华章教育

全美经典
学习指导系列

计算机体系结构习题与解答

(英文版)

COMPUTER ARCHITECTURE

Complete coverage of hardware and software design for computer systems

192 solved problems

Real-world design choices explained step-by-step

Perfect for professionals refreshing core concepts

(美) Nicholas Carter 著

机械工业出版社
China Machine Press

全球销售超过3000万册!

全美经典
学习指导系列

计算机体系结构习题与解答

（英文版）

COMPUTER ARCHITECTURE

（美）Nicholas Carter 著

机械工业出版社
China Machine Press

Nicholas Carter: Computer Architecture (ISBN 0-07-136207-x).

本书版权登记号：图字：01-2002-2182

图书在版编目（CIP）数据

计算机体系结构习题与解答/（美）卡特（Carter, N.）著. -北京：机械工业出版社，2002.8

（全美经典学习指导系列）

书名原文：Computer Architecture

ISBN 7-111-10418-8

Ⅰ.计… Ⅱ.卡… Ⅲ.计算机体系结构-习题-英文 Ⅳ.TP303-44

中国版本图书馆CIP数据核字（2002）第038500号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：华 章

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2002年8月第1版第1次印刷

787mm × 1092mm 1/16 · 20印张

印数：0 001- 3000册

定价：30.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

PREFACE

One of the most interesting aspects of computer architecture is the rate at which the field changes. Innovation occurs on an almost-daily basis, offering opportunities for individuals to contribute to the field. However, this rate of progress is one of the greatest challenges to teaching computer architecture and organization. Unlike many other fields, courses in computer architecture and organization must change on a term-by-term basis to incorporate new developments in the field without overloading students with material. Writing textbooks for the field is similarly difficult, as the author must find a balance between cutting-edge material and historical perspective.

This book includes a selection of topics intended to make it useful to readers with a wide range of previous exposure to the field. Chapters 1 through 5 cover many of the basic concepts in computer organization, including how performance is measured, how computers represent numerical data and programs, different programming models for computers, and the basics of processor design. Chapters 6 and 7 cover pipelining and instruction-level parallelism, two technologies that are extremely important to the performance of modern processors. Chapters 8, 9, and 10 cover memory system design, including memory hierarchies, caches, and virtual memory. Chapter 11 describes I/O systems, while Chapter 12 provides an introduction to multiprocessor systems—computers that combine multiple processors to deliver improved performance.

It is my hope that readers will find this book useful in their study of the field. I have tried to make my explanations of each topic as clear as possible and to avoid getting bogged down in detail. Compressing the field of computer architecture and organization into a book this size was a challenge, and I look forward to any comments that readers may have about the selection of material, the exercises, or anything else related to this work.

In conclusion, I would like to thank all those who have made this effort possible: my parents, my friends, my colleagues at the University of Illinois, and all of the teachers who contributed to my own education. In addition, I would like to thank the staff at McGraw-Hill for encouraging this work and for their tolerance of schedule delays.

NICHOLAS P. CARTER

CONTENTS

CHAPTER 1

Introduction

1.1 Purpose of This Book

This book is intended for use as a companion text for advanced undergraduate-level or introductory graduate-level courses in computer architecture. Its primary intended audience is students and faculty involved in computer architecture courses who are interested in additional explanations, practice problems, and examples to use in increasing their understanding of the material or in preparing assignments.

1.2 Background Assumed

This book assumes that the reader has a background similar to that of college sophomores or juniors in electrical engineering or computer science programs who have not yet had a course on computer organization or computer architecture. Basic familiarity with computer operation and terminology is assumed, as is some familiarity with programming in high-level languages.

1.3 Material Covered

This book covers a slightly wider range of topics than most one-term computer architecture courses in order to increase its utility. Readers may find the additional material useful as review or as an introduction to more advanced topics. The book begins with a discussion of data representation and computer arithmetic, followed by chapters on computer organization and programming models. Chapter 5 begins a three-chapter discussion of processor design, including pipelining and instruction-level parallelism. This is followed by three chapters on memory systems, including

coverage of virtual memory and caches. The final two chapters discuss I/O and provide an introduction to multiprocessors.

