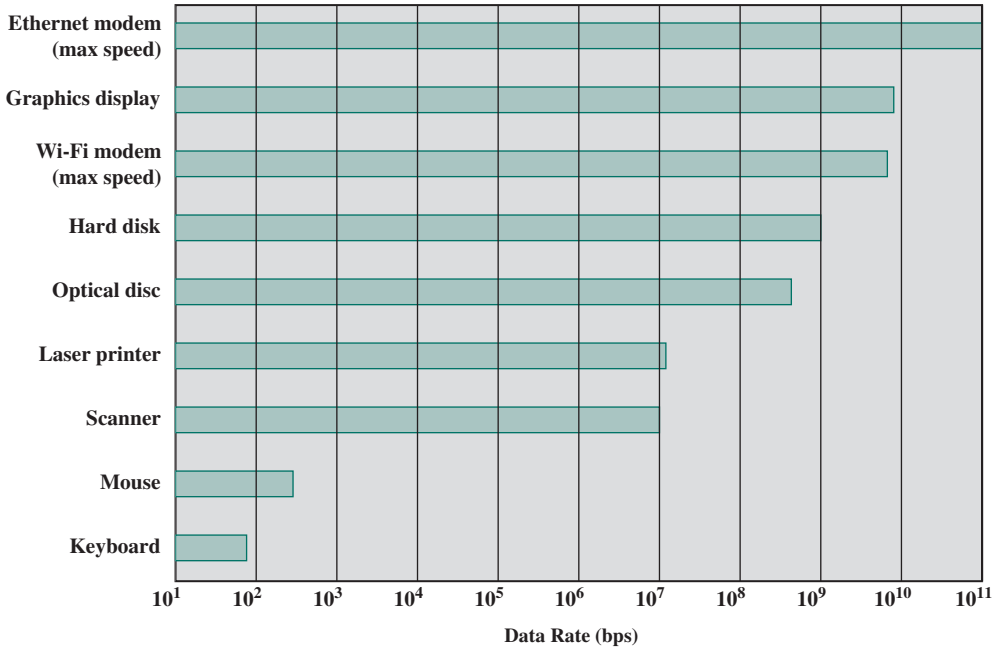


Figure 2.1 Typical I/O Device Data Rates

Thus, computer design is a constantly evolving art form. This book attempts to present the fundamentals on which this art form is based and to present a survey of the current state of that art.

Improvements in Chip Organization and Architecture

As designers wrestle with the challenge of balancing processor performance with that of main memory and other computer components, the need to increase processor speed remains. There are three approaches to achieving increased processor speed:

- Increase the hardware speed of the processor. This increase is fundamentally due to shrinking the size of the logic gates on the processor chip, so that more gates can be packed together more tightly and to increasing the clock rate. With gates closer together, the propagation time for signals is significantly reduced, enabling a speeding up of the processor. An increase in clock rate means that individual operations are executed more rapidly.
- Increase the size and speed of caches that are interposed between the processor and main memory. In particular, by dedicating a portion of the processor chip itself to the cache, cache access times drop significantly.
- Make changes to the processor organization and architecture that increase the effective speed of instruction execution. Typically, this involves using parallelism in one form or another.

Traditionally, the dominant factor in performance gains has been in increases in clock speed due and logic density. However, as clock speed and logic density increase, a number of obstacles become more significant [INTE04]:

- **Power:** As the density of logic and the clock speed on a chip increase, so does the power density (Watts/cm^2). The difficulty of dissipating the heat generated on high-density, high-speed chips is becoming a serious design issue [GIBB04, BORK03].
- **RC delay:** The speed at which electrons can flow on a chip between transistors is limited by the resistance and capacitance of the metal wires connecting them; specifically, delay increases as the RC product increases. As components on the chip decrease in size, the wire interconnects become thinner, increasing resistance. Also, the wires are closer together, increasing capacitance.
- **Memory latency and throughput:** Memory access speed (latency) and transfer speed (throughput) lag processor speeds, as previously discussed.

Thus, there will be more emphasis on organization and architectural approaches to improving performance. These techniques are discussed in later chapters of the text.

Beginning in the late 1980s, and continuing for about 15 years, two main strategies have been used to increase performance beyond what can be achieved simply by increasing clock speed. First, there has been an increase in cache capacity. There are now typically two or three levels of cache between the processor and main memory. As chip density has increased, more of the cache memory has been incorporated on the chip, enabling faster cache access. For example, the original Pentium

chip devoted about 10% of on-chip area to a cache. Contemporary chips devote over half of the chip area to caches. And, typically, about three-quarters of the other half is for pipeline-related control and buffering.

Second, the instruction execution logic within a processor has become increasingly complex to enable parallel execution of instructions within the processor. Two noteworthy design approaches have been pipelining and superscalar. A pipeline works much as an assembly line in a manufacturing plant enabling different stages of execution of different instructions to occur at the same time along the pipeline. A superscalar approach in essence allows multiple pipelines within a single processor, so that instructions that do not depend on one another can be executed in parallel.

By the mid to late 90s, both of these approaches were reaching a point of diminishing returns. The internal organization of contemporary processors is exceedingly complex and is able to squeeze a great deal of parallelism out of the instruction stream. It seems likely that further significant increases in this direction will be relatively modest [GIBB04]. With three levels of cache on the processor chip, each level providing substantial capacity, it also seems that the benefits from the cache are reaching a limit.

However, simply relying on increasing clock rate for increased performance runs into the power dissipation problem already referred to. The faster the clock rate, the greater the amount of power to be dissipated, and some fundamental physical limits are being reached.

Figure 2.2 illustrates the concepts we have been discussing.[2] The top line shows that, as per Moore's Law, the number of transistors on a single chip continues to

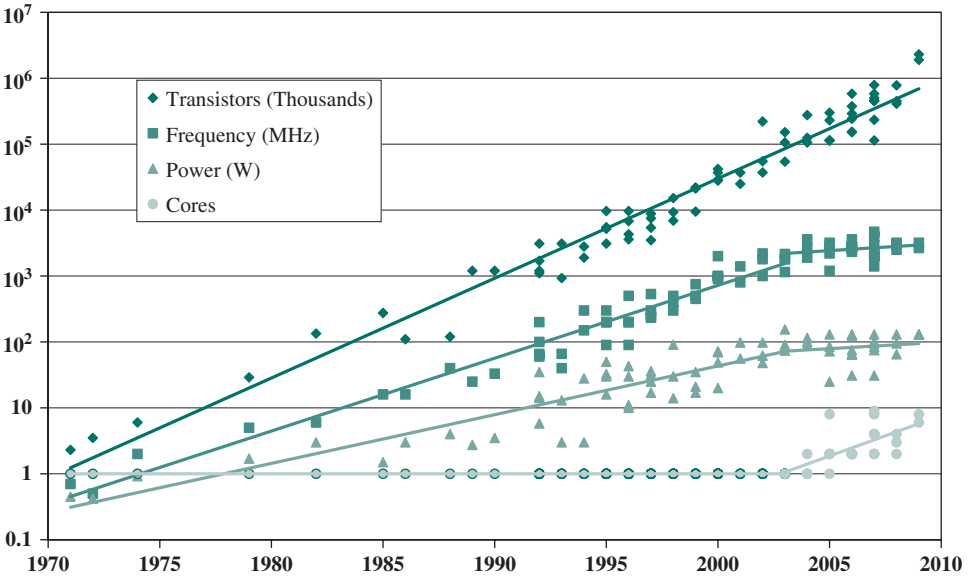


Figure 2.2 Processor Trends

[2]I am grateful to Professor Kathy Yelick of UC Berkeley, who provided this graph.

grow exponentially.[3] Meanwhile, the clock speed has leveled off, in order to prevent a further rise in power. To continue increasing performance, designers have had to find ways of exploiting the growing number of transistors other than simply building a more complex processor. The response in recent years has been the development of the multicore computer chip.

