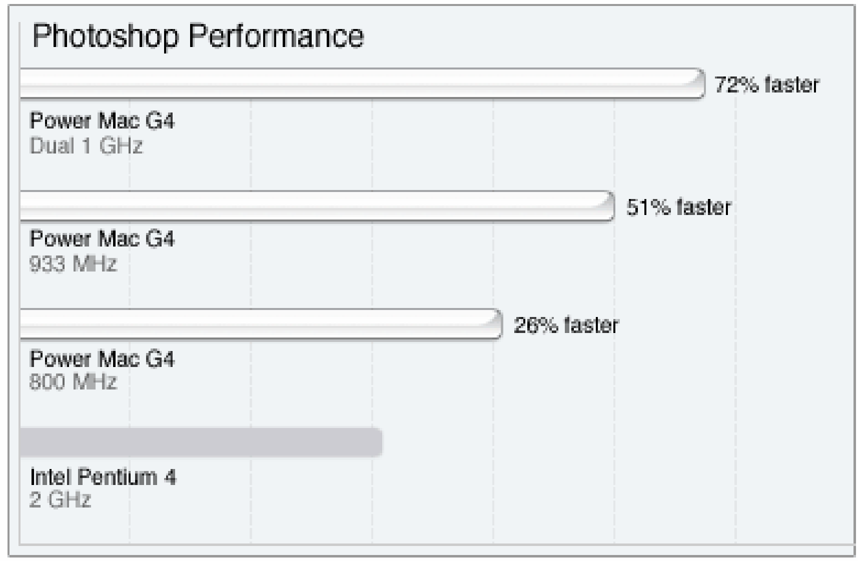
Performance

- Today we'll try to answer several questions about performance.
  - Why is performance important?
  - How can you define performance more precisely?
  - How do hardware and software design affect performance?
  - How do you measure performance in the real world?
- We'll use the ideas from today to evaluate the different CPU designs that are coming up in the next several weeks.

Performance for users

- Users want to evaluate all of the seemingly conflicting claims in the industry, and get the most for their money.



AMD

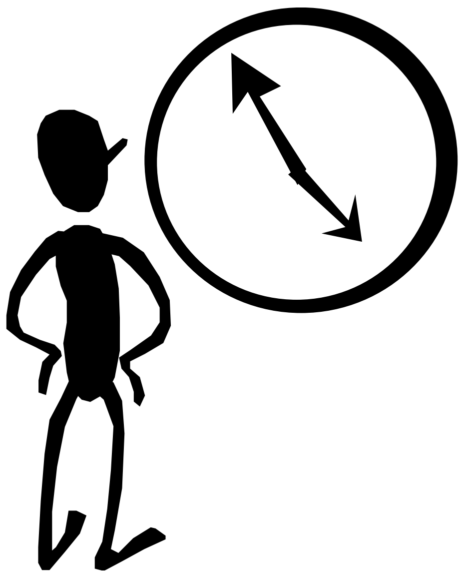
Q: *Why do end users need a new performance metric?*
A: End users who rely only on megahertz as an indicator for performance do not have a complete picture of PC processor performance and may pay the price of missed expectations.

Performance for developers

- Programmers are interested in making faster software.
  - Software can take advantage of new hardware features, such as extra instructions or graphics processors for games.
  - At the same time, programmers should avoid writing slower code that relies on complex instructions or frequent memory accesses.
- Hardware designers want to build faster systems.
  - You can add features that are likely to be used in new software and to improve performance, like SSE2 instructions in the Pentium 4.
  - You could also find ways to speed up existing systems and software, perhaps by increasing the cache size or bus speed.
- Fast software and fast hardware go hand in hand.

Execution time

- The most intuitive measure of performance is just execution time, or how long you have to wait for a program to finish running.
- Our focus for the next several weeks will be on CPU processing time, but there are other factors in determining execution time too.
  - Memory and cache accesses
  - Input and output from disks, video cards, networks, etc.
  - Other processes in a multitasking system
- We'll look at some of these later in the course.



