Homayoun

Reference 23

Third Edition

High-Performance Computer Architecture

Harold S. Stone
IBM T.J. Watson
Research Center
and
Courant Institute
New York University

Addison-Wesley Publishing Company

Reading, Massachusetts
Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England
Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan • Milan • Paris

This book is in the **Addison-Wesley Series in Electrical and Computer Engineering**

Library of Congress Cataloging-in-Publication Data

Stone, Harold S.
High-performance computer architecture / Harold S. Stone.—3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-52688-3
1. Computer architecture. I. Title.
QA76.9.A73S76 1993
004.2'2—dc20 92-32243
CIP

1 2 3 4 5 6 7 8 9 10-HA-95949392

To Jan—colleague and companion

Preface

Teaching computer architecture is an interesting challenge for the instructor because the field is in constant flux. What the architect does depends strongly on the devices available, and the devices have been changing every two to three years, with major breakthroughs once or twice a decade. Within the brief life of the first edition of this textbook, a whole generation of processor and memory chips were first offered for sale, appeared in popular computers, and then gradually disappeared from the marketplace as their successors took their places. The particular features and strengths of those devices have given way to other features in various new combinations and new relative costs. Design practices are evolving to exploit the new devices for a new generation of machines. And they will evolve again as the next wave of devices appears in the coming years.

What then should be taught to prepare students for what lies ahead? What information win remain important over the technical career of a student, and what information will soon become obsolete, of historical interest only? This text stresses design ideas embodied in many machines and the techniques for evaluating those ideas. The ideas and the evaluation techniques are the principles that will survive. The specific implementations of machines that one might choose in 1995, 2000, or 2005 reflect the basic principles described here as applied to the device technology currently prevailing. Effective designs are those that use technology cleverly and achieve balanced, efficient structures matched well to the class of problems they attack. This text stresses the means to achieve balance and efficiency in the context of any device technology.

We use a multifaceted approach to teaching the reader how to prepare for the future. The major features are the following:

1. Each topic is a general architectural approach—memory designs, pipeline techniques, and a variety of parallel structures.
2. Within each topic the focus is on fundamental bottlenecks—memory bandwidth, processing bandwidth, communications, and synchronization—and how to overcome these bottlenecks for each topic area.
3. The material addresses evaluation techniques to help the reader isolate aspects that are highly efficient from those that are not.
4. A few machines whose structure is of historical interest are described to illustrate how the concepts can be implemented.
5. Where appropriate, the text draws on examples of real applications and their architectural requirements.
6. Exercises at the end of chapters give the reader an opportunity to sketch out designs and perform evaluation under a variety of technology-oriented constraints.

The exercises are particularly important. They help the reader master the material by integrating a number of different ideas, often by working through a paper design that must satisfy some unusual set of constraints. In several exercises, the student is asked to produce a series of designs, each reflecting a different set of underlying devices. This helps the student gain experience in adapting basic techniques to new situations.

The text is intended for the advanced undergraduate and first-year graduate students. It assumes the student has had a course in machine organization so that the basic operation of a processor is well understood. Some experience with assembly language is helpful, but not essential. Programming in a high-level language such as Pascal, however, is necessary to understand the applications used as examples. Mathematical background in probability is helpful for Chapter 2, linear systems or numerical methods for Chapters 4 and 5, and some exposure to operating systems will assist understanding of Chapter 7. In no case is the material absolutely required because the text contains sufficient discussion and references to source material to support the presentation.

The text purposely avoids detailed descriptions of popular machines because in time the machines so described will inevitably be obsolete. In future years, a reader of such material may be led to think that the specific details of a successful machine represent good design decisions for the future as well as for the period in which the design was actually done. A better approach is for the individual instructor to discuss one or two current machines while using the text, with the notion that current machines can change each year at the discretion of the instructor. It is also possible to use the text without such supplementary material because the design exercises provide challenges that represent technology through the 1990s.

We jokingly tell students that the subject matter enjoys a positive benefit from the rapid change in technology. The instructor need not create new exercises and examinations for each new class. The questions may be the same each year, but the answers will be different.

A number of teaching aids are available with this edition. The exercises in Chapter 2 make use of traces of instruction execution for which a floppy disk with sample traces is available from the publisher for course adopters. The disk is in IBM-compatible format and can be accessed by programs written in a variety of programming languages.

Prior to the publication of this text, thorough studies of cache behavior required main-frame computers for analysis due to the massive amounts of data to process. The techniques described in Chapter 2 show how to reduce the processing by as much as two orders of magnitude and make possible the use of a personal computer as the primary analysis tool. The analysis techniques were first made widely available in the first edition of this text, and have now become standard among computer architects. The exercises for Chapter 2 give the student ample opportunity to practice cache analysis on the sample traces and to practice evaluating design alternatives.

An instructor's guide with solutions to selected exercises is also available from the publisher to course adopters. Among the solutions in the manual are sample solutions to some of the design exercises. The instructor should bear in mind that the design exercises can be satisfied by many different designs, and that the sample solutions are illustrative of good approaches, but are definitely not the only acceptable solutions. What is important is the reasoning used by the student to establish that a particular design meets the constraints imposed and is both efficient and effective in solving the given design problem.

Three sets of video-taped lectures provide instructional aid in a different form. A set of eight lectures that cover the highlights of the entire text can be ordered by writing to Addison-Wesley, Reading, MA 01867, Attn: Engineering Editor. A set of three lectures on the topics of multiprocessor cache coherence and synchronization is available from the IEEE Computer Society Press, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720. Another set of three lectures on advanced topics in cache behavior and cache analysis is available from the National Technological University, 700 Centre Ave., Ft. Collins, CO 80526, Attn: Richard Soderberg. The videotapes focus on central issues, and describe these topics visually and orally in a way that cannot be done in writing. Students and instructors will find the video tapes very useful for intensive study in short courses or self-paced instruction. The video medium is an effective means for fast transfer of information, and it is a useful supplement to a slower paced program of classroom lecture and intensive reading that encourages deeper understanding.

Instructors familiar with the first edition will find new material on program behavior models, RISC architecture, and parallel synchronization. The material

on program behavior has been introduced because machines have changed so quickly in recent years that designers are forced to produce new generations of processors without the benefit of traces of workloads for those processors. In such cases, the evaluation techniques described in Chapter 2 cannot be brought into play. The next best tool is to produce estimates of program behavior that can be used as input to design evaluations. We have incorporated some interesting new developments in program modeling that appeared after the publication of the first edition.

Similarly, RISC architecture and parallel synchronization have been developing very quickly in recent years and demanded additional space in the new edition. Beyond these topics, small incremental changes in the remaining topics have helped bring them up to date and streamlined their presentation.

The material in the text is structured in a modular fashion, with each chapter reasonably independent of every other chapter. The instructor can put together a course by selecting individual chapters and individual sections according to the background of the students, the prerequisites available, and the successor courses in the curriculum.

Chapters 2 and 3 form the core material. Cache memories and pipeline structures are widely used today, and they are likely to be effective in the technologies that will emerge in the next several years. These chapters should be taught in all course offerings.

For courses in which students have a good background in numerical methods, Chapters 4 and 5 show how parallel computer architectures are matched to problem domains. Students unfamiliar with the underlying mathematical applications will gain an understanding of computational methods in wide use from these chapters, and all readers will appreciate how data flow and synchronization of mathematical actions in an algorithm are directly supported by architectural features. The chapters are biased toward supercomputers and large-scale computations, but the material is useful as well for general purpose computers.

Chapters 6 and 7 treat multiprocessors, which are more general purpose than the machines of Chapters 4 and 5. Multiprocessors were almost exclusively research vehicles in the 1970s, and were in commercial use in niche areas in the 1980s. The 1990s will find a much broader use of multiprocessors as the speed of individual processors reaches the limit of metal interconnections. The highest sustainable clock rate for metal interconnections is roughly 200 to 250 MHz for a typical conductor geometry, although the clock rate can be boosted even higher at great expense by reducing the dimensions of all components and conductors. Computers in all classes from microprocessors to high-end machines started the 1990s within one to two generations of this clock limit. To sustain increases in performance through the decade, the industry must embrace multiprocessing in virtually all computers, or must abandon metal interconnection technology for another technology such as optical fiber or optical waveguide technology.

In this text, we explore the use of multiprocessors and leave the topic of optical interconnections for another time and another text. The multiprocessor discussion is oriented to where to seek performance improvement by using resources efficiently. The interplay of multiple disciplines is central to this discussion. Each specialist on a design team should have a broad shallow knowledge of the full scope of a design, including hardware, software, architecture, and applications, while enjoying a much deeper knowledge of a specialty area. Chapters 6 and 7 give a broad view of multiprocessors and delve deeply into particular topics such as algorithm design and performance models that are relevant to all specialties. These chapters are recommended especially for curricula that emphasize systems programming and computer engineering.

In one semester, it is reasonable to complete selected sections of all chapters, or to cover Chapters 2 and 3 and two other chapters in depth. Chapter 1, which has no exercises, is to be used as background reading to set the tone of the exposition. The text can easily satisfy the needs of a two-quarter or two-semester sequence if the instructor chooses to use the full material.

No matter which portion of the text is covered, working the exercises is critical for a thorough appreciation of the material. The design-oriented exercises can be rather frustrating at first because there is no clear indication of a correct answer. The reader wants to see exercises that can be answered quickly by jotting down a simple answer after a small amount of thought. What a pleasure to crank through a calculation and find the answer is 17.5. The design exercises are nothing like this. In a sense an answer is correct if it meets the constraints of the design. The reality is that the answer should be more than correct—it must be competitive.

The point of working such exercises is not the final design, but rather the process of arriving at the final design. What alternatives were considered? How does the final design overcome basic problems? Did the student consider a reasonable set of alternatives or was there a valid approach missed that should have been considered? Is the evaluation of the design reasonable? For what assumptions concerning technology factors and workload characteristics is the given design an efficient one?

After working through such problems the reader becomes familiar with the thought processes of the designer and gains both experience and insight into architectural design. Many exercises seem to capture real situations, and this is as intended. As in real situations, the reader may discover that there is no good solution, and a compromise has to be invented. Or there may be several reasonable solutions, and the reader has to pick one, possibly on the basis of characteristics that are secondary in importance because all solutions available have satisfactory primary characteristics. Many exercises have been drawn from design problems faced by the author, with constraints updated for the present and future.

The preparation of this text represents the fruits of labor of many parties.

The author's students, Tom Puzak, Zarka Cvetanovic, Dominique Thiebaut, and John Turek contributed a number of ideas to the text and exercises. They also offered helpful comments and criticisms as the project progressed. Kevin Donovan, David Epstein, and Robert Hinkley produced high-quality solutions to the exercises that appear in the instructor's guide. Other reviewers whose comments are reflected in these pages are William F. Applebe, Georgia Institute of Technology; Richard A. Erdrich, Unisys Corporation; John L. Hennessy, Stanford University; K. C. Murphy, Advanced Micro Devices; Paul Pederson, New York University; Richard L. Sites, Digital Equipment Corporation; Henry Levy, University of Washington; Glen Langdon, University of California at Santa Cruz; Peter Hsu, Sun Microcomputers, and Phil Emma, Jeff Lee, K. S. Natarajan, Howard Sachar, and Marc Surette, all with IBM. Collectively and individually, their work has aided greatly the process of developing material to make it easily accessible to the intended audience. The publication crew at Addison-Wesley did a remarkable job in putting the project together. Patsy DuMoulin, Bette Aaronson, and Karen Myer demonstrated that they know pipelining in practice better than the author does in theory, smoothly flowing the chapters through the tedious process of markup, text editing, and page composition in a remarkable example of proficiency in high-performance publishing. To Tom Robbins, we offer gratitude for support and encouragement in the project from its inception to its completion.

Chappaqua, New York H. S. S.

Contents

1

Architecture is preeminently the art of significant forms in space—that is, forms significant of their functions.
—Claude Bragdon, 1931

Introduction

This text is devoted to the study of the architecture of high-speed computer systems, with emphasis on design and analysis. We view a computer system as being constructed from a variety of functional modules such as processors, memories, input/output channels, and switching networks. By *architecture*, we mean the structure of the modules as they are organized in a computer system. The architectural design of a computer system involves selecting various functional modules such as processors and memories and organizing them into a system by designing the interconnections that tie them together. This is analogous to the architectural design of buildings, which involves selecting materials and fitting the pieces together to form a viable structure.

1.1 Technology and Architecture

Computer architecture is driven by technology. Every year brings new devices, new functions, and new possibilities. An imaginative and effective architecture for today could be a klunker for tomorrow, and likewise, a ridiculous proposal

for today may be ideal for tomorrow. There are no absolute rules that say that one architecture is better than another.

The key to learning about computer architecture is learning how to evaluate architecture in the context of the technology available. It is as important to know if a computer system makes effective use of processor cycles, memory capacity, and input/output bandwidth as it is to know its raw computational speed. The objective is to look at both cost and performance, not performance alone, in evaluating architectures. Because of changes in technology, relative costs among modules as well as absolute costs change dramatically every few years, so the best proportion of different types of modules in a cost-effective design changes with technology.

This text takes the approach that it is methodology, not conclusions, that needs to be taught. We present a menu of possibilities, some reasonable today and some not. We show how to construct high-performance systems by making selections from the menus, and we evaluate the systems produced in terms of technology that exists at the start of the 1990s. The conclusions reached by these evaluations are probably reasonable through the middle of the decade, but in no way do we claim that the architectures that look strongest today will be the best as we turn to a new millennium.

The methodology, however, is timeless. From time to time the computer architect needs to construct a new menu of design choices. With that menu and the design and evaluation techniques described in this text, the architect should be able to produce high-quality systems in any decade for the technology at that time.

Performance analysis should be based on the architecture of the total system. Design and analysis of high-performance systems is very complex, however, and is best approached by breaking the large system into a hierarchy of functional blocks, each with an architecture that can be analyzed in isolation. If any single function is very complicated, it too can be further refined into a collection of more primitive functions. Processor architecture, for example, involves putting together registers, arithmetic units, and control logic to create processors—the computational elements of a computer system.

An important facet of processor architecture is the design of the instruction set for the processor. In years past, there were controversies raging over whether instruction sets should be very simple or very complex. The controversies were not settled with a single solution; instruction sets continue to evolve with different underlying philosophies. But as part of the evolution, each different approach is influenced by the others, and incorporates advantages of other approaches where possible. We illuminate the factors that determine the quality of an instruction set, and in any technology an architect can measure those factors for a new design to guide the design process.

Computer architecture is sometimes confused with the design of computer

aesthetics, for which we have no absolute measures. We have no absolute test to conclude whether the work is a masterpiece or a piece of junk. If the art world agrees that it is a masterpiece, then it is a masterpiece.

Computer architecture, too, has an aesthetic side, but it is quite different from the arts. We can evaluate the quality of an architecture in terms of maximum number of results per cycle, program and data capacity, and cost, as well as other measures that tend to be important in various contexts. We need never debate a question such as, "but is it fast?"

Architectures can be compared on critical measures when choices must be made. The challenge comes because technology gives us new choices each year, and the decisions from last year may not hold this year. Not only must the architect understand the best decision for today, but the architect must factor in the effects of expected changes in technology over the life of a design. Therefore, not only do evaluation techniques play a crucial role in individual decisions, but by using these techniques over a period of years, the architect gains experience in understanding the impact of technological developments on new architectures and is able to judge trends for several years in the future.

Here are the principal criteria for judging an architecture:

- Performance;
- Cost; and
- Maximum program and data size.

There are a dozen or more other criteria, such as weight, power consumption, volume, and ease of programming, that may have relatively high significance in particular cases, but the three listed here are important in all applications and critical in most of them.

1.2.1 The Cost Factor

The cost criterion deserves a bit more explanation because so many people are confused about what it means. The cost of a computer system to a user is the money that the user pays for the system, namely its price. To the designer, cost is not so clearly defined. In most cases, cost is the cost of manufacturing, including a fair amortization of the cost of development and capital tools for construction. All too often we see comparisons of architectures that compare the parts cost of System *A* with the purchase price of System *B*, where System *A* is a novel architecture that is being proposed as an innovation, and System *B* represents a model in commercial production.

Another fallacious comparison is often made when relating hardware to software. In the early years of computing, software was often bundled free of charge with hardware, but, as the industry matured, software itself became a commodity of value to be sold.

We now discover that what was once a free good now commands a significant portion of a computing budget. The trends that people quote are depicted in Fig. 1.1, where we see the cost of software steadily rising with inflation and complexity, and with apparently little relief from advances in software tools. Plotted on the same curve is the general trend for hardware in the same period of time. Hardware components appear to be diminishing in cost at an unbelievable rate. If we project these trends forward ten to twenty years, we may believe that hardware might be bundled with software, given free with the purchase of the software that runs on it. But this view is rather naive.

Software and hardware costs each have two components:

1. A one-time development cost; and
2. A per-unit manufacturing cost.

The actual cost of a product, be it software or hardware, is shown in Fig. 1.2 as a function of the volume of production of a product. Note that the cost of the first unit is equal to the cost of the development. The cost curve moves upward with volume, but the slope tends to diminish with very high volumes because of manufacturing experience that tends to reduce per-unit costs over large volumes of production. The curve in Fig. 1.2 shows accumulated cost of the total volume of a product. The *price* of the product is the cost shown on the curve divided by the volume, plus a markup for profit. So price is very sensitive to volume when development costs are high.

When software was essentially free, the development costs were either bun-

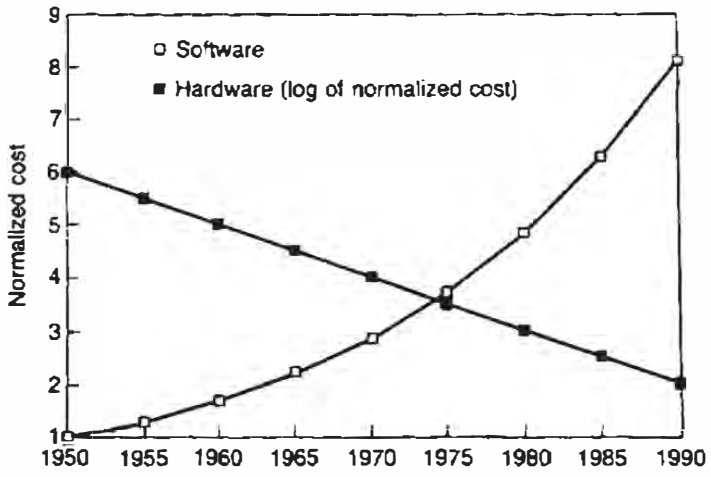

Fig. 1.1 A naive view of computer-cost trends.

of the database-management software may be sold. This alone can account for a factor-of-ten difference in price.

Our analysis also shows why in years to come hardware costs will still prove to be significant compared to software costs. At issue here is the cost of manufacturing. Software manufacturing costs are near zero today and can only go lower, so that software pricing in a competitive market mainly reflects the amortization of development costs.

Hardware manufacturing costs, while small on a per-chip basis, are many times more than software manufacturing costs. It is far less costly today to replicate accurate copies of software than it is to replicate hardware. Hardware requires assembly and testing to make sure that each copy is a faithful copy of the original design. This is far more complex today than the quality assurance on a software manufacturing line that simply has to compare each bit of information in software to see if it agrees with the original program.

Figure 1.2 suggests a strategy for the development and pricing of VLSI chips, hardware, and software. Development costs have to be amortized over the volume of units sold. The price of a unit is a strong factor in determining the ultimate volume sold. Manufacturers occasionally take a risk by setting an initial price that is profitable for a high-sales volume, but unprofitable for a lower volume. Another pricing strategy is to set an initial price somewhat higher in order to recover development costs on a lower-sales volume, with the price dropping as the product ages.

For example, memory chips have been quadrupling capacity every two to three years. The manufacturing cost per chip is usually constant per chip, regardless of the memory capacity of the chip. When a new memory chip that has four times the capacity of its predecessor is introduced, a typical strategy is to sell it initially at four or five times the price of its predecessor. Although the price per bit is about equal for new and old technologies, the newer technology leads to less expensive systems because of having only one fourth the number of memory chips.

During the initial life of the new memory technology, the manufacturer hopes to sell enough chips at the premium price to recover all of the development costs. The price will gradually diminish by a factor of four or five as it approaches a price that recovers only the production cost plus profit. The cycle begins again as the next generation of memory chips comes to market.

The profitability to the manufacturer in this scenario depends on how long the manufacturer can maintain a premium price. If competition forces prices to be lowered too soon, the manufacturer may never recoup the development cost.

Software pricing reflects development costs to a much greater extent than does hardware pricing. If a software publisher has a popular product with little direct competition, the manufacturer can recoup development cost within a short time by setting the price at a high premium over production costs. After re-

covering development costs, the publisher can gradually drop the price to generate a reasonable profit given current production costs, but too low to recoup additional profits to offset development costs. At this price, a new competitor cannot enter the market with an essentially identical product because the competitor's price cannot be high enough to recover product development costs. A competitor has to offer an enhanced product with new functions and new capabilities in order to compete.

Software and hardware pricing are starting to show some similarities, particularly in the personal computer market. The least expensive personal computers have relatively few parts because their functions have become well understood and standardized. A few VLSI chips are sufficient to implement a computer that formerly required several hundred chips. Hence, system manufacturing costs are diminishing, and a greater fraction of cost can be attributed to the development costs of the system and of the underlying VLSI chips. Nevertheless, there is still a greater production cost associated with the hardware than with software because the fabrication of hardware is more costly, and the testing of each finished unit requires substantially more effort for hardware than for software. To test computer hardware, one has to test to see if all the functions can be performed. To test software, one only has to verify that the bit pattern on the storage medium is a faithful replica of original software.

Because of the extra complexity in manufacturing hardware, in a competitive market it is very unlikely that computers of moderate or high performance will be given away to purchasers of the accompanying software.

1.2.2 Hardware Considerations

Another fallacious argument about new designs for the future concerns the lavish use of hardware components in a system. The architects state convincingly that with current trends in force, the cost of hardware will be negligible, so that we can afford to build systems of much greater hardware complexity in the future than we can today. Clearly, there is truth in this argument to the extent that future systems will surely be more powerful and complex at equal cost to today's systems. But the argument must be used with care because it does not excuse gross waste of hardware.

In the future, given System *A*, with 100 times the logic as present systems, and System *B*, whose performance is essentially identical to *A*'s but has only 10 or 20 times the logic as present systems, System *A* will be at a serious competitive disadvantage. For a few hundred or a few thousand copies of System *A* sold, System *A* may be priced competitively with System *B*. For higher volumes of production, however, the inefficiency of the architecture of System *A* will force its price higher than System *B*'s for equal system value. Of course, this presumes that both System *A* and System *B* are built from components of the same generation of technology. If *A*'s chips are 10 times as dense as *B*'s chips and therefore

10 times less costly per device, then the argument changes, and device technology, not architecture is the determining factor in the price of the system.

Throughout this text we explore the study of architecture by considering innovations of the future that depend on low-cost components. But we shall always heed the efficiency of the architectures we examine to be sure that we are using our building blocks well.

Consider, for example, a multiprocessor system in which there exists no shared memory, and suppose that we want to run a parallel program in which each processor executes the same program. Obviously, we can load identical copies of the program in all processors. When the program is small or the number of processors is rather modest, the memory consumed by the multiple copies may be quite tolerable.

But what if the program is a megabyte in size, and what if we plan to use 1000 processors in our system? Then the copies of the program account for a gigabyte of storage, which need not be present if there were some way to share one copy of code across all processors.

If System *A* uses multiple copies of programs, and System *B* through a clever design achieves nearly equal performance with a single copy, then the extra gigabyte of memory required by System *A* could well make System *A* totally uncompetitive with System *B*, unless the cost of storage becomes so insignificant that a gigabyte of memory accounts for a paltry fraction of the cost of a system. System *A*'s architect hopes that the cost per bit of memory will tumble in the future, but System *A* requires 10^{10} more bits, and this is an enormous multiplier. If current historical trends continue, a drop in cost per bit to offset an inefficiency of this magnitude would probably take 20 to 30 years.

In the example just presented, the architect of System *A* has to be aware of other approaches that could overcome a basic flaw in System *A* for the particular application. System *A* might be totally effective for other applications in which each processor requires a different program. But in the given context, System *B* has a tremendous, probably insurmountable advantage. The architect should measure the quality of the architecture across a number of applications that characterize how an architecture is to be used. The effectiveness may vary considerably from application to application, and such measurements should reveal where the architecture is truly beneficial to the user and where other approaches are superior.

A computer architecture might well have some minor but costly inherent flaws that escape the scrutiny of its designer. A different designer who can build essentially the same architecture with those flaws repaired can produce a more effective, and therefore more competitive, machine. Architects cannot hide inefficiency by arguing that hardware costs nothing.

As a simple example of this rule, consider an architecture with a rather large number of processors, such as 16,000, and assume that the processors are to be used in an application where the speedup attributed to *N* processors is pro-

portional to $\log_2 N$. (As astonishing as this sounds, such proposals have been made.) The 16,000 processors yield only a speedup of $14x$ for some constant x. The architect argues cogently that the 16,000 processors are so inexpensive that we can ignore their cost. The important fact is that the application runs $14x$ times faster than it runs on a single processor, and the speed increase is worth the small extra cost for the processors.

In this competitive world, the gross inefficiency of the architecture cannot escape notice for long. Soon there appears a System *B* to compete with this System *A*. System *B*'s architecture is identical to *A*'s in this case, except that it is a rather scaled-down version. In fact, System *B* has only 128 processors, not 16,000, so it runs only $7x$ times faster than a single processor.

System *A* is over 100 times more complex than System *B*, and yet System *A* runs only twice as fast. The cost of hardware would have to be near zero for System *B* to fail to compete with System *A*. For the next decade at least, it appears to be unjustifiable on a cost basis to double performance by replicating hardware one-hundred-fold.

The arguments in this section have taught us:

- We can evaluate architectures by their cost and performance;
- The effectiveness of an architecture must be measured on workloads for which the architecture is intended; and
- An architecture that is inefficient because of wasted resources will compete poorly against a simpler but more efficient architecture.

If computer architecture were purely an art, and aesthetics alone determined the quality of an architectural design, we would not have a basis for technical advances. Computer architecture combines the art of design with insight derived from careful analysis to create new forms of computer systems that yield ever greater service to their users.

1.3 High-Performance Techniques

Of the criteria discussed in the preceding section, this text emphasizes high performance. Our objective is to describe many different ways to improve system performance and give some additional information for evaluating those techniques. The menu of available techniques is rather extensive today, and each new generation of technology brings new ideas to the fore.

This text covers the highlights of the existing menu of design choices, but is by no means complete as of its publication date. Therefore we explore the design methodology—identify the critical design problems, generate solutions to these problems, evaluate, and select the best or most reasonable solution.

Although we emphasize performance, a thorough evaluation should consider all the criteria for comparing architectures. We simply place a greater weight

on performance. For the majority of the design space, cost and performance are treated together as a single parameter, the *cost-performance* ratio. The ratio is appropriate because it stays constant as you increase performance and cost by equal factors.

We would like to believe that users are willing to pay 10 percent more for a machine that is 10 percent faster, that is, a machine whose cost-performance ratio is equal to their current one. If a machine yields 20 percent higher performance for 10 percent higher cost, the users may see a genuine benefit in moving to the new machine, and indeed it has a lower cost-performance ratio reflecting a lower cost per computation. In most cases, users would not be interested in a machine that yields only 5 percent higher performance at 10 percent higher cost because their cost per computation goes up, not down, if they move to the new machine.

The exceptional cases occur when the present facilities are saturated, and the user absolutely must have greater capacity. Now the cost-performance ratio does not tell the whole story because the total benefit of greater capacity for the user may be much greater than the cost to achieve that capacity. The fact that the user is actually paying a higher cost per computation to obtain that capacity is incidental to the value in being able to do computations that could not be done before. However, if the user has a choice in how to obtain the necessary capacity, the user may still pick a solution based on the lowest cost-performance ratio, even though all possible solutions have higher ratios than the ratio for the user's current system.

1.3.1 Measuring Costs

We have been careful to give examples based on small changes in performance and cost. The cost-performance ratio is a good indicator of relative quality for small changes, but its usefulness breaks down when costs and performance vary by large factors.

It would be very deceptive, for example, to measure the cost-performance ratio of a small computer, such as an 8-bit video-game system, and to compare this to a much more powerful system, such as a workstation for computer-aided design. Although both systems are used to display images and interact with the images in real time, the video game probably has a much better cost-performance ratio than the workstation, assuming we can find some way of measuring relative performance. The problem is that the relative costs of the systems vary by a factor of up to 1000 to 1, and similarly, the relative performance factor is very large, although probably not as large as the relative cost.

The video game cannot do the same job as the workstation. Moreover, if you put enough copies of the video game together to have a performance equal to the workstation, the cost would be less than the workstation cost, but the collection of video games still could not do the same job. So just to be sure that

comparisons based on cost-performance ratios are valid, one should be careful to make the comparisons between computers that are similar in function and relatively close in performance.

This discussion points to two important ways to make architectural advances:

1. Make small perturbations in cost and performance that yield lower cost-performance ratios; and
2. Boost absolute performance to make new computations feasible at reasonable cost.

By "small" changes, we mean roughly a factor of 10 or less. Changes larger than this are surely welcome, but the cost-performance ratio cannot be trusted as a measure to evaluate the change. For the second point, the cost-performance ratio can actually increase, provided that the additional cost can be absorbed by the user, because the benefit of the greater capacity exceeds the cost to attain the capacity. We use both of these criteria throughout the text as informal ways to evaluate ideas.

Because absolute cost measured in currency is changing every year, it is more useful to define cost in terms of other parameters that influence cost. These parameters include the physical parameters, such as pin count, chip area, chip count, board area, and power consumption, derived from an implementation of an architecture. The parameters also include factors associated with development, such as elapsed design time, amount of associated software to be written, and size of development team required.

This text cannot easily account for all the factors that affect cost, but it can isolate the most important ones, especially when comparing two closely related architectures whose differences are limited to a few critical design choices. The intent is to focus on the differences and discuss the ways they affect the cost factors. Each different approach has its own advantages and disadvantages, and they in turn affect the cost of the approach. We cannot give absolute costs, but we can show the influence of the design decision on the cost parameters. The reader can then apply the prevailing cost functions to complete the evaluation.

1.3.2 The Role of Applications

With dramatic changes in technology ahead, how do we approach the problem of high-performance architecture design? For example, the new technology makes feasible massive parallelism. How much additional effort should be invested in increasing the performance of a single processor before we seek higher levels of performance by replicating processors? There is no simple answer to these questions. We need a combination of solutions, and what we choose almost certainly will be application dependent.

The role of applications is critical in the high-performance arena because costs tend to be very high to wring the greatest possible throughput from an architecture. Inefficiency is especially costly in this context because inefficiency adds greatly to already high cost, while contributing less than its fair share to performance. If the application area is heavily biased to some well-identified workload, then it becomes possible to design the architecture for that type of workload. The result is that the architecture can be stripped clean of irrelevant functions that might otherwise be necessary for general purposes. It can then be heavily armed with functions pertinent to the particular workload.

The objective then is to reduce inefficiency by making sure that all the functional components of the architecture contribute effectively to achieving high performance. If it were possible to build a general-purpose machine that would be equally effective for all high-performance applications, the industry would do so. And we cannot rule out this possibility in years to come. However, for the next decade, specific problem areas are so demanding of computational cycles that it is fruitful to design architectures specialized for these problem areas.

Among the important problem areas that have evolved are:

- *Highly structured numeric computations*—weather modeling, fluid flows, finite-element analysis;
- *Unstructured numeric computations*—Monte Carlo simulations, sparse matrix problems;
- *Real-time multifaceted problems*—speech recognition, image processing, and computer vision;
- *Large-memory and input/output-intensive problems*—database systems, transaction systems;
- *Graphics and design systems*—computer-aided design; and
- *Artificial intelligence*—knowledge-base-oriented systems, inferencing systems.