2.2 MULTICORE, MICS, AND GPGPUS

With all of the difficulties cited in the preceding section in mind, designers have turned to a fundamentally new approach to improving performance: placing multiple processors on the same chip, with a large shared cache. The use of multiple processors on the same chip, also referred to as multiple cores, or **multicore**, provides the potential to increase performance without increasing the clock rate. Studies indicate that, within a processor, the increase in performance is roughly proportional to the square root of the increase in complexity [BORK03]. But if the software can support the effective use of multiple processors, then doubling the number of processors almost doubles performance. Thus, the strategy is to use two simpler processors on the chip rather than one more complex processor.

In addition, with two processors, larger caches are justified. This is important because the power consumption of memory logic on a chip is much less than that of processing logic.

As the logic density on chips continues to rise, the trend for both more cores and more cache on a single chip continues. Two-core chips were quickly followed by four-core chips, then 8, then 16, and so on. As the caches became larger, it made performance sense to create two and then three levels of cache on a chip, with initially, the first-level cache dedicated to an individual processor and levels two and three being shared by all the processors. It is now common for the second-level cache to also be private to each core.

Chip manufacturers are now in the process of making a huge leap forward in the number of cores per chip, with more than 50 cores per chip. The leap in performance as well as the challenges in developing software to exploit such a large number of cores has led to the introduction of a new term: **many integrated core (MIC)**.

The multicore and MIC strategy involves a homogeneous collection of general-purpose processors on a single chip. At the same time, chip manufacturers are pursuing another design option: a chip with multiple general-purpose processors plus **graphics processing units (GPUs)** and specialized cores for video processing and other tasks. In broad terms, a GPU is a core designed to perform parallel operations on graphics data. Traditionally found on a plug-in graphics card (display adapter), it is used to encode and render 2D and 3D graphics as well as process video.

Since GPUs perform parallel operations on multiple sets of data, they are increasingly being used as vector processors for a variety of applications that require repetitive computations. This blurs the line between the GPU and the CPU

[3]The observant reader will note that the transistor count values in this figure are significantly less than those of Figure 1.12. That latter figure shows the transistor count for a form of main memory known as DRAM (discussed in Chapter 5), which supports higher transistor density than processor chips.

[ARORI12, FATA08, PROP11]. When a broad range of applications are supported by such a processor, the term **general-purpose computing on GPUs (GPGPU)** is used.

We explore design characteristics of multicore computers in Chapter 18 and GPGPUs in Chapter 19.

2.3 TWO LAWS THAT PROVIDE INSIGHT: AHMDAHL'S LAW AND LITTLE'S LAW

In this section, we look at two equations, called "laws." The two laws are unrelated but both provide insight into the performance of parallel systems and multicore systems.

Amdahl's Law

Computer system designers look for ways to improve system performance by advances in technology or change in design. Examples include the use of parallel processors, the use of a memory cache hierarchy, and speedup in memory access time and I/O transfer rate due to technology improvements. In all of these cases, it is important to note that a speedup in one aspect of the technology or design does not result in a corresponding improvement in performance. This limitation is succinctly expressed by Amdahl's law.

Amdahl's law was first proposed by Gene Amdahl in 1967 ([AMDA67], [AMDA13]) and deals with the potential speedup of a program using multiple processors compared to a single processor. Consider a program running on a single processor such that a fraction $(1 - f)$ of the execution time involves code that is inherently sequential, and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead. Let T be the total execution time of the program using a single processor. Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\text{Speedup} = \frac{\text{Time to execute program on a single processor}}{\text{Time to execute program on } N \text{ parallel processors}}$$

$$= \frac{T(1-f) + Tf}{T(1-f) + \frac{Tf}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

This equation is illustrated in Figures 2.3 and 2.4. Two important conclusions can be drawn:

1. When f is small, the use of parallel processors has little effect.
2. As N approaches infinity, speedup is bound by $1/(1 - f)$, so that there are diminishing returns for using more processors.

These conclusions are too pessimistic, an assertion first put forward in [GUST88]. For example, a server can maintain multiple threads or multiple tasks to handle multiple clients and execute the threads or tasks in parallel up to the limit of the number of processors. Many database applications involve computations on massive amounts of data that can be split up into multiple parallel tasks.

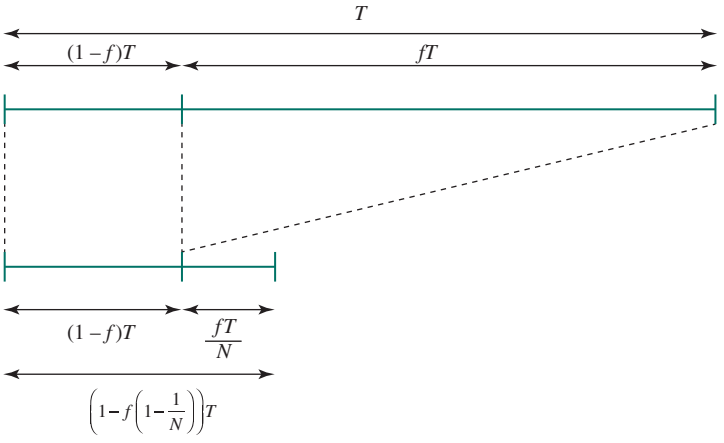


Figure 2.3 Illustration of Amdahl's Law

Nevertheless, Amdahl's law illustrates the problems facing industry in the development of multicore machines with an ever-growing number of cores: The software that runs on such machines must be adapted to a highly parallel execution environment to exploit the power of parallel processing.

Amdahl's law can be generalized to evaluate any design or technical improvement in a computer system. Consider any enhancement to a feature of a system that results in a speedup. The speedup can be expressed as

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}} \tag{2.1}$$

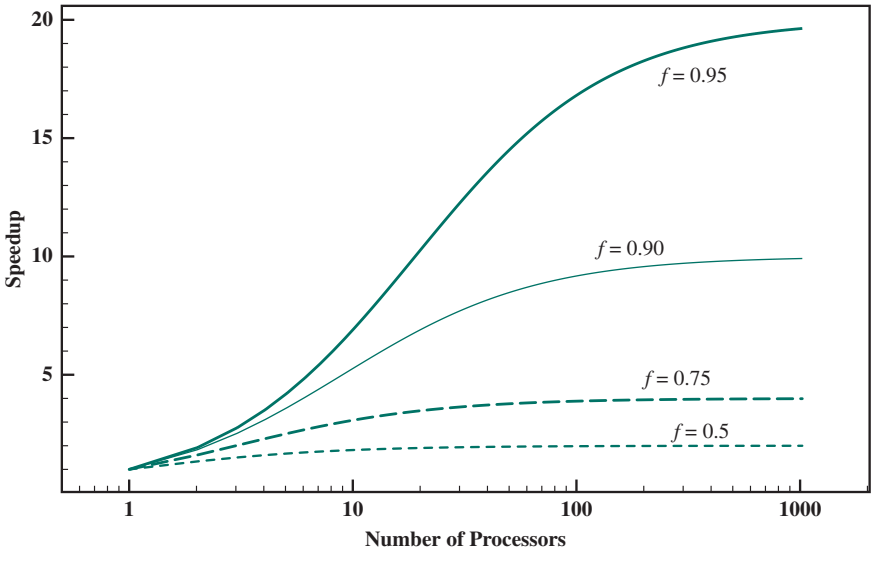


Figure 2.4 Amdahl's Law for Multiprocessors

Suppose that a feature of the system is used during execution a fraction of the time f, before enhancement, and that the speedup of that feature after enhancement is SU_f. Then the overall speedup of the system is

$$\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{SU_f}}$$

EXAMPLE 2.1 Suppose that a task makes extensive use of floating-point operations, with 40% of the time consumed by floating-point operations. With a new hardware design, the floating-point module is sped up by a factor of K. Then the overall speedup is as follows:

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{K}}$$

Thus, independent of K, the maximum speedup is 1.67.

