

Chapter 4

Accessing and Understanding Performance

Outline

- Defining performance (4.2)
- CPU performance and its factors (4.2)
- Evaluating performance (4.3)
- About benchmark (4.3)

Why Study Performance?

- Conflicting Goals
 - User
 - Find the most suitable machine to get the job done at the lowest cost
⇒ **Application-oriented metrics**
 - Vendor
 - Persuade you to buy their machine regardless of your needs
⇒ **Hardware-oriented metrics**
- Know the vocabulary and understand the issues, so that:
 - As a user/buyer, you can make better purchasing decisions
 - As an engineer, you can make better hardware/software design decision

Performance for a CPU Designer

- An attempt to quantify how well a particular computer can perform a user's applications
- **Problems:**
 - Essentially a **software+hardware** issue
 - Different machines have different strengths and weaknesses
 - There is an enormous amount of hype and outright deception in the market – **be wary**
- Key to understanding underlying organizational motivation
 - *Why is some hardware better than others for different programs?*
 - *What factors of system performance are hardware related? (e.g., Do we need a new machine, or a new operating system?)*
 - *How does the machine's instruction set affect performance?*

Performance

- Why do we care about performance evaluation?
 - Purchasing perspective
 - given a collection of machines, which has the
 - best performance ?
 - least cost ?
 - best performance / cost ?
 - Design perspective
 - faced with design options, which has the
 - best performance improvement ?
 - least cost ?
 - best performance / cost ?
- How to measure, report, and summarize performance?
 - Performance metric
 - Benchmark

Which of these airplanes has the best performance?

Airplane	Passenger Capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers x m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

- **What metric defines performance?**
 - Capacity, cruising range, or speed?
- **Speed**
 - Taking one passenger from one point to another in the least time
 - Transporting 450 passengers from one point to another

Two Notions of “Performance”

- Response Time (latency)
 - How long does it take for my job to run?
 - How long does it take to execute a job?
 - How long must I wait for the database query?
 - **Time to do the task**
- Throughput
 - How many jobs can the machine run at once?
 - What is the average execution rate?
 - How much work is getting done?
 - **Total amount of work done in a give time**
- *If we upgrade a machine with a new processor what do we increase?*
- *If we add a new machine to the lab what do we increase?*

Execution Time




- Elapsed Time
 - counts everything (*disk and memory accesses, I/O , etc.*)
 - a useful number, but often not good for comparison purposes
- CPU time
 - doesn't count I/O or time spent running other programs
 - can be broken up into system time, and user time
- Our focus: user CPU time
 - time spent executing the lines of code that are "in" our program

Execution Time

- **Execution time on a computer is typically divided into:**
 - **User time:** Time spent executing instructions in the user code
 - **System time:** Time spent executing instructions in the kernel on behalf of the user code (e.g., opening files)
 - **Other:** Time when the system is idle or executing other programs
- Use “**time**” and “**top**” commands in Unix to see these

```
90.7u 12.9s 2:39 65%  
(90.7 + 12.9)/(2:39) = 65%. 1/3 of the elapsed time for I/O  
waiting, running other programs
```



 **User time**  **Sys. time**  **Other / idle**

Performance Expressed as Time

- **Time** is the **measure** of computer performance and the **only reliable one**

- Performance $\square \text{performance}(x) = \frac{1}{\text{execution_time}(x)}$

- Bigger is better

- Improve performance = decrease execution time

- "X is n times faster than Y" means

$$n = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{Execution_Time}(Y)}{\text{Execution_Time}(X)}$$

Time Measurement

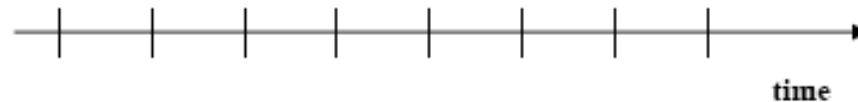
- But what does the “time” mean?
 - **Absolute** time measures
 - Difference between start and finish of an operation
 - Synonyms: running time, elapsed time, completion time, **execution time**, response time, **latency**
 - **1. Everything:** response time => **system performance**
 - Includes disk access, memory access, I/O, OS, CPU time
 - **2. CPU only:** CPU execution time or CPU time => **CPU performance**
 - the time CPU spends for this task
 - User CPU time and system CPU time
 - **Relative** (normalized) time measures
 - Running time normalized to some reference time
 - 3. In terms of **clock cycles** for computer designer

Clock Cycles

- Instead of reporting execution time in seconds, we often use cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- Clock “ticks” indicate when to start activities (one abstraction):



- cycle time = time between ticks = seconds per cycle
- clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