1.4 Chapter Objectives

The goal of this chapter is to prepare the reader for the material in later chapters by discussing the basic technologies that drive computer performance and the techniques used to measure and discuss performance. After reading this chapter and completing the exercises, a student should

1. Understand and be able to discuss the historical rates of improvement in transistor density, circuit performance, and overall system performance
2. Understand common methods of evaluating computer performance
3. Be able to calculate how changes to one part of a computer system will affect overall performance

1.5 Technological Trends

Since the early 1980s, computer performance has been driven by improvements in the capabilities of the integrated circuits used to implement microprocessors, memory chips, and other computer components. Over time, integrated circuits improve in *density* (how many transistors and wires can be placed in a fixed area on a silicon chip), *speed* (how quickly basic logic gates and memory devices operate), and *area* (the physical size of the largest integrated circuit that can be fabricated).

The tremendous growth in computer performance over the last two decades has been driven by the fact that chip speed and density improve *geometrically* rather than linearly, meaning that the increase in performance from one year to the next has been a relatively constant fraction of the previous year's performance, rather than a constant absolute value. On average, the number of transistors that can be fabricated on a silicon chip increases by about 50 percent per year, and transistor speed increases such that the delay of a basic logic gate (AND, OR, etc.) decreases by 13 percent per year. The observation that computer performance improves geometrically, not linearly, is often referred to as *Moore's Law.*

EXAMPLE

The amount of data that can be stored on a dynamic RAM (DRAM) memory chip has quadrupled every three years since the late 1970s, an annual growth rate of 60 percent.

From the late 1970s until the late 1980s, microprocessor performance was mainly driven by improvements in fabrication technology and improved at a rate of 35 percent per year. Since then, the rate of improvement has actually increased, to over 50 percent per year, although the rate of progress in semiconductor fabrication has

remained relatively constant. The increase in the rate of performance improvement has been due to improvements in computer architecture and organization—computer architects have been able to take advantage of the increasing density of integrated circuits to add features to microprocessors and memory systems that improve performance over and above the improvements in speed of the underlying transistors.

1.6 Measuring Performance

In this chapter, we have discussed how computer performance has improved over time, without giving a formal definition of what performance is. In part, this is because *performance* is a very vague term when used in the context of computer systems. Generally, performance describes how quickly a given system can execute a program or programs. Systems that execute programs in less time are said to have higher performance.

The best measure of computer performance is the execution time of the program or programs that the user wants to execute, but it is generally impractical to test all of the programs that will be run on a given system before deciding which computer to purchase or when making design decisions. Instead, computer architects have come up with a variety of metrics to describe computer performance, some of which will be discussed in this chapter. Architects have also devised a number of metrics for the performance of individual computer subsystems, which will be discussed in the chapters that cover those subsystems.

Keep in mind that many factors other than performance may influence design or purchase decisions. Ease of programming is an important consideration, because the time and expense required to develop needed programs may be more significant than the difference in execution times of the programs once they have been developed. Also important is the issue of compatibility; most programs are sold as binary images that will only run on a particular family of processors. If the program you need won't run on a given system, it doesn't matter how quickly the system executes other programs.

1.6.1 MIPS

An early measure of computer performance was the rate at which a given machine executed instructions. This is calculated by dividing the number of instructions executed in running a program by the time required to run the program and is typically expressed in *millions of instructions per second* (MIPS). MIPS has fallen out of use as a measure of performance, mainly because it does not account for the fact that different systems often require different numbers of instructions to implement a given program. A computer's MIPS rating does not tell you anything about how many instructions it requires to perform a given task, making it less useful than other metrics for comparing the performance of different systems.