Little's Law

A fundamental and simple relation with broad applications is Little's Law [LITT61, LITT11].[4] We can apply it to almost any system that is statistically in steady state, and in which there is no leakage. Specifically, we have a steady state system to which items arrive at an average rate of λ items per unit time. The items stay in the system an average of W units of time. Finally, there is an average of L units in the system at any one time. Little's Law relates these three variables as $L = \lambda W$.

Using queuing theory terminology, Little's Law applies to a queuing system. The central element of the system is a server, which provides some service to items. Items from some population of items arrive at the system to be served. If the server is idle, an item is served immediately. Otherwise, an arriving item joins a waiting line, or queue. There can be a single queue for a single server, a single queue for multiple servers, or multiples queues, one for each of multiple servers. When a server has completed serving an item, the item departs. If there are items waiting in the queue, one is immediately dispatched to the server. The server in this model can represent anything that performs some function or service for a collection of items. Examples: A processor provides service to processes; a transmission line provides a transmission service to packets or frames of data; and an I/O device provides a read or write service for I/O requests.

To understand Little's formula, consider the following argument, which focuses on the experience of a single item. When the item arrives, it will find on

[4]The second reference is a retrospective article on his law that Little wrote 50 years after his original paper. That must be unique in the history of the technical literature, although Amdahl comes close, with a 46-year gap between [AMDA67] and [AMDA13].

average L items ahead of it, one being serviced and the rest in the queue. When the item leaves the system after being serviced, it will leave behind on average the same number of items in the system, namely L, because L is defined as the average number of items waiting. Further, the average time that the item was in the system was W. Since items arrive at a rate of λ, we can reason that in the time W, a total of λW items must have arrived. Thus $w = \lambda W$.

To summarize, under steady state conditions, the average number of items in a queuing system equals the average rate at which items arrive multiplied by the average time that an item spends in the system. This relationship requires very few assumptions. We do not need to know what the service time distribution is, what the distribution of arrival times is, or the order or priority in which items are served. Because of its simplicity and generality, Little's Law is extremely useful and has experienced somewhat of a revival due to the interest in performance problems related to multicore computers.

A very simple example, from [LITT11], illustrates how Little's Law might be applied. Consider a multicore system, with each core supporting multiple threads of execution. At some level, the cores share a common memory. The cores share a common main memory and typically share a common cache memory as well. In any case, when a thread is executing, it may arrive at a point at which it must retrieve a piece of data from the common memory. The thread stops and sends out a request for that data. All such stopped threads are in a queue. If the system is being used as a server, an analyst can determine the demand on the system in terms of the rate of user requests, and then translate that into the rate of requests for data from the threads generated to respond to an individual user request. For this purpose, each user request is broken down into subtasks that are implemented as threads. We then have λ = the average rate of total thread processing required after all members' requests have been broken down into whatever detailed subtasks are required. Define L as the average number of stopped threads waiting during some relevant time. Then W = average response time. This simple model can serve as a guide to designers as to whether user requirements are being met and, if not, provide a quantitative measure of the amount of improvement needed.

2.4 BASIC MEASURES OF COMPUTER PERFORMANCE

In evaluating processor hardware and setting requirements for new systems, performance is one of the key parameters to consider, along with cost, size, security, reliability, and, in some cases, power consumption.

It is difficult to make meaningful performance comparisons among different processors, even among processors in the same family. Raw speed is far less important than how a processor performs when executing a given application. Unfortunately, application performance depends not just on the raw speed of the processor but also on the instruction set, choice of implementation language, efficiency of the compiler, and skill of the programming done to implement the application.

In this section, we look at some traditional measures of processor speed. In the next section, we examine benchmarking, which is the most common approach to assessing processor and computer system performance. The following section discusses how to average results from multiple tests.

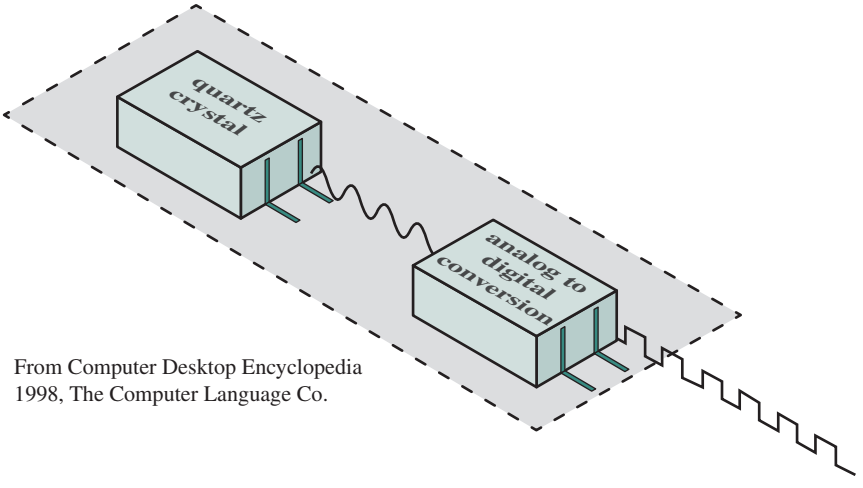
Clock Speed

Operations performed by a processor, such as fetching an instruction, decoding the instruction, performing an arithmetic operation, and so on, are governed by a system clock. Typically, all operations begin with the pulse of the clock. Thus, at the most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).

Typically, clock signals are generated by a quartz crystal, which generates a constant sine wave while power is applied. This wave is converted into a digital voltage pulse stream that is provided in a constant flow to the processor circuitry (Figure 2.5). For example, a 1-GHz processor receives 1 billion pulses per second. The rate of pulses is known as the **clock rate**, or **clock speed**. One increment, or pulse, of the clock is referred to as a **clock cycle**, or a **clock tick**. The time between pulses is the **cycle time**.

The clock rate is not arbitrary, but must be appropriate for the physical layout of the processor. Actions in the processor require signals to be sent from one processor element to another. When a signal is placed on a line inside the processor, it takes some finite amount of time for the voltage levels to settle down so that an accurate value (logical 1 or 0) is available. Furthermore, depending on the physical layout of the processor circuits, some signals may change more rapidly than others. Thus, operations must be synchronized and paced so that the proper electrical signal (voltage) values are available for each operation.

The execution of an instruction involves a number of discrete steps, such as fetching the instruction from memory, decoding the various portions of the instruction, loading and storing data, and performing arithmetic and logical operations. Thus, most instructions on most processors require multiple clock cycles to complete. Some instructions may take only a few cycles, while others require dozens. In addition, when pipelining is used, multiple instructions are being executed simultaneously. Thus, a straight comparison of clock speeds on different processors does not tell the whole story about performance.



From Computer Desktop Encyclopedia
1998, The Computer Language Co.