The components of execution time

- Execution time can be divided into two parts.
  - User time is spent running the application program itself.
  - System time is when the application calls operating system code.
- The distinction between user and system time is not always clear, especially under different operating systems.
- The Unix time command shows both.

```
salary.125 > time distill 05-examples.ps
Distilling 05-examples.ps (449,119 bytes)
10.8 seconds (0:11)
 449,119 bytes PS  =>  94,999 bytes PDF  (21%)
10.61u 0.98s 0:15.15 76.5%
```

User time

System time

Total time (including other processes)

CPU usage = (User + System) / Total

Throughput

- Another important measurement is throughput, or how many tasks can be performed in some amount of time.
- Throughput is especially important for servers.
  - How many web pages can be served per minute?
  - How many database transactions can be processed per second?
- Modern multitasking operating systems always trade some execution time in exchange for throughput—each individual program runs slower overall, but many programs can run together.
- Execution time and throughput are related.
  - Improving execution time also improves throughput.
  - It's possible to improve throughput but *not* execution time.

Execution time vs. throughput

- Let’s say you’re shopping at that new Wal-Mart by the airport. There is one register open, and the cashier takes two minutes per customer.
  - The “execution time” is 2 minutes.
  - The “throughput” would be 30 customers per hour.
- After some training, the cashier can process customers in 1.5 minutes.
  - The execution time is now improved to 1.5 minutes.
  - The throughput also improves, to 40 customers per hour.
- Or they could open three checkout lines, all with untrained personnel:
  - It still takes 2 minutes for each customer to check out.
  - But with three lines, there are 90 customers leaving per hour!

Measuring performance

- We'll measure performance according to execution time.
- A *lower* execution time is better, so we can define the performance of a computer system X on a program P as follows.

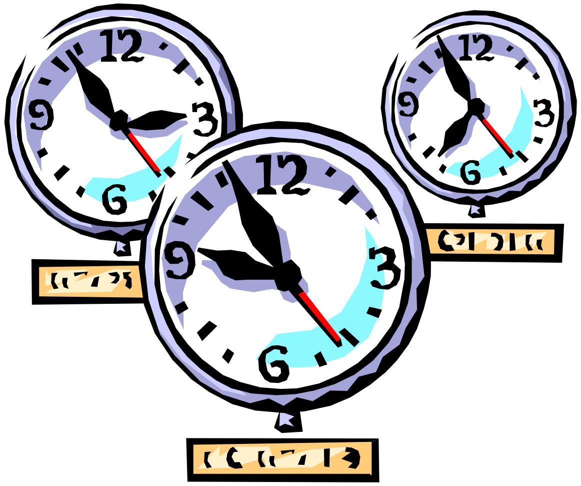
$$\text{Performance}_{X,P} = \frac{1}{\text{Execution time}_{X,P}}$$

- We can say that system X is *n* times faster than Y on program P if:

$$\frac{\text{Performance}_{X,P}}{\text{Performance}_{Y,P}} = \frac{\text{Execution time}_{Y,P}}{\text{Execution time}_{X,P}} = n$$

- This is equivalent to saying that Y is *n* times *slower* than X.

Clock cycle time



- There are three equally important components of execution time.
- One "cycle" is the minimum time it takes the CPU to do any work.
  - The clock cycle time or clock period is just the length of a cycle.
  - The clock rate, or frequency, is the reciprocal of the cycle time.
- Of course, the lower the cycle time and the higher the clock rate, the faster a given architecture can run.
- Some examples illustrate some typical frequencies.
  - A 500MHz processor has a cycle time of 2ns.
  - A 2GHz (2000MHz) CPU has a cycle time of just 0.5ns.

CPI

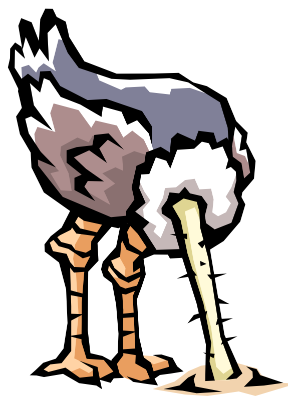
- Another important component is the average number of clock cycles per instruction, or CPI, for a particular machine and program.
  - The CPI depends on the actual instructions appearing in the program—a floating-point intensive application might have a higher CPI than an integer-based program.
  - It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster.
- In CS231 we assumed each instruction took one cycle, so we had CPI = 1.
  - The CPI is often higher in reality because of memory or I/O accesses, or more complex instructions.
  - The CPI can also be *lower* than 1, if you consider multiprocessors or superscalar architectures that execute many instructions at once.

