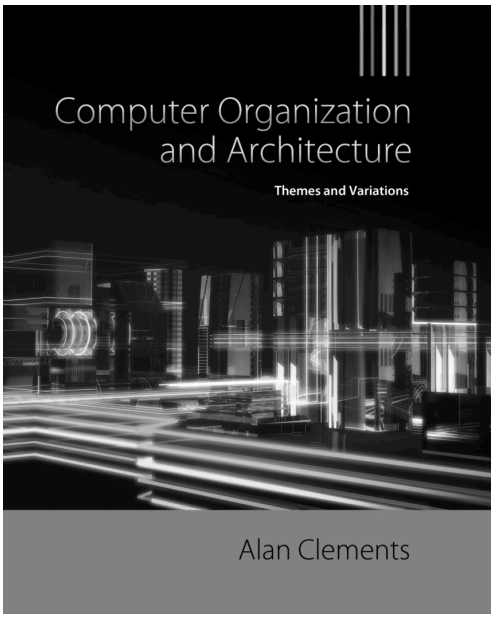
CHAPTER 6

Computer Organization and Architecture



Computer Organization and Architecture: Themes and Variations, 1st Edition Clements

Performance

A little old man goes into a computer store and a sales assistant approaches him.

"Can I help you, sir?"

"Yes. I'm looking for a new computer."

"You've come to the right place. Is there anything you're looking for in particular?"

"Yes, I want one with a SPEC 2006 rating of better than 350 and a total power dissipation of 150 watts. Oh yes, and I'd like a GPU with a rating of 1 TFLOPS"

"Would that be the one with the silver case or the black case, sir?"

These notes examine computer performance. We are interested in:

- What performance is
- How we measure performance
- What factors affect performance

In practice, the performance of a computer is very difficult to quantify as we shall see.

Moore's Law

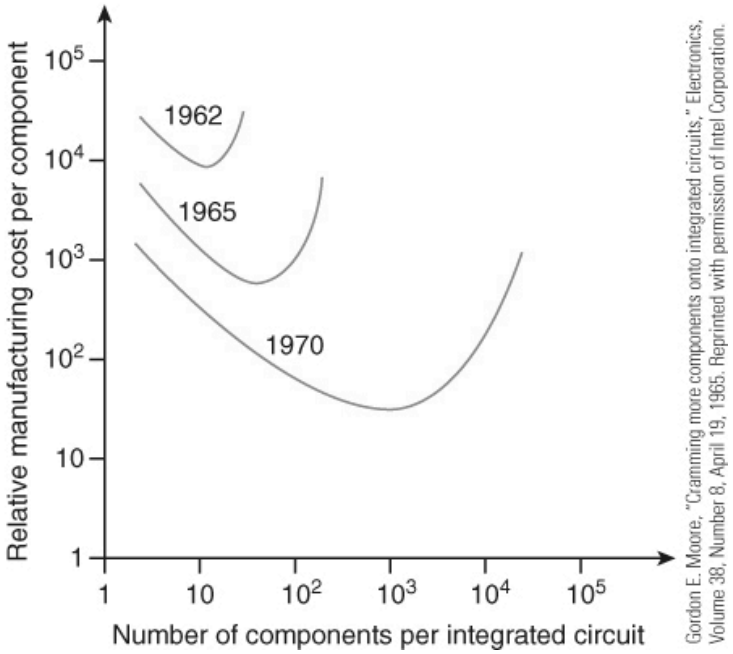
Moore's Law suggests an exponential increase in the maximum number of components on a chip with time.

There's no precise formulation of Moore's Law. The term was coined by Carver Meade in 1970. Over time, Moore's Law has also come to imply a doubling in the performance of digital systems every 18 months.

Moore's Law is based on an observation of the progress of semiconductor technology. The trends that Gordon Moore observed in the 1960s have continued largely unbroken until 2010.

This is something that is probably unique in human endeavor. If a man in 1960 was sauntering along the street and he increased his speed in accordance with Moore's law, he would be moving at 33,554,432 miles/hour in 2010 which is nearly 10,000 miles/s.

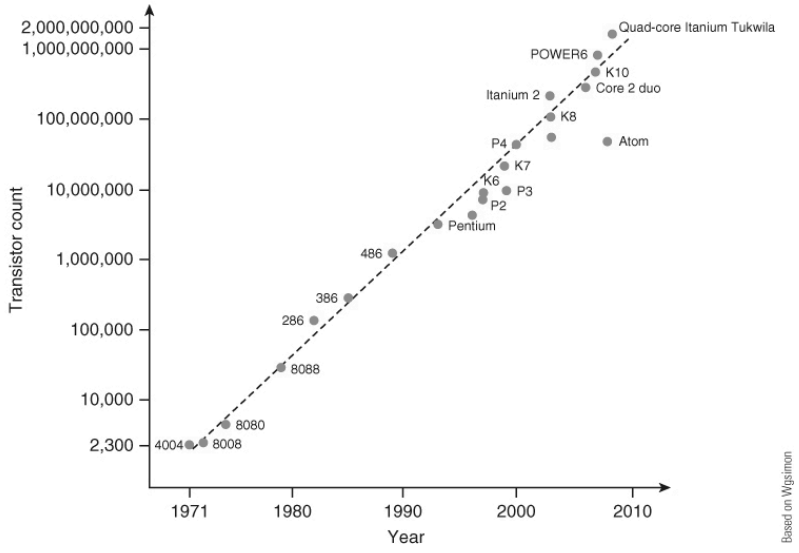
FIGURE 6.1 The basics of Moore's law



Gordon E. Moore, "Cramming more components onto integrated circuits," Electronics, Volume 38, Number 8, April 19, 1965. Reprinted with permission of Intel Corporation.

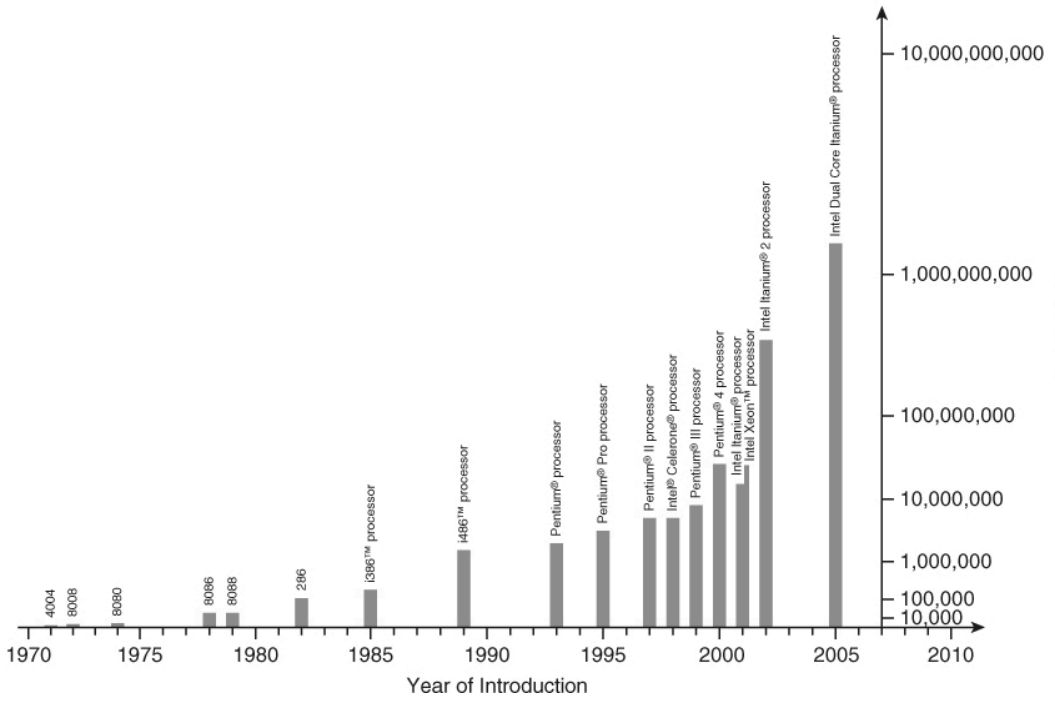
Figure 6.2 displays the number of transistors per microprocessor chip as a function of time. In barely five decades the chip density has gone from in the region of two thousand to two billion devices.