Figure 2.5 System Clock

Instruction Execution Rate

A processor is driven by a clock with a constant frequency f or, equivalently, a constant cycle time τ, where $\tau = 1/f$. Define the instruction count, I_c, for a program as the number of machine instructions executed for that program until it runs to completion or for some defined time interval. Note that this is the number of instruction executions, not the number of instructions in the object code of the program. An important parameter is the average cycles per instruction (*CPI*) for a program. If all instructions required the same number of clock cycles, then *CPI* would be a constant value for a processor. However, on any given processor, the number of clock cycles required varies for different types of instructions, such as load, store, branch, and so on. Let CPI_i be the number of cycles required for instruction type i, and I_i be the number of executed instructions of type i for a given program. Then we can calculate an overall *CPI* as follows:

$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c} \tag{2.2}$$

The processor time T needed to execute a given program can be expressed as

$$T = I_c \times CPI \times \tau$$

We can refine this formulation by recognizing that during the execution of an instruction, part of the work is done by the processor, and part of the time a word is being transferred to or from memory. In this latter case, the time to transfer depends on the memory cycle time, which may be greater than the processor cycle time. We can rewrite the preceding equation as

$$T = I_c \times [p + (m \times k)] \times \tau$$

where p is the number of processor cycles needed to decode and execute the instruction, m is the number of memory references needed, and k is the ratio between memory cycle time and processor cycle time. The five performance factors in the preceding equation (I_c, p, m, k, τ) are influenced by four system attributes: the design of the instruction set (known as *instruction set architecture*); compiler technology (how effective the compiler is in producing an efficient machine language program from a high-level language program); processor implementation; and cache and memory hierarchy. Table 2.1 is a matrix in which one dimension shows the five performance factors and the other dimension shows the four system attributes. An X in a cell indicates a system attribute that affects a performance factor.

Table 2.1 Performance Factors and System Attributes

	I_c	p	m	k	τ
Instruction set architecture	X	X			
Compiler technology	X	X	X		
Processor implementation		X			X
Cache and memory hierarchy				X	X

A common measure of performance for a processor is the rate at which instructions are executed, expressed as millions of instructions per second (MIPS), referred to as the **MIPS rate**. We can express the MIPS rate in terms of the clock rate and *CPI* as follows:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} \tag{2.3}$$

EXAMPLE 2.2 Consider the execution of a program that results in the execution of 2 million instructions on a 400-MHz processor. The program consists of four major types of instructions. The instruction mix and the *CPI* for each instruction type are given below, based on the result of a program trace experiment:

Instruction Type	*CPI*	Instruction Mix (%)
Arithmetic and logic	1	60
Load/store with cache hit	2	18
Branch	4	12
Memory reference with cache miss	8	10

The average *CPI* when the program is executed on a uniprocessor with the above trace results is $CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24$. The corresponding MIPS rate is $(400 \times 10^6)/(2.24 \times 10^6) \approx 178$.

Another common performance measure deals only with floating-point instructions. These are common in many scientific and game applications. Floating-point performance is expressed as millions of floating-point operations per second (MFLOPS), defined as follows:

$$\text{MFLOPS rate} = \frac{\text{Number of executed floating} - \text{point operations in a program}}{\text{Execution time} \times 10^6}$$

2.5 CALCULATING THE MEAN

In evaluating some aspect of computer system performance, it is often the case that a single number, such as execution time or memory consumed, is used to characterize performance and to compare systems. Clearly, a single number can provide only a very simplified view of a system's capability. Nevertheless, and especially in the field of benchmarking, single numbers are typically used for performance comparison [SMIT88].

As is discussed in Section 2.6, the use of benchmarks to compare systems involves calculating the mean value of a set of data points related to execution time. It turns out that there are multiple alternative algorithms that can be used for calculating a mean value, and this has been the source of some controversy in

the benchmarking field. In this section, we define these alternative algorithms and comment on some of their properties. This prepares us for a discussion in the next section of mean calculation in benchmarking.

The three common formulas used for calculating a mean are arithmetic, geometric, and harmonic. Given a set of n real numbers $(x_1, x_2, \ldots, x_n)$, the three means are defined as follows:

Arithmetic mean

$$AM = \frac{x_1 + \cdots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.4}$$

Geometric mean

$$GM = \sqrt[n]{x_1 \times \cdots \times x_n} = \left(\prod_{i=1}^{n} x_i\right)^{1/n} = e^{\left(\frac{1}{n}\sum_{i=1}^{n} \ln(x_i)\right)} \tag{2.5}$$

Harmonic mean

$$HM = \frac{n}{\left(\frac{1}{x_1}\right) + \cdots + \left(\frac{1}{x_n}\right)} = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{x_i}\right)} \qquad x_i > 0 \tag{2.6}$$

It can be shown that the following inequality holds:

$$AM \leq GM \leq HM$$

The values are equal only if $x_1 = x_2 = \ldots x_n$.

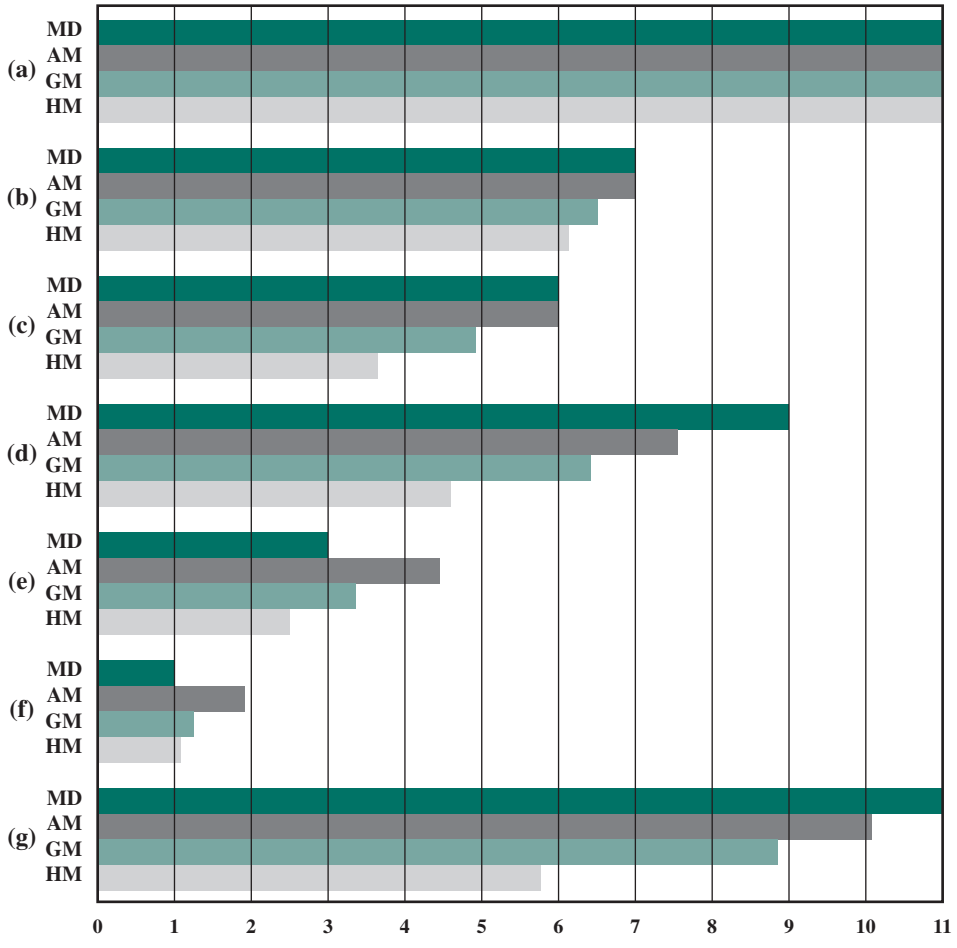
We can get a useful insight into these alternative calculations by defining the functional mean. Let $f(x)$ be a continuous monotonic function defined in the interval $0 \leq y < \infty$. The functional mean with respect to the function $f(x)$ for n positive real numbers $(x_1, x_2, \ldots, x_n)$ is defined as