Obviously, the numerical areas call for sophisticated floating-point processors in the architecture, and the more demanding applications may require hundreds of such processors. The graphics systems may be more strongly oriented to fixed-point computations to provide the mathematical support required for windowing and perspective viewing. Floating point, however, plays an important role in some graphics applications, such as those that require smooth-curve rendering and ray-tracing calculations. The artificial-intelligence systems may require very little arithmetic capability, but they are usually heavily endowed with memory.

A high-performance architecture that meets the needs of all the areas mentioned must carry a burden of inefficiency for each problem area because a substantial portion of its capability would not be useful for individual applica-

tions. If the inefficiency is high enough for any one application area, then an efficient specialized machine for that area is more attractive than a general-purpose machine because the specialized machine should cost less to manufacture.

The cost advantage depends on having a large enough market for the specialized machine so that the cost of development can be spread across many copies produced. The advantage is lost if only a few copies are sold. Consequently, even the specialized high-performance machines should be as general purpose as possible within their problem domains so that the fixed costs can be amortized over as large a base as possible.

As special-purpose architectures are extended to broaden their problem domains, their potential market increases, but at the same time they tend to make less efficient use of their hardware. So the architect faces a trade-off. The idea is to balance the efficiency of the special-purpose architecture against the broad market base of the general-purpose architecture.

The architect has to find a place in the spectrum between single-purpose and all-purpose architecture for which a new design yields high performance at competitive cost. Design decisions are changing in time because they depend both on development costs and per-unit production costs, both of which are changing dramatically as the underlying technology advances.

1.3.3 The Impact of VLSI

There have been dramatic changes in the cost structure of high-performance architecture because of the development of VLSI. In the 1950s, when hardware was so expensive that one user could not afford to purchase a 1 M-byte machine, users shared the costs of large-scale computers and ran their programs concurrently on a single machine, thereby reducing the time that the memory, processor, and peripherals were idle. There seemed to be some economy by going to increasingly larger machines. The number of users served tended to grow linearly with computational power, but the price of the machines tended to grow more slowly.

Grosch's law was a popularly believed rule-of-thumb that stated that the cost of computational power grows at the rate of the square root of computational power. Although a great deal of evidence supported Grosch's law through the early 1960s, it is not clear whether the law reflected a fundamental notion about cost/performance or merely the prices being charged for computers.

In the 1980s, VLSI changed the economics of computers dramatically. Instead of paying a manufacturing cost per logic gate or interconnection wire, VLSI production costs are incurred per chip. All gates and interconnections on a chip are created in a batch by a fixed number of production steps that does not depend on the number of components. Hence, as more and more logic gates are packed onto a single chip, the cheaper the cost per gate becomes. As the density of gates increased over the years, VLSI led to a steady reduction in the

manufactured cost per gate of digital components. The steady improvement in cost performance in computer technology has been continuing since the beginning of the computer industry in the 1950s, and has been fueled by VLSI since the early 1970s. Over four decades, the cost-performance ratio has improved by about six orders of magnitude. This much improvement over that span of time is unparalleled in the history of civilization.

With performance so inexpensive, we no longer try to put to use every last cycle from a 1 M-byte machine, and it is common to find such machines lying idle for most of a 24-hour day. So, for the huge number of small computations, the user buys a machine big enough to get the job done, and maybe a little bigger than that to have some reserve capacity. It is not particularly economical to buy enormously big machines, then gain access to the machine cycles by sharing the cycles among many users.

Strictly from a performance point of view, we do not see an economy of scale that drives all users to larger machines regardless of their needs, as once appeared to be the case. Rather, to describe the situation in simplistic terms, we see small jobs run on small machines, and large jobs run on large machines. The "small" machine of today has about the same computational power as the "large" machine of 25 years ago, so the machine formerly shared by 100 users, is now owned outright by one user.

The need to access shared data complicates the arguments here somewhat. We discover that large machines or networks of smaller machines support many concurrent users today because the need to access shared data, as opposed to the need to share machine cycles, is the driving force. And we still see the supercomputers shared among many users because these machine cycles are very expensive for any single user.

1.3.4 The Impact of Digital Communications

Having lived with continuous improvement from VLSI technology for over 20 years, the computer architect has learned to plan the next generation of machines by scaling the number of gates per design, the processing rate, the access time of memory, and the cost of components according to historic trends. The design exercise tends to be one of putting new components together to produce a more effective machine for today's workload.

Meanwhile in the sister industry of telecommunications, another technology has taken hold that has a profound future impact on computers. Fiber optics and high-bandwidth electronics together have enabled telecommunications to shift from a technology designed for 10 kHz signals to one designed for 1 GHz signals, and to achieve the new bandwidth at low cost. The cost of delivering conventional voice channels using the new technology may be 1000 times less expensive than when delivered by the old technology. Moreover, the new technology is available virtually overnight, so that we should be able to reap these

benefits in a few years instead of in a few decades. Just as computing technology has achieved an astounding advance over a short period of time, the rate of advance in communications technology is even greater.

What will be the impact on computers when we have access to multi-GHz fiber-optic networks? If we do business as usual, we will feel the impact mainly through lowered cost to operate computers in a network and a higher bandwidth within the network. But why do business as usual? Digital communications networks are much more likely to create new product opportunities. Systems that formerly interchanged characters and documents can be supplanted by systems that interchange individual images and video sequences. Starting with this premise, the role of the computer evolves from a processing center to an information server. The computer can be the window to a digital Library of Congress and then instantly switch roles to become a video telephone. Our printed output of former years becomes a multimedia presentation with sound and animation.

The architect in this scenario has to shift focus from processing of data within a computer to the movement of data into and out of the computer. The architect must synchronize low-speed audio and high-speed video data streams. The data streams have to be processed at tightly constrained rates in order to produce intelligible conversation and realistic animation.

These are new requirements. To meet such requirements requires designs that can handle proportionally greater input/output traffic relative to processing power than today's computers can handle, with the ratio increasing by one to two orders of magnitude. The architect has traditionally focused on the low-latency, heavily reused traffic between processor and memory. The new requirements call for the support of data streams with little or no reuse and controlled latency. This is a new challenge for the architect.

Consequently, a major change of underlying technology is driving computer architecture in totally new directions. In this case, the technology change is in a closely related field, and the impact overflows from that field onto computers. With VLSI, the architect has lived with change for several generations of designs. The changes are continuous and to a certain extent they can be incorporated into design plans. The change in digital communications is more difficult to absorb because it is massive and abruptly introduced.

1.3.5 The Effect of Technological Change on Cost

If we look at the underlying technology at any given time, we see curves that look something like the curve in Fig. 1.3. This figure shows performance measured in millions of instructions per second (MIPS). It shows a rough picture of relative cost per MIPS as a function of MIPS of performance. The figure is intentionally imprecise because the data on which it is based is highly volatile. The idea is that the curve consists of several plateaus. The lowest plateau rep-

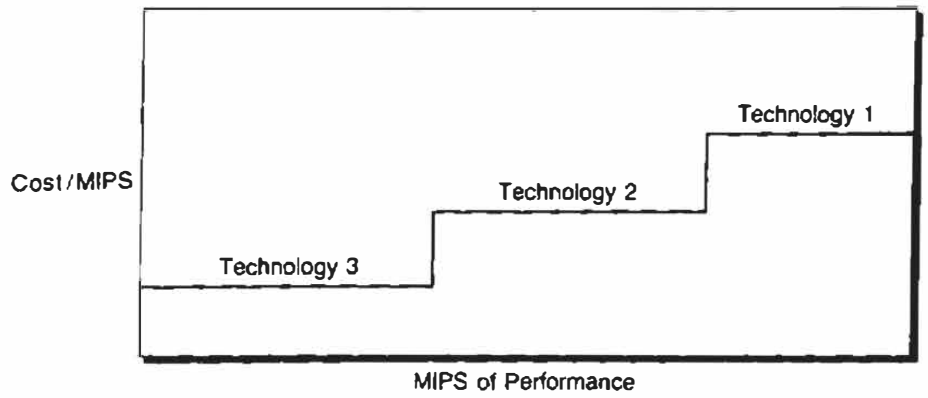

Fig. 1.3 Cost/performance as a function of performance.

resents the cost per MIPS for computers that use the dominant technology for that plateau.

In the 1980s, the dominant technology for low-performance machines was metal-oxide semiconductor (MOS), mostly NMOS (MOS devices with negatively doped channels) in the early 1980s, with CMOS (complementary MOS, devices with both negatively and positively doped channels) becoming prevalent in the later 1980s. At any given time, the cost per MIPS is fairly constant for all machines made from this technology.

The next plateau is the next level of technology, presumably a bipolar technology such as emitter-coupled logic (ECL). The cost per MIPS is nearly constant for all machines of this technology as well.

The third plateau is a more exotic device technology fabricated especially for peak performance. Gallium-arsenide devices appear in this plateau. This technology has the highest cost because of cooling requirements, manufacturing difficulties, lower chip density, or other similar factors. This plateau too has almost constant cost per MIPS for all machines produced from the technology.

Although the graph in Fig. 1.3 is imprecise, it is intended to show how a device technology influences the cost-performance relationship. The devices dictate the basic cycle time of the computer. A rough measure of processing power is the width of the address and data paths times the clock frequency; this is an upper bound on the information-transfer rate of a computer system.

For a given technology, most high-speed designs adopt a maximum or near-maximum clock frequency that is usually dependent on the technology and fairly consistent for all designs that use that technology. Consequently, the most appropriate way to improve performance is to move from 8-bit to 16-bit to 32-bit data paths, with a corresponding increase in memory capacity to support

higher levels of performance. This produces a performance gain that grows linearly with the bus width, but the cost tends to grow linearly as well in bus width, so that as bus widths increase, the additional performance produced is achieved at a constant cost per MIPS.

The assumption that cost grows linearly in bus width is certainly true in regard to the cost of data paths, drivers, and physical wires on the data paths, but the cost of memory need not grow linearly with bus width. The critical assumption that produces the plateau-like curve in Fig. 1.3 is that memory size tends to increase linearly with performance. That is, a typical configuration for a system rated at 2 MIPS might be 4 M-bytes, and faster versions of the same system that run at 4 MIPS and 8 MIPS would be configured at 8 M- and 16 M-bytes, respectively. If indeed this growth rate is true, then Fig. 1.3 is quite reasonable.

The main conclusion to draw from Fig. 1.3 is that if the curves are truly flat, then within a device technology there is no particular economy of scale. Worse yet, if an architecture's performance exceeds the capabilities of one device technology, then the move to the next higher plateau of technology may result in a higher cost per MIPS. This is directly contrary to the principle of economy of scale. At the very highest levels of performance, the device technology may be quite exotic, raising the cost per MIPS well over the cost per MIPS of less powerful systems.

If Fig. 1.3 is accurate as drawn for the beginning of the 1990s, one would conclude that once a device technology for an architecture implementation is selected, the cost-performance ratio is not strongly influenced by the absolute performance of a system, so there is no particular bias to produce high- or low-performance machines for that technology. If Fig. 1.3 is not accurate, and there exists an economy of scale, then the cost-performance ratio improves as performance goes up, and there is a strong bias toward building the highest possible performance for each different device technology. No matter what is true at the time this text is written, a future version of Fig. 1.3 may be totally different, and the architect has to take the shape of the curve into account in machine design.

Now let us reflect on the variables that the architect can control in creating a high-performance machine. By measuring performance in MIPS, we can write

$$\text{MIPS} = (\text{instructions/cycle})(\text{cycles/second}) \cdot 10^{-6}$$

The first factor is a function of the architecture, which is controlled by the architect. The second factor is determined by the devices, which are controlled by the technology.

Actually, the dichotomy between architecture and device technology is not as sharp as we depict it because the second factor, the clock speed, is partially dependent on architectural factors such as the complexity of instruction decoding. Nevertheless, to a first approximation we can affect performance by con-

centrating on the first factor, the number of instructions executed per machine cycle. What are the alternatives available?

- *Reduce the number of instructions to execute.* By using better algorithms, it may be possible to do equal work with fewer instructions.
- *Build hardware assists into the architecture to improve the architecture's efficiency.* Advances in architecture such as cache memory can increase the number of instructions executed per cycle. Another possibility is to create higher-level instructions such as SORT and SEARCH that have been optimized for particular purposes.
- *Execute many instructions concurrently.* Use parallel hardware in some fashion to increase the number of instructions that can be executed in a single cycle.

It is strange to see the first item in this list in a text on architecture. One might assume that the computer architect does not dabble with algorithms. Quite to the contrary. Since the goal is high performance as measured on some set of applications, how that goal is achieved is important because of its impact on system cost, but there are no constraints that force the solution to be architectural only. In fact, algorithmic improvements may be the most cost effective of any of the approaches mentioned previously because copies of algorithms can be manufactured for essentially zero cost as compared to the cost of hardware-intensive solutions.

1.3.6 Algorithms and Architecture

The architect has to look carefully at algorithms to decide how to achieve high performance in an architecture. Applications that are limited by the speed of floating-point division, by internal sorting, or by the ability to interpret bit-mapped representations of visual data may require extensive study by the architect. Changes to the original algorithms, sometimes simple changes and sometimes totally new approaches, may transform an application from one for which high-performance architectures are poorly suited to one that can easily be enhanced by some inexpensive hardware assists.

An algorithmic breakthrough might even eliminate the need for high-performance architectures for a particular application. Floating-point division and sorting are each reasonably well understood areas for which major changes to existing algorithms are unlikely to be developed, but many new areas are emerging for which the current crop of algorithms represents the early, immature efforts to solve the problems. Additional study of the algorithms may well produce much greater performance.

Although we cannot expect a computer architect to step into an application area and produce a breakthrough in algorithms for that area, it is possible for an architect to recast basic algorithms into forms more suitable for processing.

The architect may partition a problem in new ways to reduce the size of working memory or the number of high-speed registers required. Or the architect may find a way to structure the problem so that it fits well on a parallel architecture.

The architect actually has control of both the algorithm and the architecture. The objective is to manipulate both to create an algorithm and architecture that mutually constitute an effective solution. Usually a class of algorithms, rather than a single algorithm, must be considered, and the more difficult objective is to create a single architecture that is good for all the problems in the class.

The second solution technique involves changes to the basic architecture. In the past we have seen many different techniques used to improve performance. Such things as instruction buffers, cache memories, and pipelined execution have appeared in many commercial machine implementations. We have seen complex instructions installed in machines to reduce the number of instruction fetches, and we have seen complex instructions eliminated from instruction sets to reduce the basic instruction-cycle time for a machine.

The architect needs to know where bottlenecks may exist in a system, and then, if possible, take steps to remove those bottlenecks. At peak performance a well-designed system has many different components near saturation. A poorly designed system has some single bottleneck when running at maximum speed, and all other functional units are underutilized. By eliminating some excess capacity, this kind of system may be made less expensive at no loss of performance. Or by dealing with the bottleneck exclusively, it may be possible to improve performance relatively inexpensively.

The last choice on the list is parallelism. This is usually the most costly way to achieve high speed, but VLSI technology has changed the economics so dramatically that parallel hardware has become a viable alternative. Returning to Fig. 1.3, we see that there is some advantage in using inexpensive technology in seeking high performance. Figure 1.3 suggests that each device technology is most effective over some range of performance.

Parallel architectures, however, provide a way of using the inexpensive device technology at much higher performance ranges. An architect can attempt to exploit the plateau structure of Fig. 1.3 to create an efficient parallel machine out of low-cost devices. The objective is to increase MIPS by adding performance in a way that performance grows proportionally with cost. If this can be accomplished, the architect stays on the flat plateau of the low-cost technology while moving performance into the region dominated by high-cost technology. Certainly, this is one of the attractions of moving to parallel architectures, although the gains achieved through less-expensive device technology are negated in part by a lower efficiency in executing a program in parallel rather than on a serial machine. In fact, because of the inefficiency of parallelism the cost per MIPS grows with the number of processors in a complex of parallel processors. The challenge is to keep this growth small enough so that a parallel machine built

with low-cost devices is less costly than a serial machine of equal speed built with high-cost devices.

These three techniques for making improvements are important, but not exhaustive. All opportunities are worth investigating. Because the real world does not follow the highly idealized world of design presumed in this text, pressures that exist in the real world might lead to unbalanced configurations that are difficult to justify on the merits of cost and performance.

Consider a situation in which System *A* has 256 K-bytes of a cache memory, and System *B* competes with System *A* by offering 512 K-bytes or 1 M-byte to gain a competitive edge through larger numbers, even when other factors cannot justify the larger cache memory. If not cache, then main memory size might be offered at 4 G-bytes instead of at 1 G-byte, or 32 processors in place of eight processors. Consequently, competitive pressures could easily cause an architect to configure poorly balanced systems.

Over a period of years, however, cost and performance measures prevail. If systems are unbalanced at first release, the cost to the user or to the system producer will be too high for the performance levels actually realized. Eventually configurations are altered to bring them back into a reasonable range of cost and performance. Designing for the shorter-term view by playing a numbers game may be a fact of life, but quality, efficiency, and effectiveness dictate that a sound architectural approach drives computer design over the long term.

In closing this section, we summarize by saying that all three of the approaches—algorithms, architectural assists, and parallel architectures—must be considered by the architect. High performance may require a combination of all three approaches in any given system.

In each new design lies a significant challenge for the architect because the rules of the game change continuously. The factors that influenced the design decisions last year may no longer hold this year. The architect has new devices to use as building blocks, and new organizations that are feasible to implement. And the applications have changed, too, with totally new problem areas becoming targets of computing technology. In addition, the older areas are increasing in scope and scale. The constant in architectural design is the methodology for putting together the various components available to create effective solutions for application areas.

1.4 Historical References

We use the term *computer architecture* in a broader sense than it was used when it was first introduced by Amdahl *et al.* [1964]. Their definition of computer architecture is the computer as seen by the programmer, which is essentially the instruction set plus a model of the execution of the instruction set. The

importance of this notion is that a family of computers can have an identical architecture, yet span a large range of performance and capacity. Programs that execute on different models in one family give identical results. Different members of a family are different in implementation and may have varying degrees of hardware, microcode, and software embedded within them to support the execution of instructions defined for the family.

This narrow sense of computer architecture proved to be invaluable for defining the characteristics of a family without committing to particular implementations of architectural characteristics. The concept has been crucial in the development of the IBM 360 and 370 families, the PDP-11 and VAX families, and in recent microprocessor families such as the Motorola 680XX and the Intel 80X86 families.

Changes in technology have made the architectural definition offered by Amdahl *et al.* somewhat obsolete. The original reason for defining the architecture with the instruction-set definition was to ensure compatible execution of a program on any member of the family large enough to run the program.

Instruction-level compatibility is not sufficient in itself since program execution can depend on libraries, operating system facilities, local configuration, and other factors that are not part of this narrow sense of architecture. This has led to the standardization at other levels of interfaces, such as the operating system interface or the source language.

Meanwhile, the rapid development of VLSI and the changing cost structure of digital components forced some computer families to bring out new instruction sets. The 24-bit address of System 360 and 370 of 1964 vintage evolved to a 31-bit address in System 370 XA in 1982 and within a few years evolved again to a 44-bit address in the System 370 ESA architecture in 1988. The 16-bit address of the PDP-11 family (first offered in 1968) eventually became a 32-bit address in the VAX family in 1978.

With new devices to use in designs and the flexibility to change instruction sets, the computer designer of today faces a set of constraints somewhat different from those faced in decades past. Hence, we have enlarged the definition of computer architecture to include the design of a computer system from its instruction set and structure down to functional modules. Many topics treated in this text are issues of implementation that are not within the scope of the narrow definition of computer architecture as defined in Amdahl *et al.*

Readers interested in the historical development of computer architecture and in prerequisite material will find a wealth of information in Bell and Newell [1971] and Siewiorek, Bell, and Newell [1982]. Both books reprint a collection of historically important papers in computer architecture and include authors' commentary, which serves to organize the material and fill voids not covered in the literature.

Textbooks in the area appeared rather late in the development of computers, with Stone [1974] being among the early offerings. This is a collection of original

contributions structured as a textbook. Hayes [1978] and Baer [1980] are both high-quality texts that brought updated material into the classroom. Tanenbaum [1976] covers an interesting combination of computer architecture and operating systems, an interface of two subject areas that has become increasingly important as the operating-system level has begun to displace the instruction-set level as the standard interface for applications. Stone's second edition [1980] includes material on operating systems as well as an evolution of the architecture-based material from the first edition.

Many texts covering both specialized and general aspects of computer architecture started appearing early in the 1980s. They are too numerous to list here, but books that are useful supplements for specific topic areas covered in this book are cited in the appropriate place in the text. The overwhelming trend in texts has been to take a descriptive view. That is, the texts tend to discuss techniques by means of machine examples that have embodied those techniques. While it is useful to present historical information on computers, knowledge of history alone is insufficient to prepare for the future. The text by Hennessy and Patterson [1990] takes a different approach in that it includes the results of quantitative analyses that are helpful in evaluating alternative approaches.

2

I know of no way of judging the future but by the past.

—Patrick Henry, 1775

Memory-System Design

Some architecture researchers have called the memory system of a computer "the von Neumann bottleneck" because of the critical role it plays in affecting peak throughput. The design of the memory system is our starting point in this text, and it is frequently the starting point in machine designs. The central problem is to:

- Bring the input data from the outside world into memory;
- Buffer the data there until they can be passed to a processor;
- Compute the output data and buffer them in memory until they can be delivered outside the computer; and
- Transmit the output data from memory to the outside world.

The bandwidth between memory and the outside world limits how fast we can obtain input and deliver output. The memory system also limits how fast input data can be delivered to a processor and how fast the results can be received from the processor. Since instructions are also stored in memory, the architect must provide for concurrent demands on memory for data to process, instructions to execute, and input/output transfers between memory and the external world.

In this chapter we examine the use of *cache* and *virtual memory* to produce very efficient hierarchical memory systems. These systems are composed of a mix of memory devices that range in performance and cost. A well-designed memory system of this type tends to perform as if the entire memory were composed of the fastest devices in its structure, yet its cost tends to be dictated by slower, less expensive devices.

We explore the basic principles of the hierarchy here, but because the best possible memory design depends on workload and the available technology, we cannot give a concise formula for a good design. We do, however, present some powerful techniques for evaluating designs that will enable both the professional architect and the student to explore a range of memory designs with simple programs running on personal computers.

Why is memory so critical to performance? The major constraint imposed by high-speed memory in a von Neumann architecture is:

> A single memory module of conventional design can access no more than one word during each cycle of the memory clock.

The *bandwidth* of memory is the measure of the number of bits per second that can be accessed. If our memory system has a 100 ns cycle time and accesses 64 bits (8 bytes) per cycle, its bandwidth is 640 M-bits (80 M-bytes) per second.

If we absolutely must increase memory bandwidth to increase performance, then there are several choices available to the memory designer:

- Reduce the cycle time.
- Increase the word size of memory by accessing more bits per cycle.
- Replicate the memory modules and access two or more of them concurrently. (This is one way of increasing the word size of memory.)

The designer may also explore unconventional schemes, such as parallel-search memories, "intelligent" memories with internal sorting and searching capability, or hierarchical memories with a variety of speeds and functional capabilities. If an unconventional design proves to be an effective design, it will be incorporated in many computer systems, and eventually it will become conventional.

Advances in hardware technology have made available larger and faster memories at an almost unbelievable rate, and the trend is likely to continue through the 1990s. The designer can tap the new technology in a variety of ways, including brute-force techniques that have an inefficiency that would have been totally unacceptable in former years. For example, to increase memory bandwidth, a machine architect today can choose a very long word and wide bus, such as 256 bytes, even though there is a strong probability that many of the bytes accessed over the bus will never be used.

Inefficient techniques abound, and new technology may provide the means for using such techniques at acceptable cost. But efficient techniques are much

more difficult to invent and analyze, and they will always have an advantage over the inefficient ones. Therefore, this chapter dwells on some efficient techniques that have proved to be useful in the last few years and presents new tools for evaluating these techniques.

2.1 Exploiting Program Characteristics

The basic building block of central memory is *random-access memory* (RAM). Figure 2.1 shows a diagram of the structure of a typical memory module. Note the two registers, ADDRESS and DATA. During a READ cycle, the memory accesses the item at the location given by the contents of ADDRESS and places a copy of the item in DATA. During a WRITE cycle, the memory also accesses the item as indicated by the contents of ADDRESS, but in this case the contents of DATA are copied to the location in memory.

The term *access* refers to the physical actions that occur in the memory module during a READ or WRITE cycle. What happens is that there is a logical path set up between the selected location and DATA. The direction that data flows along this path depends on whether the operation is READ or WRITE, but in either case, to access a location the memory system uses the contents of the address register to enable or disable internal gates in such a way that for each address value, exactly one location becomes logically connected to the DATA register.

The name *random access* conveys the idea that each access to any location in memory takes a fixed amount of time, independent of what sequence of accesses occur. Suppose, for example, a READ to Location 20 takes 10 ns, and the READ is followed by a WRITE to Location 347. For a random-access memory, the WRITE also takes 10 ns because all cycles are 10 ns, no matter what location is accessed.

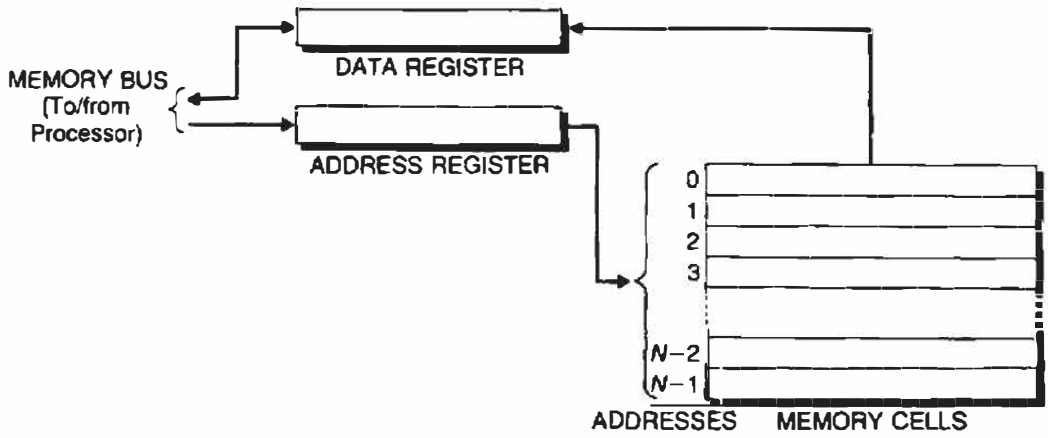

Fig. 2.1 The structure of a random-access memory (RAM).

Contrast this random-access behavior with *sequential access*. If a memory were organized as a shift-register or as a continuous magnetic tape, then access times would depend on the sequence of addresses issued to memory. An access to Location 11 immediately after an access to Location 10 would take, for example, 10 ns if this were the time required to access consecutive items. But, by this reasoning, an access to Location 17 that immediately follows an access to Location 10 would take 70 ns. To access Location 17 after accessing Location 10, the memory cycles through Locations 11 to 17, with each location requiring 10 ns to process. Access time is potentially very large in a sequential memory when items are far apart.

Obviously, there is a tremendous performance advantage for random-access memories over sequential-access memories, but the cost per bit of sequential-access memory is usually quite low compared to random-access memory.

The trade-off between cost and performance for these two types of memory is but one example of the design choices open to the computer architect. Suppose, for example, the architect can exploit the low cost of a sequential-access memory without necessarily incurring a performance penalty if the programs to be run on the computer system can be organized so that the bulk of their accesses is sequential. Then nonsequential accesses must be either negligible or executed inexpensively, perhaps by means of a small random-access memory.

If a particular site has a workload that does not directly use a sequential-access memory, then the users must convert their workload in order to capture the cost-performance benefit of the hypothetical sequential-access machine. The users may have to alter the applications programs by hand, or, better yet, they might produce a translator to alter the existing programs automatically.

A translator that minimizes access time may be quite feasible to write for this particular example, but in general there is no guarantee that program conversion will be successful, and the cost of conversion may be very high. Therefore, the decision to use a sequential memory in addition to random-access memory requires careful consideration of many related factors regarding how the software can make effective use of the new facilities. Consequently, the cost-benefits of the new architecture are less apparent to the user who must invest in a conversion with its cost, risk, and delay.

All advances, whether they are in device technology or architecture, result in the same considerations by the user community as the advances compete with existing technology for wide acceptance. If a new architecture is incompatible with existing technology, then its cost-performance benefits must be great enough and visible enough to motivate the users to convert to the new architecture. If a new architecture is compatible with existing technology to the extent that conversion can be done quickly and at low cost, it just has to be better than existing alternatives, not necessarily much better.

As we look at the history of virtual memory and cache memory, we can see how these concepts permitted systems to make use of rotating magnetic memory

for bulk store while performing most execution from high-speed random-access memory. The success of the idea is due both to the cost-performance benefit and to the fact that it is immediately useful for all programs without forcing the programs to be rewritten.

The development of virtual memory and cache memory was stimulated by a need to use magnetic tape with purely sequential access as the storage medium for bulk data. In the 1950s, computers had small central memories (usually 128 K-bytes or less) and could not contain the larger programs and their associated data. So programmers were forced to partition their programs into separate overlays, each of which was small enough to fit into central memory. Program execution moved from overlay to overlay, with a memory load required each time one overlay reached a point at which it invoked a new overlay.

The partitioning process was tedious and error-prone, but necessary for programs that were otherwise too large to fit into memory. Loading overlays from magnetic tape was very time consuming, so programmers took extra care to assure that as few overlays as possible occurred during the execution of a program.

This crude way of managing large programs eventually revealed program characteristics that can easily be exploited to create very high-performance systems at relatively low cost. The cache memories and virtual-memory systems that are widely used today have been developed largely because of the observations of program behavior that revealed the strong tendency for memory accesses to be clustered in small regions of memory during any short period of time.

The historical development of this technique received a major boost at the University of Manchester in the course of the design of the Atlas computer [Kilburn *et al.* 1962] shown in Fig. 2.2. The approach used by this design team was called *one-level store* to indicate that programs viewed memory as made up of one level of homogeneous devices, as if it were one large random-access memory. Actually, there were two levels in the memory hierarchy, a small random-access main memory with 16K words, and a much larger magnetic-drum memory, with 96K words, that held the bulk of the program and data.

The user programmed the Atlas machine as if the size of memory were the size of drum memory. The Atlas had special hardware that treated memory as composed of individual pages of 512 words each and automatically loaded main memory with 32 pages of the program and data. If the Atlas requested an item from a page not resident in main memory, the requested page would be brought into main memory, and some other resident page would be written back to drum.

The Atlas used a "learning program" that attempted to retain the most useful pages in main memory. All of the swapping between drum and main memory was totally invisible to the user. The user did not have to specify when to bring data from drum to main memory or when to move it back again. The user had

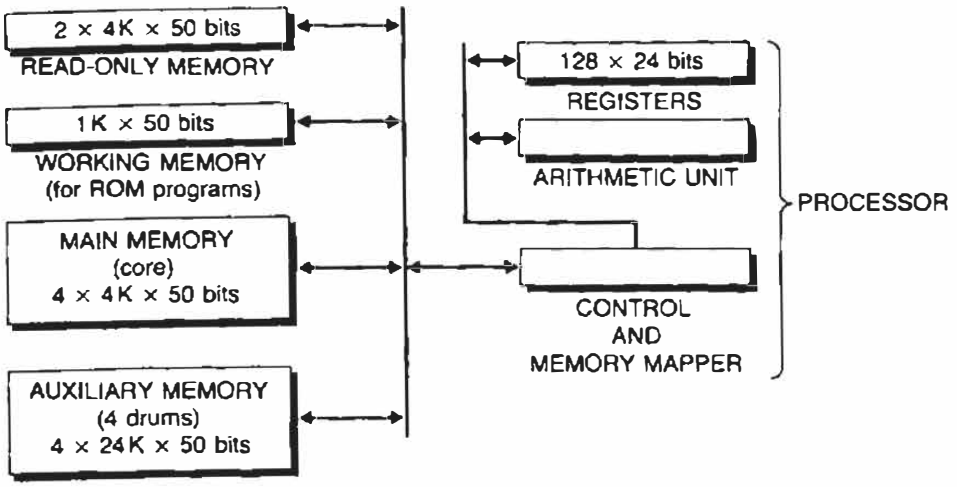

Fig. 2.2 The block diagram of the Atlas computer.

the convenience of programming with a large memory the size of the drum, whose apparent cycle time was closer to the cycle time of the central memory.