A 200 MHz. clock has a

$$\frac{1}{200 \times 10^6} \times 10^9 = 5 \text{ nanoseconds} \quad \text{cycle time}$$

Outline

- Defining performance (4.2)
- CPU performance and its factors (4.2)
- Evaluating performance (4.3)
- About benchmark (4.3)

CPU Time and its Factors

CPU Time = clock cycles for a program \times clock cycle time

$$= \frac{\text{clock cycles for a program}}{\text{clock rate}}$$

$$= \frac{\text{Instruction count} \times \text{Cycles per instruction}}{\text{clock rate}}$$

$$= \frac{\boxed{\text{Instruction count} \times \text{CPI}}}{\text{clock rate}}$$

$$\text{CPU clock cycles} = \sum_i \text{CPI}_i \times C_i,$$

C_i : the count the number of instructions of class i executed

CPI

- The average number of clock cycles each instruction takes to executed
- One way to **comparing two different implementations** of the same instruction set
- Overall CPI for a program
 - **Number of cycles** for each instruction type
 - **Frequency** of each instruction type in the program execution

Performance Equation

CPU time	=	$\frac{\text{Seconds}}{\text{Program}}$	=	$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$
----------	---	---	---	---

- Performance is determined by execution time
- Do any of the other variables equal performance?
 - # of cycles to execute program?
 - # of instructions in program?
 - # of cycles per second?
 - average # of cycles per instruction?
 - average # of instructions per second?
 - MIPS (million instructions per second)
 - When is it fair to compare two processors using MIPS?

How to determine the three factors

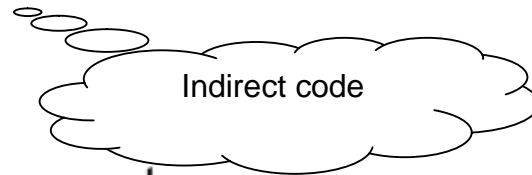
- Instruction count
 - Using software tools by **profiling**, or
 - Simulator of the architecture, or
 - Hardware counters (accuracy varies)
 - You can **measure** it without knowing the CPU implementation
- CPI
 - Depends on design details in the computer
 - By detailed simulation or hardware counter
 - **CPI should be measured**
 - You cannot get it from the “Manuals”
- Clock cycle
 - From the “manuals”

How to improve the performance

- Reduce **Instruction count** to execute
- Increase the number of instruction per cycle (reduce **CPI**)
 - Concurrent execution of instructions
- Increase **clock rate**

How Hardware and Software Affect Performance ?

- Algorithm
 - Instruction count
 - Possibly CPI
 - E.g. floating point algorithms
- Programming language
 - Instruction count
 - CPI
 - E.g. data abstraction in Java
- Compiler
 - Instruction count
 - CPI
- Instruction set architecture
 - Instruction count
 - CPI
 - Clock rate
- Other underlying architecture (from Chapter 5)
 - Pipeline
 - CPI
 - Clock rate
 - Memory system
 - CPI
 - Clock rate



Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Algorithm	X	X	
Programming Language	X	X	
Compiler	X	X	
ISA (instruction set architecture)	X	X	X

Short Summary

- Performance is determined by **execution time**
- Do any of the other variables equal performance?
 - # of cycles to execute program?
 - # of instructions in program?
 - # of cycles per second?
 - average # of cycles per instruction (CPI)?
 - average # of instructions per second (IPC)?
- Common pitfall: thinking one of the variables is indicative of performance when it really isn't.
- Remember: **Time** is the only reliable measurement for performance

Outline

- Defining performance (4.2)
- CPU performance and its factors (4.2)
- Evaluating performance (4.3)
- About benchmark (4.3)

Evaluating Performance

- Which program shall be used to evaluate performance
 - Best one: **real workload** in your daily life
 - It is not easy for everyone
 - Alternative: **benchmark**
 - To predict the performance of the real workload

Benchmarks

- Performance best determined by running a **real application**
 - Use programs typical of expected workload
 - Or, typical of expected class of applications
 - compilers/editors, scientific applications, graphics, etc. GCC, tex, spice, Excel,
- Small benchmarks
 - took small fragments of code from inside application loops
 - **nice for architects and designers**, easy to standardize but it can be abused
 - best for **isolating** performance of individual features of the machine
 - nice for architects and designers
 - easy to standardize
 - can be abused
 - Livermore Loops, LINPACK
 - Toy benchmarks: 10 ~ 100 lines
 - Sieve of Erastosthenes, Puzzle, Quicksort, N-Queen
- **Synthesis benchmarks:**
 - Try to match average frequency of a large set of programs
 - Exercise the hardware in a manner to **mimic** real-world applications, but in a small piece of code.
 - Examples: Whetstone, Dhrystone –
 - Performs a varied mix of instructions and uses the memory in various ways;
 - **How many “Whetstones” or “Dhrystones” per second** your computer can do.