Functional mean $$FM = f^{-1}\left(\frac{f(x_1) + \cdots + f(x_n)}{n}\right) = f^{-1}\left(\frac{1}{n}\sum_{i=1}^{n} f(x_i)\right)$$

where $f^{-1}(x)$ is the inverse of $f(x)$. The mean values defined in Equations (2.1) through (2.3) are special cases of the functional mean, as follows:

- AM is the FM with respect to $f(x) = x$
- GM is the FM with respect to $f(x) = \ln x$
- HM is the FM with respect to $f(x) = 1/x$

EXAMPLE 2.3 Figure 2.6 illustrates the three means applied to various data sets, each of which has eleven data points and a maximum data point value of 11. The median value is also included in the chart. Perhaps what stands out the most in this figure is that the HM has a tendency to produce a misleading result when the data is skewed to larger values or when there is a small-value outlier.



(a) Constant (11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)
(b) Clustered around a central value (3, 5, 6, 6, 7, 7, 7, 8, 8, 9, 11)
(c) Uniform distribution (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
(d) Large-number bias (1, 4, 4, 7, 7, 9, 9, 10, 10, 11, 11)
(e) Small-number bias(1, 1, 2, 2, 3, 3, 5, 5, 8, 8, 11)
(f) Upper outlier (11, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
(g) Lower outlier (1, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)

MD = median
AM = arithmetic mean
GM = geometric mean
HM = harmonic mean

Figure 2.6 Comparison of Means on Various Data Sets (each set has a maximum data point value of 11)

Let us now consider which of these means are appropriate for a given performance measure. As a preface to these remarks, it should be noted that a number of papers ([CITR06], [FLEM86], [GILA95], [JACO95], [JOHN04], [MASH04], [SMIT88]) and books ([HENN12], [HWAN93], [JAIN91], [LILJ00]) over the years have argued the pros and cons of the three means for performance analysis and come to conflicting conclusions. To simplify a complex controversy, we just note that the conclusions reached depend very much on the examples chosen and the way in which the objectives are stated.

Arithmetic Mean

An AM is an appropriate measure if the sum of all the measurements is a meaningful and interesting value. The AM is a good candidate for comparing the execution time performance of several systems. For example, suppose we were interested in using a system for large-scale simulation studies and wanted to evaluate several alternative products. On each system we could run the simulation multiple times with different input values for each run, and then take the average execution time across all runs. The use of multiple runs with different inputs should ensure that the results are not heavily biased by some unusual feature of a given input set. The AM of all the runs is a good measure of the system's performance on simulations, and a good number to use for system comparison.

The AM used for a time-based variable (e.g., seconds), such as program execution time, has the important property that it is directly proportional to the total time. So, if the total time doubles, the mean value doubles.

Harmonic Mean

For some situations, a system's execution rate may be viewed as a more useful measure of the value of the system. This could be either the instruction execution rate, measured in MIPS or MFLOPS, or a program execution rate, which measures the rate at which a given type of program can be executed. Consider how we wish the calculated mean to behave. It makes no sense to say that we would like the mean rate to be proportional to the total rate, where the total rate is defined as the sum of the individual rates. The sum of the rates would be a meaningless statistic. Rather, we would like the mean to be inversely proportional to the total execution time. For example, if the total time to execute all the benchmark programs in a suite of programs is twice as much for system C as for system D, we would want the mean value of the execution rate to be half as much for system C as for system D.

Let us look at a basic example and first examine how the AM performs. Suppose we have a set of n benchmark programs and record the execution times of each program on a given system as t_1, t_2, ..., t_n. For simplicity, let us assume that each program executes the same number of operations Z; we could weight the individual programs and calculate accordingly but this would not change the conclusion of our argument. The execution rate for each individual program is $R_i = Z/t_i$. We use the AM to calculate the average execution rate.

$$AM = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\sum_{i=1}^{n} \frac{Z}{t_i} = \frac{Z}{n}\sum_{i=1}^{n} \frac{1}{t_i}$$

We see that the AM execution rate is proportional to the sum of the inverse execution times, which is not the same as being inversely proportional to the sum of the execution times. Thus, the AM does not have the desired property.

The HM yields the following result.

$$HM = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{R_i}\right)} = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{Z/t_i}\right)} = \frac{nZ}{\sum_{i=1}^{n} t_i}$$

The HM is inversely proportional to the total execution time, which is the desired property.

EXAMPLE 2.4 A simple numerical example will illustrate the difference between the two means in calculating a mean value of the rates, shown in Table 2.2. The table compares the performance of three computers on the execution of two programs. For simplicity, we assume that the execution of each program results in the execution of 10^8 floating-point operations. The left half of the table shows the execution times for each computer running each program, the total execution time, and the AM of the execution times. Computer A executes in less total time than B, which executes in less total time than C, and this is reflected accurately in the AM.

The right half of the table provides a comparison in terms of rates, expressed in MFLOPS. The rate calculation is straightforward. For example, program 1 executes 100 million floating-point operations. Computer A takes 2 seconds to execute the program for a MFLOPS rate of $100/2 = 50$. Next, consider the AM of the rates. The greatest value is for computer A, which suggests that A is the fastest computer. In terms of total execution time, A has the minimum time, so it is the fastest computer of the three. But the AM of rates shows B as slower than C, whereas in fact B is faster than C. Looking at the HM values, we see that they correctly reflect the speed ordering of the computers. This confirms that the HM is preferred when calculating rates.

The reader may wonder why go through all this effort. If we want to compare execution times, we could simply compare the total execution times of the three systems. If we want to compare rates, we could simply take the inverse of the total execution time, as shown in the table. There are two reasons for doing the individual calculations rather than only looking at the aggregate numbers:

Table 2.2 A Comparison of Arithmetic and Harmonic Means for Rates

	Computer A time (secs)	Computer B time (secs)	Computer C time (secs)	Computer A rate (MFLOPS)	Computer B rate (MFLOPS)	Computer C rate (MFLOPS)
Program 1 (10^8 FP ops)	2.0	1.0	0.75	50	100	133.33
Program 2 (10^8 FP ops)	0.75	2.0	4.0	133.33	50	25
Total execution time	2.75	3.0	4.75	—	—	—
Arithmetic mean of times	1.38	1.5	2.38	—	—	—
Inverse of total execution time (1/sec)	0.36	0.33	0.21	—	—	—
Arithmetic mean of rates	—	—	—	91.67	75.00	79.17
Harmonic mean of rates	—	—	—	72.72	66.67	42.11

1. A customer or researcher may be interested not only in the overall average performance but also performance against different types of benchmark programs, such as business applications, scientific modeling, multimedia applications, and systems programs. Thus, a breakdown by type of benchmark is needed as well as a total.
2. Usually, the different programs used for evaluation are weighted differently. In Table 2.2, it is assumed that the two test programs execute the same number of operations. If that is not the case, we may want to weight accordingly. Or different programs could be weighted differently to reflect importance or priority.