1.6.2 CPI/IPC

Another metric used to describe computer performance is the number of clock cycles required to execute each instruction, known as *cycles per instruction*, or CPI. The CPI of a given program on a given system is calculated by dividing the number of clock cycles required to execute the program by the number of instructions executed in running the program. For systems that can execute more than one instruction per cycle, the number of *instructions executed per cycle*, or IPC, is often used instead of CPI. IPC is calculated by dividing the number of instructions executed in running a program by the number of clock cycles required to execute the program, and is the reciprocal of CPI. These two metrics give the same information, and the choice of which one to use is generally made based on which of the values is greater than the number 1. When using IPC and CPI to compare systems, it is important to remember that high IPC values indicate that the reference program took fewer cycles to execute than low IPC values, while high CPI values indicate that more cycles were required than low CPI values. Thus, a large IPC tends to indicate good performance, while a large CPI indicates poor performance.

EXAMPLE

A given program consists of a 100-instruction loop that is executed 42 times. If it takes 16,000 cycles to execute the program on a given system, what are that system's CPI and IPC values for the program?

Solution

The 100-instruction loop is executed 42 times, so the total number of instructions executed is $100 \times 42 = 4200$. It takes 16,000 cycles to execute the program, so the CPI is $16{,}000/4200 = 3.81$. To compute the IPC, we divide 4200 instructions by 16,000 cycles, getting an IPC of 0.26.

In general, IPC and CPI are even less useful measures of actual system performance than MIPS, because they do not contain any information about a system's clock rate or how many instructions the system requires to perform a task. If you know a system's MIPS rating on a given program, you can multiply it by the number of instructions executed in running the program to determine how long the program took to complete. If you know a system's CPI on a given program, you can multiply it by the number of instructions in the program to get the number of cycles it took to complete the program, but you have to know the number of cycles per second (the system's clock rate) to convert that into the amount of time required to execute the program.

As a result, CPI and IPC are rarely used to compare actual computer systems. However, they are very common metrics in computer architecture research, because most computer architecture research is done in simulation, using programs that simulate a particular architecture to estimate how many cycles a given program will take to execute on that architecture. These simulators are generally unable to predict

the cycle time of the systems that they simulate, so CPI/IPC is often the best available estimate of performance.

1.6.3 BENCHMARK SUITES

Both MIPS and CPI/IPC have significant limitations as measures of computer performance, as we have discussed. *Benchmark suites* are a third measure of computer performance and were developed to address the limitations of MIPS and CPI/IPC.

A benchmark suite consists of a set of programs that are believed to be typical of the programs that will be run on the system. A system's score on the benchmark suite is based on how long it takes the system to execute all of the programs in the suite. Many different benchmark suites exist that generate estimates of a system's performance on different types of applications.

One of the best-known benchmark suites is the SPEC suite, produced by the Standard Performance Evaluation Corporation. The current version of the SPEC suite as of the publication of this book is the SPEC CPU2000 benchmark, the third major revision since the first SPEC benchmark suite was published in 1989.

Benchmark suites provide a number of advantages over MIPS and CPI/IPC. First, their performance results are based on total execution times, not rate of instruction execution. Second, they average a system's performance across multiple programs to generate an estimate of its average speed. This makes a system's overall rating on a benchmark suite a better indicator of its overall performance than its MIPS rating on any one program. Also, many benchmarks require manufacturers to publish their systems' results on the individual programs within the benchmark, as well as the system's overall score on the benchmark suite, making it possible to do a direct comparison of individual benchmark results if you know that a system will be used for a particular application.

1.6.4 GEOMETRIC VERSUS ARITHMETIC MEAN

Many benchmark suites use the *geometric* rather than the *arithmetic* mean to average the results of the programs contained in the benchmark suite, because a single extreme value has less of an impact on the geometric mean of a series than on the arithmetic mean. Using the geometric mean makes it harder for a system to achieve a high score on the benchmark suite by achieving good performance on just one of the programs in the suite, making the system's overall score a better indicator of its performance on most programs.

The geometric mean of n values is calculated by multiplying the n values together and taking the nth root of the product. The arithmetic mean, or average, of a set of values is calculated by adding all of the values together and dividing by the number of values.

EXAMPLE

What are the arithmetic and geometric means of the values 4, 2, 4, 82?