More Benchmarks

- Drystone[Weicker84]
- Whestone[Currow & Wichmann76]
 - University computer center jobs
 - 12 loops
- SPEC Benchmarks
 - SPEC (System Performance Evaluation Cooperative)
 - companies have agreed on a set of real program and inputs
 - valuable indicator of performance (and compiler technology)
 - can still be abused
 - SPEC 89
 - SPEC 92
 - SDPEC 95
 - SPEC2000
 - SPEC2004

Application-oriented Benchmarks

- CPU performance
 - SPEC, for scientific applications
- Server performance
 - Focus on throughput, response time to individual events
 - SPECweb99
- Graphics performance
 - 3D Mark
- Embedded computing
 - EEMBC
 - Automatic, consumer, networking, office automation, telecommunication
- Other research oriented
 - MediaBench
 - CommBench

SPEC CPU 2000

Integer benchmarks		FP benchmarks	
Name	Description	Name	Type
gzip	Compression	wupwise	Quantum chromodynamics
vpr	FPGA circuit placement and routing	swim	Shallow water model
gcc	The Gnu C compiler	mgrid	Multigrid solver in 3-D potential field
mcf	Combinatorial optimization	applu	Parabolic/elliptic partial differential equation
crafty	Chess program	mesa	Three-dimensional graphics library
parser	Word processing program	galgel	Computational fluid dynamics
eon	Computer visualization	art	Image recognition using neural networks
perlbmk	perl application	equake	Seismic wave propagation simulation
gap	Group theory, Interpreter	facerec	Image recognition of faces
vortex	Object-oriented database	ammp	Computational chemistry
bzip2	Compression	lucas	Primality testing
twolf	Place and rote simulator	fma3d	Crash simulation using finite-element method
		sixtrack	High-energy nuclear physics accelerator design
		apsi	Meteorology: pollutant distribution

FIGURE 4.5 The SPEC CPU2000 benchmarks. The 12 integer benchmarks in the left half of the table are written in C and C++, while the floating-point benchmarks in the right half are written in Fortran (77 or 90) and C. For more information on SPEC and on the SPEC benchmarks, see www.spec.org. The SPEC CPU benchmarks use wall clock time as the metric, but because there is little I/O, they measure CPU performance.

SPEC CINT2000

SPEC CINT2000 Benchmark Kernels				
Benchmark	Reference Time	Language	Application Class	General Description
164.gzip	1400	C	Compression	Compresses a TIFF (Tagged Image Format File), a Web server log, binary program code, "random" data, and a tar file source.
175.vpr	1400	C	Field Programmable Gate Array Circuit Placement and Routing	Maps FPGA circuit logic blocks and their required connections using a combinatorial optimization program. Such programs are found in integrated circuit CAD programs.
176.gcc	1100	C	C Programming Language Compiler	Compiles Motorola 88100 machine code from five different input source files using gcc.
181.mcf	1800	C	Combinatorial Optimization	Solves a single-depot vehicle scheduling problem of the type often found in the public transportation planning field.
186.crafty	1000	C	Chess	Solves five different chessboard input layouts to varying search tree "depths" for possible next moves.
197.parser	1800	C	Word Processing	Parses input sentences to find English syntax using a 60,000-word dictionary.
252.eon	1300	C++	Computer Visualization	Finds the intersection of three-dimensional rays using probabilistic ray tracing.
253.perlbmk	1800	C	PERL Programming Language	Processes five Perl scripts to create mail, HTML, and other output.
254.gap	1100	C	Group Theory Interpreter	Interprets a group theory language that was written to process combinatorial problems.
255.vortex	1900	C	Object-Oriented Database	Manipulates data from three object-oriented databases.
256.bzip2	1500	C	Compression	Compresses a TIFF, a binary program, and a tar source file.
300.twolf	3000	C	Place and Route Simulator	Approximates a solution to the problem of finding an optimal transistor layout on a microchip.