Instructions executed

- Finally, we need to consider the number of instructions in a program.
  - We are not interested in the static instruction count, or how many lines of code are in a program.
  - Instead we care about the dynamic instruction count, or how many instructions are actually executed when the program runs.
- For example, there are three lines of code below, but the number of instructions executed would be 2001.

```
          li    $a0, 1000
Ostrich:  sub   $a0, $a0, 1
          bne   $a0, $0, Ostrich
```

- Programs that execute more instructions may take more time.



Computing the execution time

- Now we can express the CPU time more precisely.

$$\text{CPU time}_{X,P} = \text{Instructions executed}_P \times \text{CPI}_{X,P} \times \text{Clock cycle time}_X$$

- Make sure you have the units straight!

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- We can use this formula to determine how various changes to a program or machine will affect performance.

Programs, hardware and compilers

$$\text{CPU time}_{X,P} = \text{Instructions executed}_P \times \text{CPI}_{X,P} \times \text{Clock cycle time}_X$$

- Execution time and performance depend on *both* the particular system *and* the particular program being executed.
- How does the machine X affect performance?
  - Its implementation of the instruction set helps to determine CPI.
  - The processor's frequency will determine the clock cycle time.
- How does the program P affect performance?
  - It determines the number of instructions executed.
  - The *types* of those instructions influence the CPI.
- A good compiler is critical for optimizing program code!

Comparing ISA-compatible processors

- Let's compare the performances two 8086-based processors.
  - An 800MHz AMD Duron, with a CPI of 1.2 for an MP3 compressor.
  - A 1GHz Pentium III with a CPI of 1.5 for the same program.
- Compatible processors implement identical instruction sets and will use the same executable files, with the same number of instructions.
- But they implement the ISA differently, which leads to different CPIs.

$$\begin{aligned}\text{CPU time}_{AMD,P} &= \text{Instructions}_P \times \text{CPI}_{AMD,P} \times \text{Cycle time}_{AMD} \\ &= \text{Instructions}_P \times 1.2 \times 1.25\text{ns} \\ &= 1.5 \times \text{Instructions}_P \text{ ns}\end{aligned}$$

$$\begin{aligned}\text{CPU time}_{P3,P} &= \text{Instructions}_P \times \text{CPI}_{P3,P} \times \text{Cycle time}_{P3} \\ &= \text{Instructions}_P \times 1.5 \times 1\text{ns} \\ &= 1.5 \times \text{Instructions}_P \text{ ns}\end{aligned}$$

- So the execution time and performance are the same!

Comparing clock rates

- How about comparing a 2.5 GHz and a 3.0 GHz Pentium 4?
  - Selling the same processor at different clock rates is common.
  - Both processors will run the same code and have the same CPI.

$$\begin{aligned}\text{CPU time}_{2.5,P} &= \text{Instructions}_P \times \text{CPI}_{2.5,P} \times \text{Cycle time}_{2.5} \\ &= \text{Instructions}_P \times \text{CPI}_{2.5,P} \times 0.40\text{ns}\end{aligned}$$

$$\begin{aligned}\text{CPU time}_{3.0,P} &= \text{Instructions}_P \times \text{CPI}_{2.5,P} \times \text{Cycle time}_{3.0} \\ &= \text{Instructions}_P \times \text{CPI}_{2.5,P} \times 0.33\text{ns}\end{aligned}$$

$$\frac{\text{Performance}_{3.0,P}}{\text{Performance}_{2.5,P}} = \frac{\text{CPU time}_{2.5,P}}{\text{CPU time}_{3.0,P}} = \frac{0.40}{0.33} = 1.2$$

- As you might expect, the 3.0 GHz chip is 20% faster than the 2.5 GHz.
- Remember this is *only* CPU time, and it ignores all other factors!