Because the Atlas made main memory appear to be much larger than it actually was, the name *virtual memory* was eventually applied to this general scheme, and the term *one-level store*, used by Kilburn *et al.*, is seldom used today. Drum memory on the Atlas had a long average latency of between 2 and 14 ms to obtain the first word of a page, and the sequential access to successive words in the page occurred at the rate of about 4 μs per word. The cycle time for a random access to main memory was about 2 μs per word.

As long as the required pages were resident in main memory, computations proceeded at maximum computation rate. A missing page caused a tremendous penalty in time, since access to an item in a missing page took about 500 times longer than access to the same item when it was resident in main memory.

In current terminology the attempt to access a missing page is called a *page fault*. It is clear that maintaining a very low rate of page faults is critical to the success of a virtual-memory system. As the page-fault rate increases, the apparent cycle time of memory grows much larger than the cycle time of the faster memory, and instead approaches the cycle time of the slower memory. Performance at high fault rates is disastrously low.

The characteristic that drove the invention of virtual memory on the Atlas machine is called *locality*. Program references tend to be locally clustered in time. That is, there is a strong tendency for future patterns of access to be similar to access patterns that occurred in the near past. If an instruction stream truly

shows no sequential correlation so that the item accessed on any cycle is independent of the history of accesses, then for any given cycle, all items in the program are equally likely to be accessed. If this were the case, a small high-speed memory would be of marginal benefit. But if there is significant serial correlation, then the history of accesses can be used to predict the accesses that will occur in the future. With such predictions, the computer system can move pages between low- and high-speed memory in a way that tends to reduce faults.

There are really two questions here:

1. Is there a significant sequential correlation in typical streams of address references?
2. If there is a serial correlation, how can it be exploited?

The first question has been studied in depth. The answers obtained over a broad class of programs running on almost every possible machine consistently report a very strong sequential correlation. The findings suggest that at any given moment of time, the probability distribution for what might be referenced next looks something like the graph shown in Fig. 2.3. This figure shows the probability of access as a function of memory address (in virtual memory). Note that a few regions are highly probable, a few other regions have a low-to-moderate probability, and the remainder of the address space is very unlikely to be accessed in the near future. Note also that the regions with the highest probability of access are scattered throughout virtual memory.

One region that has a high probability is the one that contains the present program counter because it is likely to execute the next instruction in sequence.

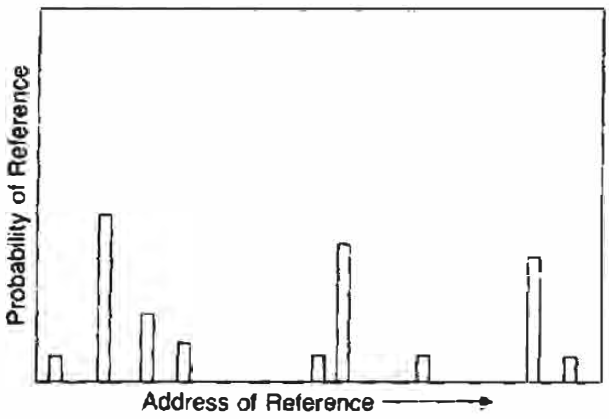

Fig. 2.3 The instantaneous value of the probability of a reference as a function of the address of the reference.

Other regions contain active data, the instructions for subroutines that might be entered, and the return point to a subroutine that called the presently executing subroutine. If the executing program were written in a block-structured language such as Pascal, then the present stack frame for local variables and parameters is another area with a high probability of access.

Another possible model for the probability distribution is shown in Fig. 2.4. In this model, the probability of access falls off with the distance from the currently executing instruction, where distance is defined to be the absolute difference of two memory addresses. This model is not a good characterization of the characteristics of programs that execute on the machines most commonly used today, but it too displays sequential correlation.

This type of correlation is easily exploited because the computer system would attempt to retain in main memory those items whose addresses are closest to the address of the executing instruction. Moreover, this model suggests that it is a good idea to make pages fairly large because once an item on a page is referenced, the probability is very high that other items on the same page will be referenced.

Early designs of virtual-memory systems occasionally made the assumption that memory references were characterized better by Fig. 2.4 than by Fig. 2.3, with the result that these systems tended to use too large a page size and had more traffic between low- and high-speed memory than was necessary. The large page size resulted in many words being transferred to high-speed memory that were never accessed while resident in high-speed memory. That portion of high-speed memory would have been better used for other regions of memory, and a small page size would have made more high-speed memory available.

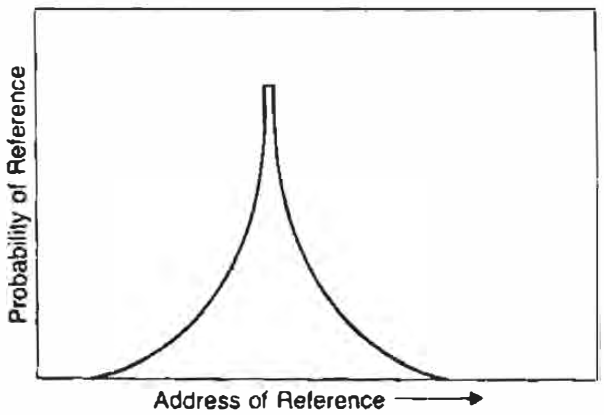

Fig. 2.4 A possible, but unrealistic, model of address-reference probability.

Although Fig. 2.3 is a more accurate characterization of streams of address references than is Fig. 2.4, there is always the possibility that a designer can bias the statistics of accesses through some unusual characteristics of a design. For example, compilers and loaders may attempt to place subroutines together on the same page if there is evidence that the subroutines work together in some way. Or through a combination of instruction-set design and compiler design, it may be possible to reduce branches to far away regions.

Machines with vector instructions may behave more nearly like Fig. 2.4 than Fig. 2.3. Nevertheless, no matter what the details of the correlation are, there is overwhelming evidence that streams of address references exhibit strong sequential correlation. Hence there is an opportunity to exploit this correlation through schemes such as the one-level store of Atlas.

In the next two sections we examine cache memory and then virtual memory as we seek ways to reduce the memory bottleneck.

2.2 Cache Memory

2.2.1 Basic Cache Structure

Two years after the publication of the paper that described the Atlas one-level store, there appeared a brief article by Wilkes [1965] that describes an evolution of this idea to a different level of the memory hierarchy. Wilkes describes a system that contains two kinds of main memory. One kind is conventional; the other is a high-speed unconventional memory that Wilkes calls a *slave memory*. Present terminology calls such memories *cache memories*.

The idea of cache memories is similar to virtual memory in that some active portion of a low-speed memory is stored in duplicate in a higher-speed cache memory. When a memory request is generated, the request is first presented to the cache memory, and if the cache cannot respond, the request is then presented to main memory.

The difference between cache and virtual memory is a matter of implementation; the two notions are conceptually the same because they both rely on the correlation properties observed in sequences of address references.

Cache implementations are totally different from virtual memory implementations because of the speed requirements of cache. If we assume that cache memory has an access time of one machine cycle, then main memory typically has an access time anywhere from 4 to 20 times longer, not 500 times longer, which we cited previously for the delay due to page faults.

Earlier we defined a *page fault* to be a reference to a page in virtual memory that is not resident in main memory. The corresponding concept for cache memories is an access to an item that is not resident in cache, but is resident in main memory. This is called a *cache miss* to distinguish it from a page fault.

For cache misses, the fast memory is cache and the slow memory is main

memory. For page faults the fast memory is main memory, and the slow memory is auxiliary memory, the next level of the memory hierarchy. In many virtual-memory systems of the 1980s and 1990s, auxiliary memory is high-speed disk, but in the higher performance systems, auxiliary memory is itself a buffer memory for disk memory at the next level of the memory hierarchy. Regardless of the implementation of auxiliary memory, its access time is longer, possibly much longer, than the access time to main memory. Although misses are still rather costly for cache-based systems, they are not nearly as costly as page faults are, and we can afford to sustain cache misses more frequently than we can sustain page faults.

The time available for updating the status of a cache during a cache miss is minuscule compared to the time available during a page fault. Consequently, caches are controlled by hardware algorithms that can process cache misses automatically within the constraints dictated by the time available during a cache miss.

In the following material we describe in detail the operation of a cache and then consider practical cache designs. Then we examine efficient ways to use traces of programs to evaluate different designs.

Figure 2.5 shows the structure of a typical cache memory. Each reference to a cell in memory is presented to the cache. The cache searches its directory of address tags shown in the figure to see if the item is in the cache. If the item is not in the cache, a miss occurs. In the figure, the reference to address 01173 matches the tag 0117X, where the X designates any octal digit from 0 to 7. Since there is a match, the item sought is in the cache. The data associated with tag 0117X have addresses 01170 through 01177, so the access must be made to the fourth item, whose address is 01173. This datum, which has the value 30, is

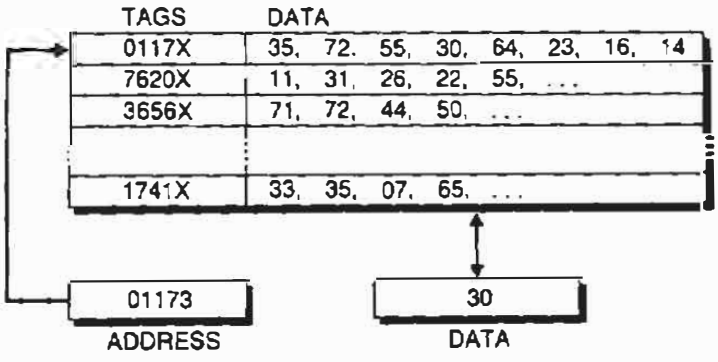

Fig. 2.5 A cache-memory reference. The tag 0117X matches address 01173, so the cache returns the item in the position $X = 3$ of the matched line.

copied to the data register of the cache. A reference to address 01163 produces a miss for the tags shown since no tag matches this address.

For READ operations that cause a cache miss, the item is retrieved from main memory and copied into the cache. During the short period available before the main-memory operation is complete, some other item in cache is removed from the cache to make room for the new item.

The cache-replacement decision is critical; a good replacement algorithm can yield somewhat higher performance than can a bad replacement algorithm. The effective cycle-time of a cache memory is the average of cache-memory cycle time and main-memory cycle time, where the probabilities in the averaging process are the probabilities of hits and misses.

If we consider only READ operations, then a formula for the average cycle-time is:

$$t_{eff} = t_{cache} + (1 - h)\, t_{main} \tag{2.1}$$

where h is the probability of a cache hit (sometimes called the *hit ratio*), and the time t_{cache} and t_{main} are the respective cycle times of cache and main memory. The quantity $(1 - h)$, which is the probability of a miss, is known as the *miss ratio*.

Equation (2.1) is a convenient tool for estimating performance changes due to cache. A very crude approximation for processing time for a particular workload is the sum of the time required to access memory. The change in the effective cycle time produced by using cache gives an estimate of performance improvement. This change has to be weighed against the cost of the cache. The user is interested in the ratio of cost to performance and is willing to pay extra if the performance achieved is worthwhile.

Cache is an efficient technique for improving the cost-performance ratio. For example, if main memory is 10 times slower than cache, then a decrease in the hit ratio from 0.99 to 0.98 (roughly 1 percent fewer hits) results in an increase in t_{eff} of roughly 10 percent. Thus, small changes in the hit ratio are amplified by the ratio of main-memory cycle time to cache-memory cycle time and the resulting average cycle time is very sensitive to small changes in the hit ratio.

A 10-percent decrease in the hit ratio from 0.99 to 0.89 almost doubles the effective cycle time and halves net performance when the cycle-time ratio is 10. If the cycle-time ratio is 20, that same 10-percent decrease in hit ratio increases the effective cycle time by more than a factor of 2.5. It is clear that we must have as high a hit ratio as possible, and that under many circumstances techniques that result in marginal improvements of the hit ratio, such as just 1 or 2 percent, may yield substantial performance improvement.

This leverage explains why cache is effective on the performance side of the ratio, but it is also effective on the cost side of the ratio. The cost of cache is a small fraction of the total system. The cost component of the cost-performance ratio is the total system cost, not the cost of cache alone. If a designer elects to

quadruple cache size in a new generation of machines, and the cost for the new cache is 50 percent higher than the cost of the prior cache, the system cost in the cost-performance ratio will be only 5 percent higher, not 50 percent higher, if the cost of cache was originally 10 percent of system cost and becomes 15 percent of system cost in the new design.

For our discussion in this chapter, we will use Eq. (2.1) as a rough measure of performance, and will measure the cost of cache by counting the number of bits in cache. Equation (2.1) is an exact measure of performance if the memory system is busy continuously satisfying one reference at a time generated by the processor. But, the formula is only an approximation to the true performance because

1. systems composed of independent random-access memories can satisfy more than one access at a time,
2. a processor need not generate a memory reference on every machine cycle, particularly if the processor is waiting for input/output, and
3. memory cycles can take longer than stated in the formula because some requests can arrive at memory while memory is busy honoring earlier processor requests or requests generated by the input/output system.

Hence, the model underlying Eq. (2.1) should be viewed as a useful way to estimate the impact of cache on performance, but a detailed performance model of a processor and its accompanying memory system is required for more accurate and more credible data.

Continuing with the discussion of the details of cache organization, in Fig. 2.5 we show an item in the cache surrounded by nearby items, all of which are moved into and out of the cache together. This group of cache data corresponds to the memory page for virtual-memory systems. For cache memories, we call such a group of data a *line* of the cache, although some papers refer to this group as a *block* of the cache. The smallest a line can possibly be is a single addressable item, which is anywhere from 1 byte to 4 bytes for the most popular computer systems. If items are as small as possible, however, then the cache directory becomes larger because there is a cache directory entry for each item in the cache. Doubling the size of a cache line while holding the number of bytes in the cache fixed reduces the size of the directory by a factor of 2 because two items in the same line share the same directory entry.

The cache in Fig. 2.5 requires the directory to behave *associatively;* that is, the cache directory retrieves information by key rather than by address. To determine if a candidate address is in the cache, the directory uses the tag bits from the candidate address as a key and compares this key to all tags now in the cache directory. To maintain high speed, this operation must be done as quickly as possible, which should be within one machine cycle.

A parallel memory that has the search capability just described is called an

associative memory. An associative memory, however, has a longer cycle than a random-access memory built from identical technology. This is strictly a consequence of the need to propagate signals through a larger number of gates in the associative memory than in a random-access memory of equal size. If we attempt to speed up the associative memory by adding more gates, the effect generally is to introduce additional delays that partially offset the gains attributable to the additional hardware. So, for practical reasons, the associative memory is less attractive than is an implementation that uses ordinary random-access memory technology.

2.2.2 Cache Design

Figure 2.6 shows a conceptual implementation of a cache memory. This system is called *set associative* because the cache is partitioned into distinct sets of lines, and each set contains a small fixed number of lines. The sets are represented by the rows in the figure. In this case, the cache has N sets, and each set contains four lines. When an access occurs to this cache, the cache controller does not search the entire cache looking for a match. Instead, the controller maps the address to a particular set of the cache and searches only that set for a match.

If the line is in the cache, it is guaranteed to be in the set that is searched.

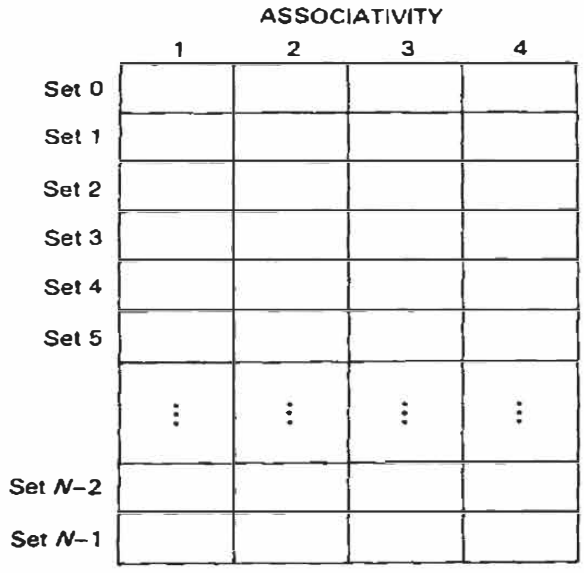

Fig. 2.6 The logical organization of a four-way set-associative cache.

Hence, if the line is not in that set, the line is not present in the cache, and the cache controller searches no further. Because the search is conducted over four lines, the cache is said to be *four-way set associative* or, equivalently, to have an associativity of four. For machines built at the beginning of the 1990s, the number of lines per set is as few as one and as many as 16.

The number of sets in commercial caches grows every year, and depends if the cache is on-chip with a processor where room is limited, or is off-chip where room is less constrained and the cache can be larger. A typical size for an on-chip cache in the late 1980s was 2K bytes arranged as 16 bytes by 64 sets by two-way set associative. A typical off-chip in the same era was 32K bytes arranged as 64-byte lines by 128 sets by four-way set associative. In the early 1990s cache sizes climbed to 16K on chip and 512K off chip, and the growth continues as memory technology supports increased density at lower cost per bit.

Figure 2.7 shows a physical implementation of a four-way set-associative cache. The implementation is organized around conventional random-access memories to take advantage of fast and effective lookup by means of address decoding. The mapping from requested address to cache set is a very simple operation on the reference address. The reference address, 03261 in the figure, is partitioned into two pieces, one piece called a *tag* consisting of the leading digits 0326, and the remaining portion, the digit 1 in this example, used as an address for the cache lookup.

The cache in Fig. 2.7 is composed of eight conventional random-access memories ganged together to operate as a single memory. The address tags are stored in the memory modules shown in the left-hand side of the figure. This region of the cache is often called the *cache directory*. The remainder of the cache modules hold the cache data lines. In typical caches, the number of bytes expended on an address tag is a small fraction of the total number of bytes of the line it identifies. An address tag may occupy two to four bytes, whereas typical line sizes run from 8 to 128 bytes.

Figure 2.7 shows a cache reference mapped to Set 1. Since there are N sets in the cache, the portion of the address that identifies the set to access must be able to take on any of N values. Hence, the single digit in Fig. 2.7 represents what is often implemented as a field of bits with a nominal size of 10 to 12 bits, but possibly has more or less than nominal. The four tags and four lines of data that comprise Set 1 are the contents of address 1 in each of the eight memory modules. Similarly, Set 17 is composed of the entries at address 17 in each of the modules. Each set is distributed across the eight memories rather than concentrated in a single memory. Since each memory can access at most one item in one cycle, by distributing each set across all modules, all of the components of the set can be accessed in a single cycle.

The cache access proceeds by reading simultaneously all four directory entries. Also, the data lines in the set are read concurrently so they will be available at the end of the READ cycle. If a cache line is larger in size than the size of

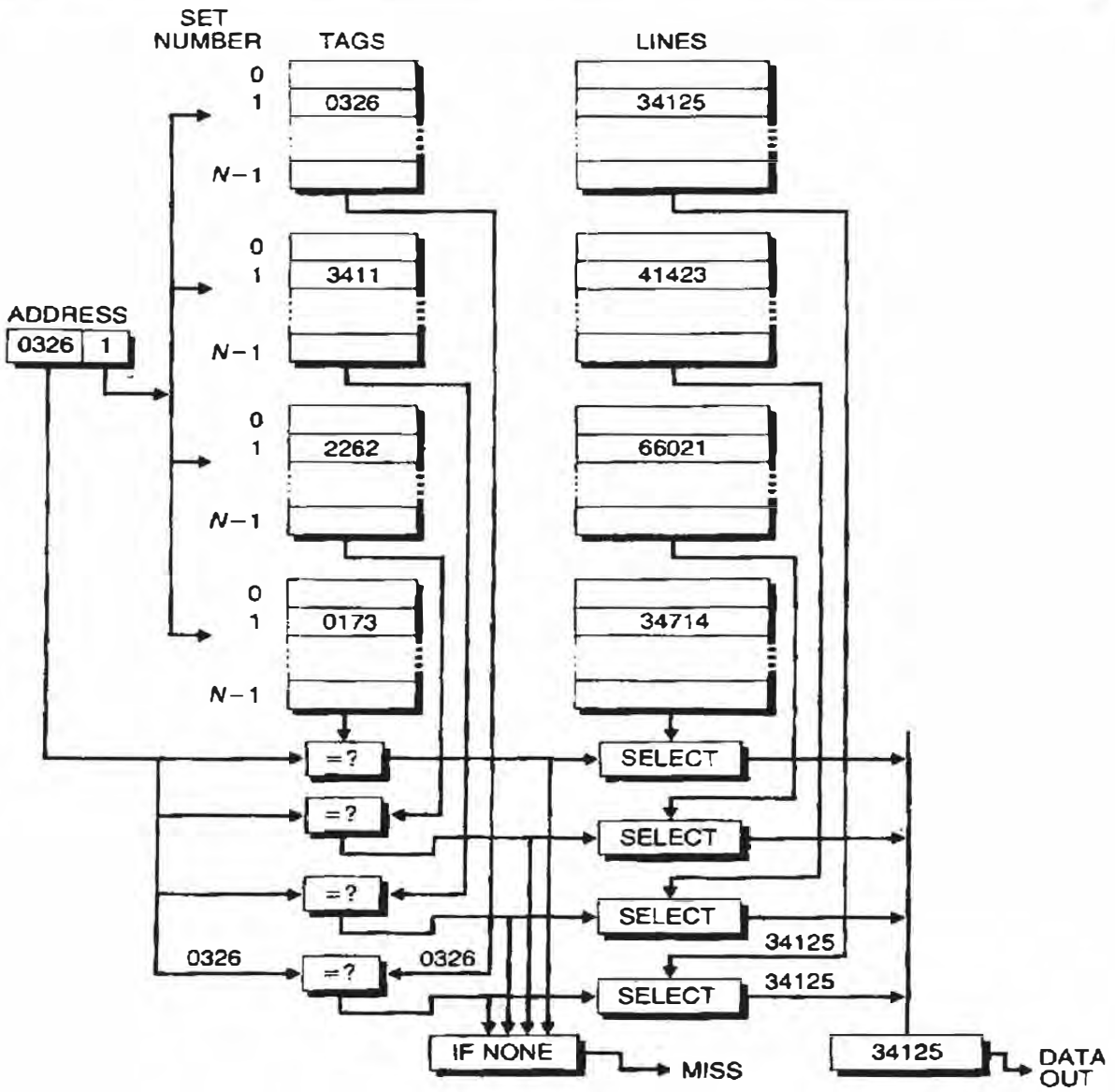

Fig. 2.7 An implementation of a four-way set-associative cache with N sets.

the data bus to the processor, then only the portions of a cache line that will be sent to the processor are read from the cache.

At the end of the READ operation, all four tags in the directory are compared to the tag of the address reference. If a match is found, then the corresponding data line of the cache is gated to the cache output-data buffer, and from there it is transmitted to the processor.

The timing of the cache activity is such that all reads from the memory occur as early as possible to allow maximum time for the comparison to take place. At least three of the four items accessed, and possibly all four, are discarded at the end of the cycle. Which line to use, if any, is decided late in the cache cycle, but at that time the data required have reached high-speed registers, so the data can be gated to the processor very quickly after the cache comparison-logic discovers a match.

Now let us reexamine Fig. 2.7 to see which parameters describe the cache design. This cache has:

1. L bytes per line;
2. K lines per set; and
3. N sets.

The total number of bytes in the cache is the product LKN. A cache in which the directory search covers all lines in the cache is said to be *fully associative*. In this case, $N = 1$, and the number of bytes in the cache is the product LK. For reasons mentioned earlier, fully associative caches are less attractive to build than are set-associative caches. The logic to compare two, four, or eight directory entries concurrently can be made sufficiently fast that the comparison and subsequent line selection can be completed without a significant impact on the machine cycle-time. But as the number of entries to compare increases to 16, 32, and above, cycle time starts to climb and the advantage of the larger set associativity is negated by the longer cycle time.

At the other end of the spectrum is the case for which $K = 1$, that is, the case in which there is only one line per set. Here, for any given candidate address, there is only one line in the cache that may contain the address reference. The cache in this case consists of one ordinary random-access memory with a simple comparator for the directory check. This special case is called *direct mapping* because address references map directly to a unique place in the cache.

There are several questions regarding cache design that are suggested by Fig. 2.7. The figure shows a possible mechanism for mapping address references into cache references.

Figure 2.8 provides more detail on this mapping. The address shown in this figure is a physical address M bits long that will be sent to main memory if the item is not in the cache. In this case, we assume that each byte in memory has

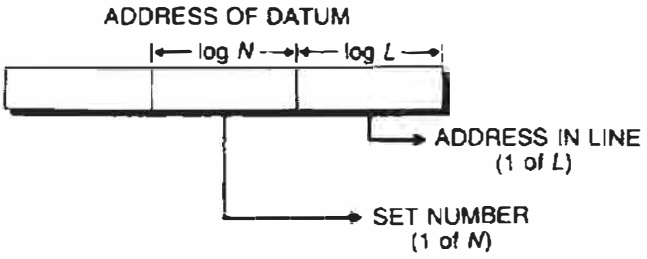

Fig. 2.8 Address partitioning for a cache search.

a unique address. Since all L bytes within one line are present or absent as a group, to determine if that line is present, we must strip off the least significant $\log_2 L$ bits of the address to prepare to interrogate the cache. The remaining address bits are common to all members of a single cache line, and these are the bits that we must check when looking for a hit.

The next facet of the mapping operation is to determine which of N sets to interrogate. We must find some way of mapping the remaining address bits into $\log_2 N$ bits, which are then used to select among N different sets in the cache directory. The method used most frequently is to use the least significant $\log_2 N$ bits of the remaining address bits, which has the effect of scattering lines with successive addresses to successive sets of the cache. This tends to randomize address references through the cache and reduce clustering by mapping contiguous active regions in main memory across many sets of the cache, thereby making for the best use of the cache.

Figure 2.8 shows the low-order $\log_2 L$ bits of the address reference being stripped away to account for the number of bytes per line, and then shows the next low-order $\log_2 N$ bits being used as the address for access to a conventional random-access memory. From this memory, we read all K tags simultaneously. Also, the required data from the lines in the cache are read from random-access memories that hold the cache data.

The latter memories use the $\log_2 N$ bits together with some of the $\log_2 L$ bits to access specific regions within a line in case the processor cannot accept the entire cache line. Portions of all K lines in a set are accessed, and the greater the number of bits from the L field used to address the lines, the smaller will be the size of the data fields read from memory.

In making the directory comparison, note that it is necessary to store only the leading bits of the address reference, $M - \log_2 N - \log_2 L$ bits in this case. All lines stored in a set have the same values for the set number, so it is not necessary to store the $\log_2 N$ bits that identify the set number.

In Fig. 2.7, the set number 1 is stripped from the address 03261 to create a

tag of 0326 for comparisons. The bottom tag shown in the same set has a value 0173, so the corresponding memory address is 01731.

The parameters mentioned thus far give us at least three degrees of freedom in designing a cache, and there are more choices yet to be discussed. Let us reflect a moment on the choices at hand to see what trade-offs are available and what guidance we have to complete our design.

Figure 2.9 shows the general form of the curves that describe cache behavior as a function of some single parameter choice. The x-axis plots caches of increasing size, but the curve does not indicate the structure of the successively larger caches. For the moment it is not important which of the parameters L, K, or N increases in this figure. The y-axis plots the relative number of misses against cache size, with the number of misses for a cache of size 0 normalized to unity. Note that the curve drops sharply at first and then bends and drops less steeply as the cache size increases.

Most of the improvement in this graph is obtained by the initial small changes in X. As X increases beyond the knee of the curve, relatively little additional benefit is obtained. Hence, a good design point is a value of X around the knee of the curve.

One problem that cache designers face is that the data available are not nearly as clean as the data in Fig. 2.9. The data are often at best sketchy and are highly dependent on the method in which they were gathered. So the designer has to make critical choices using a combination of hunches, skill, and experience to supplement the meager information at hand.

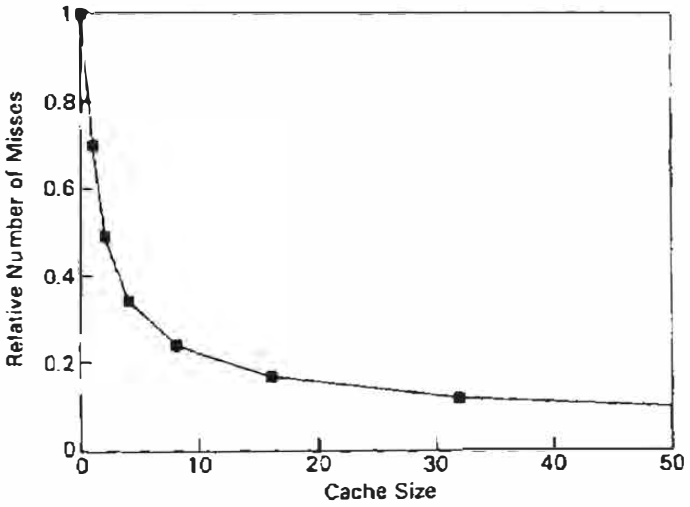

Fig. 2.9 Cache performance—the number of misses versus cache size.

Good engineering sometimes requires a small degree of overdesign and inefficiency to protect against unusual cases. For the data in Fig. 2.9, the smallest cache that operates well is a cache of size 8 (relative misses = 0.24) or possibly size 16 (relative misses = 0.17). A cache of size 32 (relative misses = 0.11) would be difficult to justify because the performance change is small considering that the cost of cache is doubled to obtain this change. Nevertheless, a reasonable cache design may well incorporate a cache of size 32 or even size 64 as an intentional strategy to assure a low number of misses over a wide range of workloads. In this case, the designer is protecting against workloads whose characteristics are quite different from those in Fig. 2.9.

How can we be sure that all workloads are accurately modeled by Fig. 2.9? In fact, we cannot. Some workloads might have a much shallower slope and exhibit a knee at larger values of cache size. If we design a cache that just barely runs a workload of the type characterized in Fig. 2.9 at an acceptable performance, that cache may deliver unacceptably poor performance for more stringent workloads. If we overdesign for the workload in Fig. 2.9, we can still run acceptably well on some workloads that demand larger caches.

The designer has to obtain the most useful performance data possible and then use good judgment to estimate the characteristics of other important types of workloads that are not reflected in the data available to decide how much the architecture should be overdesigned. The idea is to examine the cost of the excess capacity against the possibility that the capacity will be necessary and beneficial. Decisions of this type are usually driven by cost considerations because the cost has a major impact on the competitive marketing of the machine, whereas the value of the excess capacity is more difficult to assess if it does not contribute identifiably to higher performance on normal workloads.

Returning to the problem of cache design, how can we develop data that will enable us to select a cache size as well as the values of the cache-structure parameters K, L, and N? To answer this question, we need to develop data, as shown in Fig. 2.9, that plot cache misses against cache size.

Because a cache of a given size may be organized in various ways through different choices for K, L, and N, we suggest that K and L be fixed and N varied when this study is conducted. That is, the set associativity and the line size should be fixed while the number of sets is varied. What is typically observed is that the number of misses decreases, as shown in Fig. 2.9, and the knee of the curve will be at a point that is dependent on the particular processor and the workload.

Extensive data on the subject has appeared in the literature. Specifically, A. Smith [1982, 1985, 1987] has excellent collections of typical results. Empirical observations of typical programs turned up a simple rule of thumb: each doubling of the size of the cache reduces misses by roughly 30 percent. Figure 2.9 shows this characteristic and demonstrates what is often observed in real systems.

The 30-percent rule is useful for rough estimates, but should not be used when accurate data are needed. Specific programs and processors do not obey this rule. The reader should establish a similar formula when designing a cache for a particular architecture and workload and use the new formula to evaluate various cache designs.

Given a total size of cache, how should the cache be organized? We recommend choosing line size L next and then the set associativity factor K, although they could be chosen in the opposite order. To find the cache performance as a function of line size, fix the parameter K to a value that is likely to be its final value.