FIGURE 6.2 The number of devices on a circuit



Based on Wgsimon

FIGURE 6.3 Intel's progress in chip design and manufacturing



Reprinted with permission of Intel Corporation.

FIGURE 6.4 3D illustration of microprocessor progress. "De-Mystifying Software Performance Optimization," Intel® Software Network, http://software.intel.com/en-us/articles/de-mystifying-software-performance-optimization/. Reprinted with permission of Intel Corporation.

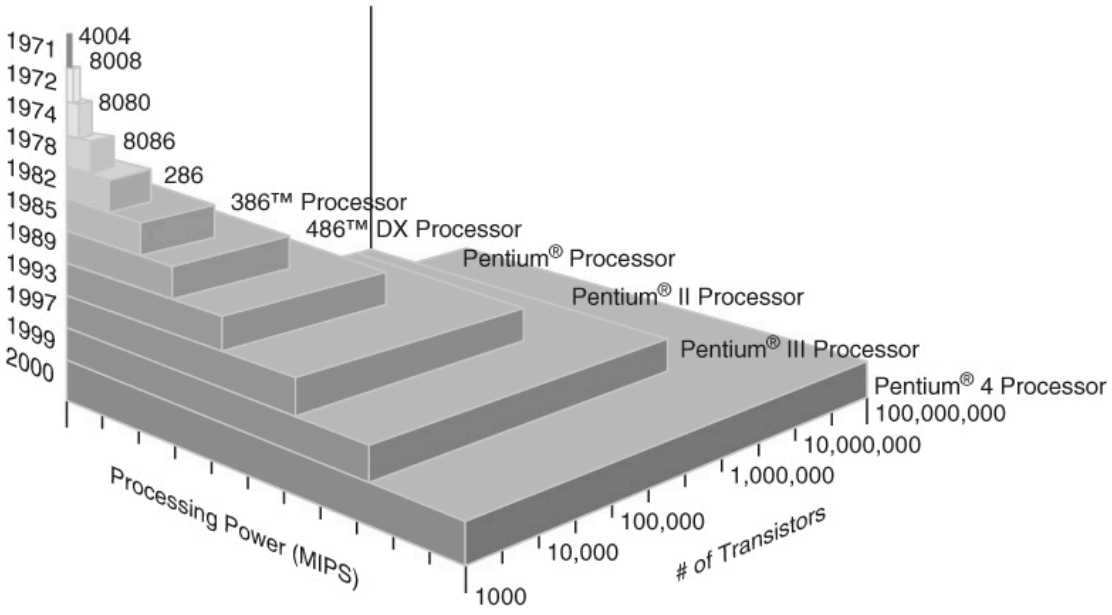


FIGURE 6.5 The growth of memory capacity

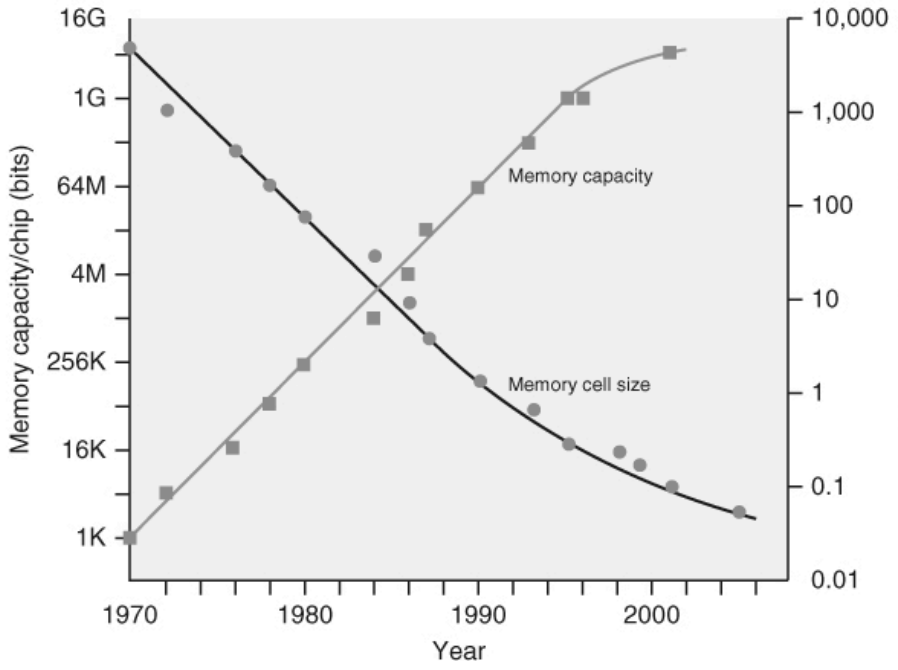


FIGURE 6.6 Hard disk trends – areal density growth by year

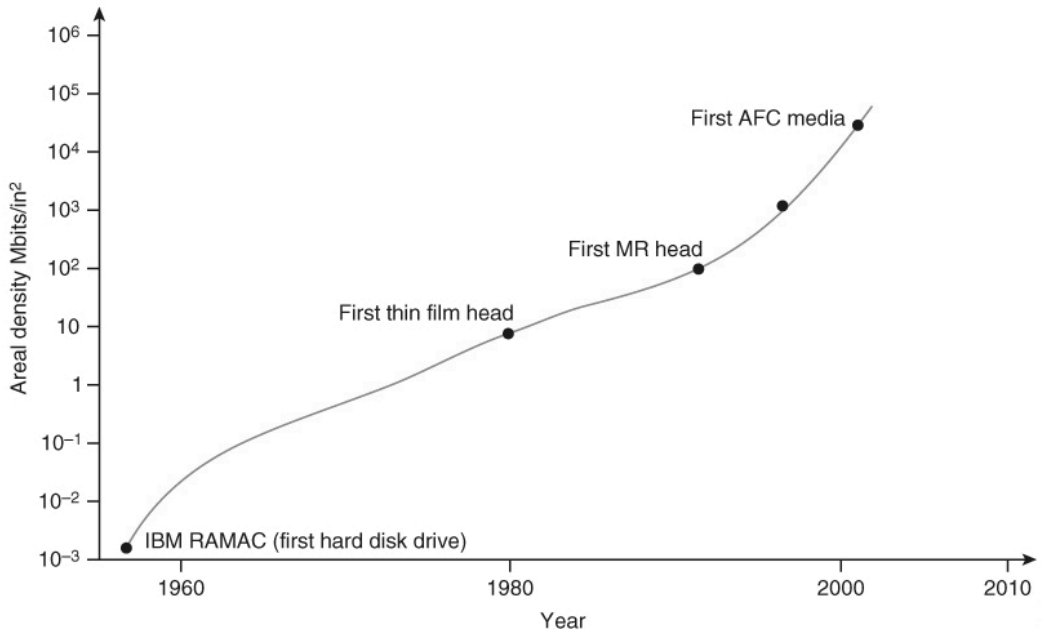
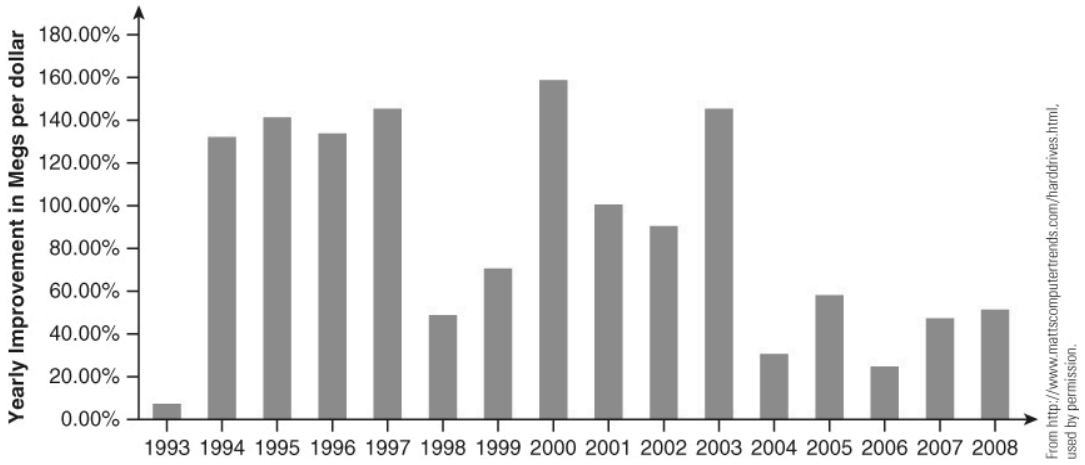


FIGURE 6.7 Disk improvement as a function of price



Performance and Design

Figure 6.8 shows the structure of a typical PC from the point of view of the systems designer.

The performance of a computer is dependent on factors like cache memory, main memory, buses, and secondary storage such as hard disks and optical drives.

Every one of these systems contributes to the overall performance of the computer and you cannot maximize the performance of a computer by improving only one of its components.

The designer's view of the computer is characterized by the parameters and specifications that have to be refined and optimized to suit any specific computer application. Only by generating appropriate metrics can anyone optimize the system being designed.

In principle, you can analyze the performance of a computer mathematically by modeling all its features. However, computers are generally too complex to model sufficiently accurately.

David Lilja provides a list of criteria by which we can judge the metrics of computer performance: linearity, reliability, repeatability, ease of measurement, consistency, and independence .

The criterion suggests that a metric should be linear; that is, increasing the performance of a computer by a fraction should be reflected by an increase of fraction in the metric. If computer has a metric of, say 200, and is twice as fast as computer , then computer 's metric should be 100.

Performance metrics should be and correctly indicate whether one computer is faster than another. You could also call this property *monotonicity*; that is, an increase in the value of a metric should indicate an increase in the speed of the computer and never vice versa. This isn't true of all metrics. Sometimes, a computer may have a metric implying a higher-level performance than another computer when its performance is worse. This situation arises when there is a poor relationship between what the metric actually measures and the way in which the computer operates.