Let us see what the result is if test programs are weighted proportional to the number of operations. Following the preceding notation, each program i executes Zi instructions in a time ti. Each rate is weighted by the instructions count. The weighted HM is therefore:

$$WHM = \frac{1}{\sum_{i=1}^{n}\left(\left(\frac{Z_i}{\sum_{j=1}^{n} Z_j}\right)\left(\frac{1}{R_i}\right)\right)} = \frac{n}{\sum_{i=1}^{n}\left(\left(\frac{Z_i}{\sum_{j=1}^{n} Z_j}\right)\left(\frac{t_i}{Z_i}\right)\right)} = \frac{\sum_{j=1}^{n} Z_j}{\sum_{i=1}^{n} t_i} \quad \textbf{(2.7)}$$

We see that the weighted HM is the quotient of the sum of the operation count divided by the sum of the execution times.

Geometric Mean

Looking at the equations for the three types of means, it is easier to get an intuitive sense of the behavior of the AM and the HM than that of the GM. Several observations, from [FEIT15], may be helpful in this regard. First, we note that with respect to changes in values, the GM gives equal weight to all of the values in the data set. For example, suppose the set of data values to be averaged includes a few large values and more small values. Here, the AM is dominated by the large values. A change of 10% in the largest value will have a noticeable effect, while a change in the smallest value by the same factor will have a negligible effect. In contrast, a change in value by 10% of any of the data values results in the same change in the GM: $\sqrt[n]{1.1}$.

EXAMPLE 2.5 This point is illustrated by data set (e) in Figure 2.6. Here are the effects of increasing either the maximum or the minimum value in the data set by 10%:

	Geometric Mean	Arithmetic Mean
Original value	3.37	4.45
Increase max value from 11 to 12.1 (+10%)	3.40 (+ 0.87%)	4.55 (+ 2.24%)
Increase min value from 1 to 1.1 (+10%)	3.40 (+ 0.87%)	4.46 (+ 0.20%)

A second observation is that for the GM of a ratio, the GM of the ratios equals the ratio of the GMs:

$$GM = \left(\prod_{i=1}^{n} \frac{Z_i}{t_i}\right)^{1/n} = \frac{\left(\prod_{i=1}^{n} Z_i\right)^{1/n}}{\left(\prod_{i=1}^{n} t_i\right)^{1/n}} \tag{2.8}$$

Compare this with Equation 2.4.

For use with execution times, as opposed to rates, one drawback of the GM is that it may be non-monotonic relative to the more intuitive AM. In other words there may be cases where the AM of one data set is larger than that of another set, but the GM is smaller.

EXAMPLE 2.6 In Figure 2.6, the AM for data set d is larger than the AM for data set c, but the opposite is true for the GM.

	Data set c	Data set d
Arithmetic mean	7.00	7.55
Geometric mean	6.68	6.42

One property of the GM that has made it appealing for benchmark analysis is that it provides consistent results when measuring the relative performance of machines. This is in fact what benchmarks are primarily used for: to compare one machine with another in terms of performance metrics. The results, as we have seen, are expressed in terms of values that are normalized to a reference machine.

EXAMPLE 2.7 A simple example will illustrate the way in which the GM exhibits consistency for normalized results. In Table 2.3, we use the same performance results as were used in Table 2.2. In Table 2.3a, all results are normalized to Computer A, and the means are calculated on the normalized values. Based on total execution time, A is faster than B, which is faster than C. Both the AMs and GMs of the normalized times reflect this. In Table 2.3b, the systems are now normalized to B. Again the GMs correctly reflect the relative speeds of the three computers, but now the AM produces a different ordering.

Sadly, consistency does not always produce correct results. In Table 2.4, some of the execution times are altered. Once again, the AM reports conflicting results for the two normalizations. The GM reports consistent results, but the result is that B is faster than A and C, which are equal.

It is examples like this that have fueled the "benchmark means wars" in the citations listed earlier. It is safe to say that no single number can provide all the information that one needs for comparing performance across systems. However,

Table 2.3 A Comparison of Arithmetic and Geometric Means for Normalized Results

(a) Results normalized to Computer A

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (1.0)	1.0 (0.5)	0.75 (0.38)
Program 2	0.75 (1.0)	2.0 (2.67)	4.0 (5.33)
Total execution time	2.75	3.0	4.75
Arithmetic mean of normalized times	1.00	1.58	2.85
Geometric mean of normalized times	1.00	1.15	1.41

(b) Results normalized to Computer B

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (2.0)	1.0 (1.0)	0.75 (0.75)
Program 2	0.75 (0.38)	2.0 (1.0)	4.0 (2.0)
Total execution time	2.75	3.0	4.75
Arithmetic mean of normalized times	1.19	1.00	1.38
Geometric mean of normalized times	0.87	1.00	1.22

Table 2.4 Another Comparison of Arithmetic and Geometric Means for Normalized Results

(a) Results normalized to Computer A

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (1.0)	1.0 (0.5)	0.20 (0.1)
Program 2	0.4 (1.0)	2.0 (5.0)	4.0 (10.0)
Total execution time	2.4	3.00	4.2
Arithmetic mean of normalized times	1.00	2.75	5.05
Geometric mean of normalized times	1.00	1.58	1.00

(b) Results normalized to Computer B

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (2.0)	1.0 (1.0)	0.20 (0.2)
Program 2	0.4 (0.2)	2.0 (1.0)	4.0 (2.0)
Total execution time	2.4	3.00	4.2
Arithmetic mean of normalized times	1.10	1.00	1.10
Geometric mean of normalized times	0.63	1.00	0.63

despite the conflicting opinions in the literature, SPEC has chosen to use the GM, for several reasons:

1. As mentioned, the GM gives consistent results regardless of which system is used as a reference. Because benchmarking is primarily a comparison analysis, this is an important feature.
2. As documented in [MCMA93], and confirmed in subsequent analyses by SPEC analysts [MASH04], the GM is less biased by outliers than the HM or AM.
3. [MASH04] demonstrates that distributions of performance ratios are better modeled by lognormal distributions than by normal ones, because of the generally skewed distribution of the normalized numbers. This is confirmed in [CITR06]. And, as shown in Equation (2.5), the GM can be described as the back-transformed average of a lognormal distribution.

2.6 BENCHMARKS AND SPEC

Benchmark Principles

Measures such as MIPS and MFLOPS have proven inadequate to evaluating the performance of processors. Because of differences in instruction sets, the instruction execution rate is not a valid means of comparing the performance of different architectures.

EXAMPLE 2.8 Consider this high-level language statement:

```
A = B + C   /* assume all quantities in main memory */
```

With a traditional instruction set architecture, referred to as a complex instruction set computer (CISC), this instruction can be compiled into one processor instruction:

```
add   mem(B), mem(C), mem (A)
```

On a typical RISC machine, the compilation would look something like this:

```
load  mem(B), reg(1);
load  mem(C), reg(2);
add   reg(1), reg(2), reg(3);
store reg(3), mem (A)
```

Because of the nature of the RISC architecture (discussed in Chapter 15), both machines may execute the original high-level language instruction in about the same time. If this example is representative of the two machines, then if the CISC machine is rated at 1 MIPS, the RISC machine would be rated at 4 MIPS. But both do the same amount of high-level language work in the same amount of time.

Another consideration is that the performance of a given processor on a given program may not be useful in determining how that processor will perform on a very different type of application. Accordingly, beginning in the late 1980s and early 1990s, industry and academic interest shifted to measuring the performance of

systems using a set of benchmark programs. The same set of programs can be run on different machines and the execution times compared. Benchmarks provide guidance to customers trying to decide which system to buy, and can be useful to vendors and designers in determining how to design systems to meet benchmark goals.