Comparing compilers

- Let's compare two different compilers for the same system.
- The better compiler generates programs that need 5% fewer instructions and have a 10% lower CPI than the base compiler.

$$\text{CPU time}_{X,Base} = \text{Instructions}_P \times \text{CPI}_{X,Base} \times \text{Cycle time}_X$$

$$\text{CPU time}_{X,Opt} = (0.95\ \text{Instructions}_P) \times (0.90\ \text{CPI}_{X,Base}) \times \text{Cycle time}_X$$
$$= 0.86 \times \text{CPU time}_{Base,P}$$

$$\frac{\text{Performance}_{X,Opt}}{\text{Performance}_{X,Base}} = \frac{\text{CPU time}_{X,Base}}{\text{CPU time}_{X,Opt}} = \frac{1.00}{0.86} = 1.17$$

- So there are many ways to speed up a system.

Amdahl's Law

- Amdahl's Law states that optimizations are limited in their effectiveness.

$$\text{Execution time after improvement} = \frac{\text{Time affected by improvement}}{\text{Amount of improvement}} + \text{Time unaffected by improvement}$$

- For example, doubling the speed of floating-point operations sounds like a great idea. But if only 10% of the program execution time T involves floating-point code, then the overall performance improves by just 5%.

$$\text{Execution time after improvement} = \frac{0.10\ T}{2} + 0.90\ T = 0.95\ T$$

- A corollary of this law is that we should always try to *make the common case fast*—enhance the parts of the program that are used most often, so "time affected by improvement" is as large as possible.

Benchmarking

- What programs should we use to measure real-world performance?
  - Ideally we'd test each computer with our favorite programs.
  - But there are too many computers and programs out there!
- Instead people often rely on a few benchmark programs in an attempt to characterize the performance of systems.
- A good benchmark should reflect the performance of other applications too—in particular, it should have a realistic mix of instructions.
- Some common benchmarks include:
  - Adobe Photoshop for image processing
  - BAPCo SYSmark for office applications
  - Unreal Tournament 2003 for 3D games