From experience with other cache designs, we intuitively know that the set associativity will be a small number, and it will probably not be 1. So we set $K = 2$ and examine cache behavior as a function of line size. Note that we have fixed the total size of cache, so that as line size doubles, we must reduce N, the number of sets, by a factor of 2 to keep the product LKN constant.

The best performance is obtained with $L = 1$ because each individual address is cached independently. But in this case, the directory may be enormous and rather costly. When L is maximum, $N = 1$, and there is but a single set in the cache. This has the worst performance, but it is the least expensive to build.

By plotting L along the x-axis in Fig. 2.9, with misses on the y-axis, we obtain the knee of the curve for some value of L. Note that to obtain the shape of the curve shown in the figure, fix N and let L increase. Then produce a family of curves, each for a different value of N.

Since the size of the directory depends on the value of L, the selected value of L may be very small and require too large a directory to be practical. Consequently, for a fixed cache size, it may be necessary to increase L while reducing N to obtain a practical directory size. The cost of this change is greater bus traffic per miss.

The final step is to choose the set-associativity factor K. This too can be accomplished by plotting a curve similar to the one in Fig. 2.9. In this case we perform cache analyses that hold the line size L and number of sets N fixed while varying set associativity K. The resulting curve should have the shape of the curve in Fig. 2.9 when K is plotted along the x-axis, increasing to the right, and with misses plotted on the y-axis.

If the study suggests that a better choice for K is 8 instead of 2, then we should restudy the effect of line size on performance, but use the new value of K in place of $K = 2$.

Eventually we can find a collection of values for K, L, and N that represent a satisfactory trade-off between cost and performance. Generally speaking, we obtain better performance as we increase the absolute size of cache. We estimate that performance by estimating the average memory-cycle time

$$t_{\text{eff}} = t_{\text{cache}} + (1 - h)\, t_{\text{main}} \tag{2.2}$$

where h is the hit-ratio for the given cache. Then we factor in the effect of technology. How much does that extra performance cost? If we are willing to pay for the performance, then we use larger caches. If the cost is very high—too high for the performance gained—then we use a smaller cache.

The cache-parameter values used in commercial systems have tended to increase in time as technology has made it possible to build larger caches at reasonable expense. High-performance minicomputers were produced with caches as small as 2 K-bytes at the start of the 1970s and moved to larger caches that reached 8 K-bytes in one decade and 256 K-bytes in two decades. In that same timeframe, cache memories for high-end machines evolved from 16 K-bytes to 64 K-bytes to 1 M-bytes.

Although we expect the trend toward larger caches to continue, it is certainly not clear that they will increase in size in the future at the same rate as they have increased in the past. As main memory capacity increases from 10^7 bytes to 10^8 and 10^9 bytes, there is a strong possibility that cache memory need not grow linearly with main memory. Instead, it may grow as some slowly growing function that reflects the growth of the active areas of memory as a fraction of the total size of memory. In fact, several manufacturers such as Amdahl and Hitachi have produced machines with two levels of cache memory, with the first level very fast, very expensive, and relatively small, and the second-level cache much less expensive, but still costly compared to main memory.

The second-level cache may be the architectural feature that grows larger with new generations of device technology. The first-level cache captures most of the hits. As cache size grows and the performance curve bends around a knee, the additional hits obtained are rather infrequent. These should be fielded in the second-level cache, whose cost is relatively low compared to first-level cache, but whose performance is much better than main memory.

Two levels of cache further complicate the design picture. Now we have to consider three different memory costs and two different cache structures. The design possibilities are very rich, but rather overwhelming in their number, making thorough analysis of alternative designs very costly to perform. The next two sections treat the efficient use of address traces for exploring the design space.

2.2.3 Cache Analysis: Trace Generation and Trace Length

In the previous section we glibly assumed that the reader can construct curves such as those in Fig. 2.9 from data on hand. This is hardly the case. Cache-analysis input data usually consist of extremely lengthy address-reference sequences obtained through great effort. The fastest way to obtain such information is through special hardware attached to an operational machine. The special hardware monitors memory requests and logs each individual reference on a tape for later use by a cache-evaluation program.

Although this method is very fast, if the operational machine happens to be a very high-performance machine, then the specialized hardware must run several times faster than the high-performance machine to keep up with it. Such hardware monitors are costly, and are difficult or impossible to build for the fastest machines. They are quite useful, however, for studies involving slower machines.

By far the most popular means for generating an address-reference stream for studying cache performance is the machine simulator. This is a program that simulates the instruction execution of a computer under study. The input to the simulator is a typical workload. As each instruction is executed by the simulator, the simulator writes to an external file the sequence of address references generated during the simulation.

Some processor architectures have the capability of trapping to the operating system after the execution of each instruction. On such architectures, simulation can be done very efficiently because each instruction is executed at full machine speed. Software in the operating system is required only to determine what addresses were generated by the instruction and to transmit these addresses to the output file.

Since we presume that cache design is to be done by examining the performance of various design alternatives on address traces, we have to be sure that the address trace is representative and does not have particular biases that could produce misleading evaluations. Actually there are three distinct problems.

1. The workload on the trace may not be representative of the actual workload for which the machine is to be used;
2. The initialization transient during which the cache is filled with relevant data may grossly affect the evaluation; and
3. The trace may be too short to obtain an accurate measure of the miss ratio.

The first problem is particularly nasty. Because simulations run 10,000 to 1,000,000 times slower than real time, it is not feasible to create traces that cover long periods of real time. Typical simulations cover hundreds of milliseconds at most, which raises a question about the fidelity with which the simulation captures the characteristics of the workload.

High-performance machines specialized to particular applications typically spend the bulk of their time in predictable ways, which should account for the majority of an address trace for such machines. But a representative fraction of the address trace should also be devoted to other activities, such as input/output and loop initialization.

General-purpose machines present a much more difficult problem because their workloads are not easily characterized. Moreover, any user may choose to dedicate a computer to an unusual function whose characteristics are vastly different from normal uses of the same computer. So the cache designer can at

best evaluate a cache for some good estimate of the workload. Individual users may experience performance that deviates from the expected performance if their workload has dramatically different characteristics from the workload used for cache design.

Various models of address references have been postulated with the idea that the models can be used in place of traces for cache evaluation. Trivedi [1982, pp. 305–308], for example, describes a statistical model that captures the notion of locality of references in a way that is useful for designing an operating system for a virtual-memory system. This model in turn is a refinement of the models of Coffman and Denning [1973, pp. 275–278]; Denning, Savage, and Spirn [1972]; Shemer and Gupta [1969]; and Shemer and Shippey [1966]. Later in this section we describe a useful model that captures the 30-percent rule depicted in Fig. 2.9, and is due to Thiebaut [1989] with refinements by Singh, Stone, and Thiebaut [1992].

Such models give insight into the characteristics of address-reference strings, but because cache design is so critical and so much is at stake, cache designs have to be validated by testing them on the address references produced by a real workload when this is possible. So the use of actual address traces for evaluating cache designs will continue to be the primary tool for cache analysis. For large caches, the length of traces required to yield accurate data becomes prohibitive, and models of program behavior become an acceptable alternative. When designing caches for machines with new instruction sets, workloads do not exist, so that a combination of analytical models and data from caches for similar architectures may be required.

Apart from the fidelity of a trace in capturing workload behavior, there are two problems related to the accuracy of the results obtained for a particular trace. The first problem is that data are corrupted by an initialization transient, but fortunately the corrupted data can be removed as indicated below. The second problem is more serious. The trace has to be long enough to capture enough misses to produce an accurate measure of the miss ratio. Unfortunately, the trace length required grows roughly as the cache size raised to the 1.5 power. For each quadrupling of cache size, the trace length increases by roughly a factor of 8. For 2 M-byte caches, trace lengths required for moderate precision can exceed 100 million references.

When miss ratios are so small that extremely long traces are required to measure them, they are also relatively unimportant factors in the overall performance model of a machine, and other factors tend to be more important. The other factors may be cache related, and may include the bus traffic generated by processor WRITEs to the cache, as well as traffic to and from the cache due to input/output and interaction with other processors in a multiprocessor system. These other factors are often measured by trace simulation in much the same way that hit ratios are measured, but the traces need not be as long because the

events of interest occur more frequently. In any case, the cache designer can rely heavily on detailed trace information in order to evaluate performance as a function of cache size and structure, but the designer should rely on the techniques that follow to remove bias in the data and to determine what confidence should be placed on the interpretation of the data.

The problem of cache initialization is that during a cache simulation, the first reference to each line in the cache will generate a miss, whereas in a real environment the corresponding reference may have been a hit because in the real environment the cache would have been holding a recently used item. The cache simulator cannot preload the environment so that it cannot generate a hit where the actual cache it simulates produces a hit.

One may argue that the beginning of a cache simulation is something like the behavior of the real computer at a context swap. In that situation a new process takes control and generates cache misses until it loads itself into the cache. In this way the cache simulator captures not only the steady-state effect but the cache-reload transient as well. The problem is that the two effects are combined in a fashion determined by the total length of the trace. This is rather arbitrary, and may not reflect the true ratio of transient reload effects to steady-state effects. Moreover, when caches of different sizes are simulated with a trace of fixed length, the larger caches use a larger portion of the trace to initialize the state of the cache, and thus the transient contributes a greater proportion of the simulated miss ratio for the larger caches than for the smaller caches. To the extent that the transient effect is a distortion of reality, the larger caches suffer a larger distortion in their simulated data. As we shall see in this section, the distortion is so bad on large caches that most published data on simulated miss ratios of large caches report effects almost entirely due to the initialization transient and the steady-state miss ratio is totally lost in the transient.

We recommend an approach that measures the steady-state miss ratio in isolation, and then factors in the reload transient by a means described later in this chapter. To do so, it is essential to remove the bias introduced by misses attributed to the cache initialization. Such misses are quite easy to recognize and can be factored out with a minimum of effort.

The cache-simulator program can easily be modified to record which misses are true misses and which are artifacts of the initial state of the cache. Simply initialize the cache with address tags that are illegal. Since address tags of a physical cache are several bits shorter than a full address, and since a cache-simulator program can manipulate data as wide as a full address, we can write the cache simulator to store address tags that are wider than actual tags. Consequently, we can initialize the values of address tags to some illegal value, for example, by setting the sign bit, if we know that while simulating the cache no valid tag can be generated with a sign bit set.

The value of initializing tags in this way is that we can examine the address

tag of each line that is removed from the cache. If the tag is invalid, then the line removed is one that was placed there during cache initialization, and in a real environment it might have produced a hit.

Now that we can recognize a miss due to cache initialization, what action should we take? Focus attention on any one of the N sets in the cache. The addresses that can be stored in this set are disjoint from the remainder of the addresses. In essence, this set is a small cache that operates on $1/N$th of the address space of the computer. If the cache is direct mapped ($K = 1$), the cache contains one entry. The first reference to this set produces a miss due to initialization. If we simply ignore the first reference, then this set is properly initialized for subsequent simulation. That is, we do not count the first reference, a miss, to a set. The trace length is effectively shortened by one address per set referenced. (If our intent is to measure the transient together with the steady-state miss ratio, then we should record both the reference and the miss.)

Thus when simulating a direct mapped cache with N sets, when the initialization transient is factored out, the trace length is reduced by N references, and N misses are removed (assuming that all N sets are touched during the trace). The impact on total trace length is negligible, but the miss ratio may be greatly affected, depending on how many additional misses are on the remainder of the trace.

As an example of how initialization misses can dominate steady-state misses, we draw data from a study by J. E. Smith and J. R. Goodman [1985]. Smith and Goodman study a model in which the cache-reload transient is present and can impact performance. Their goal is to measure the impact of that transient, and their experiment indeed produced a large reload transient. For example, their data includes a table entry with a hit ratio of 0.994 for an 8 K-byte direct-mapped cache with a line size of 16. The trace length used for this simulation is 100,000. This cache has 512 lines and there are 600 misses recorded for this trace. Although we do not know how many of these 600 misses are due to cache-initialization misses, at one extreme all 512 lines in the cache could have suffered one initialization miss. In this extreme case, the miss ratio reported would be produced by 512 initialization misses and 88 steady-state misses. Their hit-ratio data for 2K-line cache structures with 4-byte line size approaches the asymptote of 0.985, which is about 1500 misses per 100,000 references or about 0.75 misses per line, which is not enough to initialize the full cache. If the cache simulations did not specifically remove the initialization misses, then the reported data on the large caches are due largely to the initialization of the cache rather than to the steady-state misses. For the smaller caches, the initialization effects are much less dominant in the data. It is clear that for their parameters the treatment of initialization data can make very large changes in the relative size of the measured miss ratio, and this is consistent with their conclusions regarding the effect of the cache-reload transient. Had their traces been longer, their miss ratios for the

8 K-byte caches would be lower to the extent that the initialization effect becomes small relative to the remainder of the trace.

We have argued that the initialization of direct-mapped caches can be treated very simply with minimal shortening of the trace lengths. Set-associative caches can be treated equally easily, but the effect is to shorten the trace length more dramatically than for direct-mapped caches. When creating traces, the shortening phenomenon caused by cache initialization has to be considered, and thus requires the trace lengths to be somewhat longer than the length estimates produced by statistical considerations. For set-associative caches, the rule is to treat each of the N sets independently. After K misses are recorded to one set of a K-way associative cache, the statistics for that set can be recorded. Until that point, no statistics on that set are kept. The reason for this rule is that until the set has been fully initialized, the set is acting as if it were smaller than its true size. For example, after one miss is recorded in a four-way set, one line holds a valid datum and the other three hold uninitialized data. At this point in the simulation, the effect of having only one initialized entry in the set is the same as if the set were one-way associative. It experiences a miss somewhat sooner than expected for a four-way cache. If the references were recorded at this point for that cache, the next three misses to the set produce a higher than average miss ratio and influence the final data of the simulation. Consequently, all references to that set should be ignored until the set is fully initialized, and then the recording can begin for that set. This notion first appeared in Laha, Patel, and Iyer [1988] where they introduce the term *primed set* for a set that has its initial contents purged.

A variety of techniques for dealing with the initialization transient have been reported in the literature. Some researchers "warm" the cache by running a fixed number of references through it. This technique works to the extent that it removes the transient. However, it may not remove all of the transient, or even may remove very little, depending on the size and structure of the cache. Moreover, if the length of the trace for initialization is fixed for all caches rather than dependent on the cache structure, then as a variety of structures are simulated, the transient effects will contribute differently to different cache structures, and the data obtained will not be comparable across the various caches. To be absolutely certain that the transient is absent and to use the fewest possible references for initialization, we recommend beginning a simulation on each distinct set when the set is fully initialized.

The next question is how long should a trace be? We cannot give a precise answer because the statistical behavior of cache misses is not well understood. But we can model the cache misses by a different, but understood process that enables us to obtain a length estimate. This length estimate is a lower bound on what is required for the cache-miss process. At this writing, we do not know how much longer than the lower bound the traces should be. The lower bound

is very long for large caches, which casts doubt on the practicality of producing statistically accurate measures of miss ratios of large caches.

To obtain a bound on trace length, we assume that the cache-miss process is a Bernoulli process. That is, each address reference has a probability h of being a hit and $m = 1 - h$ of being a miss. Each reference is generated independently in time. This is equivalent to flipping a coin whose probability of coming up tails is m. We know that the independence assumption is false for the process that produces cache misses. Misses tend to cluster in time and occur in bursts rather than as predicted by a Bernoulli process. To measure the average of clustered references over long periods of time requires more observations than the measurement of the average of statistically independent references. In the following paragraph we use the independence assumption to produce trace lengths, but we recognize that the lengths produced are only lower bounds on the actual lengths required.

Let a trace contain T references. Then the mean number of misses, M, is $m\,T$ and the variance in M is $m\,T\,h = M\,h$. The estimate of the miss ratio is M/T and the variance in the estimate is $Mh/T^2 = mh/T$. This is true because given a random variable X with variance V, the variance of the random variable cX, for constant c, is c^2V. The true miss ratio of the underlying process is not necessarily equal to the estimate, but it lies close to the estimate. In fact, the standard deviation, which is the square root of the variance, gives us an estimate of how far away the true mean lies from the observed mean. For Bernoulli processes, we know with over 95 percent confidence that the true mean lies within two standard deviations of the observed mean [*cf.*, Brunk, 1960]. To obtain the 95 percent confidence interval for small miss ratios, we can safely approximate the hit ratio h by 1. This yields

$$\text{True Mean Miss-Ratio} = m \pm 2\sqrt{\frac{m}{T}} \tag{2.3}$$

Figure 2.10 shows typical hypothetical confidence intervals based on Eq. (2.3) for a trace of fixed length starting with fully initialized caches. The central curve is the 30-percent rule plotted in Fig. 2.9. The trace length is assumed to be 10 million references after all caches are fully initialized. The caches are 4-way set-associative and the line size is 64. The number of T references in the standard deviation term is the number of references per set, which is the trace length divided by the number of sets. The large confidence interval around this curve is the 95 percent confidence interval for which Eq. (2.3) holds. The inner interval is a plot of the 90 percent confidence interval for which the multiplier in Eq. (2.3) is reduced from 2 to approximately 1.65. Note that we have greater confidence when we bound the true miss ratio to within a larger region. The curves plotted are for the Bernoulli model, and the actual confidence intervals will

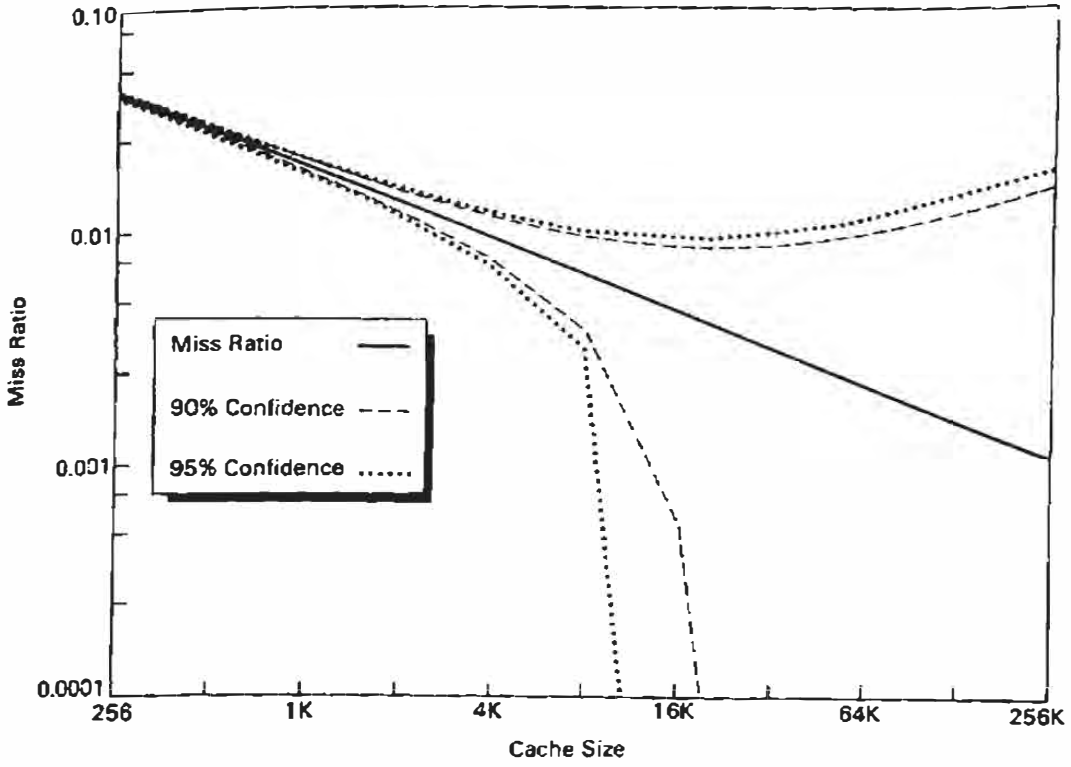

Fig. 2.10 Confidence intervals for miss-rate measures for the Bernoulli model. This does not include initialization effects.

probably be much greater than those shown when statistical dependencies are taken into account.

As a numerical example, let the miss ratio be on the order of 1 percent, and find the size of T that produces a confidence interval of 20 percent of the miss ratio. The standard deviation term is $2\sqrt{m/T}$ and we require this to be less than 0.002. Equivalently, we require T to be greater than 10,000 for $m = 0.01$. Note that we bound the relative size of the confidence interval about the mean by forcing the standard deviation to be a fixed percentage of the mean, say β times the mean. To achieve this bound we increase the length of the trace T until twice the standard deviation decreases to that percentage of the mean. For a specified β, the confidence interval is within two standard deviations of the measured mean when $2 \leq \beta\sqrt{(mT)}$. Since this inequality involves the product of m and T, for a fixed value of β, T grows inversely with m. As the hit ratio falls away from unity, the approximation used here for the variance becomes inaccurate,

and it becomes necessary to use the factor h in the variance expression in order to improve accuracy. In terms of M, the number of misses observed, a little manipulation of the formulas shows that $M \geq 4/\beta^2$. To double the relative precision of an estimate by reducing β by a factor of 2, the number of misses observed must quadruple.

This calculation suggests that by observing as few as 10,000 references, the calculated values of miss ratios lie within 20 percent of the actual values. This assumes all of the references lie in a single set. What happens if those references are distributed across N sets? For the Bernoulli process, the variance per set increases by a factor of N when the number of observations per set is reduced by a factor of N, but the average of N independent observations across the N different sets decreases the variance by a factor of N, so that there is no net change in the size of the confidence interval.

Unfortunately, the cache-miss process has highly clustered references as reported by Voldman *et al.* [1983]. The activity across N sets has very high correlation, and thus the distribution of references across N sets does not reduce the variance in the estimated variance as predicted. When one set experiences misses, many sets experience misses. When a reference stream produces a relatively long miss-free period, all of the sets see a proportion of that long miss-free period. If the correlation across sets is near unity, then observations of the activity in N sets produces very little additional accuracy than the observation of a single set.

More specifically, assume that the variance in the number of misses per set is V, for each set, and that the correlation coefficient between any pair of sets is ρ. Then for N sets, the variance in the average number of misses per set is equal to $(V/N)(1 + (N - 1)\rho)$. When the correlation coefficient is zero, N observations reduce the variance by a factor of N and the standard deviation by a factor of $\sqrt{N}$. When the correlation coefficient is unity, the variance in the observed mean for N observations is the same as the variance in a single observation, so that the additional observations do not reduce the confidence interval.

Figure 2.11 illustrates the potential reduction for a drop in confidence interval. Figure 2.11(a) shows a data point surrounded by its confidence interval based on 10 independent observations. The interval size happens to be 32 percent of the size of the variable, and thus we have the value of the variable pinned within a precision of approximately plus or minus 16 percent. Figure 2.11(b) shows what happens when we increase the number of observations to 160 independent observations. Since the number of observations has increased by a factor of 16, the confidence interval has been reduced by a factor of 4. We now have a bound on the variable to within plus or minus 4 percent.

If the additional observations are not independent of the original variables, the bound does not decrease and the effort to obtain the additional variables is lost. Polls prior to presidential elections take this into consideration when seeking

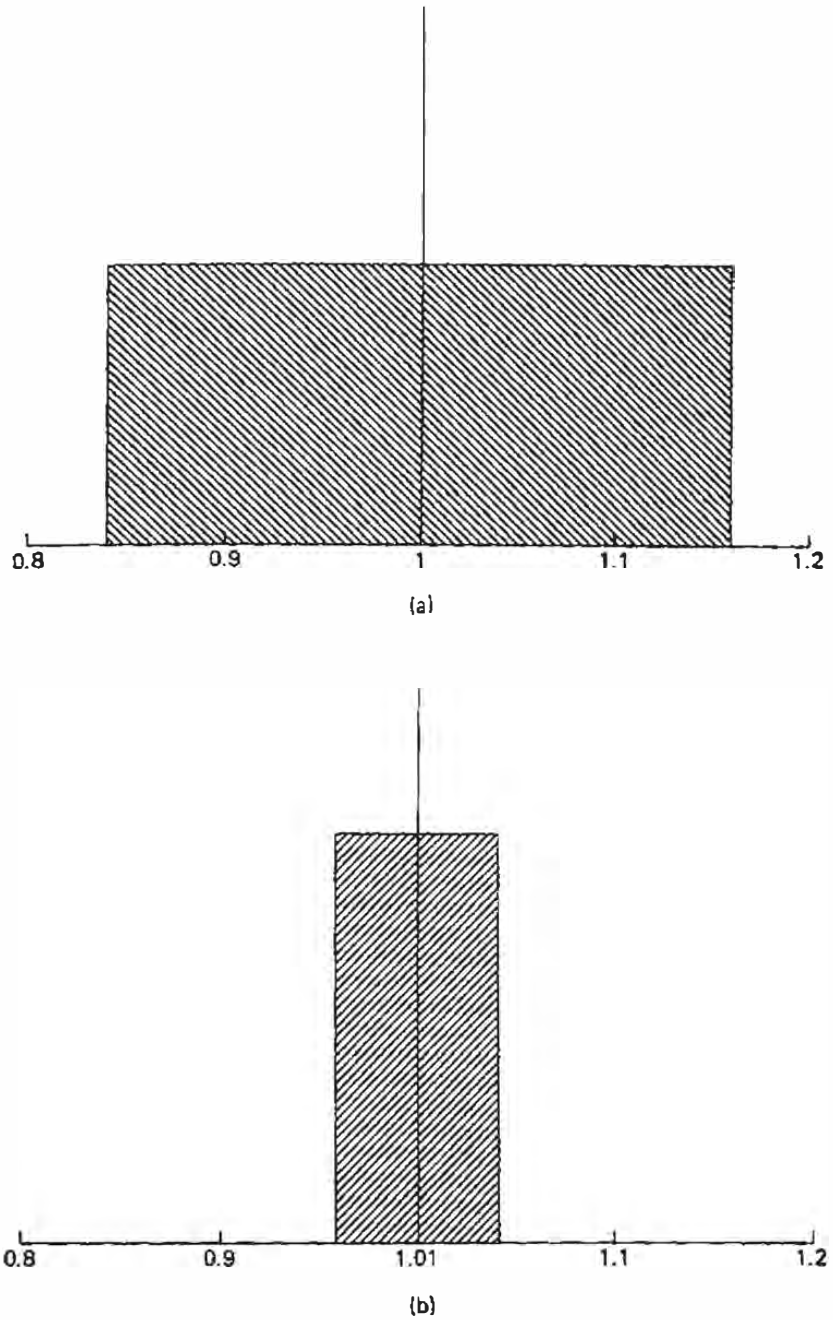

Fig. 2.11 Confidence intervals for
(a) 10 samples of an event, and
(b) 160 samples of an event.

opinions of voters. Pollsters seek samples across a representative slice of voting population. Geographic location tends to be a correlation factor, and thus polling many residents who live in the same community is less effective in reducing the confidence interval of the sample than is obtaining the same number of data points spread out geographically in some proportion to the voting population where the samples are taken.

Because of the potential for high correlation of activity among different sets, it is essential to record enough misses per set to obtain reasonably tight confidence intervals for one set rather than rely on the averaging of the misses across many sets to reduce the confidence interval to acceptable accuracy. Ultimately we are attempting to measure the average length of an interval between two events. We must record at least two events per set to obtain a rate of occurrence, and ideally we should have many events to obtain an accurate measure of that rate.

When caches are sufficiently large, the miss ratios may drop to a region where high relative accuracy is not important. At miss ratios of 0.005 or below, we may be satisfied with a relative accuracy of ±100 percent. The Bernoulli bound requires only four misses per set in this case, or about a factor of 25 less than is required for a confidence interval five times tighter. Traces too short to produce at least four misses per set yield confidence intervals on a per set basis that are larger than the observed mean. Because of the clustering in the cache-miss process, we do not know what the overall confidence interval on the observed mean may actually be, but when the misses per set drop below four, there may be a problem in the precision of the answers obtained. The designer must be cautious in using the results of the trace data, and should seek independent means of estimating or measuring the miss ratios of a prospective design to confirm the results of trace simulation.

To give some idea of the potential lack of precision of cache measurements, recall the data reported by Smith and Goodman [1985] mentioned earlier. The number of misses observed for the cache with 512 lines is on the order of 600 misses, or just over 1 miss per set out of roughly 200 references per set. For one set, the Bernoulli bound on the 95 percent confidence interval is 0.006 ± 0.011. For the caches with 2K lines, the number of misses observed were fewer than 1500, or about 0.75 misses per set out of roughly 50 references per set. The Bernoulli bound on the confidence interval for a single set of this collection of caches is 0.015 ± 0.0346. For both the 512-line and 2K-line caches the uncertainty in the size of the mean in one direction is about twice the size of the mean. Also, the interval includes negative miss ratios, which is not meaningful. By taking the measurements over many sets, the confidence interval of the overall mean is smaller than that of a single set, but the true size of the interval is unknown because of the high correlation of the activity of the sets. For these caches, Smith and Goodman appear to have too few misses per set to produce

trustworthy measures of miss ratios. To be sure of the data, it is necessary to repeat the experiments with longer traces to obtain tighter confidence intervals. The bounds on the confidence intervals for the smaller caches studied by Smith and Goodman are tighter and useful, so that the uncertainty in their data shows up mainly in the extreme points in their study.

Given this information we can develop some estimates for trace lengths required to evaluate caches. Let's start with a design point of a cache of size 32 K-bytes, with four-way set associativity, and 16-byte lines. This cache has 512 sets. Assuming a nominal miss ratio of 1 percent, to achieve 4 misses per set, we require 400 hits per set or about 200,000 references after initialization. For higher precision, a trace length of 5 million references produces 100 misses per set, which should yield satisfactory accuracy. A cache four times as big has roughly half the miss ratio and four times the number of sets. To achieve the same number of misses per set for this cache, the length of the trace must increase by a factor of 8, or, equivalently, the trace length grows roughly as the cache size raised to the 1.5 power under these assumptions. Hence, by this approximation, to obtain only 4 misses per set, we need a trace of 1.6 M references for a 128 K-byte cache, 12.8 M for a 512 K-byte cache, and 102 M for a 2 M-byte cache after initialization. If we insist on 100 misses per set, the trace for the 2 M-byte cache reaches a length of 2.5 billion, which is prohibitively large. These are all nominal estimates and must be calculated more carefully when doing cache designs by using miss-ratio data for the architecture or for related machines. What may be devastating is the effect of initialization misses. For four-way set associative caches we need 4 misses per set before we record data, and we need to simulate long enough to record at least another 4 misses per set. For direct mapped caches, the additional trace length required is negligible. For four-way set associative caches, the additional trace length for initialization is considerable and may be from 25 percent to 50 percent of the trace length needed for simulation. Note that in this case both the initialization and the simulation portions of the trace produce four misses per set, but the initialization trace is shorter because it has a higher miss ratio.

Laha, Patel, and Iyer [1988] report a successful technique for higher miss ratios in which they average 35 trace samples taken at different points in time. Each sample is large enough to initialize all sets, and still have a sufficient number of misses per set to obtain meaningful information. The effect of using 35 different samples is to obtain a better estimate of the overall miss ratio than can be obtained by a long continuous observation. Although this scheme is useful for larger miss ratios, at very low miss ratios the number of references used for initialization of the sets becomes excessive. The cost of 34 additional initializations negates some or all of the gain in shortening the trace by sampling in time. The only foolproof way to obtain accurate estimates of the rate of rare events is to observe many of them.

This analysis suggests that very long traces are required to analyze the behavior of large caches, and traces of such lengths are much larger than lengths actually used in the literature. Can we trust the data in the literature? An interesting study by Agarwal, Hennessy, and Horowitz [1989] produces a model of cache behavior that matches closely with cache simulation data. The largest caches studied in this paper are of size 256 K-bytes. These are large enough to require longer traces than used by Agarwal *et al.* in their study, and thus the question of the accuracy of the measurements arises.