A good metric should be *repeatable* and always yield the same result under the same conditions. Not all computer systems are deterministic because the number of parameters that affect a computer's performance is very large indeed and you can't always achieve the same results for each test run of a program.

Suppose you are carrying out a test that requires the reading of data from a disk. In the first run, the data may be about to fall under the read head at the time it is required and, therefore, the data will be immediately ready.

In the next run of the same test, the data may have just passed under the read head and the system will have to wait for a complete rotation to access the data. In a third run, the data might be cached in RAM and the system will entirely bypass the disk's hardware.

Consequently, we can have three runs of a test yielding three different metrics using the same data.

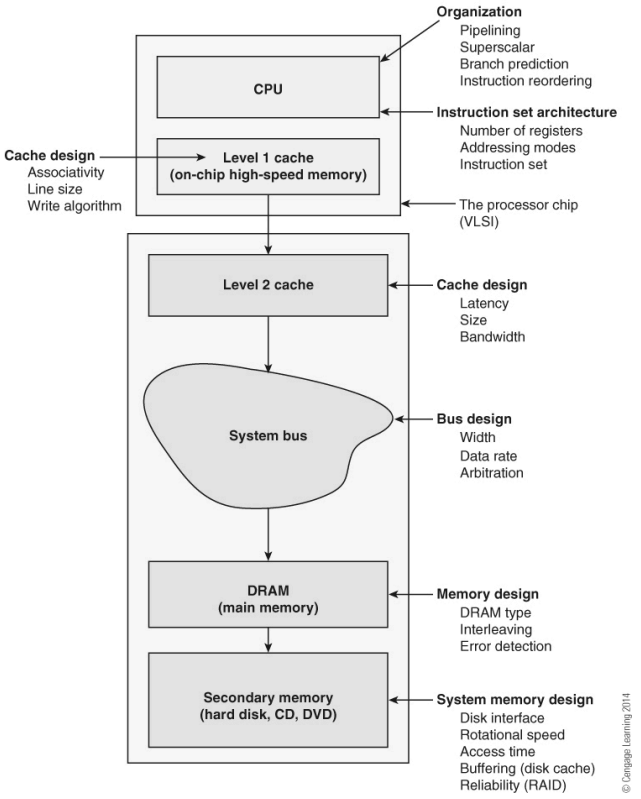
Lilja's *ease of measurement* criterion is self-explanatory. If it is difficult to measure a performance criterion, then few users are likely to make that measurement. Moreover, if a metric is difficult to measure, an independent tester will have great difficulty in confirming it.

A metric is *consistent* if it is precisely defined and can be applied across different systems. Perhaps this metric should be called *universality* or *generality* to avoid confusion with *repeatability*.

Consistency can be difficult to achieve if the metric measures a feature of a specific processor and that feature is not constant across all platforms. Using clock rate as a metric demonstrates a lack of consistency because the relationship between clock rate and performance is not consistent across different platforms. The relationship between clock rate and the performance of a PowerPC processor is not the same as the corresponding relationship between clock rate and a Core i7 processor.

One metric that was commonly used to indicate the performance of graphics cards in PCs in the late 1990s was the maximum number frames per second at which the *Quake* game could refresh the display.

FIGURE 6.8 Performance, the computer, and the designer



Computer Metrics

We're going introduce some of the ways of comparing the performance of various computers. However, we have to state here that computer metrics can be notoriously unreliable. We begin by introducing some of the terminology of performance and then demonstrate why clock frequency alone is not a reliable indicated of performance.