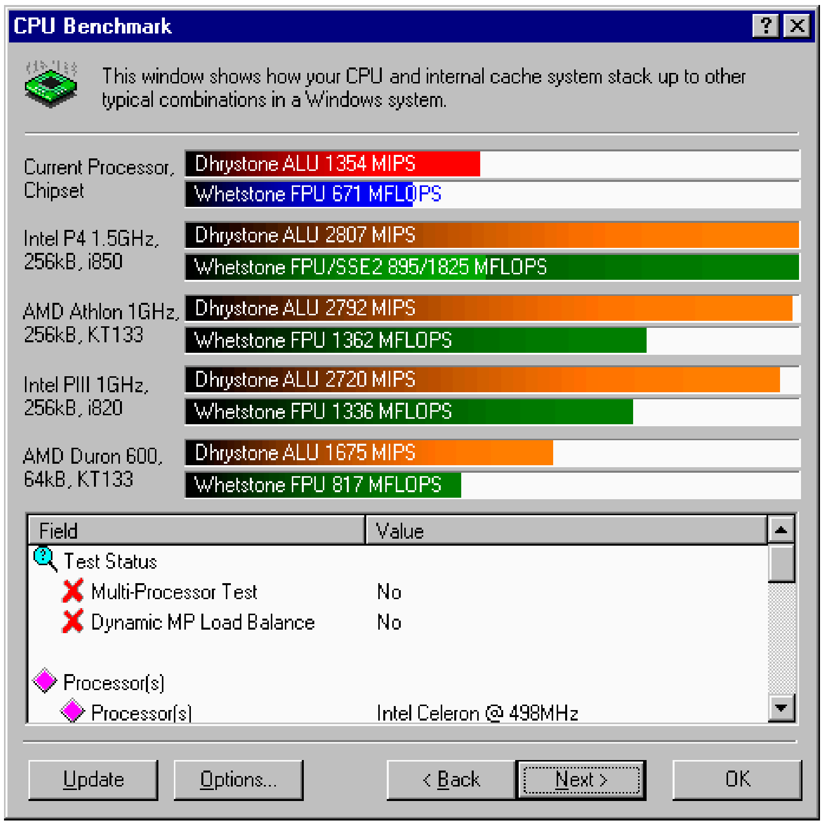
Synthetic benchmarks

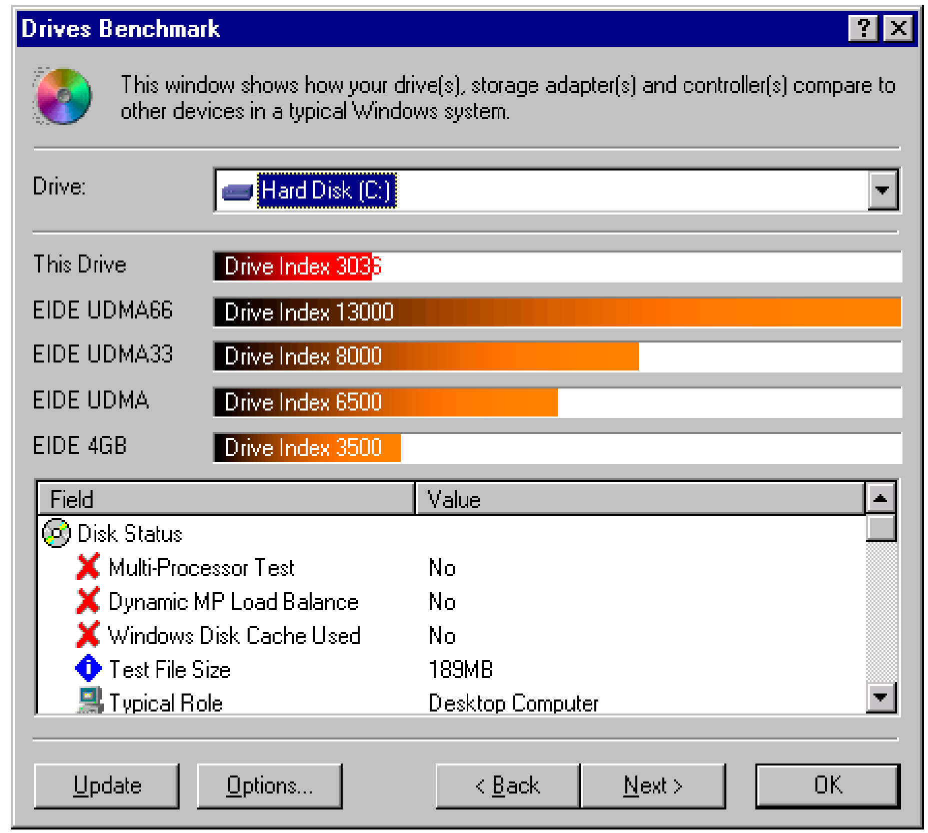
- A synthetic benchmark is a program whose only purpose is to measure performance.
- They are usually small and easy to port to different CPUs and operating systems, so they are convenient for comparing systems.
- However, there are many disadvantages too.
  - They're small enough that it's easy for compiler writers and CPU designers to cheat and make improvements that apply only to that particular benchmark.
  - Synthetic benchmarks may not contain a realistic instruction mix and may not reflect performance of typical applications.
- Many synthetic benchmarks are in use today.
  - SiSoft Sandra measures general system performance.
  - Futuremark 3DMark03 tests Windows 3D game performance (it does include code from actual games).

Some actual benchmarks

- SiSoft Sandra tests several system components.