Figure 2.12 illustrates the approximate shape of the simulated data and the predicted data produced by the model. Note how the model tracks the simulated data reasonably closely. Both curves flatten out horizontally at an asymptote that is equal to the initialization transient for the trace. This is equal to the number of distinct lines in the trace divided by the length of the trace. So their model explicitly incorporates the initialization transient and the cache simulation captures the same transient. The trend line in the figure shows an extrapolation of the straight portion of the data. The extrapolation follows the rule that each doubling of cache size reduces the miss ratio by a constant percentage, and is essentially the 30-percent rule for a different percentage. The curve that veers

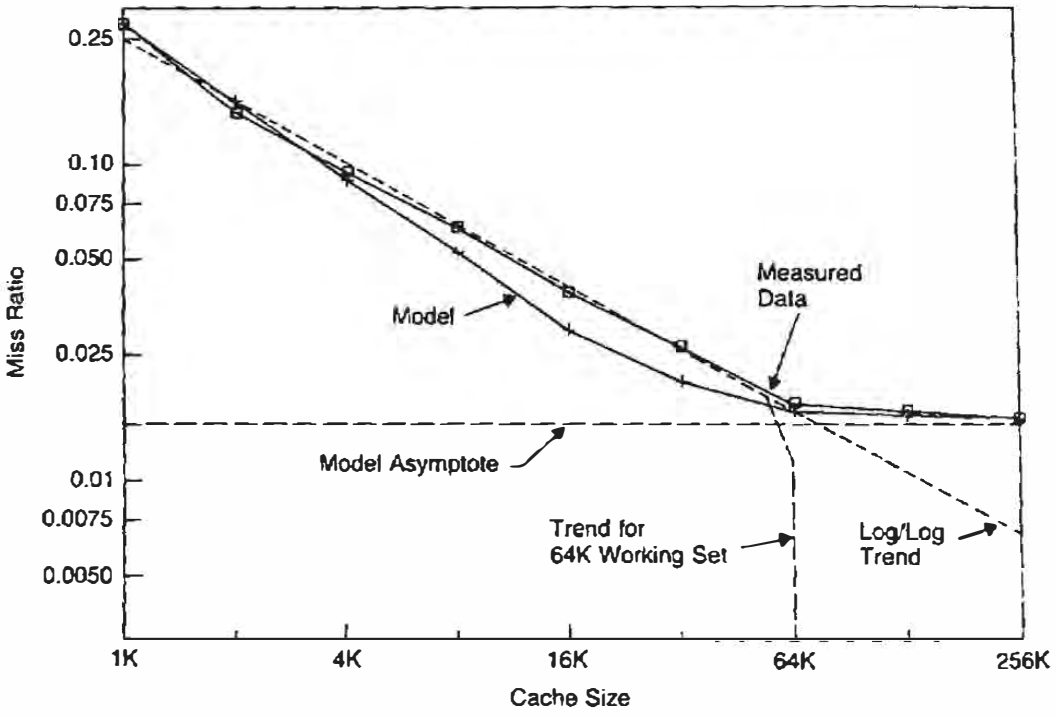

Fig. 2.12 The cache-miss ratio model of Agarwal, Hennessy, and Horowitz.

downward vertically at 64K shows what we expect to be the case for a workload that fits entirely within a 64 K-byte cache.

In analyzing the data in the figure, note that at 64K the miss ratio is close to 1 percent. This 64K cache has 4K lines, and at that miss ratio requires roughly 400K references to produce enough misses to fill the cache. Although they used many traces for their analysis, the longest trace was only 500K. Hence the longest of their traces barely filled the 64K cache in the initialization period before it ended and other traces may not have filled the cache completely prior to completion. For the 256K cache, at most 25 percent of the cache was initialized by any one trace. Because the miss-ratio during initialization is higher than the steady-state miss ratio, the data give us very little indication of what the steady-state miss ratio might be for a 256K cache.

Because their model specifically estimates the initialization misses, Agarwal *et al.* correctly conclude in their paper that the observed miss ratios for the caches larger than 64K are almost totally due to the initialization transient. Many other studies similar to the one conducted by Agarwal *et al.* produce curves with the same flattening of the miss ratio for large caches, but Agarwal *et al.* produce along with the data a correct explanation of the phenomenon. It is interesting that the vertical trend is what we expect to see when caches are sufficiently large, but published data tend to show a horizontal trend rather than a vertical trend because the traces are generally too short to eliminate the initialization effects.

2.2.4 Efficient Cache Analysis

At this point we presume that the cache designer has a collection of traces available for cache studies. Our earlier remarks suggest that the designer will try to evaluate many different caches, and therefore may have to use one address trace several different times. This could be extremely time consuming and costly. A trace with 5 million references may contain 4 bytes per reference, for a total of 20 million bytes. Processing this trace may require an hour of computer time on a high-speed computer. To evaluate 100 variations of cache designs on separate passes of the trace would be an enormous computational burden for an analysis that is conceptually very simple. The remainder of this section treats a set of techniques that together reduce processing requirements by as much as a factor of 1000.

We use three different techniques to reduce processing requirements:

1. Multiple analyses per run;
2. Elimination of hits to the most recently used line; and
3. Set sampling.

The following sections treat each of these in turn.

Multiple Analyses per Run The idea of examining several different caches in one pass of a trace is due originally to Mattson *et al.* [1970]. The idea is that a single simulation run for one pass of a trace can produce data for several cache evaluations. However, this result depends on the replacement policy that specifies what line to remove when a new line enters the cache. We describe the work of Mattson *et al.* by example in the context of cache analysis, and explore more fully the impact of replacement policies later in this section.

Figure 2.13 illustrates a situation in which an eight-way set-associative cache is being analyzed. We shall see that we can obtain analyses for K-way set-associative caches for each K less than eight while performing the analysis for the eight-way cache. Figure 2.13 shows the directory for one of N sets in a cache. Note that this directory has eight positions because the cache is eight-way set associative.

Let us examine a typical sequence of address references to this set and observe the effects of a particular replacement policy. Suppose the set initially contains the addresses A through H as shown in Fig. 2.13(a). If the next address reference to this set is a miss, the new item is brought into the cache and entered into the set. But which item is displaced to make room for the new item? A policy that is implemented almost universally is the least-recently used (LRU) policy, which says that the item displaced is the least-recently used item in the set.

If the addresses in Fig. 2.13(a) are arranged in order of their last reference so that A is the most-recently used item and H is the least-recently used item, then the new state of the set will be as shown in Fig. 2.13(b), which shows the new reference Z at the top of the set, references A through G moved down one

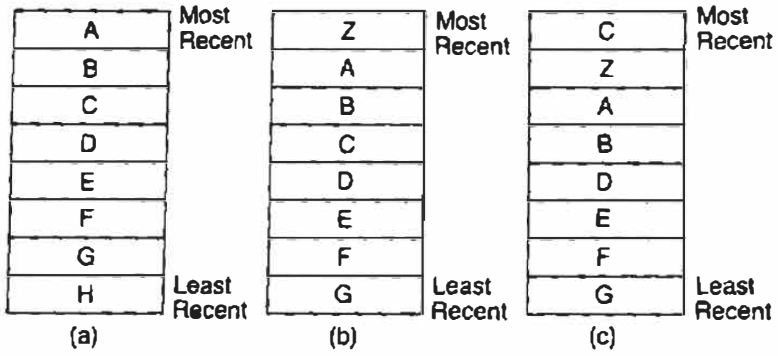

Fig. 2.13 An eight-way cache directory maintained with an LRU policy:
(a) Initial state;
(b) After reference to Line Z; and
(c) After reference to Line C.

cell in the set, and reference *H* discarded from the bottom. Suppose in this state that the next reference is to address *C*, a hit in the set. Then the next state should be as shown in Fig. 2.13(c), which places *C* at the top of the set, pushes down *Z*, *A*, and *B*, and leaves other items unchanged. Even though no item is removed from the set when a hit occurs, the contents of the set should be reordered because we must be able to locate the least-recently used item at any given time. The reordering maintains the set in the order from most- to least-recently used item.

The key idea contributed by Mattson *et al.* is that for the LRU replacement policy the contents of a set for a *K*-way set-associative cache contains the contents of sets for all *K'*-way set-associative caches for each *K'* less than *K*. This is called the *inclusion principle*. In Fig. 2.13, the eight-way set contains the contents of one-way, two-way, and so on up to seven-way set-associative caches whose number of sets and line size are equal to the number of sets and line size for the eight-way cache. In fact, if we look at the behavior of a seven-way set-associative cache, we discover that the items held in that cache occupy the first seven positions of the sets for an eight-way cache.

To keep track of the performance of stacks with one-way to eight-way set associativity, we simply have to note the position of each hit in the stack. For example, the reference to item *C* in Fig. 2.13(b) touches *C* lying in position 4. This is a hit in a four-way cache, but a miss in a three-way cache. In fact, this is a hit in a *K*-way cache if and only if $K > 3$.

Let us keep track of the position of a cache hit in a vector of counts that we call HIT(*I*), where *I* runs from 1 to 8. The HIT vector is initialized to 0. If a hit occurs at position *I*, then we increment HIT(*I*). At the end of the cache evaluation, if we want to know how many hits there will be for an eight-way cache, we simply sum HIT(*I*) for $I = 1$ to 8.

To find the hits for a four-way cache, we compute the sum of HIT(*I*) for $I = 1$ to 4. Since HIT(5) counts the number of hits at Position 5 in the set, none of those hits are hits in a set with four or fewer lines. Hence, the contents of HIT(5) must be excluded from the hit count for a four-way set-associative cache. Similarly, we can reason that the count for HIT(3) must be included in the hit count for a four-element set because hits in a three-way cache are also hits in a four-way cache under the LRU-replacement policy. That is why the number of hits for a *K*-way cache can be found by summing HIT(*I*) for $I = 1$ to *K*.

Mattson *et al.* [1970] treat other replacement algorithms in addition to LRU replacement. Some of the replacement strategies have the same property that LRU has. That is, as you increase the size of a set, all of the hits of a *K'*-way set-associative cache are hits in a *K*-way set-associative cache for all $K' < K$. But some of the replacement strategies do not have this property. In particular, if you select the item to be replaced at random, then it is perfectly possible, for example, for a miss in a three-way cache to hit in a two-way cache.

A replacement policy that exploits the inclusion principle is called a *stack-*

replacement policy because the candidates to be replaced can be placed in a push-down stack as shown in Fig. 2.13. The stacks for K-way set-associative caches nest one inside the other as K increases. As a general rule, when evaluating stack-replacement policies, one can simulate many different caches during one pass of the trace. This technique could reduce evaluation effort by as much as a factor of 10 over the process of performing separate passes of the input data for each set size evaluated.

Hill and Smith [1989] exploited the algorithm of Mattson *et al.* by generalizing it to analyze in one pass of a trace caches that differ in the number of sets per cache as well as in their associativity. The algorithm is remarkably simple to implement, and it is a very effective tool to have available. Figure 2.14 shows a collection of sets from caches of various sizes, and illustrates how the algorithm keeps track of all caches concurrently. At the left in the figure is a set from a two-way set-associative cache of size N. To its right are shown two different sets from a two-way set-associative cache of size $2N$, and to their right are four sets from a two-way set-associative cache of size $4N$. The sets have the property that the addresses that map into the four sets shown for the $4N$-set cache all map to same set in the N-set cache. The upper pair of sets of the $4N$-set cache map into the upper set of the $2N$-set cache as depicted in the figure. That set in turn maps to the upper set of the N-set cache.

In Fig. 2.14, since the sets on the right are from a cache with four times as many sets as the one on the left, the sets on the right are selected by decoding two more address bits than those on the left. The decoded values of the two additional bits are shown with the sets on the right. These values are called the *selection fields* of the addresses. The top set of two lines has a selection field with the value 00. The lines in this set map to a set in the middle cache whose selection field uses one fewer bit of the address for selection. The bit drops off from the more significant end of the selection field according to the decoding scheme shown in Fig. 2.8. Hence, the selection field of the set in the middle cache has the value X0, where the X represents a bit that does not participate in selection. The other set from the right-hand cache that maps into the top set of the middle cache has the selection field 10.

Figure 2.15 shows the eight entries from the right hand sets of Fig. 2.14. They are tabulated in order from most recently used to least recently used, and each entry carries with it the corresponding selection field shown in Fig. 2.14. The interesting point of this figure is that the selection field with each entry plus its position in the table when considered together are sufficient to determine where each entry lies in all three caches. To find the entries in the set with selection field 00 in the right-hand cache, scan the table from most recently used entry to least recently used entry for the occurrence of field 00. The first one encountered will be the most recently used entry in the set selected by field 00 and the next one will be the second most recently used in that set. In the figure, these two entries are A and B in that order.

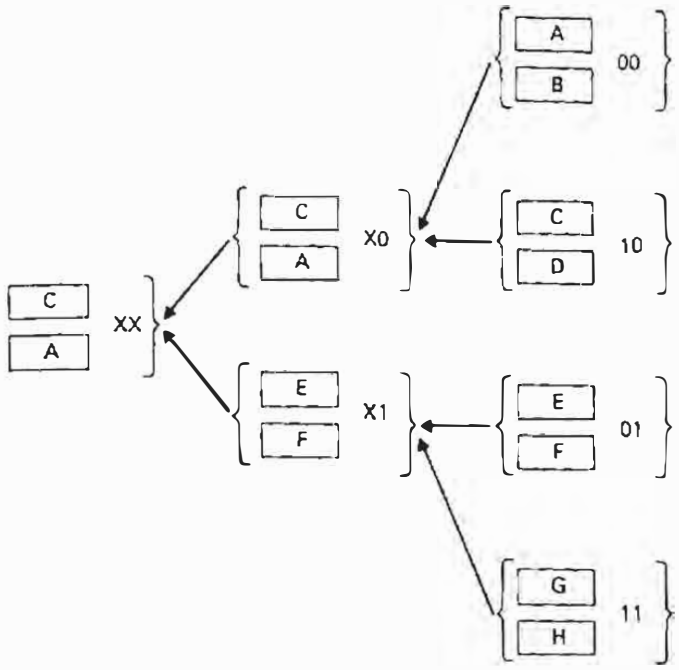

Fig. 2.14 A diagram of the relation between sets of caches derived by doubling the number of sets per cache. Two sets of the $2N$-set cache map into one set of the N-set cache. The value of the selection field is shown with each set.

To determine the contents of a set in the middle cache, say set X0, scan the table from most recently used entry to least recently used entry for the first two entries that match the tag X0. These are C and A in that order. Since the table in Fig. 2.15 contains sufficient information to determine the contents of all three caches, it follows that we can also determine if a reference is a cache hit or cache miss in each of the three caches. In this example, a table 8 deep for each of N sets is used to evaluate two-way associative caches with up to $4N$ sets. In general, to evaluate K-way associative caches for caches with a number of sets ranging from N to M N, the algorithm uses N tables of size K M.

By keeping track of LRU depth using the algorithm of Mattson *et al.*, all caches with associativities from 1 to K can be evaluated in the same sweep of the table. The algorithm works as follows:

1. Map the reference address to one of N tables by extracting the set-field for an N-set cache from the reference address.
2. Extract the additional bits in the selection field from the address.

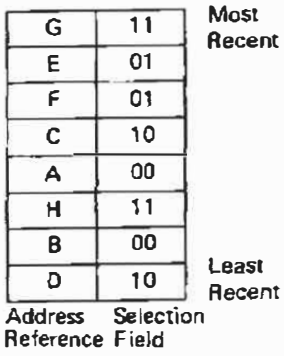

Fig. 2.15 A table of cache entries for simulating several cache directories concurrently. The entries are arranged from most recently used to least recently used.

3. Scan the table entries in succession starting at the most recently used entry.
4. Each table entry represents the contents of one line of several different caches. If the table entry matches the reference address, it is a hit in all of the caches to which the entry maps. If the table entry does not match the address reference, it is a miss in those caches for which the table entry and address reference map into the same set. For the remaining caches to which the address reference maps, the entry is neither a hit nor a miss. In this step, the algorithm determines for which caches the entry is a hit or a miss.

   If the address matches the address in the table, it is a hit in each of the caches to which it maps. For a selection field of 00, the hit is recorded for set XX of an N set cache, for set X0 of a $2N$ set cache, and for set 00 of a $4N$ set cache. If the selection field of table entry is the ith field that matched position X0, then the hit occurs at LRU depth i in the $2N$-set cache. By this reasoning, the algorithm can record the LRU depth of a hit in each cache, and thereby preserve enough information to calculate miss rates for all associativities less than the highest associativity among the caches analyzed.

   If the address does not match the table entry, it is a miss in some of the sets represented by the table entry. There is a miss in the N-set cache whose selection field is XX..X. To determine which other sets and which caches should log a miss, compare the selection field of the table entry to the selection field of the address bit by bit, starting at the rightmost bit of the selection field. Move to the left and stop at the first bit that does not match. For each bit that matches, log a miss in the set that corresponds to the matched bits of the selection field. For example, consider an address with selection field 1001 and a table entry with selection field 1101. The algorithm

logs a miss in an N-set cache in set XXXX, in a $2N$-set cache in set XXX1, and in a $4N$-set cache in set XX01. Since the next bit of the selection field of the reference address and the table entry do not match each other, no additional misses are logged.

5. If a table entry produces a hit, the scan of the table terminates. The entry at the hit position becomes the most recently used entry in the table, and the entries that were formerly ahead of it are moved down one position.

   If a table entry is the K-th miss in a cache with the maximum number of sets (a cache with no X's in its selection field), the scan of the table stops at this entry. This entry is cast out of the table, the entries above it are moved down one, and the new address becomes the most recently used entry in the table.

Wang and Baer [1991] extended these ideas further to encompass techniques for simulating caches in multiprocessor systems. Their reasoning relies on the inclusion property to enable multiple analyses to be carried out during a single pass of the input data.

Trace Stripping: Filtering Puzak [1985] discovered two techniques for reducing traces that together can reduce effort by two orders of magnitude. The first, trace filtering, introduces no error in counting misses, and is the subject of this section. The second, set sampling, is statistical by nature, and produces a sampling error in the final data. It is discussed in the following section. By using both techniques, an analyst can reduce a trace with 100 million references to a trace length of only 1,000,000 references, yet the reduced trace can give extremely accurate estimates of the miss and hit ratios of the cache.

The shorter length of a filtered trace permits extensive cache analysis to be done with microcomputers, whereas an analysis of the full trace would be far too large a task to be done in reasonable time on a microcomputer. Both of Puzak's techniques rely on stripping from the address trace a large number of references that do not affect the final results. The first stripping technique, *trace filtering*, is the following:

> Assume that a set of analyses is to be done for caches with a fixed line size L and at least N sets. Then prepare a reduced address trace by simulating a one-way associative cache with N sets and line size L operating on the full trace. Output a reduced trace that contains only the addresses that produce misses in the N-set, one-way set-associative cache.

The trace produced by this simulation process throws away all hits to a one-way associative cache with N sets. That is, it throws away all address references to a line in a set that was the most recently referenced line in that set. What remains on the reduced trace are just the misses experienced by the N-set, one-

way set-associative cache. Typically, a cache of reasonable size with this structure should have a miss ratio of 10 percent or less, so that the trace reduction should produce a new trace that is about 10 percent of the original length.

When the reduced trace is used to evaluate the same cache that produced the reduced trace, the number of misses will be identical to the number obtained from the unreduced trace. Of course, the number of hits will be different, so the observed hit and miss ratios will also be different for the reduced trace. If we know the original length of the address trace, however, then from the absolute number of misses observed we can compute the number of hits and the hit and miss ratios. So all information relevant for a cache analysis is still available on the reduced trace, provided that we know the length of the original trace.

What makes this technique more interesting is that we can use the reduced trace to evaluate many different cache structures in a single pass and still obtain exact or near-exact values of the hit and miss ratios. Puzak proved the following result:

> Create a reduced trace by simulating a one-way cache with N sets and line size L, retaining on the reduced trace only the addresses that produce cache misses. Simulate a K-way set-associative cache with N sets and line size L on the original trace and the reduced trace. The two simulations produce the same number of cache misses.

Puzak's proof of this statement is a modification of the argument of Mattson *et al.*, which says that as you increase the stack depth (in this case K), the contents of stacks for smaller K are subsets of the contents of the stack for larger K. The key idea in the proof is that each miss on the reduced trace is a miss on the full trace, and conversely, each miss on the full trace is a miss on the reduced trace.

The process of producing a reduced trace by discarding cache hits for the one-way cache discards no misses for the K-way cache. Because of the stack-algorithm property developed by Mattson *et al.*, no misses for the K-way cache are discarded either. The misses for the K-way cache are a subset of the misses of the one-way cache and will appear in the reduced trace. Moreover, the references discarded from the full trace to produce the reduced trace are hits to the most-recently used set in a one-way cache, but each of these is a hit to the most-recently used set of a K-way cache, and each such hit does not result in a reordering of the K lines within a K-way set. When such references are removed from a trace, the number of misses observed does not change. From this argument we conclude that the reduced trace and the full trace yield an identical number of misses for any K-way cache with N sets and line size L.

Not only can you vary set associativity on a reduced trace, but you can also study the effects of varying N. Puzak showed that the following result is true:

> Let N be a power of 2. Prepare a reduced trace by simulating an N-set, one-way set associative cache with a line size L, and retain on the reduced trace only those address references that produce cache misses. Simulate a one-way set-associative cache with

$2N$ sets and line size L on the full trace and on the reduced trace. The two simulations produce the same number of misses.

To prove this statement it is sufficient to show that any cache miss on the full trace is a miss on the reduced trace, and conversely, any miss on the reduced trace is a miss on the full trace.

To illustrate the proof, consider Fig. 2.14 which shows how sets of $2N$-set and $4N$-set caches map into sets of an N-set cache. When we double the number of sets from N to $2N$, we do so by increasing the set field by one bit. The effect is to split into two groups the addresses that map into one set of an N-set cache. Each group maps into distinct sets of the $2N$-set cache as shown in the figure.

For example, consider an address for a cache with $2N$ sets and line size L. Let us break up the left-most field of the address shown in Fig. 2.8 into two fields, one of which contains only the rightmost bit of the field, and the other of which contains the remaining bits on the left.

This produces a total of four fields in the address, which we denote as (T, B, S, L) for tag, bit, set, and line. The L field gives an address within line and is ignored by the cache when matching addresses. The S field gives a set number for an N-way cache and is $\log_2 N$ bits long. The B field has a length of 1, and the B field concatenated with the S field gives the set number for a cache with $2N$ sets. The B-bit for this example is the selection field shown in Fig. 2.14. The T field is the tag field for a cache with $2N$ sets, and the T field concatenated with the B field is the tag field for a cache with N sets.

When a cache lookup is in progress, a K-way cache uses the set number to initiate a read in each of K memories and compares the tag stored there to the tag derived from the address. Hence, when an address (T, B, S, L) is used by an N-set cache, the tag is (T, B), and the set number is S. When that same address is used by a $2N$-set cache, the tag is T, and the set number is (B, S). Since B is a single bit, the set number is either (0, S) or (1, S). Consequently, any address that is in Set S of an N-set cache falls in either Set (0, S) or (1, S) of a $2N$-set cache.

We will simulate a $2N$-set one-way cache to show why each miss on the full trace appears as an address on the reduced trace and yields a miss there as well. Without loss of generality, let us focus on a particular set, Set (0, S), of the $2N$-set cache and observe an address sequence that produces a miss. Suppose that the address (T_1, 0, S, L) produces a miss on the full trace. The prior reference to Set (0, S) in the $2N$-set cache must have a different tag in order for the present reference to be a miss. Consequently, the prior reference must be an address of the form (T_0, 0, S, L) where tags T_0 and T_1 are not equal, and the values of the L field for the two addresses do not matter. Is (T_1, 0, S, L) on the reduced trace? Yes it is, because this reference produces a miss in an N-set cache.

On the full trace, the prior reference to the Set (S) of an N-set cache is either the reference (T_0, 0, S, L) or a reference of the form (T, 1, S, L) for some tag

field T. If the prior reference were (T, 1, S, L), this reference would appear between (T_0, 0, S, L) and (T_1, 0, S, L), and it would be treated as a reference to Set (1, S), not (0, S), when simulating the $2N$-set cache.

Since the prior reference to Set S for the N-set simulation has either tag (T_0, 0) or tag (T, 1), neither of which is equal to tag (T_1, 0), the reference to (T_1, 0, S, L) is a miss on the full trace and does appear on the reduced trace. This reference is a miss on the reduced trace because of the rule used in discarding addresses to produce a reduced trace. We can discard address reference (T_1, 0, S, L) from the full trace only if the prior reference on the full tape to Set S in an N-set cache has tag (T_1, 0). But this was not the case, since the preceding discussion indicates that the prior reference to Set (0, S) in a $2N$-set cache must have a tag different from T_1.

To prove that during the simulation of a $2N$-set cache every miss observed on the reduced trace is a miss on the full trace, we use a similar argument. Now we assume that the address reference (T_1, 0, S, L) on the reduced trace produces a miss when simulating a cache with $2N$ sets. On the reduced trace the prior reference to Set (0, S) must have a different tag, so it must be an address of the form (T_0, 0, S, L) where T_0 and T_1 are unequal, and the values of the L fields are immaterial. Both of these references must occur in sequence on the full trace. Between these references there may be other references to Set (0, S) that do not appear on the reduced trace, but all such references are eliminated from the full trace only if they are hits to Set S of a one-way N-set cache. They must be address references of the form (T_0, 0, S, L) because the tag must be of the form (T, 0), and the latest reference to Set S with a tag of this form is the reference to (T_0, 0, S, L).

We have now shown that when we simulate a one-way $2N$-set cache on the full and reduced traces we obtain the same number of misses. Puzak actually used the proof technique given here to prove the following statement:

> Let N be a power of 2, and let M be a power of 2 no less than N. Create a reduced trace by simulating a one-way N-set associative cache with line size L. Retain on the reduced trace only those addresses that produce misses. Now simulate a K-way M-set cache for any $K > 0$ on both the full and reduced traces. The number of misses observed during the two simulations will be equal.

This statement says that a reduced trace can be used to simulate caches with any combination of set associativity and number of sets, provided that

- The line size L for the simulated cache is equal to the line size of the cache used for the trace reduction;
- The simulated cache uses at least as many bits in the set-number field as the cache uses for the trace reduction; and
- The set associativity is arbitrary.

Given that trace reduction is useful for studying caches with increased values of set associativity and number of sets, is it also useful for studying caches with increased values of line size? The answer is a qualified yes. When a trace reduced by simulating a cache with line size L is used to evaluate a cache with line size $2L$, we quickly discover that the sets for the cache with line size L do not have a direct relation to the sets for the cache with line size $2L$.

Wang and Baer [1991] made an interesting discovery that permits a single trace to be used to simulate caches with different line sizes at perfect fidelity. The idea is based on the observation that a reference that is a hit to a cache with a certain line size is probably also a hit to a cache of equal or larger size whose line size is different. Where Puzak recommends producing a stripped trace for each different line size, Wang and Baer produce a single stripped trace that contains the union of the references of Puzak's stripped traces. To create such a trace from a full trace, use the full trace as input to a set of cache simulators. Simultaneously simulate the smallest one-way set associative cache of each line size of interest. Record on the stripped trace a reference if it is a miss in any simulated cache. In other words, cast out hits only if they hit in every simulated cache. Because so many references hit in all the caches, the trace reduction is effective. Wang and Baer report that it increases the length of a single trace by a percentage that was observed as high as 40 to 48 percent for capturing data for five line sizes, but it removes the necessity for creating a single trace for each different line size, and thereby substantially reduces the volume of data saved. The trace reductions they obtained even with this additional burden were factors of 7 to 20.

Trace Stripping: Set Sampling Having reduced the original trace by roughly 90 percent, let's explore how to make a second reduction that again reduces the reduced trace by 90 percent. The trick here is to observe that each of the N sets behaves statistically like any other set, so that the performance of the full cache can be estimated by observing only one set. In fact, as we mentioned earlier, the references across sets are highly correlated. The measurement of miss ratios by using all sets does not increase the accuracy of the estimate of the miss ratio by as much as it would if the references were independent. Hence some effort can be saved without a severe loss of accuracy by restricting attention to only a fraction of all the sets in the cache.

In most designs, N is fairly large, usually 64 to 1024, so the opportunity for reducing effort by a factor of N has a high payoff. There is some danger in selecting a single set, however, because it might just happen to be an unusual set whose statistics are not representative. To be safe, the designer can use two, three, or more sets, with accuracy increasing as the number of sets increases, but at the cost of additional processing time. The idea is to select a few sets and examine their behavior characteristics.

Using standard statistical techniques, one can obtain confidence intervals

on the evaluation measures produced from a few sets, where the confidence intervals give an estimate of the error introduced by sampling a few sets instead of using all of them. Similar techniques are used for quality control and for predicting the outcome of elections. In both of these instances, the sampling process is used on a small population to determine the characteristics of a much larger population.

Puzak discovered that selecting 6 of 64 sets was sufficient to reduce the 95-percent confidence interval to less than 1 percent of the measured data. That is, the data obtained from 6 sets would be within 1 percent of the data values for the full trace in 95 percent of the experiments that sample random populations with similar statistical distributions. Hence, the reduced trace can be stripped again to retain the references from some small number of sets. The number of sets to use does depend on the variance in the observed data, so we cannot give a specific number that holds for all cases. Puzak's experience indicates that for reasonable data, retaining only 10 percent of the sets is sufficient.

To summarize how the various techniques described in this section can be put to use, consider the design of a cache that nominally has from 128 K-bytes to 1 M-byte. The designer has to determine how this cache can be organized. Here is one typical sequence of steps that might be followed:

1. Pick a candidate line size for the 128K cache. This is usually determined by the width of the path between main memory and the processor. The line size can be a multiple of this width, but should not be smaller than the width. In the running example, we assume that the path width is 16 bytes, but we choose to have a line size of at least 32 bytes to reduce the number of directory entries in the cache.
2. Determine what cache structures are to be studied. In our case we want to examine caches with at least two-way set associativity. Hence, the 128K caches of interest are 2 by 2048, 4 by 1024, and so on, and larger caches are obtained by doubling and quadrupling the number of sets. If we choose to examine larger line sizes, we halve the number of sets for each doubling of the line size.
3. The largest number of lines among the caches under consideration occurs for the 1 M-byte cache with 32-byte lines. The number of lines in this cache is 32K lines. Assuming a miss ratio during initialization of about 1 percent, it takes about 3.2 million references to fill this cache initially. To produce accurate estimates of hit ratios, we must find the trace length required for measuring misses after cache initialization. To obtain good precision, we would like at least 16 misses per set. Since the associativity is at least two-way among all of the caches studied, the 32K lines of the largest cache fall into 16K sets. Hence, we need to produce at least 256K misses from a trace in order to obtain sufficient precision. If the steady-state miss ratio is ap-

proximately 0.25 percent, we require more than 100 million references to produce a confidence interval of no less than 50 percent of the miss ratio. If either the initialization miss ratio or steady-state miss ratio are lower than estimated, the trace has to be longer. To give some additional margin of safety, we choose to make the traces 200 million references in length.

4. Prepare a collection of programs that comprises a representative workload. Prepare an address trace from these programs of a length of at least 200 million addresses. The proportion of the trace devoted to each type of program should reflect the anticipated workload, and the transients caused by changing from one program to another should also reflect the expected frequency of such transients.
5. Prepare a reduced trace by stripping from the full trace all hits to a cache with line size 32, set associativity 1, and 1024 sets. If other line sizes are of interest, simulate 128K direct-mapped caches with those line sizes as well. Retain an address in the stripped trace if it is a miss in any cache. Also, select some fraction of the sets at random, for example, 12.5 percent, and strip out all references in the trace to sets other than the selected sets. During the stripping process, observe the total miss ratio, the miss ratio on each of selected sets, and the composite miss ratio for the collection of selected sets. If the composite miss ratio differs significantly from the actual miss ratio, use the data obtained from the individual sets to find a sample variance for the observed miss ratio. From this, estimate how many sets are actually required to reduce the sampling error to a tolerable amount. If more sets are needed, obtain them from the original trace by repeating the process given here.
6. With the reduced trace as input data, simulate the following caches in one pass of the trace using the algorithm of Hill and Smith [1989]: 32-byte line size, 2K and 4K sets with associativities 2, 4, and 8; 8K sets with associativities of 2 and 4; 16K sets with an associativity of 2.
7. Calculate the miss-rate for each cache and estimate the cost of each cache. Use the miss-rate data to estimate relative performance for each cache. Estimate the relative system cost for each cache. Determine the most reasonable trade-off of performance and cost.