William Thomson (1824 – 1907), the British scientist whose work on transmission lines underpins all modern electronics said:

"When you can measure what you are speaking about, and express it in numbers, you know something about it. But when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind. It may be the beginning of knowledge but you have scarcely in your thoughts advanced to the state of science."

Efficiency

A computer is, of course, always executing instructions unless it is in a halt state or suspended state.

A computer may not always be executing *useful* (i.e., application level) instructions because, for example, it may be repeatedly going round a polling loop waiting for data.

The efficiency of a computer is an indication of the fraction of time that it is doing useful work.

$$Efficiency = Total\ time\ executing\ useful\ work / Total\ time = Optimal\ time / Actual\ time$$

For example, if a computer takes 20 s to perform a computational task and 5 s is taken waiting for a disk that has been idle to spin up to speed, the efficiency is 20s/(20s + 5s) = 20/25 = 80%.

Throughput

The *throughput* of a computer is a measure of the amount of work it performs per unit time.

The upper limit to a system's throughput can normally be determined from basic system parameters; for example, if a computer has a 500 MHz clock and it can execute up to two instructions in parallel per clock cycle and each instruction takes 1, 2, or 4 clock cycles, then the upper limit on throughput occurs when all instructions are being executed in parallel in one cycle; that is 10^9 instructions/s.

Note that the definition of throughput includes the term *amount of work* because instruction execution is meaningful only if the instructions are performing useful calculations; a computer executing an endless stream of NOPs (no operations) may be operating at its peak rate but is achieving nothing other than to wait.

Instructions per second is a very poor indicator of the actual performance of a computer.

Latency

Latency is the delay between activating a process (for example, a memory write or a disk read, or a bus transaction) and the start of the operation; that is, latency is the *waiting time*.

Latency is an important consideration in the design of rotating disk memory systems where, for example, you have to wait on average half a revolution for data to come under the read-write head.

In some computer applications, the effects of latency may be negligible in comparison with processing time.

In some systems, the effects of latency may have an important effect on system performance.

Some define latency at the time to *finish* a process.

Relative Performance

We are interested in how one computer performs with respect to another. The relative performance of computers *A* and *B* is the inverse of their execution times; that is

$$Performance_{A\_to\_B} = \frac{Performance_{ComputerA}}{Performance_{ComputerB}} = \frac{ExecutionTime_{ComputerB}}{ExecutionTime_{ComputerA}}$$

If system *A* executes a program in 105 s and system *B* executes the same program in 125 s, we can calculate the relative performance as 125/105 = 1.190. You can say that machine *A* is 19% faster than *B*.

When you are trying to improve a system, you are often most interested in how much better the new system is in comparison with the old system. The *old* system may be a previous machine, the same machine without the improvement, or even a competitor's machine, and is called the reference machine or baseline machine. The speedup ratio is a measure of *relative performance* and is defined as

$$Speedup\ ratio = \frac{Execution\ time\ on\ reference\ machine}{Execution\ time}$$

If a reference machine takes 100 seconds to run a program and the test machine takes 50 seconds, the speedup ratio is 100/50 = 2.

Time and Rate

Benchmarks can be expressed as the *time* required to execute a task or as the *rate* at which tasks are executed.

For example, one benchmark may yield a time of 20 s, whereas another benchmark may yield a rate of 12 tasks/s.

The computer game *Quake* has become a popular benchmark for PCs with the figure of merit being the rate at which frames are displayed by the processor (although the Quake frame rate is probably a reasonable indication of how your computer performs relative to other computers running Quake, it is not a good general benchmark).

They say people feel more comfortable with metrics that increase numerically with performance (i.e., rate) rather than those that reduce with performance (time).

Time and *rate* benchmarks don't behave in the same way with respect to averaging.

Suppose we benchmark a computer and get execution times for tasks *A* and *B*, respectively, of 2 and 4 seconds.

We can also say that the rates at which tasks *A* and *B* execute are 0.5/s and 0.25/s, respectively.

The average *execution time* is ½(2 + 4) = 3s. The average rate of execution for the tasks is ½(0.5 + 0.25) = 0.375.

The average execution time is 3s. This is an average rate of 1/3s = 0.333/s, which is not the same as the 0.375/s that we calculated by averaging the rates.

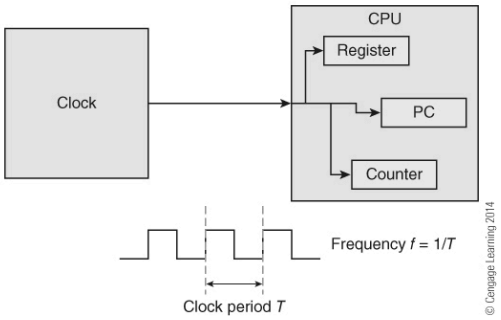
Clock Rate

The obvious indicator of a computer's performance is its *clock rate*, the speed at which internal operations are carried out within the computer.

Using the clock rate as a metric to compare processors is probably the worst metric by which to judge computers.

Figure 6.9 illustrates the CPU clock. At each clock cycle, the processor carries out an internal operation. At first sight it's tempting to think that the processor's performance is proportional to its clock rate and therefore clock rate is a precise metric.

FIGURE 6.9 The CPU's clock



There are so many flaws in this argument that it's difficult to know where to begin.

There is no *single* clock in most computers. Some systems may have entirely independent clocks (i.e., there's a separate clock generator for each functional part such as the CPU, the bus, and the memory).

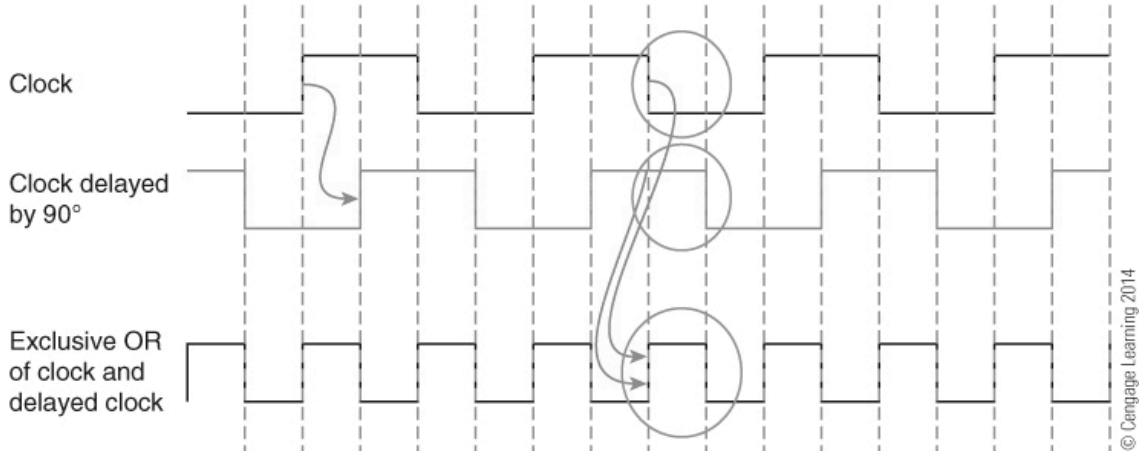
The CPU itself may have several functional units, each driven by its own clock. Some systems have a single master clock that generates pulses at the highest rate required by any circuit and all other clocks run at a sub-multiple of this frequency.