http://www.sisoftware.co.uk



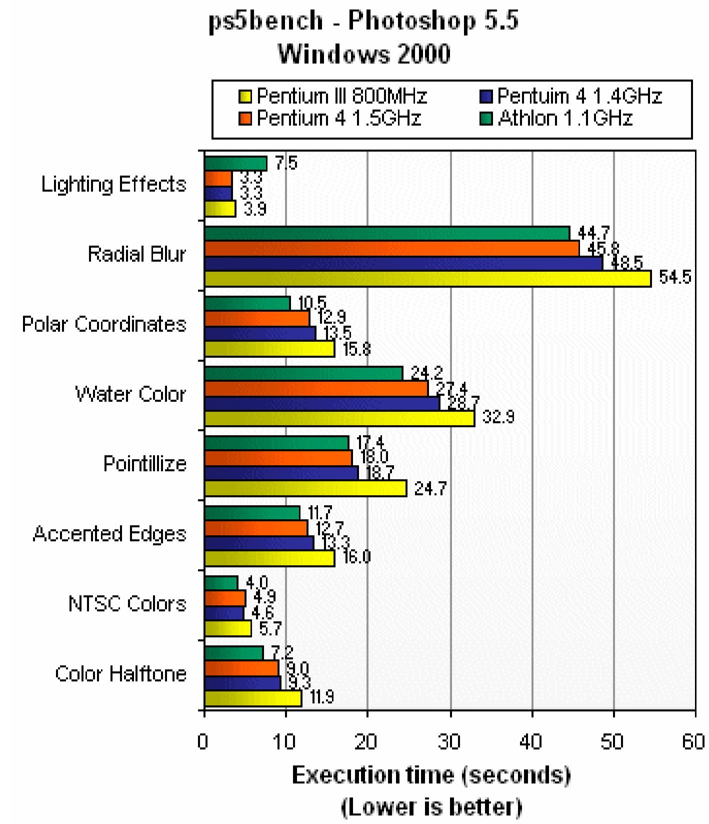


- It produces pretty bar graphs.

Performance of many programs

- The best way to see how a system performs for a variety of programs is to just show the execution times of all of the programs.
- Here are execution times for several different Photoshop 5.5 tasks, from

  http://www.tech-report.com

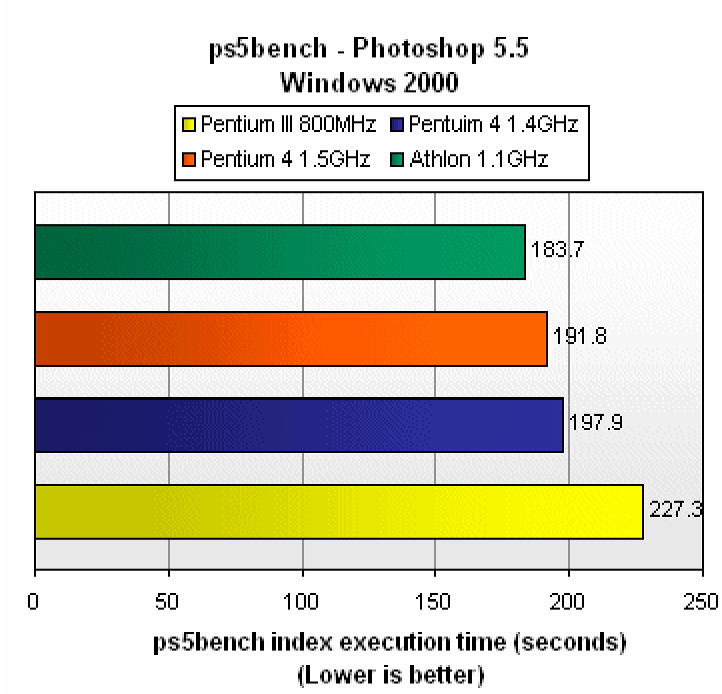


Summarizing performance

- Summarizing performance with a single number can be misleading—just like summarizing four years of school with a single GPA!
- If you must have a single number, you could sum the execution times.

  This example graph displays the total execution time of the individual tests from the previous page.
- A similar option is to find the average of all the execution times.

  For example, the 800MHz Pentium III (in yellow) needed 227.3 seconds to run 21 programs, so its average execution time is 227.3/21 = 10.82 seconds.



- A weighted sum or average is also possible, and lets you emphasize some benchmarks more than others.

Summary

- Performance is one of the most important criteria in judging systems.
- There are two main measurements of performance.
  - Execution time is what we'll focus on.
  - Throughput is important for servers and operating systems.
- Our main performance equation explains how performance depends on several factors related to both hardware and software.

$$\text{CPU time}_{X,P} = \text{Instructions executed}_{P} \times \text{CPI}_{X,P} \times \text{Clock cycle time}_{X}$$

- It can be hard to measure these factors in real life, but this is a useful guide for comparing systems and designs.
- Amdahl's Law tell us how much improvement we can expect from specific enhancements.
- The best benchmarks are real programs, which are more likely to reflect common instruction mixes.