Note that some of the data collected is for caches larger than the design point. This gives additional information on the merits of moving to a larger cache in the future and should be useful if there is some need to plan for the larger cache in present designs.

Together the methods described in this section should make cache-memory analysis accessible to all designers. It becomes feasible to use personal workstations to conduct such studies that formerly taxed the facilities of the largest computers. In closing, we make one additional observation that greatly simplifies

the collection of the data. It is quite feasible to strip the trace while collecting address references. Simply record only those addresses that are misses to a few selected sets of an N-set one-way set associative cache. If we randomly select three bits from the set field and record only the misses to the sets for which these bits have a specific value, for example, (0, 0, 0), then we will be recording references to 12.5 percent (one-eighth) of the sets, and only the misses to those sets.

Because this scheme selects only one address in a hundred for output, the address references can easily be gathered in real-time, even for very fast machines. However, it is necessary to have a buffer that can accept references at very high instantaneous data rates because the cache misses that are captured do not necessarily occur uniformly through a simulation, but rather may bunch together in small regions of the simulation. Nevertheless, the almost 100-to-1 reduction in the volume and data rate of the data to capture makes this technique very attractive.

2.2.5 Replacement Policies

In this section we look into the replacement policies and their impact on cache performance. Nearly all caches in commercial production use least-recently used (LRU) policies to manage the lines in a set. Recent work by Puzak [1985] points out ways to obtain improvements over LRU replacement at reasonable cost. This section explores the characteristics of LRU and compares them with an optimal (but nonrealizable) replacement policy to conjecture how one might design a realizable, near-optimal replacement policy for a cache.

The main objective of a replacement policy is to retain the lines likely to be referenced in the near future and discard lines that are no longer useful or whose next access is in the more distant future. We can easily evaluate any replacement policy by comparing it to an optimum policy that has perfect knowledge of the future. Belady [1966] described such a policy in the context of virtual-memory systems. The same algorithm holds for cache memories. The characterization of this algorithm described here first appeared in Mattson *et al.* [1970].

Assume that a cache has perfect knowledge of the future: What should its replacement policy be? In fact, the optimal replacement policy (OPT) is identical to an LRU replacement policy that operates on the reference stream reversed in time. More specifically:

> The optimal replacement policy (OPT) discards the line of a set whose next reference is furthest in the future of any other line in the same set.

Figure 2.16 shows the OPT policy in action. Figure 2.16(a) shows a set of lines ordered so that the line at the top is the next of the set to be referenced, and the remaining lines appear in the order in which their next reference appears in the address-reference stream. We assume that the future reference stream is

OPT has a miss, OPT must at some earlier time produce a hit when the other policy produces a miss. In fact, OPT has at least one hit for every hit of the other policy, and it may have more.

It is rather interesting that LRU, which looks only backward, works well compared to OPT, which looks only forward. The recent past appears to be a good estimate of the near future. Perhaps this is due to the nested structure of programs, which leads to the characteristic that the recent past is a reversal of the near future. Consider, for example, a series of nested loops.

```
for i = 1 to N do
    for j = 1 to M do
        for k = 1 to L do
        (body of inner loop)
        end; {* k loop *}
    end; {* j loop *}
end; {* i loop *}
```

In this nesting the indices are incremented and tested on a last-in, first-out basis. In the loop body the index that is next to be touched is the last to have been touched, and similarly, the index to be touched furthest in the future is the index touched furthest in the past. The last-in, first-out data access characteristic associated with nested loops, nested subroutine calls, and nested interrupts accounts for the future being similar to the past.

If this were the only characteristic of programs, then LRU would almost always closely mimic OPT. But other characteristics of programs strongly interfere with cache management. One problem with the LRU replacement policy is that it does not anticipate the future well when sequential or cyclical activity is in progress. In either of these cases, once an item has been processed, it can be removed from the cache. If it is to be used again, the next access occurs further in the future than the access to other items available in the same cycle.

Consider, for example, the difference between LRU and OPT when each processes a cycle of references of length 6. Let the reference string be *A, B, C, D, E, F, A, B, C, ... , F, A, ... , F, ...* , and observe how LRU manages this cycle in a set of size 2. LRU retains the last two references and misses on each new reference. But OPT obtains two hits in the cycle of six references because the two entries in its cache will be retained to obtain hits on those entries.

Assume for a moment that the cache contains *A* and *B* and the references to these entries have just been processed to produce hits.

The next reference is to *C*. OPT looks ahead in the future and sees that *A* and *B* will be referenced before *C*, so that OPT chooses to retain *A* and *B* in the cache rather than bring in *C* after experiencing the miss on *C*. LRU always brings in the most recently used item, but in this example, the most recently used item is discarded in favor of the present contents of the cache. And as a result OPT

is able to outperform LRU. In reality, the safe bet is that the most recently used item will be reused soon, and it is wise to hold this in cache. But when a process touches a sufficient number of distinct references to exceed cache capacity, the safe bet is no longer the best bet.

Therefore, to create a replacement policy that performs nearly as well as OPT we must do some replacements that are not LRU replacements, and we should try to do these when references are sequential or cyclical or in some other pattern that is poorly handled by LRU. From the preceding description of the characteristics of OPT, we discover the following interesting fact about OPT:

> For any set-associativity size K, OPT considers only one of two lines for replacement. One candidate is the line *most* recently used, the other is the line referenced furthest in the future.

LRU has only one possible candidate for replacement, the line least recently used. It never replaces the line most recently used unless the set-associativity factor K is 1. Presumably, LRU does as well as it does because the least-recently used line is frequently the line to be referenced furthest in the future.

Puzak's analysis [1985] of OPT and LRU policies turned up another interesting characteristic of LRU replacements. Consider a situation in which a set managed by an LRU policy happens to contain exactly the same lines that OPT would retain. Now assume that LRU elects to replace a line of the cache that OPT elects to keep, and conversely, OPT replaces a line, Line A, that LRU elects to retain.

Puzak notes that Line A is a *dead line* in the LRU cache and that it must leave the cache before it is touched again. If this were not the case, then OPT would have retained Line A and cast out some other line. Since Line A is dead, the set associativity is effectively reduced by one until Line A is swept from the cache by an LRU replacement decision. If Line A happens to be the most-recently referenced item when OPT disposes of it, then in a K-way set-associative cache managed with LRU, references to $K - 1$ distinct lines must occur before Line A is replaced. Many of these references are likely to be misses. As each of the $K - 1$ distinct references occurs, Line A moves down one position in the set, until at last it reaches the least-recently used position from where it is removed from the cache. Here is an opportunity for a better policy!

For example, if a cache-management algorithm were clever enough to prefetch data in anticipation of future references, the obvious place to store the new data is in place of dead lines because these lines will not be referenced again. If we replace lines that are not dead, then each such replacement might change a future hit to a future miss. Therefore, there is some risk in replacing live lines, and no risk in replacing dead lines.

Quite apart from prefetching, there is a great deal that can be done just to improve LRU replacement. For the cache parameters he studied, Puzak found

that OPT's performance for a cache with M lines is approximately the same as LRU's performance for a cache with $2M$ lines. The actual performance difference is not a factor of two. Although the cache sizes differ by a factor of 2, OPT replacement produces only about 30 percent fewer misses than does LRU replacement.

These data are strictly empirical and depend on the architecture, the workload, and the ranges of cache parameters. There is no reason to believe that his observations hold in general. Cache designers should make their own observations based on their specific context and then compare their results with Puzak's.

In any case, by comparing LRU with OPT we can obtain an estimate of the improvement available. Although we can try to improve the cache as much as possible, in reality, we are likely to gain only from 10 to 30 percent of the available improvement because the hardware cannot have perfect knowledge of the future.

Here is a description of one scheme for improving LRU that is based on work by Pomerene *et al.* [1984]. The objective is to distinguish between transient lines that must be flushed from cache quickly and lines that become active after long periods of inactivity.

Pomerene *et al.* propose to use a *shadow directory*, as shown in Fig. 2.17. On the left side of Fig. 2.17 is an ordinary cache divided into a directory and data area. Let us presume that this is a K-way associative cache. To the right in the figure is a duplicate of the cache, except that the duplicate contains only the directory and no data area. This part of the cache is the shadow directory. The

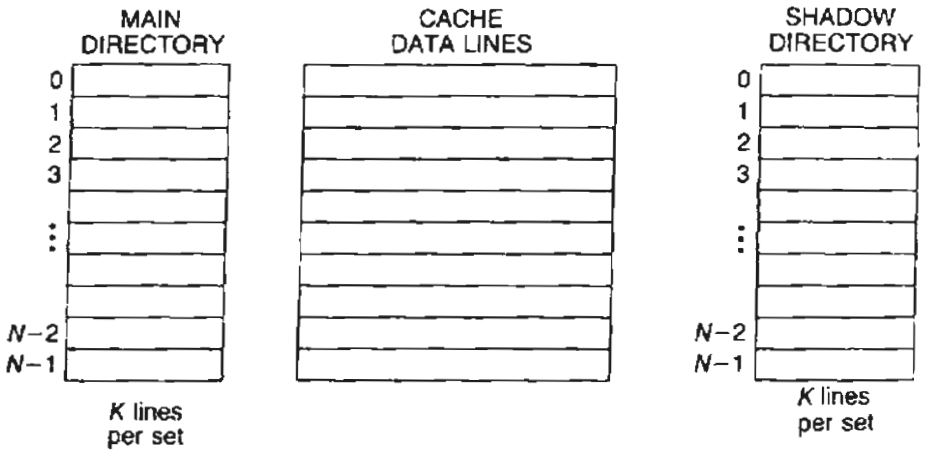

Fig. 2.17 The organization of a cache with a shadow directory. The main cache has N sets, K lines per set. The shadow directory has only the directory entries for an additional K lines per set.

cache is generally managed as if it were a 2K-way set associative cache, except that a directory hit in the right half of the cache produces no data, just directory information.

When a new item is brought into the main cache, one of K items in the same set is discarded from that cache. The discarded item is entered in the shadow directory, displacing one of K items from the corresponding set in that directory. If in each case, the item removed is the least recently used item of the K in its set in the respective directories, the effect of this strategy is to create a cache that is 2K-way set associative, managed by LRU replacement. There is usually plenty of time available to update the cache and both directories because the update occurs during a cache miss, when cache activity essentially comes to a standstill.

The key to the cache's operation is that there are two kinds of misses:

- A *transient miss*, in which the datum is not in the cache, and there is no entry in the shadow directory; and
- A *shadow miss*, in which the datum is not in the cache, but there is an entry for the datum in the shadow directory.

A shadow miss is a miss to line that was used in the distant past and is being used again. There is some likelihood that it will repeat the same behavior in the future by having a lengthy period between two successive accesses.

As each new item is loaded into the cache, a bit is set to indicate whether the item was a transient miss or a shadow miss. That information is used to control replacement. When a replacement decision occurs, the cache manager can examine how lines entered the cache. It can tend to retain the lines that were in the shadow in favor of the lines that were transient misses, and in this way it will tend to flush transients from the cache more quickly than an LRU algorithm will flush them.

As the replacement algorithm chooses lines closer and closer to the most-recently used line, however, the risk becomes greater that the replacement algorithm will make a mistake and cast out a line that should be retained for a future hit. Puzak discovered that it is effective to place a limit on the region of the cache over which the cache manager can give preference to shadow misses over transient misses. For a four-way cache, a reasonable policy is to limit non-LRU decisions to the bottom two cells, that is, the LRU and next-to-LRU entries.

In terms of cost, the shadow directory is surprisingly inexpensive. Most of the cost of a cache is in the data memory. For example, for a line size of 16 bytes and an address-tag size of 4 bytes, a data memory will have four times the number of bytes as has a cache directory. For larger line sizes, such as 64 bytes, the shadow directory will have less than 10 percent of the storage capacity of the data memory.

Since the directory also has comparators, the costs are not in storage alone,

but the storage ratio of data memory to directory does give some idea of relative costs. Consequently, it is conceivable to put 10-percent additional cost into a shadow directory to obtain 5- to 10-percent performance improvement. From a cost-performance view, such improvement could be better than doubling the size of the cache. Note that the improvement range for performance corresponds to a very small absolute change in the miss ratio, roughly 0.5 to 1 percent. The percentage reduction in the number of misses is somewhat larger, possibly 10 to 30 percent. The point is that the shadow directory does not have to be extremely accurate to achieve the improvement we seek.

The shadow directory also avoids a serious problem that develops as caches become larger. Generally, the larger a cache, the slower the cache cycle becomes. Since the shadow directory is not accompanied by a data memory, the volume and power consumed by the data memory is avoided, which is a tremendous advantage for high-speed systems.

Moreover, the shadow directory need not increase the cache-cycle time since the shadow does not have to be consulted on every memory access. The only time it needs to be consulted is on a cache miss, and at this point there are many cycles available to handle updating and replacement. To conserve on power and cooling, it is feasible to build the shadow directory with logic slower than that used in the main cache. The shadow can be included fairly inexpensively to obtain a small, but worthwhile, increase in performance.

2.2.6 Footprints in the Cache

In this section, we expand upon some of the ideas of the shadow directory to derive a simple and useful model of transient misses in a cache. Voldman and Hoevel [1981] and Voldman *et al.* [1983] conducted empirical studies of misses in caches. Their data show that cache misses are not distributed uniformly through an address trace, but instead tend to be clustered into clumps. Between the clumps are relatively long periods of time during which cache misses are rare.

Attempts to model this behavior statistically have not been very successful because the distributions that best characterize the behavior do not have finite variance. Voldman *et al.* [1983] showed a characterization based on fractals, which is helpful for explaining an empirically observed sequence of misses, but is not directly useful in predicting the effects of cache parameters on miss ratios.

The importance of the transient effect on cache performance led Strecker [1983] to develop a model of miss ratios for the case when two or more processes compete alternately for a cache. Strecker observed that as each process takes control, it expends its initial references reloading the cache. As the cache becomes partially loaded, the misses decline, and eventually the miss ratio reaches the long-term steady-state miss ratio. Strecker's model estimates the average miss ratio for a process over an interval of time that includes the transient period

when the process is reloading the cache. Although the model is fairly complex, Strecker showed that it gives reasonably accurate results for the specific processes he modeled. The value of the method is limited because it relies on fitting a curve at two points to calculate the values of two parameters that define the curve. If you have the data to fit the two points, you probably have the data for the remainder of the points. The objective is to determine the transient using as little additional information as possible. Another interesting study of note is by Laha, Patel, and Iyer [1988] who show that reasonable estimates of the reload transient can be obtained by piecing together many different small segments of a larger trace.

Both of the studies cited here attempt to explore the reload transient in conjunction with the steady-state miss process. The material in this section shows how to calculate the reload transient in isolation. This permits the transient to be combined with the steady-state miss ratio in various ways to reflect the true cache reload transients that take place in a computer system. The trick is to count the number of lines that have to be reloaded without attempting to measure instantaneous miss rates. Since each cache miss carries a penalty, and for many architectures the penalty is a fixed cost, the model can give the total cost penalty of a reload transient. The penalty drops off for larger caches in a predictable way, and the predictions have been confirmed by experiment. The work described here is by Thiebaut and Stone [1987].

The model for determining how different processes compete for the cache is illustrated in Fig. 2.18. Figure 2.18(a) shows Process *A* and Process *B* running alternately in time. These two processes may be quite independent, as is the case if Process *A* is an interrupt-driven program servicing some input/output device, and Process *B* is a compute-bound main program. Or the processes could be quite dependent on each other, as is the case if Process *A* invokes Process *B* repeatedly because of a call on *B* placed within a loop in *A*.

In a cache-based architecture, what actually happens is shown in Fig. 2.18(b), where we see a reload transient at the beginning of the second iteration of Process *A*. Before calling Process *B*, Process *A* fills the cache with various instructions and data that were referenced frequently and will be referenced frequently again. When Process *B* runs, it displaces many of *A*'s data and instructions in the cache with data and instructions that belong to *B*. When Process *A* reinitiates, it spends some time reloading the cache while displacing *B*'s lines. The shaded area shown in the figure represents this transient.

We show the transient occurring at the beginning of *A*'s second cycle. Actually it occurs throughout the cycle, with the initial miss-ratio quite heavy but gradually diminishing until the transient is over, or until Process *B* is reinvoked, whichever occurs first. The miss-ratio may be as high as 40 or 50 percent at the beginning of the transient and eventually falls off to a steady state of 1 or 2 percent.

The average miss-ratio over the period of Process *A*'s activity depends on

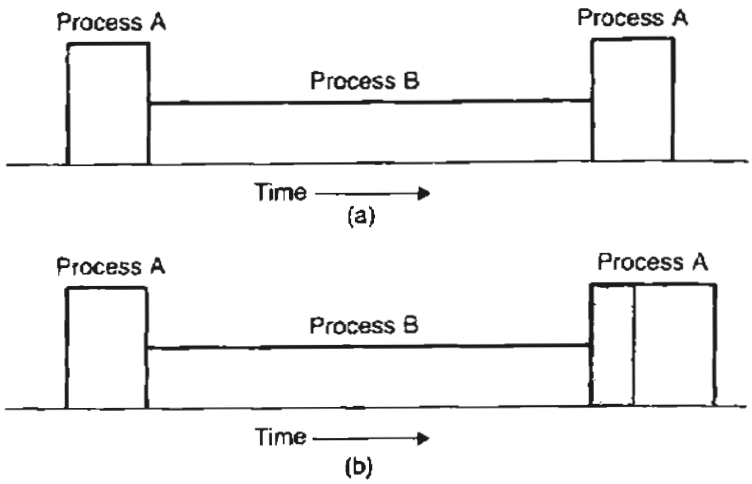

Fig. 2.18 Execution profile of two processes that share one processor:
(a) Ideal execution profile; and
(b) Actual execution profile of Process *A* when it contends for the cache with Process *B*. The shaded area denotes lost time from a cache reload when lines of *A* are displaced by *B*.

the relative size of the transient as compared to the length of the reference string for Process *A*. A similar transient not shown in Fig. 2.18 occurs for Process *B*. In fact, the lines belonging to *A* discarded by *B* are those that are reloaded by *A* when *A* takes control, so that the number of misses in *B*'s transient due to Process *A* is equal to the number of misses in *A*'s transient due to Process *B*.

The key to measuring the size of a transient is the notion of a *footprint*, as illustrated in Fig. 2.19. Figure 2.19 shows an *N*-set cache with potentially infinite associativity. The lines in the cache marked with an *A* are the lines that Process *A* touches when it runs in isolation. We call this set of lines the *footprint* of Process *A*, and the number of such lines is the *footprint size*. For fixed line size *L*, footprint size is fixed. That is, we can double or halve the value of *N* in Fig. 2.19, and the lines marked *A* will redistribute in the infinite cache accordingly, but no lines will disappear. The footprint shape changes with cache structure, but the footprint size is independent of *N*.

The double vertical lines in Fig. 2.19 isolate the four left-hand columns from the remainder of the cache, and show Process *A*'s footprint in a finite cache with set associativity 4. Note that some sets (rows) of the footprint contain more than four entries, and therefore these sets do not fit into the finite cache. When *A* runs by itself, these sets cause cache misses at a rate that depends on the frequency and exact sequence of references to those sets.

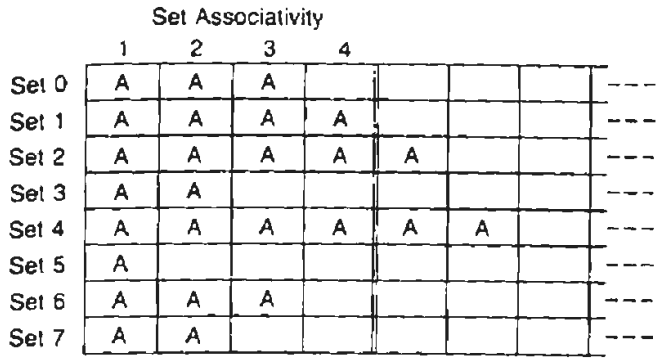

Fig. 2.19 The footprint of Process *A* in an eight-set potentially infinite set-associative cache. Three lines fall outside a 4-way set-associative cache and produce steady-state cache misses.

The model of cache behavior is very simple to state. We assume that Process *A* runs first and firmly implants its footprint in the cache. Then Process *B* runs. If Process *B* is agile enough to step around Process *A*'s footprint, then many lines of Process *A* will be resident when *A* restarts after *B* finishes. If Process *B* steps on part or all of Process *A*'s footprint, those lines from Process *A* will be displaced from the cache and will have to be reloaded when *A* restarts. How many lines will have to be reloaded? The relevant parameters are the footprint sizes of *A* and *B* and the cache size and structure. We show here how to estimate the number of lines to reload using statistical assumptions that turn out to be very good.

If the cache is very large compared to the footprint sizes of *A* and *B*, then with very high probability Processes *A* and *B* can run together without interference, just as two mice can ramble in a football stadium without bumping into each other. But if the cache is small relative to the footprint sizes, *B*'s footprint will land directly on *A*'s, and most or all of Process *A* will have to be reloaded.

The size of the footprint that Process *A* actually occupies in the finite cache is equal to the number of entries posted in the first four columns on the cache shown in the figure. How big is this footprint? We can estimate its size rather easily for a *K*-way cache by considering the probability of having more than *K* lines per set in Process *A*'s footprint for an infinite cache.

If we assume that the lines are distributed uniformly to the sets of the cache so that each set is equally likely to be the target of any line in the footprint, then the probability that the first line referenced by Process *A* falls into a set, such as Set 1, is $p = 1/N$, since there are N sets, each equally likely to receive this

line. The probability of not falling into this set is $q = 1 - p = 1 - 1/N$. Let the size of Process A's footprint in an infinite cache be S_A.

The model developed here is a binomial probability model in which we toss a coin with a probability $p = 1/N$ of landing head's up, which represents a line falling into Set 1. We flip the coin S_A times, once for each line in the footprint, and count the number of heads. The probability distribution of heads tells us the probability distribution of lines to Set 1. In an infinite cache, the distribution is given by the formula

$$Pr[i \text{ lines of Process } A \text{ in a set}] = \binom{S_A}{i} p^i (1 - p)^{S_A - i} \tag{2.4}$$

If the cache is finite, with only K-way set associativity, then Eq. (2.4) holds for $i < K$, and the probability of having K entries in a set is obtained by summing the probabilities in the tail of the binomial distribution. Thus we have,

$$\begin{aligned} Pr[i \text{ lines of Process } A \text{ in a set}] &= \binom{S_A}{i} p^i (1 - p)^{S_A - i} \text{ for } i < K \\ &= \sum_{j=K}^{S_A} \binom{S_A}{j} p^j (1 - p)^{S_A - j} \text{ for } i = K \end{aligned} \tag{2.5}$$

This is the probability distribution that we use in the remainder of the derivation.

Process B is governed by a similar probability distribution, except that its footprint size is S_B. For example, the equation corresponding to Eq. (2.5) for $i = K$ is

$$Pr[i \text{ lines of Process } B \text{ in a set}] = \sum_{j=K}^{S_B} \binom{S_B}{j} p^j (1 - p)^{S_B - j} \text{ for } i = K \tag{2.6}$$

Now we can estimate the cache reload transient. Figure 2.20 shows two possible states of the cache with both footprints resident. Figure 2.20(a) shows the cache in the state that exists when Process A runs first, then Process B, and we are about to reload Process A. The entries within a set (shown as a row in the figure) are ordered so that the most-recently used items appear on the left, and the least-recently used items appear on the right. All B's in this cache are to the left of all A's because Process B's references are more recent than Process A's.

Figure 2.20(b) shows the same cache in a state in which Process B runs first, then Process A, and we are about to reload Process B. In Fig. 2.20(a) the A's that appear in Columns 0–3 are lines that do not have to be reloaded when A is restarted. The A's in the other columns represent lines that are reloaded during the reload transient, and the number of such A's is the size of the transient. In Fig. 2.20(b), the reload transient for Process B is equal to the number of B's that appear outside of Columns 0–3.

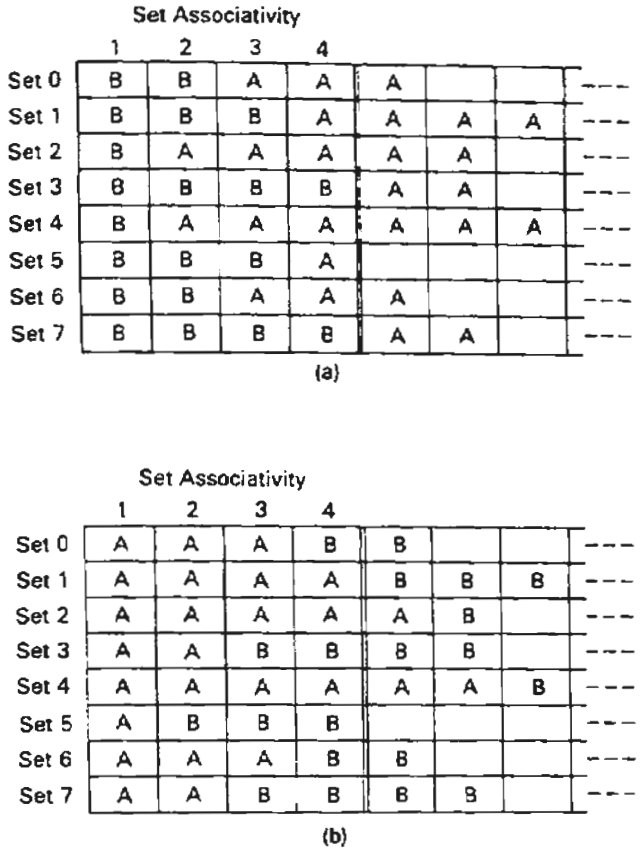

Fig. 2.20 The footprints of two processes that compete for the cache:
(a) An eight-set, four-way set-associative cache in a state obtained by running Process *A*, then Process *B* (the *A*'s to the right of Column 3 are the lines that form the reload transient);
(b) The same cache in a state obtained by running Process *B*, then Process *A*.

The binomial probability model makes the computation of the size of the transient quite straightforward. Let us focus attention on Set 1, since all sets are assumed to behave the same. There are three related random variables of interest to us for this set:

- X is the number of lines of Process *A*'s full footprint in this set;
- Y is the number of lines from Process *B*'s full footprint in this set; and
- $Z = X + Y$ is the total number of lines in this set.

If Z does not exceed K, then Set 1 contributes nothing to the reload transient. The probability of this event is the probability

$$Pr[Z \leq K] = \sum_{i=0}^{K} \left(Pr[X = i] \sum_{j=0}^{K-i} Pr[Y = j] \right) \tag{2.7}$$

If X and Y are binomially distributed, both with probability $p = 1/N$, then Z is also binomially distributed. That is, Eq. (2.7) is the probability of having K or fewer heads among $S_A + S_B$ coin flips. Hence, Eq. (2.7) is the area under the tail of a binomial density. For values of p near 0.5, Eq. (2.7) is closely approximated by a normal distribution. For the values of p of interest to us, however, Eq. (2.7) is only crudely approximated by a normal distribution, although the general shape of the curve is the same.

The interesting situation occurs when Z exceeds K. Let W be the number of lines of Process A that are overwritten by B in Fig. 2.20(a). Then the probability that exactly i lines of Process A are overwritten is given by

$$Pr[W = i] = \sum_{j=i}^{K} Pr[X = j] Pr[Y = K + i - j] \text{ for } 1 \leq i \leq K \tag{2.8}$$

Each term in the summation of Eq. (2.8) accounts for a case in which precisely i lines of Process B fall on i lines of Process A in a K-way cache. Note that the first and last terms of the summation involve summations from Eqs. (2.5) and (2.6).

To compute the cache-reload transient from Eq. (2.8), we note that the transient to reload Process A is S_A minus the number of lines of Process A left in the cache when A resumes. This number is given by

$$\text{Cache-Reload Transient} = S_A - N(E[X] - E[W]) \tag{2.9}$$

The term in parentheses is the expected number of lines from Process A remaining in each set of the cache. The term is equal to the number of lines in the full footprint reduced by the number of lines overlaid by Process B.

Figure 2.21 shows an example of the cache-reload transient for caches of various sizes and structures. This figure is based on actual data, and the curves produced by the model have been confirmed in practice up to the ability to determine which misses are part of the reload transient and which are not.

The shape of the curve is rather interesting because it is similar to the appearance of the area under the tail of a normal density function. We would obtain that curve exactly if the binomial parameter $p = 1/N$ were not so small, and if the only lines in Fig. 2.20(a) that lie outside the first K columns belonged to Process A.

Note that, for a fixed cache size, the curve becomes steeper as set associativity increases. There is a threshold phenomenon displayed here. If the cache is sufficiently large to hold Processes A and B concurrently, the reload transient is very small. If the cache cannot hold both processes comfortably, they conflict

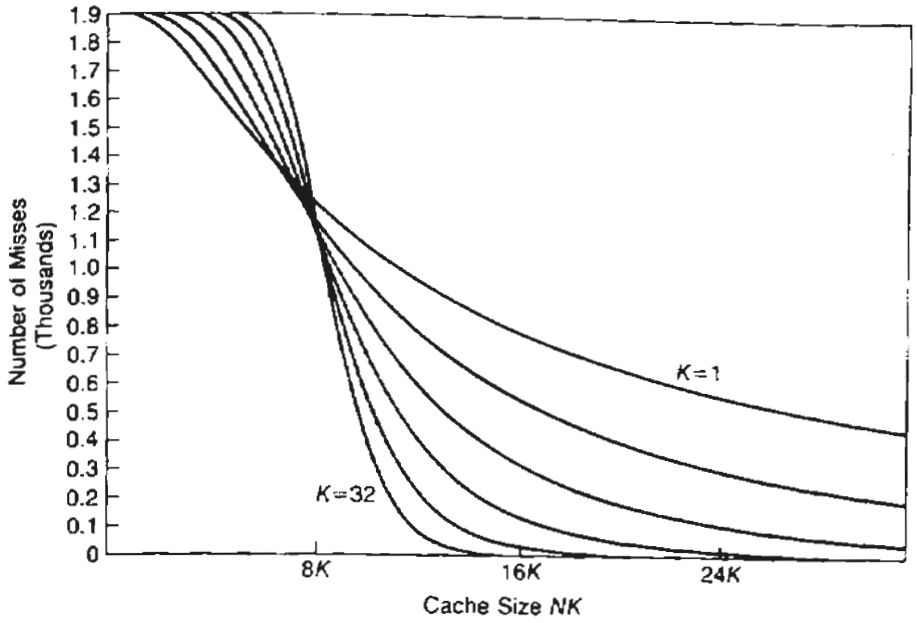

Fig. 2.21 The cache-reload transient (Program *A* footprint = 1900; Program *B* footprint = 7900).

with each other. When a cache is smaller than the footprint of Process *B*, when Process *B* reloads the cache it tends to displace Process *A* almost completely, and the displacement is greater as the associativity becomes greater.

This particular model has been successfully used to select a cache size for a computer system in which an interrupt-driven process had to remain cache resident between interrupts. The interrupt-driven process had to execute its task in real time, and could not pay a large cache-reload penalty. The objective was satisfied by making the cache large enough to hold both the interrupt-driver and the background process so that both could run with the full benefit of cache.

The probabilistic model gives better answers than does a deterministic model because the probabilistic model shows the effects of different background processes on the reload transient. With this model we can obtain fairly accurate estimates of the reload transient under the most adverse conditions likely to be encountered, as well as for typical conditions, and thereby have a very good estimate of the real-time performance of the interrupt-driver.

The model is also useful for explaining cache behavior in ordinary programs. Processes *A* and *B* in Fig. 2.20 might well be two processes within one program that are executed alternately. Figure 2.21 shows that the benefit of a large cache falls off fairly rapidly when the cache is big enough to hold contending processes.