Some processors have variable clock rates. For example, *mobile* processors designed for use in laptops can reduce the clock rate to conserve power. This is called *clock throttling* or *dynamic frequency scaling*.

Some processors switch to a lower clock speed if the core temperature rises and the chip is in danger of overheating.

Note – the frequency of a clock can be doubled using simple logic

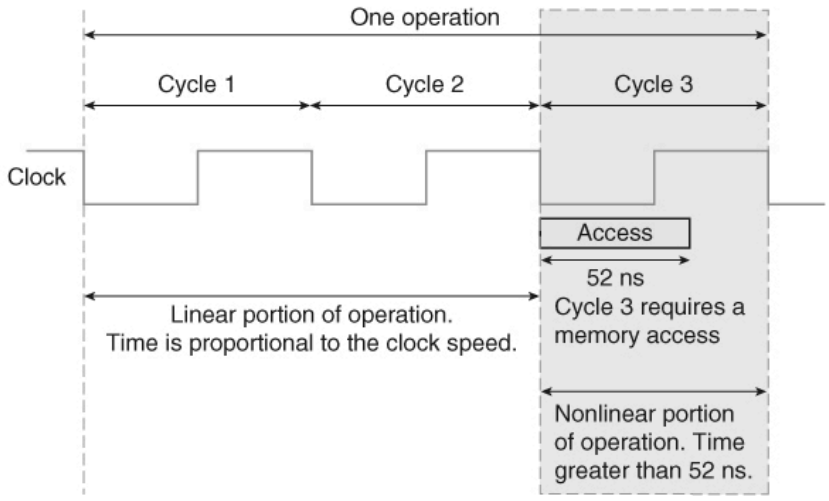
FIGURE 6.10 Clock doubling



Clock scaling

Let's look at speeding up the clock. Consider Figure 6.11.

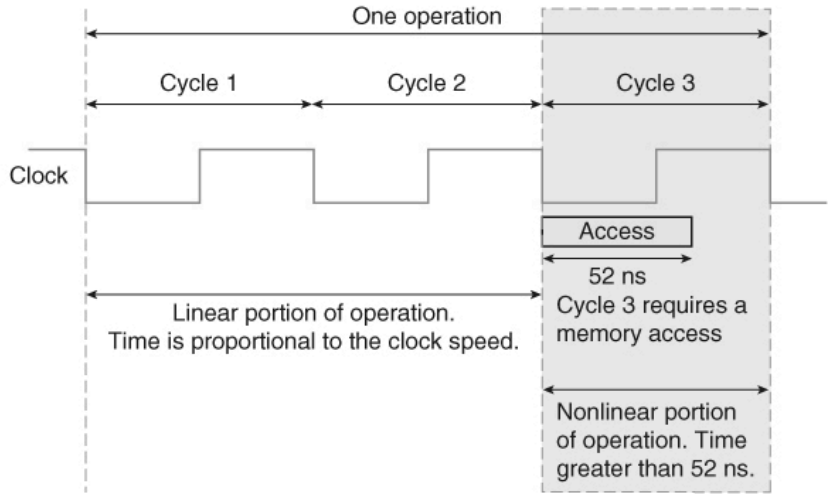
FIGURE 6.11 Demonstrating the non-scalability of clocks



Three cycles are required to perform an operation. The first two clock cycles are *scalable* and we can increase the clock rate.

The third cycle has an operation that requires 52 ns for its completion. If the clock cycle is less than 52 ns then cycle three **must** be stretched either by slowing the clock or by lengthening cycle three by two or wait states.

FIGURE 6.11 Demonstrating the non-scalability of clocks

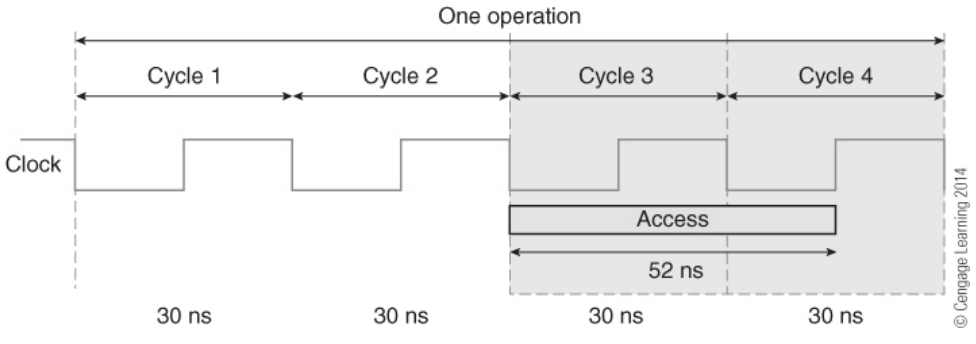


It's difficult to alter the clock rate from clock pulse to clock pulse.

Most systems extend an operation by an integer number of cycles called *wait states*. In figure 6.12, the cycle time is 30 ns and the 52 ns memory access requires two cycles. The operation now takes 4 x 30 ns to complete.

The non-scalable clock property of systems described by Figure 6.11 demonstrates why some computers become unstable if you try and *overclock* them (a mechanism beloved of hackers who clock processors at a rate higher than that specified by the manufacturer).

FIGURE 6.12 The effect of introducing wait states



Is Megahertz Enough?

The following is taken from an IDC White Paper written by Shane Rau *and sponsored by AMD .*

PC buyers often rely on the clock speed of a PC's microprocessor to determine their purchasing decision. Because the industry lacks a simple, universally accepted way to judge performance, users have become conditioned to substituting clock speed to gauge how fast their applications will run. This practice has grown common over many years because:

- The popularization of the PC among general consumers has increased the available pool of buyers unfamiliar with factors in PC performance.
- The growth of the direct model of PC purchases has made it more likely that the actual end user will buy a PC without the help of a third party familiar with factors that influence PC performance.
- The increasing sophistication of the PC exposes the buyer to a growing number of often arcane technical specifications, from which clock speed promises a convenient escape.

In some ways, the power barrier has rendered clock speed arguments rather moot.

Since about 2008, clock speed has ceased to increase dramatically because the limits of power dissipation have been reached and power dissipation is proportional to the *square* of the clock frequency.

Manufacturers have directed their efforts towards multicore processors rather than faster processors.

Clock speeds may rise again if power consumption falls because of the introduction on new semiconductor materials or because of circuit innovations such as asynchronous clocking where circuits are not driven by a master clock and one event triggers another.

Asynchronous circuits can operate at up to 70% lower power levels than their clocked counterparts, but design, testing, and verification is problematic.

MIPS

A slightly better metric than clock rate is MIPS, or *millions of instruction per second*.

This metric removes the discrepancy between systems with different numbers of clocks per operation by measuring instructions per second rather than clocks per second.

For a given computer

$$MIPS = \frac{n}{t_{execute} \times 10^6}$$

where n is the number of instructions executed and $t_{execute}$ is the time taken to execute them.

The MIPS rating is a poor metric that fails for the same reason as the clock rate.

MIPS tells you only how fast a computer executes instructions, but doesn't tell you what is actually achieved by the instructions being executed.

Consider the following example of computation on two computers *A* and *B*, where computer *A* has a load/store architecture without a multiplier and *B* has a memory-to-register architecture. Computer *A* is more verbose than *B*. Both the computers evaluate the expression z = 4(x + y).