Solution

The arithmetic mean of this series is

$$\frac{4+2+4+82}{4} = 23$$

The geometric mean is

$$\sqrt[4]{4 \times 2 \times 4 \times 82} = 7.16$$

Note that the inclusion of one extreme value in the series had a much greater effect on the arithmetic mean than on the geometric mean.

1.7 Speedup

Computer architects often use the term *speedup* to describe how the performance of an architecture changes as different improvements are made to the architecture. Speedup is simply the ratio of the execution times before and after a change is made, so:

$$\text{Speedup} = \frac{\text{Execution time}_{\text{before}}}{\text{Execution time}_{\text{after}}}$$

For example, if a program takes 25 seconds to run on one version of an architecture and 15 seconds to run on a new version, the overall speedup is 25 seconds/15 seconds = 1.67.

1.8 Amdahl's Law

The most important rule for designing high-performance computer systems is *make the common case fast.* Qualitatively, this means that the impact of a given performance improvement on overall performance is dependent on both how much the improvement improves performance when it is in use and how often the improvement is in use. Quantitatively, this rule has been expressed as *Amdahl's Law,* which states

$$\text{Execution Time}_{\text{new}} = \text{Execution Time}_{\text{old}} \times \left[\text{Frac}_{\text{unused}} + \frac{\text{Frac}_{\text{used}}}{\text{Speedup}_{\text{used}}}\right]$$

In this equation, $\text{Frac}_{\text{unused}}$ is the fraction of time (not instructions) that the improvement is not in use, $\text{Frac}_{\text{used}}$ is the fraction of time that the improvement is in use, and $\text{Speedup}_{\text{used}}$ is the speedup that occurs when the improvement is used (this would be the overall speedup if the improvement were in use at all times). Note that $\text{Frac}_{\text{used}}$ and $\text{Frac}_{\text{unused}}$ are computed using the execution time *before* the modifica-

tion is applied. Computing these values using the execution time after the modification is applied will give incorrect results.

Amdahl's Law can be rewritten using the definition of speedup to give

$$\text{Speedup} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}} = \frac{1}{\text{Frac}_{\text{unused}} + \frac{\text{Frac}_{\text{used}}}{\text{Speedup}_{\text{used}}}}$$

EXAMPLE

Suppose that a given architecture does not have hardware support for multiplication, so multiplications have to be done through repeated addition (this was the case on some early microprocessors). If it takes 200 cycles to perform a multiplication in software, and 4 cycles to perform a multiplication in hardware, what is the overall speedup from hardware support for multiplication if a program spends 10 percent of its time doing multiplications? What about a program that spends 40 percent of its time doing multiplications?

In both cases, the speedup when the multiplication hardware is used is $200/4 = 50$ (ratio of time to do a multiplication without the hardware to time with the hardware). In the case where the program spends 10 percent of its time doing multiplications, $\text{Frac}_{\text{unused}} = 0.9$, and $\text{Frac}_{\text{used}} = 0.1$. Plugging these values into Amdhal's Law, we get Speedup $= 1/[.9 + (.1/50)] = 1.11$. If the program spends 40 percent of its time doing multiplications before the addition of hardware multiplication, then $\text{Frac}_{\text{unused}} = 0.6$, $\text{Frac}_{\text{used}} = 0.4$, and we get Speedup $= 1/[.6 + (.4/50)] = 1.64$.

This example illustrates the impact that the fraction of time an improvement is used has on overall performance. As $\text{Speedup}_{\text{used}}$ goes to infinity, overall speedup converges to $1/\text{Frac}_{\text{unused}}$, because the improvement can't do anything about the execution time of the fraction of the program that does not use the improvement.

1.9 Summary

This chapter has been intended to provide a context for the rest of the book by explaining some of the technology forces that drive computer performance and providing a framework for discussing and evaluating system performance that will be used throughout the book.

The important concepts for the reader to understand after studying this chapter are as follows:

1. Computer technology is driven by improvements in semiconductor fabrication technology, and these improvements proceed at a geometric, rather than a linear, pace.
2. There are many ways to measure computer performance, and the most effective measures of overall performance are based on the performance of a system on a wide variety of applications.
3. It is important to understand how a given performance metric is generated

in order to understand how useful it is in predicting system performance on a given application.