The cache-design question centers on how large to make the cache so that contending processes do not step on each other. Since the footprint size is the critical parameter, the distribution of footprint sizes of processes within programs gives valuable information regarding how large to make caches. The architect should measure footprints for a variety of subroutines, inner loops, and other identifiable processes, especially processes that are invoked frequently.

We also need to know the cumulative sum of the footprint sizes of processes invoked between successive runs of a given process. With such information the architect can develop a model for cache transients that gives an estimate of total performance as a function of cache size. This can be used for gross estimates before detailed estimates are produced from simulation experiments on long traces.

2.2.7 Writing to the Cache

The discussion up to this point has not mentioned any special actions to take for WRITE operations, whether they hit or miss in the cache. Handling the WRITE operations is somewhat tricky because of the interaction of the cache with the input/output system. Figure 2.22 shows typical organizations of processor, cache, and input/output processor.

Figure 2.22(a) shows an organization in which all references, whether from the input/output processor or the central processor, go through the cache. This scheme is seriously flawed because there is too much activity in the cache. The two ports to the cache require interlocks and arbitration, which tend to affect performance adversely.

The scheme shown in Fig. 2.22(b) is definitely preferable to that of Fig. 2.22(a) because the central processor and the input/output processor do not conflict with each other on the majority of the accesses. The central processor operates mostly with the cache memory, and independently, the input/output processor operates mostly with main memory.

Although the latter scheme is good from a performance view, it is not good from the view of logical consistency unless we embellish the scheme in some way. The problem is that each item has two places where it may be resident—main memory or cache memory. If the item is in both places, the two values must be identical. If ever the values are not identical, then we can have a situation in which the processor accesses the cache to find one value for the item, while the input/output processor accesses main memory and discovers a totally different value. We must forbid this situation from happening, and, in so doing, some designers have opted to implement the organization of Fig. 2.22(a), which solves the problem directly.

Figure 2.23 shows one way to approach the problem. The idea is to have two copies of the cache directory, one read by the central processor and the

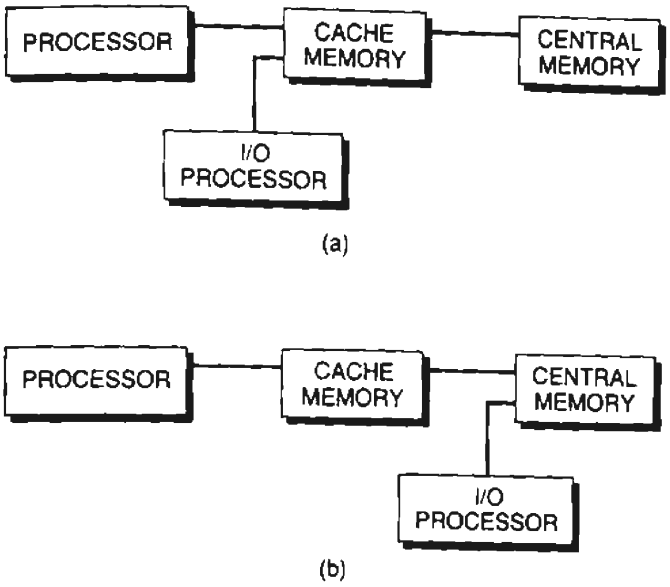

Fig. 2.22 Two possible ways of organizing a cache memory with respect to an I/O system: (a) The cache multiplexes requests from the I/O processor and central processor; and (b) The I/O processor has a direct path to memory. This scheme requires interlocks between the cache and the I/O system.

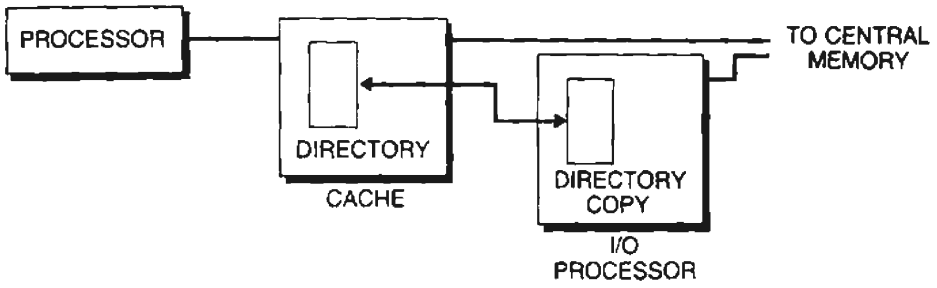

Fig. 2.23 A system organized with a direct route to memory for the I/O processor. All changes to the cache directory are maintained in a copy in the I/O processor. The I/O processor invalidates entries in the cache directory when the entries are updated by an I/O operation to central memory.

other read by the input/output processor. With two separate copies, each processor can read the directory without interfering with the other processor.

Figure 2.23 shows the directories resident in two physically separate regions of the computer system, but they obviously can both be resident in the cache. The cache directory is read for every READ or WRITE operation, but the directory is changed only when a miss occurs. Since this happens rather rarely, roughly every 25 to 100 memory operations, there is very little overhead from contention between the central processor and the input/output processor. The key idea is to make sure that the input/output processor always reads the correct datum, and that every datum written by the input/output processor to main memory is made available to the central processor.

Let's consider the details of operation of Fig. 2.23. WRITE operations are tricky to handle in this structure; READ operations depend on how WRITEs are implemented. If the input/output processor writes an item to main memory, it must also check to see if the item is also in the cache. If so, the input/output processor should invalidate the cache entry to be sure that the central processor will access main memory when that item is next requested. Otherwise, the central processor might discover an out-of-date value for the item if the processor happens to find the item in the cache. Thus the input/output processor invalidates the cache entry for each cache hit it observes while writing new data in main memory.

Another possible strategy is to rewrite the new data to the cache instead of invalidating the data. For most systems, however, the probability of that update leading to a cache hit for the processor is rather low and does not justify the update of the cache.

The central processor actually has two different ways of handling WRITE operations, both of which have been implemented in commercial machines. One method is called *write-through*, in which every WRITE operation to the cache is accompanied by a write of the same data to main memory. If this is implemented, then the input/output processor need not consult the cache directory when it reads memory, since the state of main memory is an accurate reflection of the state of the cache as updated by the central processor. Although this scheme simplifies the accesses for the input/output processor, it does result in fairly high traffic between central processor and memory, and the high traffic tends to degrade input/output performance.

A different scheme is sometimes called *write-back* or *write-in cache*. In this scheme the central processor updates the cache during a write, but the actual updating of memory is deferred until the line that has been changed is discarded from the cache. At that point the changed data are *written back* to main memory.

The advantage of the write-back policy is that updates of main memory can be avoided completely whenever a cache line experiences a WRITE hit. The disadvantage is that an entire line is written back from cache to main memory when the line is eventually replaced, but a write-through policy usually writes

only the bytes within a line that are addressed by the WRITE operation. When a line size is long enough to require several bus cycles, a single write-through operation will require less bandwidth than the bandwidth required to write back a full line. We expect that the reduction in the number of updates under the write-through policy is much greater than the extra bandwidth required to update a full cache line. But this assumption has to be examined carefully for designs that involve small caches or large line sizes.

Because the input/output processor must be informed of WRITEs to the cache, just in case an input/output operation has to move such data from the computer system to an external device, the input/output processor must consult a cache-directory copy when it reads an item from main memory. If there is a hit, the input/output processor requests the item from the cache. Note that it is not necessary to update the cache directory read by the input/output processor on every WRITE by the central processor; it is sufficient to change this directory only when the main cache directory changes, which occurs on every miss, not on every WRITE.

2.2.8 Other Cache Metrics

The connection between main memory and processor is by means of a high speed bus. Because the bandwidth is finite and because it may at any given time have to respond to competing requests, there is a possible performance degradation due to contention on this bus. For example, in the previous section we learned that WRITEs to caches can be treated in different ways. In this section we examine schemes for evaluating performance degradation due to bus traffic caused by WRITEs and by multiple bus cycles when line size is a multiple of bus width.

The two possible ways to treat WRITEs in cache raise a question of evaluating the impact of these alternatives on system performance, and the main difference is attributed to the difference in bus traffic produced by the two schemes. The finite bandwidth of the bus also produces degradation on READ misses because the various components of a long cache line arrive at cache from main memory in a sequence of cycles. It is possible to generate a series of misses to the same cache line in such circumstances when our model to this point predicts that after the first miss the subsequent references to the same line will generate hits. This section considers the mechanisms that impact performance, and indicates how the cache simulation techniques can be extended to measure their effects.

The effect of the WRITE policy is the starting point of this discussion. What is the difference in the WRITE traffic generated by a write-through policy as compared to that generated by a write-in cache policy? Every item in cache that has been altered while resident there must be rewritten to main memory before the item is removed from cache. The write-through policy copies the item to main memory as early as possible and the write-in cache copies the item as late

as possible. The latter policy avoids some traffic when it modifies items in cache multiple times before they leave cache. The earlier modifications need not be reported to main memory. But the write-in policy requires more cycles per individual update because it updates a full line of memory rather than just the part of the line modified by a WRITE.

The difference in the traffic produced by the two policies can be computed by answering the following questions:

1. What is the WRITE rate of the workload, measured in the number of bus cycles per instruction devoted to modifying data?
2. What is the rate at which WRITEs hit modified data in the cache, measured in the number of bus cycles per instruction required to rewrite modified data hits?

The first measure is the bus traffic generated by writes in a write-through cache policy. The second measure counts the reduction in the number of bus cycles of write traffic that are available from a write-in cache policy. To capture the first set of performance data, a simulator counts bus cycles of write traffic for each instruction simulated. If a reduced trace is used for input to the simulator, then as part of the trace reduction process when a sequence of references is removed from the full trace, a special record that contains the sum of the write bus-cycles produced by the discarded references is written in its place. Since these are hits in a one-way set-associative cache, the sums of such records contribute to the measures for both the cycles expended for write-through policy and the cycles deducted from that measure for the write-in policy.

To capture the second set of performance data, it is not immediately clear that one-way, two-way, and four-way caches can be simulated on one pass of the trace as suggested by the method of Mattson *et al.* [1970]. A little reflection shows that the basic algorithm of Mattson *et al.* is not sufficient. Suppose, for example, that a modified item is read and a hit is recorded in a four-way cache, but not in a cache with less associativity. The one-way and two-way caches will not contain the item so that they will retrieve a clean copy of it and put it in the most recently used position of the cache. But the four-way cache produces a hit and moves the modified item to the most-recently used position of the cache. A subsequent write to the item that produces a hit cannot tell if the item is modified (moved from the four-way cache on a hit) or unmodified (copied from main memory on a miss in a one-way or two-way cache). Thus it is not clear how to account for bus cycles saved by writing to a modified item.

The problem was neatly solved by Thompson and Smith [1989] who show that it is sufficient to keep track of the stack depth at which an item is written. In the preceding example, if a modified item is the target of a read hit for a stack depth of 4 (four-way cache), place the number 4 with the item as a reminder of where it came from. Then when the subsequent write-hit occurs, we know

that bus cycles are saved for four-way caches but not for caches with less associativity. To treat all cases correctly, the tag stored with a modified item is set to 1 when the item is written. The 1 signifies that caches with an associativity of 1 or more must write back this item when it leaves cache. Otherwise the tag is made equal to the stack depth at which the item is hit by a read hit if the stack depth of the hit is greater than the current tag stored with the item. This signifies that caches with an associativity as large as the tag retrieve a modified version of the item from the cache, and that the item has to be written back to storage when it is flushed eventually. Caches with an associativity less than the stored tag value suffer a cache miss and reload clean copies of the item that do not have to be rewritten to memory. When a write hit finds an item, assume that the value of the tag it contains is k before the tag is updated. Then the bus cycles caused by a write modification are avoided by a write-in policy for all caches with associativity k or more. The simulation maintains a vector of running sums and updates the item at position k in this vector. At the close of the simulation, the number of cycles saved in a k-way cache is the sum of the components with indices k or larger. This is another application of the stack replacement technique discovered by Mattson *et al.*

Having discussed the need for measuring the bus cycles produced by WRITEs, we recognize that we cannot strip WRITEs arbitrarily from traces. Do we have to retain all WRITE hits on traces, or are we permitted to strip some during the process of preparing tapes for simulation? Wang and Baer [1991] discovered that the filter can strip out all WRITE hits except those that modify data in cache for the first time. This depends on the fact that the cache used to filter out references from the stripped trace is a direct-mapped cache. No other WRITEs that hit in the direct-mapped cache change the value of the tag required by the Thompson-Smith algorithm, so that they can be discarded without changing either the number of misses or the number of cache writebacks experienced during a cache simulation.

Apart from the rate at which bus requests are issued, there is an additional performance concern regarding the time required to reload a cache line from main memory. The line size of a cache line need not be identical to the bus width between main memory and cache. It is usually a multiple of the bus width, and varies from one to eight times the bus width in practice, but could be larger in principle. Consider an extreme example in which the factor is eight so that at least seven additional bus cycles elapse after the first part of the line reaches cache, and before the last part of the line reaches cache. Assuming that instruction execution suspends momentarily on a cache miss and restarts when the first part of a cache line reaches cache, during the next seven cycles it is quite likely for another reference to the same line to occur, possibly to a part of the line that has not yet reached the cache. Such a reference also causes a processor idle period, but the idle period lasts only until the missing line elements reach the cache. Since the access to those elements is in progress, the

wait is not as long as the wait for the first part of line. We distinguish the performance degradation due to the two kinds of misses by calling the wait associated with a normal miss the *leading-edge effect* and the wait due to line transfers in progress the *trailing-edge effect*.

With very little additional work we can compute the additional degradation due to the trailing edge effect, which we have ignored up until now, while computing the leading edge effect, which has been the primary subject of the discussion. The trick here is to produce a stripped trace with sufficient additional information to compute the trailing edge effect for a variety of bus widths. We record on the stripped trace, not just the misses to a one-way set-associative cache, but we also record summary information for the references that have been stripped from the trace. For each miss, the cache simulator stores with the miss the simulated time of the miss. Any subsequent hit to that line that occurs within a fixed period of time might result in a delay that contributes to the trailing edge fact. So when such a hit is detected the simulator writes summary information to the stripped trace. When the stripped trace is used as an input trace for cache evaluation, the summary information should be detailed enough to compute the additional performance degradation due to the trailing edge effect.

The trailing-edge effect and the bus traffic for the write-in cache policy are just two of several fine details of processor performance that merit attention. From the discussions in this chapter regarding techniques for measuring performance, the reader should have no difficulty adapting these techniques or developing similar new techniques to other aspects of performance not covered in this text.

2.2.9 Modeling System Performance

With the various techniques described thus far at our disposal, how do we put them to use in the design of high-performance systems? Ultimately, the goal is to compare on the basis of cost and performance. We can estimate the cost of cache fairly easily because it is proportional to the total number of bytes in cache. The proportionality constant is a function of the device technology used in the cache design. Performance is somewhat trickier to measure in the context of complex systems. This section describes how to construct a performance model of a computer system that helps to quantify the effect of cache behavior on total system performance.

The usual starting point for a performance measure for computer systems is *MIPS*, Millions of Instructions Per Second of computer execution. To relate MIPS to cache performance, we have to make assumptions regarding memory accesses. Let's suppose that we analyze a number of benchmark programs for a computer system and discover that each instruction produces 1.85 memory accesses on the average. To a first approximation, the number of MIPS for the

system is the number of memory references per second divided by $1.85 \cdot 10^6$. The number of memory references per second is the reciprocal of the effective cycle time given at the beginning of this chapter in Eq. (2.1). Hence, the bulk of the effect of cache is wrapped up in a very simple formula. But there are several other factors that require some deeper investigation.

A major perturbation to this model is a processor design that uses two caches, one for instructions and one for data. Figure 2.24 depicts this type of structure. It is very common in current processors. The two caches are independent, and each can respond to one request per cycle. As long as the processor generates one instruction request and one data request each cycle, the memory system produces two results per cycle. Equation (2.1) expresses the average time per memory request and has to be modified to reflect that there are two requests completed per cycle, not just one. The completion of two requests per cycle reduces effective cycle time by a factor of 2, and doubles the MIPS rate of the processor. This is the actual effective cycle time in the absence of misses, and can be viewed as the performance rate for a machine that has an infinite cache and no input/output overhead. Finite cache size produces misses, and misses slow the processing rate. Input/output occupies some bandwidth of main memory, which in turn produces a likelihood of contention between a processor and an input/output controller as the processor seeks access to main memory.

It is convenient to cast a performance model into one in which the various effects that lower performance enter the model through simple additive terms. A popular approach is to measure performance in CPI (Cycles Per Instruction). This is independent of clock speed, so it is a fairer measure of the architectural component of a processor than is MIPS. Recall from Chapter 1 that the relation definition of MIPS as a product of two factors:

$$\text{MIPS} = \left(\frac{\text{instructions}}{\text{cycle}}\right)\left(\frac{\text{cycles}}{\text{second}}\right) \cdot 10^{-6} \qquad (2.10)$$

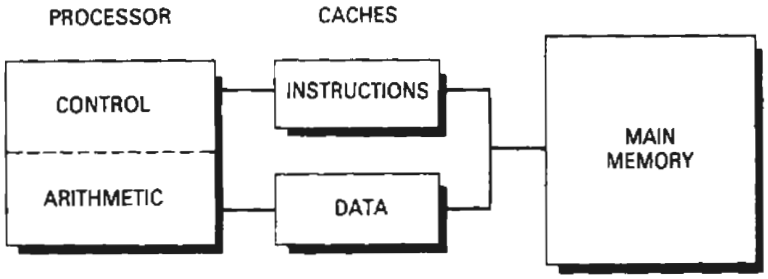

Fig. 2.24 A computer system with separate caches for data and instructions.

The factor "cycles per second" is a measure of the clock speed, and the factor "instructions per cycle" is 1/CPI. CPI in turn is computed by measuring the value under an ideal set of conditions and adding to this the effects of finite cache size and finite memory bandwidth. We can express CPI as

$$CPI = CPI_{Ideal} + CPI_{Finite\ Cache} + CPI_{Trailing\ Edge} + CPI_{Contention} \quad (2.11)$$

At this point we present approximate models for each of the other terms. These are generic and are intended to be representative, but the models need to be fitted and refined to reflect the actual characteristics of specific designs. This discussion raises the issues that need to be addressed by the performance models, but we are unable to produce a universal formula because there is no single model that fits all designs. The designer has to build a model with each design to express the ideal CPI and the various increments to CPI from different sources of performance degradation.

The ideal case assumes a cache hit on every memory access. The finite cache term charges an access delay for each miss. Measure the miss rate on typical workloads, and examine the architecture to find the penalty per miss. The miss rate has to be normalized to misses per instruction to be valid in this performance model. For example, a miss rate of 1 percent per memory access is a 2 percent miss rate per instruction in an architecture that produces two memory references per instruction. For a miss penalty of 10 cycles per miss, this term adds 0.2 cycles per instruction to the CPI.

Suppose those two references per instruction are done concurrently because the architecture has separate instruction and data caches, and supports a reference to each in each machine cycle. The miss penalty per miss may be different if both references miss in the same cycle, than if one records a miss while the other records a hit. If both miss in the same cycle, it may be possible to restart the processor before paying an access time delay for two misses. Then instead of charging a fixed penalty for each miss, the penalty is reduced for those cases in which the instruction fetch and the data access both miss in the same cycle. The frequency of this occurrence can be measured from detailed processor simulations of trace data, or through measurements made in real-time on machines in execution.

This discussion points out that the additive terms in the CPI formula are not necessarily additive in reality. The underlying assumption for the use of additive terms is that the delays associated with different components of CPI cannot be overlapped. If two or more of the delays occur concurrently, then they may be fully or partially overlapped, and the degradation is less than the figure produced by the CPI equation.

The access delay between the occurrence of a cache miss and the retrieval of the data from main memory is the leading-edge effect. There is also a trailing-edge effect when the line size is a multiple of the memory-bus width. In this case, the line is transmitted by breaking it into pieces, each of which is trans-

mitted in one bus cycle. When the first portion of a line arrives at the processor, the bytes that arrive are those bytes within the line that were requested by the instruction that missed. This allows the processor to restart without waiting for the entire line to arrive. The remaining bytes arrive closely spaced in time on subsequent cycles.

If the processor tries to access another byte in the line, it will experience another miss if that byte is still in transit. This miss will be resolved fairly quickly because the byte has been already retrieved from storage and is currently in some buffer or on its way to the processor. The additional delay experienced for these data is much less than initial delay accounted for by the leading-edge effect.

To compute the penalty due to the trailing edge effect, use a detailed simulation of typical workloads, find how many extra cycles are expended waiting for the trailing edge of a line to arrive, and then determine the frequency of delays due to this effect. For example, assume that the average penalty is 4 cycles per event, and the event occurs at the rate of 1 per thousand instructions. Then the CPI attributed to the trailing-edge effect is 0.004 cycles per instruction. As line size of a cache increases, the trailing effect increases for at least two reasons. The penalty is greater because it takes more cycles to load a line when the line size increases and the bus size remains fixed. The probability of experiencing a trailing-edge delay is higher because more instructions are executed during the time a line is in transit since the length of the transit time is longer. Detailed simulations need to look carefully at this problem when line sizes are sufficiently long to require eight or more clock cycles to transfer a line between main memory and cache memory.

The CPI penalty due to contention accounts for the increase in access time to main memory caused by contention between processor requests and input/output requests at main memory. A processor request that attempts to access main memory while an input/output operation is active will reach a point where the processor request has to be queued pending the completion of the input/output operation. The delay experienced while the request is queued increases the CPI if the processor cannot continue until the memory request has been completed. This is the usual situation for READ misses. A processor need not wait for WRITE misses, so that memory contention experienced while satisfying WRITE misses does not directly impact processor performance, but the contention can interfere with input/output performance. The processor can be delayed by contention on WRITEs when they are delayed sufficiently long to cause internal processor buffers to fill and prevent further activity until the WRITE completes and breaks the data logjam.

To find the CPI penalty for contention, one follows the familiar pattern mentioned above. That is, run detailed simulations that keep track of all pertinent factors and obtain the data from the simulations. In this case the simulation has to keep track of input/output operations and the memory traffic that they gen-

erate. An alternative method is to monitor a system in execution and obtain the values of the parameters from measurements made in real time. Find the average number of cycles lost per contention event, and find the average number of contention events per instruction. If measurements reveal 250 instructions per contention event, and each contention event contributes an average of five cycles of additional delay, then the increment to CPI is 5/250, or about 0.02 cycles per instruction.

All of the preceding example data are exemplary but are not necessarily typical of any system. A workload oriented to visual data, graphics, animation, and to communications-intensive applications, in general, experiences a great deal more contention than a workload that is engaged in extensive floating-point arithmetic operations on data that fit into cache memory.

Systems with large line sizes tend to experience greater contention delays than systems with short line sizes. For a design with large line sizes, cache data transfers occupy a greater proportion of available memory bandwidth because the cache line transfers take more cycles than for systems with short line sizes. With greater fraction of memory bandwidth devoted to cache reloads, input/output operations are forced to fit within a smaller fraction of the cycles available, and hence it becomes more likely for the processor to experience contention with an input/output operation in progress.

When a designer wishes to explore several different alternatives, the designer can calculate a CPI for each design. If the cycle rate for all alternatives is equal, the CPI by itself gives the relative performance of the several alternatives. If some alternatives support a faster clock (more cycles per second), this has to be weighed into the evaluation. The designer has to have good cost-estimates for the alternatives, and from there can compute relative cost-performance for each possible choice. The process of performing a detailed comparison of alternatives is conceptually simple, but it is far from simple to carry out.

The hardest step to take is the step that involves measuring typical workloads to obtain values for unknown model parameters. What is a typical workload today? Is that workload typical of future workloads? If you design a machine based on data from today's workloads you should be able to produce a machine capable of performing well on today's workloads, but how well will it do on workloads of the future? For example, the future evolution of digital communications, image manipulation, multimedia, and digital video will place vastly different demands on machines than the workloads characteristic of the past. Can machines designed for today work efficiently in such environments? New generations of old machines can completely miss new markets if their designers fail to address the next generation of applications.

The common thread to performance evaluation so far has been the use of traces of real workloads. The next section addresses what to do in the absence of such traces.

2.2.10 Modeling Cache Behavior

One important trend produced by the development of VLSI technology is the reduction in the number of different processors in wide use. When a single chip contained only a few logic gates or registers, designers put them together in a variety of ways to form different kinds of processors with different instruction sets. VLSI changed the design rules somewhat by offering computer architects entire processors on a single chip to use as building blocks in their designs. Moreover, the cost of those processors drops very low when the volume of production is very high, so that there is an incentive to incorporate widely used processors in a design. Another advantage is that a widely used processor usually has a large base of applications and systems software already in place. Hence, the cost-performance and marketplace trends are driving toward a state in which relatively few different processor types are used in the majority of computers sold.

In spite of this trend, there is also an incentive to produce novel machines that incorporate advances in some form or other. The appearance of reduced instruction-set computer (RISC) architecture with its potential reduction in cycle time resulted in the development of very high speed processor chips from each of several manufacturers. Meanwhile, more conventional processors evolved in an orderly fashion that brought along enormous changes in the typical workload on such processors. The greatest impact in workload change has been caused by the increase in address space and basic processor speed. For example, in the Intel family of microprocessors, the 2 MHz 8080 became the 5 MHz 8086, the 10 MHz 80286, the 20 MHz 80386, and the 40 MHz 80486 in successive generations. A significant change between the 8080 and the 8086 was the increase of the memory address from 16 bits to 20 bits, and an even more dramatic change was the increase to the 32-bit address of the 80286 and 80386. Even though the 8080 ancestry is quite evident in an 80486, there is very little in common in the typical workloads of an 8080 and 80486. The differences between two successive generations may be very large, as indicated in this example by the differences between an 8086 and an 80286. When such differences exist, workloads for the present generation are not likely to capture the features of workloads likely for the next generation, and thus cache designs based on present workloads may not perform as predicted on the workloads of the next generation.

Most RISC processors and the later chips in the 80X86 family are designed to work with caches. Many RISC processors and the 80486 processor have caches on-chip, and provide for larger second-level caches off-chip. No real workloads existed for these chips during their design. How were the caches for these processors designed? They were not designed by the trace-driven techniques presented earlier in this chapter. They had to be designed by estimating the performance of various cache structures on the projected workload. What can you do to estimate performance when you cannot perform detailed simulation?

We would like to have in hand a general method for estimating miss ratios as a function of cache size and structure on which we can rely for crude but close estimates of performance in the absence of precise data. Figure 2.25 illustrates a model due to Thiebaut [1989] that can answer many of the questions posed. Recall that the footprint of a process is the number of unique cache lines touched by the process. The function plotted in Fig. 2.25(a) is the footprint function for a workload as a function of time. The first time that each line is accessed, it increases the footprint function by one. The function is essentially the number of misses in an infinite cache as a function of time.

Figure 2.25(a) is plotted on a log/log scale. Notice that it is composed of two straight lines—an initial line with a steep slope and a steady-state line with a gentler slope. Thiebaut observed this behavior when analyzing a number of different processes on a number of different machines. Independently the same observation was made by Kobayashi and MacDougall [1989] for seven different workloads on a 370 architecture. The dotted line at the right end of the curve shows a trend in the data of Kobayashi and MacDougall where the footprint function tapers off. Their data may cover just the initialization part of the curve plotted by Thiebaut, and it may be possible that the dotted line shown in Fig. 2.25(a) itself has a long-term straight trend, but the paper does not give sufficient information to determine if this is the case.

In Fig. 2.25(a) in the regions where the footprint function is approximated by straight lines the footprint function $u(t)$ (for *unique* references) obeys the power law

$$u(t) = At^B \tag{2.12}$$

for some constants A and B. B determines the slope of the curve and A determines the y-intercept. Thiebaut's model indicates that the footprint function can be used to predict miss ratios of fully associative caches.

The idea behind the model is illustrated in Fig. 2.25(b) where the same function is plotted on linear axes to show the curve of the power-law function. Thiebaut claims that the miss rate of a fully associative cache of size C lines is the derivative of the footprint function (measured in lines) evaluated where the footprint function takes on the value C. The reason as shown in the figure is that at this point in time a fully associative cache with C lines has just filled. When the next unique reference appears, this cache will experience a miss. Hence, the instantaneous miss rate at this point in time is the slope of the footprint function.

Now consider the future. At any time in the future, the fully associative cache with C lines produces a hit if a new reference is among the lines in the cache and otherwise produces a miss. The miss rate in the future is assumed to be a function only of the size of the cache. Consequently, the future miss rate of the cache is equal on the average to the instantaneous miss rate at the point it first fills since it depends only on the fact that C lines are in the cache

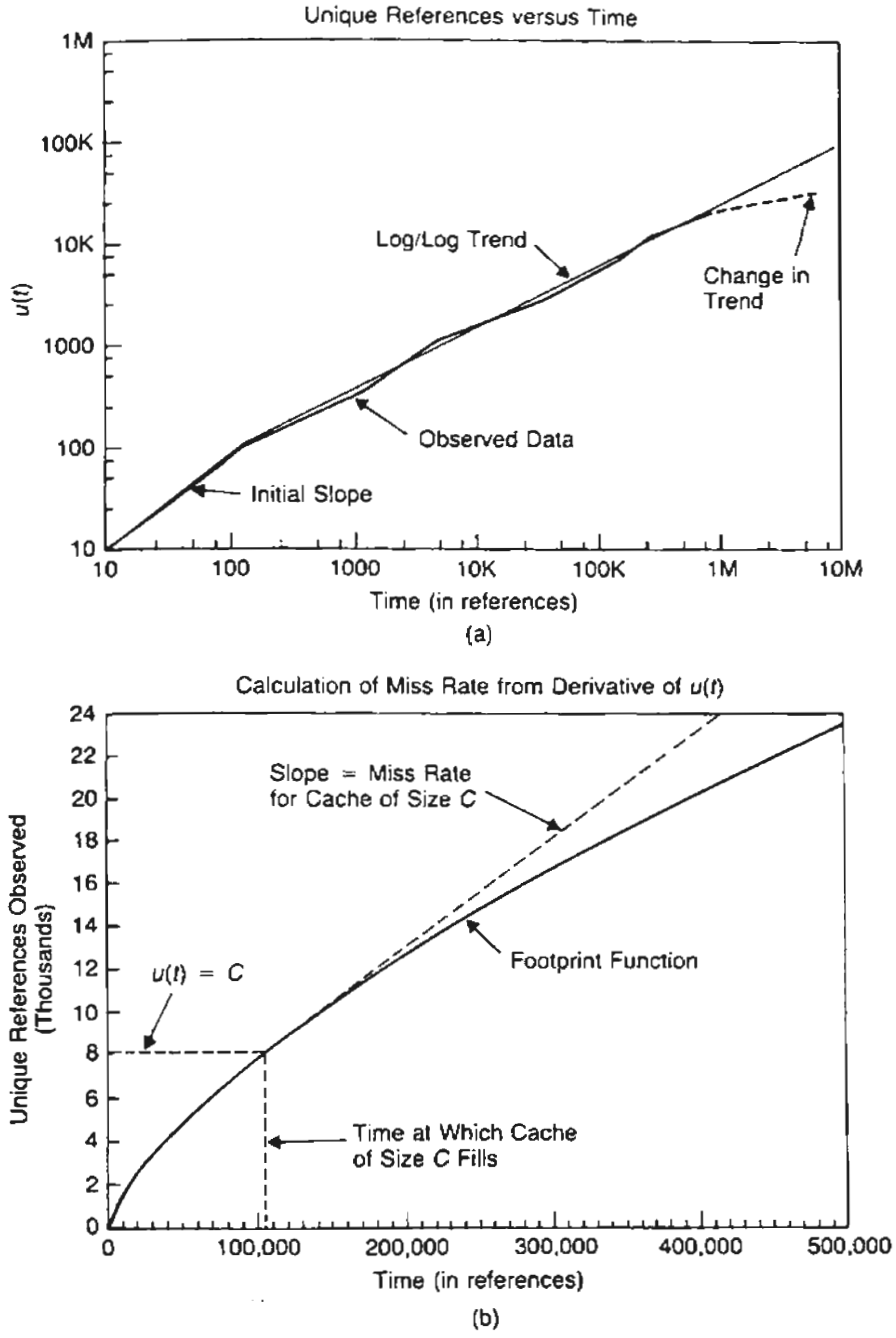

Fig. 2.25 (a) The footprint function as a function of time plotted on a log/log scale. (b) The same footprint function plotted on linear axes to illustrate its relation to miss ratio.

memory. When you work through the calculus, you discover that the miss rate of a cache of size C is:

$$\text{Miss Rate}(C) = BA^{1/B}C^{1-1/B} \tag{2.13}$$

for a footprint function given by Eq. (2.12). The power law in Eq. (2.13) is the general form of the 30-percent rule we have informally used throughout this chapter. Specifically, the 30-percent rule holds when B has the value 0.6603, at which point the exponent of C in Eq. (2.13) has the value -0.5146. When cache size C doubles to $2C$ the number of misses is multiplied by $2^{-0.5146} = 0.700$, for a 30 percent reduction in miss rate. Thiebaut reports values of B (the reciprocal of his coefficient θ) that range from 0.484 to 0.544, and produce, respectively, a 52 percent and 44 percent reduction in cache misses for each doubling of cache size for his sample workloads. Although Thiebaut's workloads show greater locality than indicated by the 30-percent rule, Kobayashi and MacDougall's values of B range from 0.43 for scientific workloads to 0.75 for supervisor workloads, which is a somewhat larger variation than observed by Thiebaut. These exponents produce, respectively, 60 percent and 21 percent reduction in miss ratios for each doubling of cache size. This range of reductions brackets the 30-percent rule and the observations of Thiebaut.