Computer A (LOAD/STORE)	Computer B (memory-register)

LDR r1,[r0] ;load x
LDR r2,[4,r0] ;load y
ADD r2,r1,r2 ;x+y
ADD r2,r1,r2 ;2(x+y)
ADD r2,r2,r1 ;4(x+y)
STR r2,[8,r0] ;store z
	LDR r1,[r0] ;load x
	ADD r1,[4,r0] ;x+y
	MUL r1,#4 ;4(x+y)
	STR r1,[8,r0] ;store z

Suppose computers *A* and *B* have the same MIPS and execute code equally rapidly.

If you relied solely on MIPS as a metric of performance, you'd conclude that the computers offer the same performance.

As you can see, computer *B* is faster than computer *A* because only four instructions are required to do the work (this argument is based on the assumption that all instructions take the same time).

In practice, computer *A* might be faster than *B* because memory-to-register architectures are slower than register-to-register architectures.

MIPS is sensitive to the way in which a compiler generates code.

The duration of a single instruction is *cycles* x t_{cycle}, where *cycles* is the number of machine cycles required to execute the instruction and t_{cycle} is the cycle time (usually the clock period).

The total execution time for a program is given by:

$t_{execution} = t_{cycle} \times \sum n_i \times c_i$

where n_i is the number of times instruction i occurs in the program and c_i is the number of cycles required by instruction i.

If we plug this formula into the equation for MIPS, we get

$$MIPS = \frac{n}{t_{cycle} \times \sum n_i \times c_i \times 10^6}$$

Consider the following example. A program is compiled to run on a computer and the compiler generates two million one-cycle instructions and one million two-cycle instructions. If we assume that the cycle time is 10 ns, the time taken is given by:

$2 \times 10^6 \times 1 \times 10$ ns + $1 \times 10^6 \times 2 \times 10$ ns = $4 \times 10^6 \times 10$ ns = 4×10^{-2}s.

Suppose that a different compiler generates code for this problem but with 1.5 million one-cycle instructions and 1.2 million two-cycle instructions. Now the time required to execute the code is

$1.5 \times 10^6 \times 1 \times 10$ ns + $1.2 \times 10^6 \times 2 \times 10$ ns = $3.9 \times 10^6 \times 10$ ns = 3.9×10^{-2}s.

The second compiler generated faster code. Now let's evaluate the MIPS for each case. In the first case, the MIPS is given by

$$\frac{n}{t_{cycle} \times \sum n_i \times c_i \times 10^6}$$

MIPS = $3 \times 10^6/(10$ ns x $(2 \times 10^6 \times 1 + 1 \times 10^6 \times 2) \times 10^6) = 0.75 \times 10^2 = 75$ MIPS

In the second case, the MIPS is given by

MIPS = $2.7 \times 10^6/(10$ ns x $(1.5 \times 10^6 \times 1 + 1.2 \times 10^6 \times 2) \times 10^6)$
$= 0.69 \times 10^{-2} = 69$ MIPS

The faster computer has a lower MIPS even though it is clearly superior to the slower computer.

This failure of the MIPS metric is inevitable because instruction throughput takes no account of how much work each instruction actually performs.

The average number of clock cycles per instruction is determined by the instruction mix (i.e., the relative number of 1-cycle, 2-cycle, 3-cycle instructions etc.) and by the organization or logic design of the processor.

$$CPI = \sum_{i=1}^{N} F_i \, x \, C_i$$

F_i is the fraction of instructions taking C_i clock cycles to execute.

In a *non-scalar* processor, the value of i ranges from 1 (one instruction per cycle) to N.

The value of N is the length of the longest instruction in terms of clock cycles.

The clock period is governed by: device physics, the ability to dissipate heat, and the chip's logic design.

The rate at which signals propagate through semiconductor devices can be improved only either shrinking the device or by changing the semiconductor's electronic properties.

When a signal changes level in a chip, it is necessary to charge inter-electrode capacitances of the transistors on the chip. This process requires energy and raising the switching rate increases the energy consumption.

Since all energy eventually ends up as heat, increasing the clock rate causes the chip to become hotter. This heat can be dissipated only via conduction to the outside. The maximum clock rate is often limited by the ability of system to dissipate heat.

A benchmark runs on a hypothetical processor to give the results in Table 6.3. We can obtain the cycles per instruction for each class of operation by multiplying the frequency by the instruction to get the values in Table 6.4.

TABLE 6.3 Relative Instruction Frequency and Cycle Count for a Computer

Machine Operation	Relative Frequency	Cycles per Instruction
Arithmetic/logical instruction	53%	1
Register load operation	20%	4
Register store operation	7%	2
Unconditional branch instruction	12%	1
Conditional branch instruction	8%	2

TABLE 6. 4 Calculating the Average CPI for the System in Table 6.3

Machine Operation	Frequency	Cycles	CPI
Arithmetic/logical instruction	53%	1	0.53
Register load operation	20%	4	0.80
Register store operation	7%	2	0.14
Unconditional branch instruction	12%	1	0.12
Conditional branch instruction	8%	2	0.16
Average cycles per instruction			1.75

We can also relate the relative instruction frequencies to the percentage of time an instruction class takes by multiplying the instruction class frequency by the number of cycles and dividing the result by the average cycles per instruction (i.e., 1.75), Table 6.5.

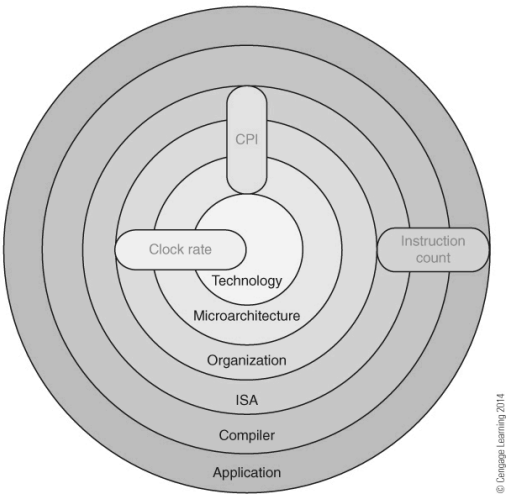
In Table 6.5 the register load instruction takes up 20% of the code, but is responsible for 45.71% of the processor cycle time; this is telling us that loading registers from memory is expensive.

TABLE 6.5 Calculating the Average Time Spent Executing Each Instruction Class

Machine Operation	Frequency	Average Cycles	Average Time
Arithmetic/logical instruction	53%	$1 \times .53 = 0.53$	30.29%
Register load operation	20%	$4 \times .2 = 0.80$	45.71%
Register store operation	7%	$2 \times .07 = 0.14$	8.00%
Unconditional branch instruction	12%	$1 \times 0.12 = 0.12$	6.86%
Conditional branch instruction	8%	$2 \times .08 = 0.16$	9.14%

Figure 6.14 sums up the relationship between clock rate, MIPS, instruction count, and computer systems design. The concentric layers represent the components of a computer from its technology to the programs that run on it. The three ovals in blue represent the factors that affect the performance of the system, such as clock rate. This figure demonstrates (approximately) the relationship between the design layers and the performance factors.

FIGURE 6.14 Factors affecting computer performance



MFLOPS

MFLOPS indicates *millions of floating-point instructions per second*.