[WEIC90] lists the following as desirable characteristics of a benchmark program:

1. It is written in a high-level language, making it portable across different machines.
2. It is representative of a particular kind of programming domain or paradigm, such as systems programming, numerical programming, or commercial programming.
3. It can be measured easily.
4. It has wide distribution.

SPEC Benchmarks

The common need in industry and academic and research communities for generally accepted computer performance measurements has led to the development of standardized benchmark suites. A benchmark suite is a collection of programs, defined in a high-level language, that together attempt to provide a representative test of a computer in a particular application or system programming area. The best known such collection of benchmark suites is defined and maintained by the Standard Performance Evaluation Corporation (SPEC), an industry consortium. This organization defines several benchmark suites aimed at evaluating computer systems. SPEC performance measurements are widely used for comparison and research purposes.

The best known of the SPEC benchmark suites is SPEC CPU2006. This is the industry standard suite for processor-intensive applications. That is, SPEC CPU2006 is appropriate for measuring performance for applications that spend most of their time doing computation rather than I/O.

Other SPEC suites include the following:

- **SPECviewperf:** Standard for measuring 3D graphics performance based on professional applications.
- **SPECwpc:** benchmark to measure all key aspects of workstation performance based on diverse professional applications, including media and entertainment, product development, life sciences, financial services, and energy.
- **SPECjvm2008:** Intended to evaluate performance of the combined hardware and software aspects of the Java Virtual Machine (JVM) client platform.
- **SPECjbb2013 (Java Business Benchmark):** A benchmark for evaluating server-side Java-based electronic commerce applications.
- **SPECsfs2008:** Designed to evaluate the speed and request-handling capabilities of file servers.
- **SPECvirt_sc2013:** Performance evaluation of datacenter servers used in virtualized server consolidation. Measures the end-to-end performance of all system components including the hardware, virtualization platform, and the virtualized guest operating system and application software. The benchmark supports hardware virtualization, operating system virtualization, and hardware partitioning schemes.

The CPU2006 suite is based on existing applications that have already been ported to a wide variety of platforms by SPEC industry members. In order to make the benchmark results reliable and realistic, the CPU2006 benchmarks are drawn from real-life applications, rather than using artificial loop programs or synthetic benchmarks. The suite consists of 12 integer benchmarks written in C and C++, and 17 floating-point benchmarks written in C, C++, and Fortran (Tables 2.5 and 2.6). The suite contains over 3 million lines of code. This is the fifth generation of

Table 2.5 SPEC CPU2006 Integer Benchmarks

Benchmark	Reference time (hours)	Instr count (billion)	Language	Application Area	Brief Description
400.perlbench	2.71	2378	C	Programming Language	PERL programming language interpreter, applied to a set of three programs.
401.bzip2	2.68	2472	C	Compression	General-purpose data compression with most work done in memory, rather than doing I/O.
403.gcc	2.24	1064	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	2.53	327	C	Combinatorial Optimization	Vehicle scheduling algorithm.
445.gobmk	2.91	1603	C	Artificial Intelligence	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	2.59	3363	C	Search Gene Sequence	Protein sequence analysis using profile-hidden Markov models.
458.sjeng	3.36	2383	C	Artificial Intelligence	A highly ranked chess program that also plays several chess variants.
462.libquantum	5.76	3555	C	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	6.15	3731	C	Video Compression	H.264/AVC (Advanced Video Coding) video compression.
471.omnetpp	1.74	687	C++	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	1.95	1200	C++	Path-finding Algorithms	Pathfinding library for 2D maps.
483.xalancbmk	1.92	1184	C++	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.

Table 2.6 SPEC CPU2006 Floating-Point Benchmarks

Benchmark	Reference time (hours)	Instr count (billion)	Language	Application Area	Brief Description
410.bwaves	3.78	1176	Fortran	Fluid Dynamics	Computes 3D transonic transient laminar viscous flow.
416.gamess	5.44	5189	Fortran	Quantum Chemistry	Quantum chemical computations.
433.milc	2.55	937	C	Physics / Quantum Chromody-namics	Simulates behavior of quarks and gluons.
434.zeusmp	2.53	1566	Fortran	Physics / CFD	Computational fluid dynamics simulation of astrophysical phenomena.
435.gromacs	1.98	1958	C, Fortran	Biochemistry / Molecular Dynamics	Simulates Newtonian equations of motion for hundreds to millions of particles.
436. cactusADM	3.32	1376	C, Fortran	Physics / General Relativity	Solves the Einstein evolu-tion equations.
437.leslie3d	2.61	1273	Fortran	Fluid Dynamics	Models fuel injection flows.
444.namd	2.23	2483	C++	Biology / Molecular Dynamics	Simulates large biomolecu-lar systems.
447.dealII	3.18	2323	C++	Finite Element Analysis	Program library targeted at adaptive finite elements and error estimation.
450.soplex	2.32	703	C++	Linear Pro-gramming, Optimization	Test cases include railroad planning and military airlift models.
453.povray	1.48	940	C++	Image Ray-Tracing	3D image rendering.
454.calculix	2.29	3,04	C, Fortran	Structural Mechanics	Finite element code for linear and nonlinear 3D structural applications.
459. GemsFDTD	2.95	1320	Fortran	Computa-tional Elec-tromagnetics	Solves the Maxwell equa-tions in 3D.
465.tonto	2.73	2392	Fortran	Quantum Chemistry	Quantum chemistry pack-age, adapted for crystallo-graphic tasks.
470.lbm	3.82	1500	C	Fluid Dynamics	Simulates incompressible fluids in 3D.
481.wrf	3.10	1684	C, Fortran	Weather	Weather forecasting model.
482.sphinx3	5.41	2472	C	Speech Recognition	Speech recognition software.

processor-intensive suites from SPEC, replacing SPEC CPU2000, SPEC CPU95, SPEC CPU92, and SPEC CPU89 [HENN07].

To better understand published results of a system using CPU2006, we define the following terms used in the SPEC documentation:

- **Benchmark:** A program written in a high-level language that can be compiled and executed on any computer that implements the compiler.
- **System under test:** This is the system to be evaluated.
- **Reference machine:** This is a system used by SPEC to establish a baseline performance for all benchmarks. Each benchmark is run and measured on this machine to establish a reference time for that benchmark. A system under test is evaluated by running the CPU2006 benchmarks and comparing the results for running the same programs on the reference machine.
- **Base metric:** These are required for all reported results and have strict guidelines for compilation. In essence, the standard compiler with more or less default settings should be used on each system under test to achieve comparable results.
- **Peak metric:** This enables users to attempt to optimize system performance by optimizing the compiler output. For example, different compiler options may be used on each benchmark, and feedback-directed optimization is allowed.
- **Speed metric:** This is simply a measurement of the time it takes to execute a compiled benchmark. The speed metric is used for comparing the ability of a computer to complete single tasks.
- **Rate metric:** This is a measurement of how many tasks a computer can accomplish in a certain amount of time; this is called a **throughput**, capacity, or rate measure. The rate metric allows the system under test to execute simultaneous tasks to take advantage of multiple processors.

SPEC uses a historical Sun system, the "Ultra Enterprise 2," which was introduced in 1997, as the reference machine. The reference machine uses a 296-MHz UltraSPARC II processor. It takes about 12 days to do a rule-conforming run of the base metrics for CINT2006 and CFP2006 on the CPU2006 reference machine. Tables 2.5 and 2.6 show the amount of time to run each benchmark using the reference machine. The tables also show the dynamic instruction counts on the reference machine, as reported in [PHAN07]. These values are the actual number of instructions executed during the run of each program.