4. The impact of a change to an architecture on overall performance is dependent not only on how much that change improves performance when it is used, but on how often the change is useful. A consequence of this is that the overall performance impact of an improvement is limited by the fraction of time that the improvement is not in use, regardless of how much speedup the improvement gives when it is useful.

Solved Problems

Technology Trends (I)

1.1. As an illustration of just how fast computer technology is improving, let's consider what would have happened if automobiles had improved equally quickly. Assume that an average car in 1977 had a top speed of 100 miles per hour (mi/h, an approximation) and an average fuel economy of 15 miles per gallon (mi/g). If both top speed and efficiency improved at 35 percent per year from 1977 to 1987, and by 50 percent per year from 1987 to 2000, tracking computer performance, what would the average top speed and fuel economy of a car be in 1987? In 2000?

Solution

In 1987:

The span 1977 to 1987 is 10 years, so both traits would have improved by a factor of $(1.35)^{10} = 20.1$, giving a top speed of 2010 mi/h and a fuel economy of 301.5 mi/g.

In 2000:

Thirteen more years elapse, this time at a 50 percent per year improvement rate, for a total factor of $(1.5)^{13} = 194.6$ over the 1987 values. This gives a top speed of 391,146 mi/h and a fuel economy of 58,672 mi/g. This is fast enough to cover the distance from the earth to the moon in under 40 min, and to make the round trip on less than 10 gal of gasoline.

Technology Trends (II)

1.2. Since 1987, computer performance has been increasing at about 50 percent per year, with improvements in fabrication technology accounting for about 35 percent per year and improvements in architecture accounting for about 15 percent per year.

1. If the performance of the best available computer on 1/01/1988 was defined to be 1, what would be the expected performance of the best available computer on 1/01/2001?
2. Suppose that there had been no improvements in computer architecture since 1987, making fabrication technology the only source of performance improvements. What would the expected performance of the best available computer on 1/01/2001 be?
3. Now suppose that there had been no improvements in fabrication technology, making improvements in architecture the only source of performance improvements. What would the expected performance of the fastest computer on 1/01/2001 be then?

Solution

1. Performance improves at 50 percent per year, and 1/01/1988 to 1/01/2001 is 13 years, so the expected performance of the 1/01/2001 machine is $1 \times (1.5)^{13} = 194.6$.
2. Here, performance only improves at 35 percent per year, so the expected performance is 49.5.
3. Performance improvement is 15 percent per year, giving an expected performance of 6.2.

Speedup (I)

1.3. If the 1998 version of a computer executes a program in 200 s and the version of the computer made in the year 2000 executes the same program in 150 s, what is the speedup that the manufacturer has achieved over the two-year period?

Solution

$$\text{Speedup} = \frac{\text{Execution time}_{\text{before}}}{\text{Execution time}_{\text{after}}}$$

Given this, the speedup is $200\,\text{s}/150\,\text{s} = 1.33$. Clearly, this manufacturer is falling well short of the industrywide performance growth rate.

Speedup (II)

1.4. To achieve a speedup of 3 on a program that originally took 78 s to execute, what must the execution time of the program be reduced to?

Solution

Here, we have values for speedup and Execution $\text{time}_{\text{before}}$. Substituting these into the formula for speedup and solving for Execution $\text{time}_{\text{after}}$ tells us that the execution time must be reduced to 26 s to achieve a speedup of 3.

Measuring Performance (I)

1.5.
1. Why are benchmark programs and benchmark suites used to measure computer performance?
2. Why are there multiple benchmarks that are used by computer architects, instead of one "best" benchmark?

Solution

1. Computer systems are often used to run a wide range of programs, some of which may not exist at the time the system is purchased or built. Thus, it is generally not possible to measure a system's performance on the set of programs that will be run on the machine. Instead, benchmark programs and suites are used to measure the performance of a system on one or