In principal, the same objections to MIPS apply to the MFLOPS metric and therefore you might expect MFLOPS to be as poor an indicator of performance as MIPS.

MFLOPS is a better metric than MIPS because MFLOPS measures the *work* done rather than instruction throughput.

MIPS counts *all* instructions executed by a computer, many of which perform no useful work in solving a problem (e.g., data movement operations).

MFLOPS considers only *floating-point operations* which are at the heart of the algorithm being implemented.

The MFLOPS metric isn't easy to understand because all computers don't implement floating-point arithmetic in the same way.

One computer might use dedicated hardware to calculate a trigonometric function such as a sine, whereas another computer might evaluate sin(x) by directly evaluating the appropriate series for sin(x).

Amdahl's Law

The most famous *law* governing computer performance is *Amdahl's law*.

It's also an *infamous* law because it appears to place a limit on the maximum performance increase that can be achieved by optimizing a computer's subsystems.

Amdahl's law describes the performance increase you get when a program is run in a system where some of the operations can take place in parallel.

Amdahl's law tells you what performance increase you get for greater parallelism.

Amdahl's law is applicable to any system where you are interested in the effect of local improvements on the system globally.

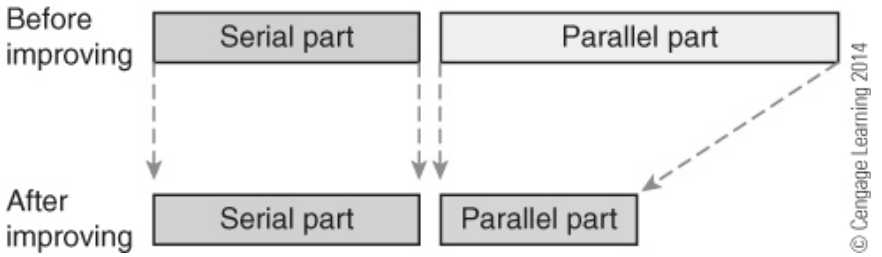
Amdahl's law highlights the effects of bottlenecks in a system.

Figure 6.15 illustrates the effect of parallelizing part of an activity.

The diagram demonstrates that the serial (irreducible) part of the process remains the same while the parallel (reducible or improvable) part of the system is reduced.

Ultimately, system performance is dominated by the serial part of the system and the motto of the computer designer has become *make the common case fast*.

FIGURE 6.15 Illustration of Amdahl's law



Suppose a computer executes a program on a single processor in time t_s seconds. If we have p processors and the program is divided into p equal chunks, the same program will run in t_s/p seconds.

Assume that a fraction of the program f_p can run on the p processors and a fraction f_s can run only on one processor.

The execution time on the parallel computer system T_p will be:

$T_p = t_s \cdot f_s + t_p \cdot f_p$

Since $f_s + f_p = 1$ and $t_p = t_s/p$, we can write $T_p = t_s(f_s + (1 - f_p)/p)$

The speedup ratio for this system is the ratio of the speed without parallelization to the speed with parallelization; that is, $S = t_s/T_p$.

$$S = \frac{1}{f_s + \frac{1 - f_s}{p}} = \frac{p}{pf_s + 1 - f_s}$$

Amdahl's law illustrates the effect of a bottleneck on system performance and shows that there is a limit beyond which attempts at further improvement are futile, unless the bottleneck can be removed.

The most popular formulation of Amdahl's law is

$$S = \frac{1}{1 - Fraction_{enhanced} + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Consider the two cases in Table 6.9 in which $f_s = 0.2$ and $f_s = 0.1$, respectively for various degrees of parallelism.

The speedup ratio falls off rapidly once the serial part of the process begins to dominate the equation.

In the limit, an infinite number of processors cannot make the speedup ratio greater than the reciprocal of the fraction of time devoted to serial processing.

TABLE 6.9 Effect of Amdahl's Law

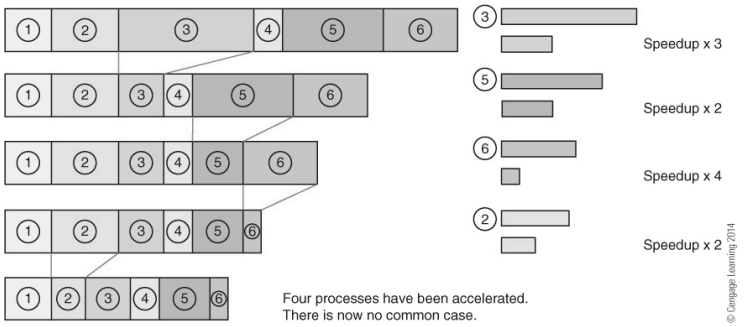
Processors	Speedup Ratio $f_s = 0.2$	Speedup Ratio $f_s = 0.1$
1	1	1
2	1.667	1.818
3	2.143	2.500
4	2.500	3.077
5	2.778	3.571
10	3.571	5.263
100	4.808	9.174
∞	5.000	10.00

Let's look at a second example of Amdahl's law where we have an operation that consists of six sequential operations or processes as described in Figure 6.16.

Five of the processes are accelerated using the stated factors. This shows the change in the total time after each process has been accelerated.

By the time the bottom line has been reached, the six processes are now of roughly similar durations. We have reached the state at which the common case has been eliminated and there is nothing left to accelerate.

FIGURE 6.16 Amdahl and the common case



Benchmarks

The ideal program with which to evaluate a computer is the one you are going to run on that computer. Unfortunately, it's normally impractical to carry out such a test. Moreover, computers execute a mix of programs that changes from moment to moment.

One approach to benchmarking is based on *kernels* or fragments of real programs that require intense computation, such as the LINPACK benchmark. Another approach is to run *synthetic* benchmarks that are programs constructed for the express purpose of evaluating computer performance and which purport to be similar to the type of code that users might actually execute.

Benchmarks can be divided into two categories: *fine-grained* and *coarse-grained*. The granularity of a benchmark is a function of the object being measured; for example, a benchmark that measures the performance of a complete computer system can be considered coarse-grained, whereas a benchmark that measures the performance of, say, branch instructions, can be considered fine-grained.

SPEC

Various organizations exist to offer the consumer *unbiased* advice, whether it be about the quality of wines or the safety of automobiles.

Such an organization has arisen to create benchmarks. SPEC, the *Standard Performance Evaluation Corporation,* is a non-profit making organization with the status of a charity that's been formed to *establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers*.

The user is responsible for compiling the benchmarks before they are run. SPEC allows two types of compilation. One type of compilation is restrictive in terms of compiler flags and switches and that provides a *base* result. The other type of compilation is more liberal and allows switches to be optimized for each individual program. This provides a *peak* metric.

SPEC benchmarks appeared in 1988 with SPEC89, a suite of ten programs. The next major change was in 1992 when SPEC CINT92 (6 integer programs) and SPEC CFP92 (14 floating-point programs) were published. In 1995 SPEC CINT95 and SPEC CFP95 (8 integer, 10 floating-point programs, respectively) were introduced. The fourth revision to the benchmarks was SPEC CPU2000 with SPEC CINT2000 and SPEC CFP2000 (12 integer and 14 floating-point programs). SPEC 2000 also introduced benchmarks in C, C++, Fortran 77, and Fortran 90. SPEC 2006 uses 12 integer programs and 17 floating-point programs.