We now consider the specific calculations that are done to assess a system. We consider the integer benchmarks; the same procedures are used to create a floating-point benchmark value. For the integer benchmarks, there are 12 programs in the test suite. Calculation is a three-step process (Figure 2.7):

1. The first step in evaluating a system under test is to compile and run each program on the system three times. For each program, the runtime is measured and the median value is selected. The reason to use three runs and take the median value is to account for variations in execution time that are not intrinsic to the program, such as disk access time variations, and OS kernel execution variations from one run to another.

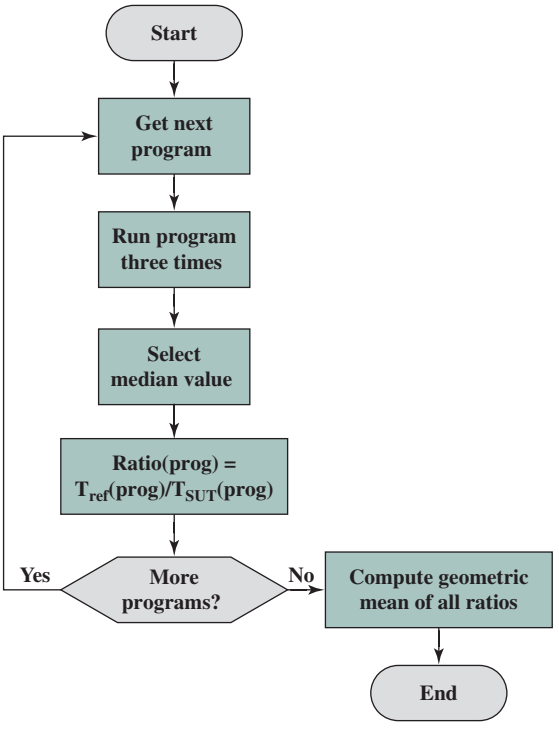


Figure 2.7 SPEC Evaluation Flowchart

2. Next, each of the 12 results is normalized by calculating the runtime ratio of the reference run time to the system run time. The ratio is calculated as follows:

$$r_i = \frac{Tref_i}{Tsut_i} \tag{2.9}$$

where $Tref_i$ is the execution time of benchmark program i on the reference system and $Tsut_i$ is the execution time of benchmark program i on the system under test. Thus, ratios are higher for faster machines.

3. Finally, the geometric mean of the 12 runtime ratios is calculated to yield the overall metric:

$$r_G = \left(\prod_{i=1}^{12} r_i\right)^{1/12}$$

For the integer benchmarks, four separate metrics can be calculated:

- **SPECint2006:** The geometric mean of 12 normalized ratios when the benchmarks are compiled with peak tuning.
- **SPECint_base2006:** The geometric mean of 12 normalized ratios when the benchmarks are compiled with base tuning.
- **SPECint_rate2006:** The geometric mean of 12 normalized throughput ratios when the benchmarks are compiled with peak tuning.
- **SPECint_rate_base2006:** The geometric mean of 12 normalized throughput ratios when the benchmarks are compiled with base tuning.

EXAMPLE 2.9 The results for the Sun Blade 1000 are shown in Table 2.7a. One of the SPEC CPU2006 integer benchmark is 464.h264ref. This is a reference implementation of H.264/AVC (Advanced Video Coding), the latest state-of-the-art video compression standard. The Sun Blade 1000 executes this program in a median time of 5,259 seconds. The reference implementation requires 22,130 seconds. The ratio is calculated as: 22,130/5,259 = 4.21. The speed metric is calculated by taking the twelfth root of the product of the ratios:

$$(3.18 \times 2.96 \times 2.98 \times 3.91 \times 3.17 \times 3.61 \times 3.51 \times 2.01 \times 4.21 \times 2.43 \times 2.75 \times 3.42)^{1/12} = 3.12$$

The rate metrics take into account a system with multiple processors. To test a machine, a number of copies N is selected—usually this is equal to the number of processors or the number of simultaneous threads of execution on the test system. Each individual test program's rate is determined by taking the median of three runs. Each run consists of N copies of the program running simultaneously on the test system. The execution time is the time it takes for all the copies to finish (i.e., the time from when the first copy starts until the last copy finishes). The rate metric for that program is calculated by the following formula:

$$rate_i = N \times \frac{Tref_i}{Tsut_i}$$

The rate score for the system under test is determined from a geometric mean of rates for each program in the test suite.

EXAMPLE 2.10 The results for the Sun Blade X6250 are shown in Table 2.7b. This system has two processor chips, with two cores per chip, for a total of four cores. To get the rate metric, each benchmark program is executed simultaneously on all four cores, with the execution time being the time from the start of all four copies to the end of the slowest run. The speed ratio is calculated as before, and the rate value is simply four times the speed ratio. The final rate metric is found by taking the geometric mean of the rate values:

$$(78.63 \times 62.97 \times 60.87 \times 77.29 \times 65.87 \times 83.68 \times 76.70 \times 134.98 \times 106.65 \times 40.39 \times 48.41 \times 65.40)^{1/12} = 71.59$$

Table 2.7 Some SPEC CINT2006 Results

(a) Sun Blade 1000

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio
400.perlbench	3077	3076	3080	9770	3.18
401.bzip2	3260	3263	3260	9650	2.96
403.gcc	2711	2701	2702	8050	2.98
429.mcf	2356	2331	2301	9120	3.91
445.gobmk	3319	3310	3308	10,490	3.17
456.hmmer	2586	2587	2601	9330	3.61

(*Continued*)

Table 2.7 (*Continued*)

(a) Sun Blade 1000

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio
458.sjeng	3452	3449	3449	12,100	3.51
462.libquantum	10,318	10,319	10,273	20,720	2.01
464.h264ref	5246	5290	5259	22,130	4.21
471.omnetpp	2565	2572	2582	6250	2.43
473.astar	2522	2554	2565	7020	2.75
483.xalancbmk	2014	2018	2018	6900	3.42

(b) Sun Blade X6250

Benchmark	Execution time (secs)	Execution time (secs)	Execution time (secs)	Reference time (secs)	Ratio	Rate
400.perlbench	497	497	497	9770	19.66	78.63
401.bzip2	613	614	613	9650	15.74	62.97
403.gcc	529	529	529	8050	15.22	60.87
429.mcf	472	472	473	9120	19.32	77.29
445.gobmk	637	637	637	10,490	16.47	65.87
456.hmmer	446	446	446	9330	20.92	83.68
458.sjeng	631	632	630	12,100	19.18	76.70
462.libquantum	614	614	614	20,720	33.75	134.98
464.h264ref	830	830	830	22,130	26.66	106.65
471.omnetpp	619	620	619	6250	10.10	40.39
473.astar	580	580	580	7020	12.10	48.41
483.xalancbmk	422	422	422	6900	16.35	65.40

2.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Amdahl's law	functional mean (FM)	microprocessor
arithmetic mean (AM)	general-purpose computing on GPU (GPGPU)	MIPS rate
base metric	geometric mean (GM)	multicore
benchmark	graphics processing unit (GPU)	peak metric
clock cycle	harmonic mean (HM)	rate metric
clock cycle time	instruction execution rate	reference machine
clock rate	Little's law	speed metric
clock speed	many integrated core (MIC)	SPEC
clock tick		system under test
cycles per instruction (*CPI*)		throughput