SPEC CFP2000

SPEC CFP2000 Benchmark Kernels				
Benchmark	Reference Time	Language	Application Class	General Description
168.wupwise	1600	FORTRAN 77	Quantum Chromodynamics	Simulates quark interactions as needed by physicists studying quantum chromodynamics.
171.swim	3100	FORTRAN 77	Shallow Water Modeling	Predicts weather using mathematical modeling techniques. Swim is often used as a benchmark of supercomputer performance.
172.mgrid	1800	FORTRAN 77	3D Potential Field Solver	Computes the solution of a three-dimensional scalar Poisson equation. This kernel benchmark comes from NASA.
173.applu	2100	FORTRAN 77	Parabolic-Elliptic Partial Differential Equations	Solves five nonlinear partial differential equations using sparse Jacobian matrices.
177.mesa	1400	C	3-D Graphics Library	Converts a two-dimensional graphics input to a three-dimensional graphics output.
178.galgel	2900	FORTRAN 90	Computational Fluid Dynamics	Determines the critical value of temperature differences in the walls of a fluid tank that cause convective flow to change to oscillatory flow.
179.art	2600	C	Image Recognition	Locates images of a helicopter and an airplane within an image. The algorithm uses neural networks.
183.quake	1300	C	Seismic Wave Propagation Simulation	Uses finite element analysis to recover the history of ground motion ensuing from a seismic event.
187.facerec	1900	FORTRAN 90	Face Recognition	Uses the "Elastic Graph Matching" method to recognize faces represented by labeled graphs.
188.amp	2200	C	Computational Chemistry	Solves a molecular dynamics problem by calculating the motions of molecules within a system.
189.lucas	2000	FORTRAN 90	Primality Testing	Begins the process of determining the primality of a large Mersenne number ($2^p - 1$). The result is not found; the intermediate results are measured instead.
191.fma3d	2100	FORTRAN 90	Finite-Element Crash Simulation	Simulates the effects of the collision of inelastic three-dimensional solids.
200.sixtrack	1100	FORTRAN 77	High Energy Nuclear Physics Accelerator Design	Simulates tracking particle behavior through a particle accelerator.
301.apsi	2600	FORTRAN 77	Pollutant Distribution	Finds the velocity of pollutant particles from a given source using parameters of initial velocity, wind speed, and temperature.

Arithmetic Mean vs. Geometric Mean

- Problem
 - How you combine the normalized results or **Can you ?**
- When arithmetic mean applied to the **normalized execution time**
 - A is 5.05 times faster than B
 - B is 5.05 times faster than A
 - This is used in SPEC ratio
 - Result is strongly affected by the **choosing reference machine**
- Geometric means produces the same “relative” results whether we normalize to A or B
 - Pros: independent of the running time
 - Cons: Geometric mean **does not track total execution time** and thus can't be used to predict relative execution time for a workload
- So what's the **solution** to summary the performance
 - **Measure the workload and weighted by their frequency of execution**


Amdahl's Law

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

$$\begin{aligned} \text{Speedup} &= \frac{\text{Performance after improvement}}{\text{Performance before improvement}} \\ &= \frac{\text{Execution time before improvement}}{\text{Execution time after improvement}} \\ &= \frac{\text{Execution time before improvement}}{\frac{\text{Execution time affected}}{\text{Amount of improvement}} + \text{Execution time unaffected}} \end{aligned}$$

Amdahl's Law

- Speedup due to enhancement E:

$$\text{Speedup (E)} = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}$$


- Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected, then:

$$\text{ExTime (E)} = ((1-F) + F/S) \times \text{ExTime (without E)}$$

$$\text{Speedup (E)} = \frac{1}{(1-F) + F/S}$$

Amdahl's Law

- Floating point instructions improved to run 2X; but only 10% of actual instructions are FP

$$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 + .1/2) = 0.95 \times \text{ExTime}_{\text{old}}$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.95} = 1.053$$

Example #3

- Our favorite program runs in 10 seconds on computer A, which has a 400 Mhz. clock. We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program. What clock rate should we tell the designer to target?"

$$\frac{\text{Execution_Time (A)}}{\text{Execution_Time (B)}} = \frac{10}{6} = \frac{C \times \frac{1}{400 \times 10^6}}{1.2C \times \frac{1}{x}}$$

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

Speedup

Assume

execution time unaffected = $f \times$ Execution Time before improvement

Amount of improvement = n

$$Speedup = \frac{\text{Execution time before improvement}}{\frac{\text{Execution time affected}}{\text{Amount of improvement}} + \text{Execution time unaffected}}$$

$$= \frac{1}{\frac{1-f}{n} + f}$$

$f = 0$, speedup = n
 $f \sim 1$, speedup = $1/f$, **less speedup**

– $N \Rightarrow$ infinity, speedup = $1/f$

- Opportunity of improvement depend on **how much** the time event occurs
- Principle: **Make the common case fast**
 - Frequency of one even may be much higher than another
 - Common case is often the simple case, thus easier to speedup

Summary

- Performance is specific to a particular program/s
 - **Total execution time** is a consistent summary of performance
- For a given architecture performance increases come from:
 - increases in clock rate (without adverse CPI affects)
 - improvements in processor organization that lower CPI
 - compiler enhancements that lower CPI and/or instruction count
 - Algorithm/Language choices that affect instruction count
- Amdahl's law