SPEC suites have changed to remove some of their inefficiencies and errors; for example, in 2001 SPEC CPU95 was *retired* and replaced by CPU2000.

In turn, CPU2000 was replaced by CPU2006. The benchmark programs of a SPEC suite vary over a wide range of applications; from Lisp interpreters to compilers to data-compression programs in older SPECs, and from Quantum mechanics to fluid dynamics in more recent SPECs. The complexity of applications grows with each new set of SPEC benchmarks to reflect the increasing demand on computers.

SPEC Methodology

The SPEC methodology is to measure the time required to execute each program in the test suite; for example, the times might be T_{p1}, T_{p2}, T_{p3},… where the subscripts p1, p2, etc refer to the components of the suite.

The SPEC suite is also executed on a so-called *standard basis machine* to give the values B_{p1}, B_{p2}, B_{p3},…

The measured times are divided by the reference times to give the values T_{p1}/B_{p1}, T_{p2}/B_{p2}, T_{p3}/B_{p3},… This step normalizes the execution times of the components of the SPEC suite. Finally, the *normalized* times are averaged to generate a final SPEC metric for the machine being tested.

Here, we will simply state that the individual SPEC benchmarks are averaged by taking the *geometric mea*n, rather than the arithmetic mean.

The geometric mean is calculated by multiplying together *n* values and then taking the *n*th root; for example, the arithmetic mean of the values 4,5,6 is (4+5+6)/3 = 3 and the geometric mean is $\sqrt[3]{4 \times 5 \times 6} = 3.915$.

The advantage of normalizing a machine with respect to a standard machine is that large differences in individual benchmarks is reduced.

Suppose that a reference machine takes 10, 100, and 5 seconds to perform tasks A, B, and C, respectively. Now, imagine that three test machines M1, M2, and M3 are tested to give the results specified in Table 6.14.

Machine M2 has the best execution (total) time.

TABLE 6.11 Test Results for Three Hypothetical Machines

	Task A	Task B	Task C	Total
Reference machine	10	100	5	115
Machine M1	10	200	5	215
Machine M2	20	100	5	125
Machine M3	20	100	20	140

Table 6.15 gives the same results normalized to the reference machine.

As you can see, test machine M3 has the worst case execution time.

TABLE 6.12 Test Results for Three Hypothetical Machines after Normalization

	Task A	Task B	Task C	Total
Machine M1	1	2	1	4
Machine M2	2	1	1	4
Machine M3	2	1	4	7

The SPEC benchmarks are sometimes called the *best of a bad lot* or the *best metrics in the absence of anything that is more successful*.

A criticism of SPEC is that it is CPU intensive and doesn't test the computer system. In particular, the effect of memory systems is not fully taken into account. Moreover, the SPEC tests are not necessarily representative of the type of workload found in a multitasking environment – although the situation appears to have improved with the introduction of SPEC CPU2006.

The SPEC organization has been criticized for periodically changing its benchmarks. You could say that this prevents all systems being compared against an agreed baseline. Equally, you could say that the nature of computers is changing and the way in which they are used is changing, and therefore it is essential to update the basis for comparison.

The SPEC benchmarks are sometimes said to be given in *unitless* values because the actual measurements for a given machine are divided by the same measurements for the reference or baseline machine.

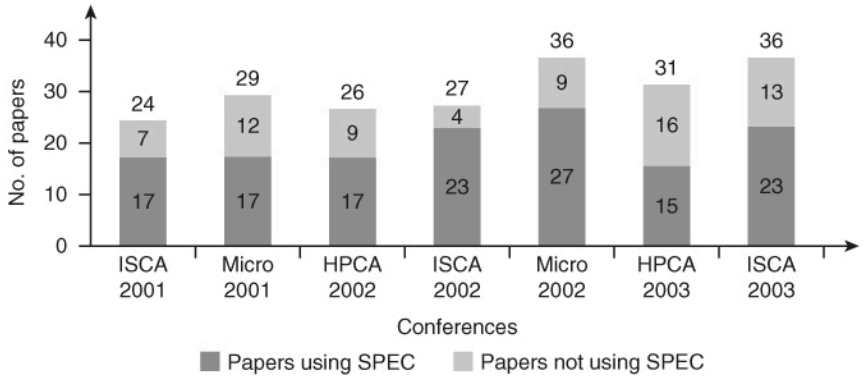
For example, if the test machine gives the values 1s 10s 10s and the reference machine gives 2s 10s 20s, the normalized unitless results are ½ 1 ½.

However, we will see that these values do have the properties of either *time* or *rate* with respect to averaging and it is necessary to employ the appropriate averaging technique.

The SPEC benchmarks have proved very successful in some areas. Figure 6.19 gives the number of papers on computer architecture submitted to computer architecture conferences between 2001 and 2003.

These papers are divided into those that use the SPEC benchmark and those that don't. SPEC is employed by a large majority of researchers.

FIGURE 6.22 The use of SPEC benchmarks in published papers. © 2003 IEEE. Reprinted, with permission, from Hennessy, J.; Citron, D.; Patterson, D.; Sohi, G. "The Use and Abuse of SPEC: An ISCA Panel," IEEE Micro, vol. 23, no. 4, pp. 73–77, July/Aug. 2003.



SPEC CPU2006 and the Academic Community

One of the key users of SPEC benchmarks is the computer architecture research community. A paper by Sarah Bird provides an insight into how SPEC benchmarks are used. This paper looks at CPU2006 benchmarks as applied to Intel's Core processors. Both CPU2006 integer and floating point benchmarks are examined in terms of their interactions with the processor architecture; for example, the benchmarks are broken down in terms of instruction mix. It is instructive to see the percentages of branches, loads and studies in the 12 integer and 17 floating-point benchmarks.

In the integer benchmarks, the percentage of branches ranges from 7.5% to 27.3%; the percentage of loads ranges from 14.4% to 35%, and the percentage of stores ranges from 4.6% to 17.7%.For the floating-point benchmarks, the corresponding ranges are 0.2 – 16.4% (branches), 23.3% - 46.5% (loads), and 3.0% - 14.5% (stores).

These figures tell you what aspects of the computer architecture are being tested (or *stressed*) by the benchmarks.

SPEC CPU2006 Benchmarks

SPEC CPU2006 integer benchmarks are written in either C or C++, two languages that are widely used in academia, the graphics world, and high-performance computing.

These benchmarks cover both commercial and scientific applications. One of the key features of CPU2006 is that the benchmarks represent real-world applications rather than synthetic applications.

The 17 floating-point SPEC CPU2006 benchmarks are described by are written in C, C++, and FORTRAN (used in numerical applications in the engineering world.

SPEC and Power

Power is now a first-class consideration in computer design and had become a component of benchmarking. SPEC has created a standard to measure a server's power and performance across multiple loads, SPECpower_ssj2008.

The metric reports the server's performance in ssj_ops (*server side Java operations per second*) divided by the power consumed in watts.

This benchmark is primarily intended for servers in commercial applications where power consumption is of importance; that is, the cost of buying the power at a time when energy costs are rising, the cost of cooling and ventilation to remove the power, and the additional cost due to increased failure rate in hot environments.

Figure 6.24 from Dell provides an example of the normalized performance per watt for four commercial servers.

FIGURE 6.24 SPECpower_ssj2008 performance per watt