In essence, the slope of the curve of the footprint function is a measure of the locality of references of the workload. If the slope is steep, the workload touches many new items per unit time. If the slope is shallow, the workload tends to touch a greater proportion of items seen in the past. The coefficient of 0.43 for scientific workloads is very shallow and indicates that cache is very effective for such applications.

Since Fig. 2.25(a) shows the footprint curve to be composed of two different straight lines, the derivative of the footprint curve plotted on log/log axes also consists of two straight lines as indicated in Fig. 2.26. For small caches, the process has a high miss rate that is not strongly affected by cache size. As the cache becomes large, the slope steepens, and the miss rate changes more rapidly with cache size. This exemplifies the *working-set* model described later in this chapter, and is originally due to Denning [1968b]. The working set of a process is some minimal set of lines that have to be resident in fast memory in order for the process to execute mostly out of fast memory. Until the full working set is resident in fast memory, the process experiences a high miss rate. The miss rate drops quickly when the full working set resides in cache. If this is the case, then the intersection point in Fig. 2.25(a) of the straight lines occurs when the number of unique lines in the footprint function approximates the working set size.

Singh, Stone, and Thiebaut [1992] refined the model further to show how miss rate depends on line size as well as cache size. Figure 2.27 shows the general form of the curves they derived. The footprint functions plotted in Fig. 2.27(a) show the footprint functions for different line sizes plotted as a function

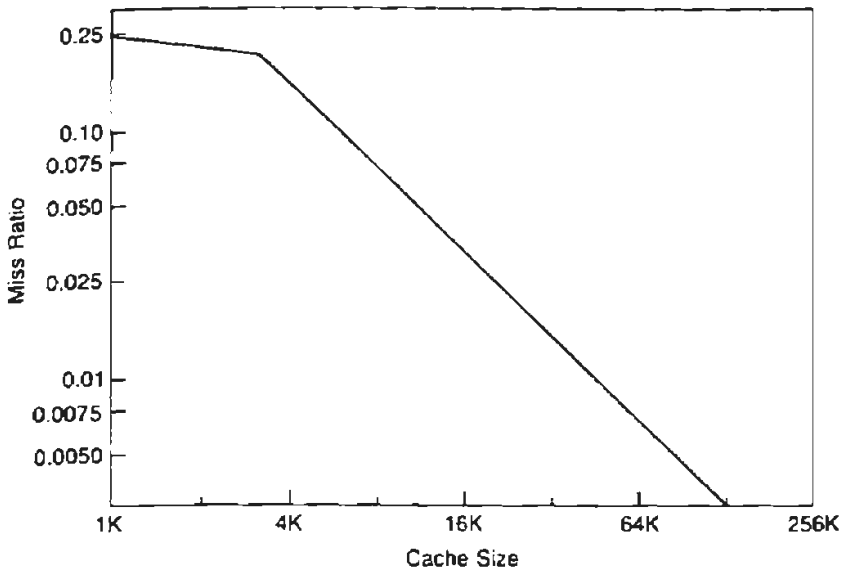

Fig. 2.26 The time derivative of a footprint function plotted on a log/log scale.

of time. Note that each footprint function has a straight-line trend on the log/log scale. The slopes of the footprint functions as a function of line size are related in a way that makes the footprint function a power function of both time and line size. That is, in addition to satisfying Eq. (2.12), the footprint function satisfies an equation of the form

$$u(L) = DL^E \tag{2.14}$$

for line size L and constants D and E. Putting Eq. (2.12) and Eq. (2.14) together yields the most general composite function of this type which is:

$$u(t,L) = W L^a t^b d^{\log L \log t} \tag{2.15}$$

where W is a measure of working-set size, a is a measure of spatial locality, b is a measure of temporal locality, and d is a measure of the interaction between spatial and temporal locality. *Temporal locality* is the tendency for references to cluster together in time and reflects the probability of referencing something that has recently been referenced. *Spatial locality* is a measure of the probability of referencing something located near to an item recently referenced. The two locality measures are not independent, and their interaction is reflected in the coefficient d. The derivative of this function gives an estimate of the miss ratio of a fully associative cache. It is plotted in Fig. 2.27(b) for a variety of line and cache sizes. Hill and Smith [1989] suggest how to change this to the miss ratio

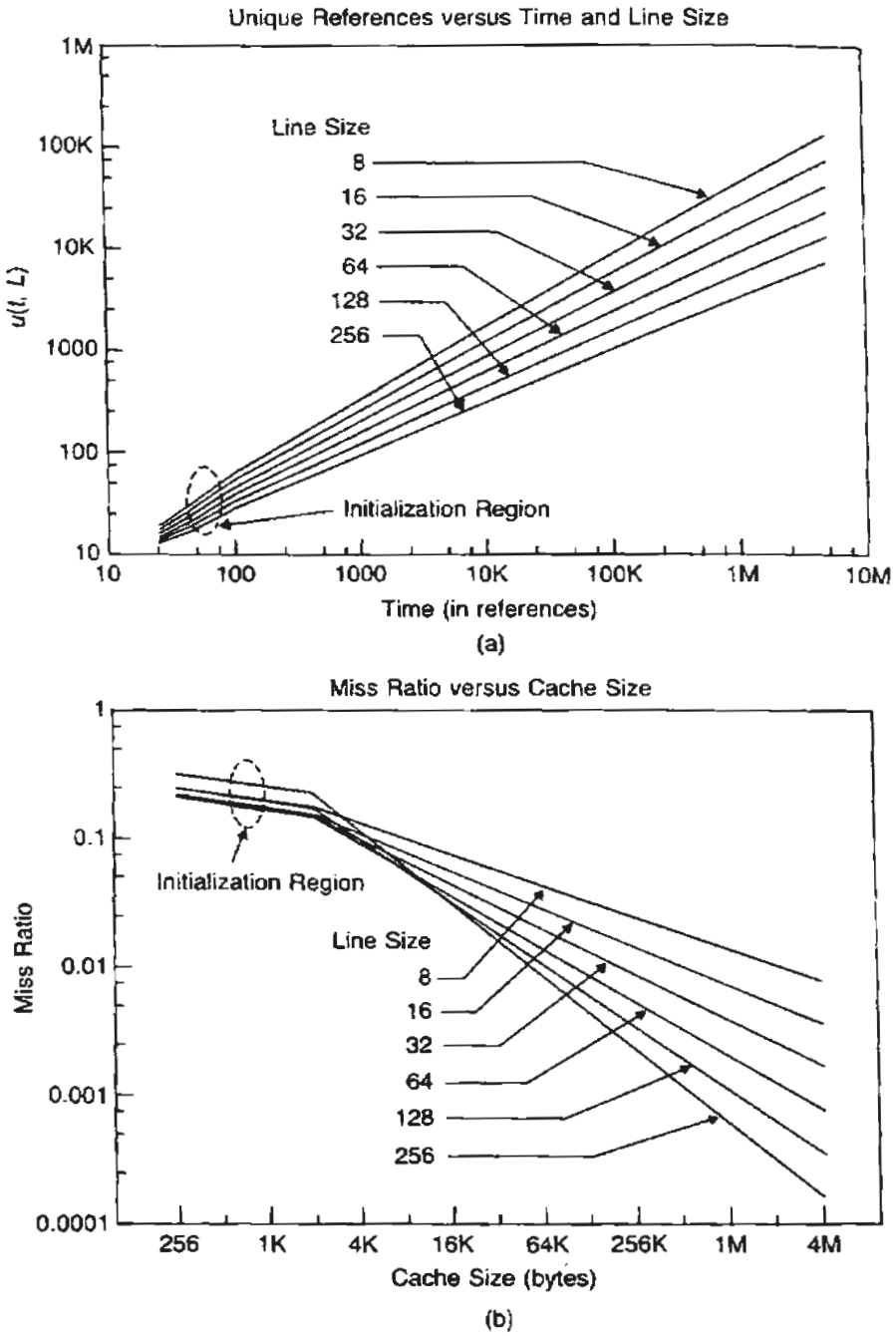

Fig. 2.27 (a) The footprint function model as a function of time and line-size, plotted on a log/log scale.
(b) The miss-rate function model as a function of line size and cache size.

of a set-associative cache. Although fully associative caches have better miss ratios than four-way set associative caches of equal size, the miss ratios are fairly close, and the fully associative cache can serve as an approximate model to the four-way set-associative cache or Hill and Smith's techniques can be used for specific cache structures.

There are a number of concepts from this model that are useful to the cache designer. Here are the major ones:

1. The power-law function indicates that there is a law of diminishing returns. Each successive doubling of cache size to exploit temporal locality gives less absolute improvement in miss ratio for double the expenditure of hardware. Each doubling of line size gives less absolute improvement in miss ratio for double the expenditure in bus traffic per miss. Very large caches can cost more than their performance justifies.
2. No cache size and structure is characterized by one miss ratio. The cache performance depends strongly on the workload. Miss rates are low if the working set of the workload fits in cache, and they are high otherwise.
3. The size of a cache should be large enough to contain the full working set of the majority of the workloads to be run on the cache. Although, in general, performance improves with increasing cache size, the most performance gain per change in cache size is in the region where the cache reaches and exceeds the working set of most typical workloads. If the cache is too small, the performance of larger workloads is compromised. If the cache is too large, the machine will be priced higher than its performance justifies. It is probably best to produce too large a cache than too small a cache because the working-set size of applications tends to increase in time as new applications with large working-sets are released. Consequently, excess cache size will be put to use eventually.
4. To estimate the cache size needed for a new processor design, estimate the working-set size of typical programs. This tends to be much larger in large address spaces than in small address spaces.
5. Steady-state miss-rate functions depend on the nature of the workload but all workloads are likely to have exponents in Eq. (2.12) that lie between 0.400 and 0.700. Pick sample exponents in this interval, and model cache behavior for such exponents. The smaller exponents tend to be associated with scientific or numerically intensive applications and the larger ones tend to be associated with less structured applications such as operating systems, database management, and artificial intelligence.
6. The line-size effects are also workload dependent. Kobayashi and MacDougall [1989] have a limited amount of data on line-size effects as a function of workload type, and A. Smith [1987] has extensive data on line-size effects

independent of workload. Both of these studies provide typical data that are useful in the absence of simulation data.

The cache designer can explore the impact of workload parameters through Eq. (2.15). Singh *et al.* [1992] give values to a, b, and d in Eq. (2.15) of 0.0333, 0.827, and 0.740, respectively, for a general workload that includes operating system functions and user applications. Scientific applications can be expected to have greater spatial locality than the workload used in the study, which can be modeled by decreasing the coefficients a and d in combination.

Cache design is considered again later in this text when we discuss cache design for multiprocessor systems. The important principles of cache design are:

1. Cache memories retain needed information physically close to the central processor where the information is quickly accessible. As a general rule for high-performance systems, the data most frequently accessed should be physically close to where it is used.
2. The traffic density between the central processor and main memory is anywhere from 10 to 30 times lower than the traffic density between central processor and cache. An important goal in high-performance systems is to keep traffic density low on long interconnections and on shared interconnections.
3. The cache mechanism works only because programs exhibit particular behavior that can be exploited by the cache. If programs behaved differently, caches as we know them could fail badly. Programs are not forced to work the way they do; they just happen to do so. Other facets of programs might be exploitable to attain high performance, especially if processors are designed for particular applications.
4. The cache mechanism adapts to execution streams by learning what items have been used and favoring recently used items over items that have not been used recently. It is possible to incorporate other kinds of hardware into a system that help the system adapt to observed behavior in an execution stream. The question is open as to where and what kind of hardware to use, and whether or not the cost of the extra hardware is justified by the performance gained.

2.3 Virtual Memory

The designers of the Atlas computer gambled heavily on program characteristics that tend to keep the active pages in high-speed memory. The cache memories described in the previous section are successful because address references show

strong sequential locality, and cache management easily exploits such characteristics.

Virtual-memory systems, as they exist today, fulfill a role similar to cache memories, except that virtual-memory systems manage a different portion of the memory hierarchy. Cache-management algorithms attempt to make optimum use of a high-speed memory for which main memory serves as a backup buffer. Active items tend to move from main memory to cache, and inactive items tend to migrate back to main memory.

Virtual-memory systems attempt to make optimum use of main memory, while using an auxiliary memory, usually a rotating magnetic disk memory, for backup. Therefore, to the first order of approximation, the high-speed buffer memory of a cache system corresponds to main memory of a virtual-memory system, and the main memory of a cache system corresponds to an auxiliary memory of a virtual-memory system. The principles that govern the behavior of cache and virtual-memory systems are largely the same. Namely,

1. Keep active items in the memory that has the higher speed;
2. As items become inactive, migrate them back to the lower-speed memory; and
3. If the management algorithms are successful, the performance will tend to be close to the performance of the higher-speed memory, and the cost will tend to be close to the cost per bit of the lower-speed memory.

We have learned some implementation techniques for cache memories in the previous section, so one might believe that those implementation techniques carry over to virtual-memory systems. Unfortunately, they do not carry over directly because the details of costs and timing are dramatically different when you move from cache memories to virtual memories.

Effective designs are driven by details of performance and costs. Because cache and virtual memory are dramatically different in such details, implementations of the two memory-management schemes may be quite different. In this section we examine a very simple virtual-memory system to identify the design parameters. Then we look more closely at available implementation techniques to satisfy the needs of the design.

2.3.1 Virtual-Memory Structure

A simplified view of virtual memory is illustrated in Fig. 2.28. In this figure the address produced by the processor, which is called a *virtual address*, is mapped by hardware to a physical location in central memory if the item is located in main memory. If not, the result is a page fault that moves the page containing the item being moved to main memory. The size of the virtual address-space that contains the addresses produced by the processor need not bear any relation

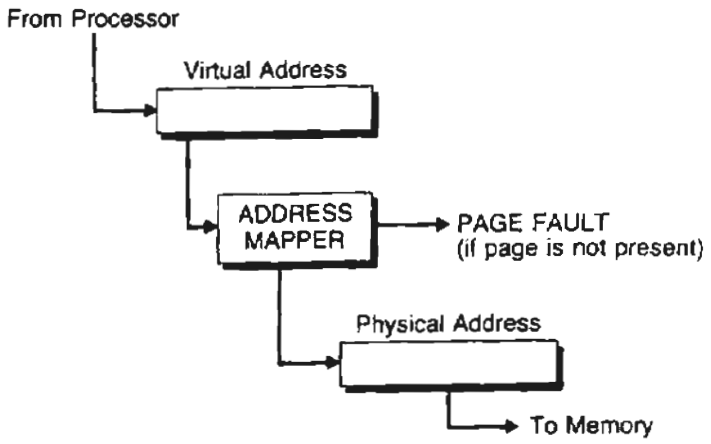

Fig. 2.28 The structure of a virtual-memory mapper.

to the size of the physical address-space that contains the addresses in central memory produced by the mapper shown in the figure.

We tend to view virtual memory as the Atlas designers originally viewed it. That is, virtual memory is much larger than physical memory, and the objective of the virtual-memory system is to produce a large memory with high performance and low cost per byte. But the mapping scheme has been used successfully in situations in which virtual memory is much smaller than physical memory, although such uses are becoming rarer. The applications in question arose because of technological changes that led to large, central memories whose costs were dramatically lower than the costs of prior generations.

Some computer families had been designed with relatively small address spaces, which cannot be changed because of compatibility requirements. Designers can create a machine whose physical memory is many times larger than the addressable memory available in the family, and then use virtual-memory mapping to permit software to run unchanged in the large physical memory. To make effective use of the large physical memory, the systems run several independent applications concurrently in a time-shared mode of operation frequently called *multiprogramming*.

In the early 1970s, for example, a limitation of 16 bits for addresses was natural, and, therefore, the typical virtual-memory space in minicomputers was 64K. When a memory of size 1M became available, then approximately 16 independent programs, each of maximum size 64K, could be run concurrently in the one physical memory of size 1M. Moreover, if the memory manager were successful in retaining the active pages in main memory and returning inactive pages to auxiliary memory, perhaps the main memory used per program could

drop from 64K to something less, such as 32K. Then the number of independent programs that can run concurrently increases to about 32 programs, which makes the system reasonably cost-effective per user program.

The use of virtual memory in this example remains attractive only as long as it is necessary to run the software developed for the 16-bit address space. New programs should be written to take full advantage of the larger address space when the extra memory can be put to good use.

In the late 1970s the first machines with 32-bit addresses appeared, and by the mid-1980s, the multigigabyte virtual-address space was firmly entrenched in machines that ranged in size from engineering workstation to the high-end mainframe computer. Even with this large virtual address, it is just a matter of time before it becomes economical to deliver physical memories larger than four gigabytes. In recognition of this possibility, some architectures evolved to 64-bit virtual addresses in the early 1990s. Nevertheless, for any 32-bit architecture that does not evolve to a larger virtual address, in the present and immediate future the virtual-memory system maps a large address space into a smaller one. When technology can provide gigabyte memories at an economical price for the prevailing 32-bit environment, these same systems are likely to be redesigned to map the virtual space into larger physical spaces.

A significant difference between virtual-memory and cache-memory systems lies in the relative penalty of a page fault and a cache miss. In present technology, a cache miss is 4 to 20 times as costly as a cache hit, but a page fault is 1000 to 10,000 times as costly as a page hit. Rotating memory has a latency time fixed by mechanical limitations. Although electronic random-access memory has had speed improvements on the order of 1000 to 1 over the last two decades, the latency of mechanical memories has not improved by more than a factor of 10. Moreover, the mechanical limitations inherent in the design of rotating memories suggest that disks will not spin 1000 times faster, nor are they likely to have 1000 heads, which exhaust the two obvious ways to reduce latency.

For the near future, we are more likely to see the relative cost of a page fault increase as semiconductor memories continue to improve performance, while no significant improvements reduce disk latency. Over a longer period, we may see a new memory technology filling the gap between semiconductors and rotating mechanical memories. Such a technology would have a profound impact on virtual-memory implementation as we know it today.

The huge cost of page faults results in very different strategies for cache and virtual-memory management. During a cache miss, the processor becomes idle while waiting for data to arrive from main memory. Some activity pertaining to table maintenance may take place during the miss, but there is insufficient time available for other processes to do useful work on the processor. Hence, a cache miss is not accompanied by a change in task for the duration of the cache miss.

In a virtual-memory system, relatively large amounts of unused time are available while awaiting a page transfer from auxiliary memory. This time is so

long that it is reasonable to put the processor to work on other tasks during the latency period attributed to a page fault. In typical systems, the latency experienced is from 1 to 100 ms, and a 10-MIPS processor can execute somewhere between 10,000 and 1,000,000 instructions of other programs during this period.

The earliest commercial implementations of virtual memory attempted to improve efficiency by turning the processor over to other pending tasks. The processor costs made the processor cycles a precious resource that should be conserved, if possible. Consequently, virtual memory was implemented in several different multiprogramming systems and in remote-access time-sharing systems. The idea was to create queues of pending tasks by amalgamating many users on a single system. When one user was delayed by a page fault, the processor could be dispatched with a second user's task in the interim.

The sharing of the processor is a natural solution when processor cycles are expensive. But sharing has its own negative factors. As processor utilization goes up and approaches 100 percent, each user sees a longer response time because the time to process a job depends on how long the job takes when running without contention and how long it spends in queues waiting for other users to terminate. Increased efficiency in the use of a processor generally is accompanied by increased waiting time for each job because of contention with other jobs for access to the processor.

As the cost per machine cycle has become very small, a new alternative has become possible. Instead of turning over control to a different job while waiting for a page from auxiliary memory, it is reasonable in some circumstances to retain the processor and simply wait for the new page to arrive. In such cases, the performance gain due to lack of contention for the processor is more valuable than the loss due to cycles lost by the processor during page faults.

There is another major negative impact on system design if a virtual-memory manager forces an application to relinquish the processor on a page fault. The policy interferes greatly with the ability to evaluate designs by using address traces. Each page fault is accompanied by the execution of 10,000 to 1,000,000 new instructions that would not be executed if the processor were not reassigned during a page fault. If a simulation run is used to evaluate the effects of a new control strategy, then what should be simulated during page faults?

In essence, the attempt to evaluate new policies inevitably increases or decreases the page faults observed. But there is no convenient mechanism for modifying a trace dynamically to obtain an accurate description of the execution that actually takes place during page faults.

We have no difficulty evaluating cache designs from trace tapes, but we have a great difficulty evaluating virtual-memory designs the same way. Moreover, we can simulate a few seconds of processor time to obtain thousands of cache misses, but the same simulation produces only tens or hundreds of page faults, so trace data are subject to large statistical errors.

Consequently, virtual-memory evaluation is best performed in real time with

hardware or software measurements of activity. To obtain repeatability and to evaluate different strategies on a common workload, the architect must rely at least in part on a synthetic workload.

There are three major design considerations described in the next sections:

1. The mapping mechanism;
2. Partitioning for locality; and
3. The replacement strategy.

As these topics are presented, bear in mind how different the approaches are from approaches that address similar functions for cache memories. The differences are all attributable to the difference in the values of performance and cost figures. This clearly shows the impact of specific values of design parameters on architectural decisions and suggests how major technological advances that alter these values will affect designs.

2.3.2 Virtual-Memory Mapping

The mapping device shown in Fig. 2.28 is grossly simplified for purposes of exposition. Let us consider the requirements for a mapper and then discover what additional complexity is required to make an effective mapper.

The basic function is to map a large address space into a much smaller one, so that we may view the virtual-memory mapper as performing the function shown in Fig. 2.29, where some large field of bits in a virtual address is replaced by a smaller field of bits to create a physical address. In Fig. 2.29, the displacement field describes the offset from the base of a page. The displacement is not changed by the mapper because the offset within a page is the same for a virtual address as it is for a physical address. We need to know only where the page begins in physical memory, and by adding the offset to this address, we can find the physical address of any item. Hence, the mapper uses the virtual-address bits other than the offset bits as it transforms addresses.

What makes the problem challenging is the very large number of pages in the virtual address. Consider the difference in the mapping problem for a virtual memory with 64K addresses (16 bits) as compared to a virtual memory with 4G addresses (32 bits). For purposes of comparison, in both cases we assume that the page size is 1K (10 bits).

In the smaller memory, there are only 6 bits, or 64 pages permitted in a program of maximum size. It is perfectly reasonable to store the translation table in a set of 64 registers and consult the translation table on each reference.

The larger memory system permits programs to grow to as large as 4M pages. A translation table with 4M entries is far too large to place in a set of dedicated registers, and it is costly by present standards to store in memory, although this is a possible solution in the future.

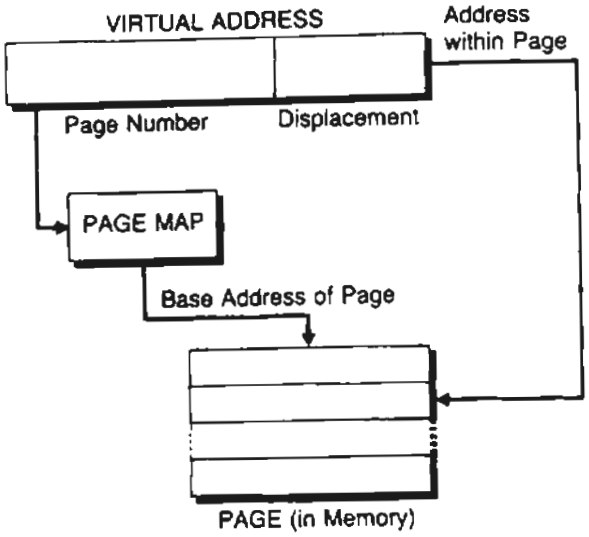

Fig. 2.29 A typical virtual-address translation.

How do you deal with such a large translation table? To reduce the memory demands for storing the translation table, most solutions in use today break up the one-level translation into two translations. The effect of having two levels could be disastrous on performance because each access then becomes three accesses, and worse yet, each of the two accesses into the translation table could generate its own page faults before the access to the requested page has occurred. So performance could be dramatically poorer just because of the overhead of the mapping process.

The overhead of mapping is reduced by means of an artifice called a *translation-lookaside buffer* (TLB), which is a cache for holding recently used mappings. Figure 2.30 gives the general structure of a virtual-memory mapper. We examine the details of the pieces in the following discussion.

In Fig. 2.30, a virtual address is broken into two fields, one for the offset and one that identifies a virtual page. The virtual-page field is presented to the translation-lookaside buffer, which checks its cache-like memory to see if a recent translation for that page took place. If so, the translation-lookaside buffer returns the base address of the page, and the mapping is completed. Just from our knowledge of cache behavior, we would expect almost all references to be satisfied by the lookaside buffer.

If an address misses in the lookaside buffer, the two-level mapping is performed. We describe the details later in this section. A miss in the two-level mapping is a page fault, and the virtual-memory manager must intervene to

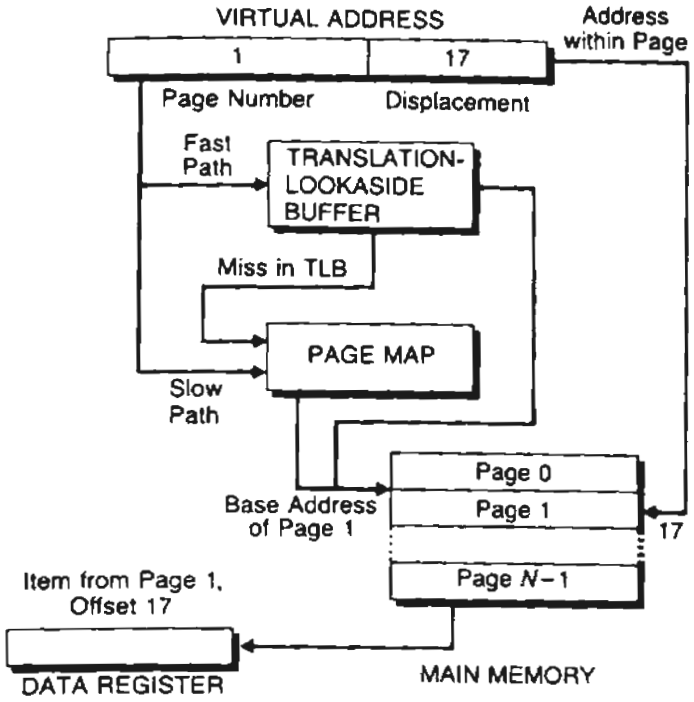

Fig. 2.30 A virtual-address translation with a translation-lookaside buffer (TLB) for fast operation.

correct it. Otherwise, the mapping produces the base address of a physical page that can be added to the offset to obtain the full physical address.

In the last operation, if we force pages to begin on addresses that are multiples of the page size, then we can save an addition operation in the mapping transformation. In this case, since the low-order bits of the full base address are known to be zero, the page offset can be concatenated to a shortened base address rather than be added to the full base address.

One possible two-level mapping is shown in Fig. 2.31(a). The idea is to break up the large field into two smaller fields. Common terminology is to designate the high-order field as a *segment number* and the next field as a *page number*, although the term *segment* is used to denote other concepts related to virtual memory.

In this example, the 22 bits remaining after stripping off the 10-bit offset are broken into an 11-bit segment number and 11-bit page number. The segment number is used in the first level of the transformation as the index to a Segment Table. From the Segment Table we obtain the base address of a page table. The

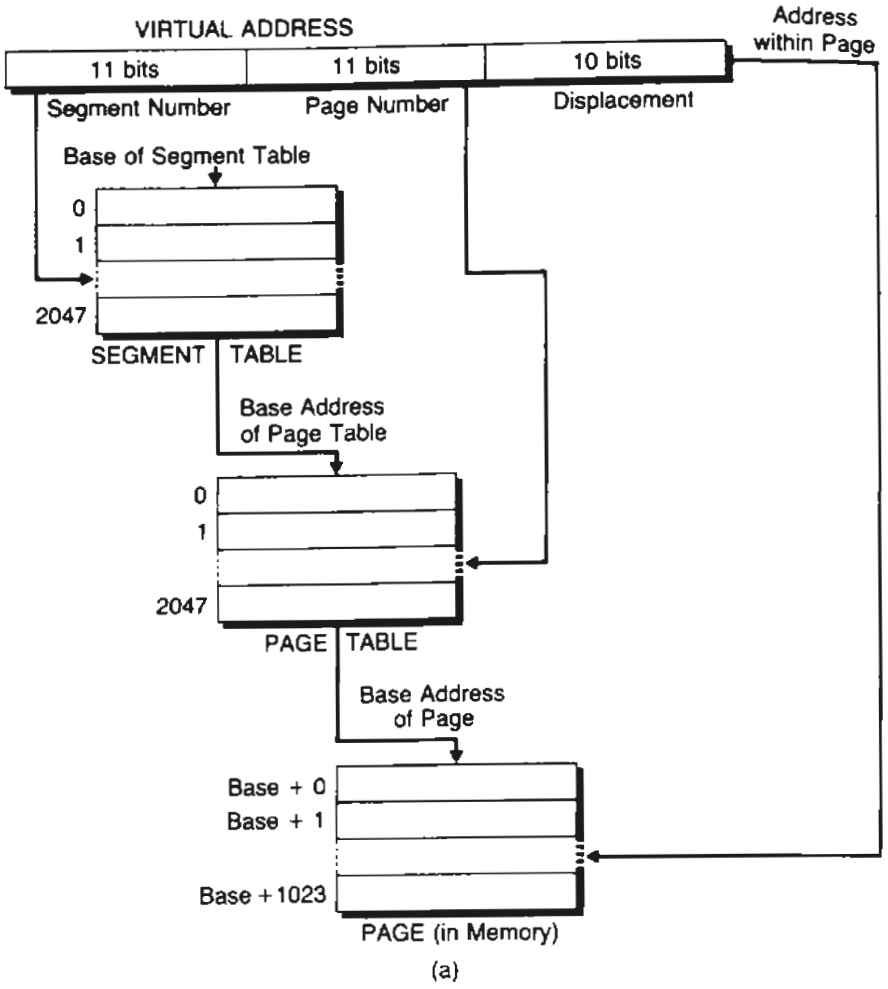

Fig. 2.31 Two-level mappings:
(a) A typical two-level mapping; and
(b) A two-level mapping used in the VAX architecture.

page number is combined with this base address to consult a page-table entry that has the base address of the page itself.

The effect of using two levels is to reduce the page-table number from 22 bits to two indices, each of which is no more than 11 bits, so that no single table needs to be larger than 2048 entries. What has happened in the two-level mapping is that a very large page table has been broken into many pieces, each of which is no larger than 2048 entries. The smaller tables need not reside in main

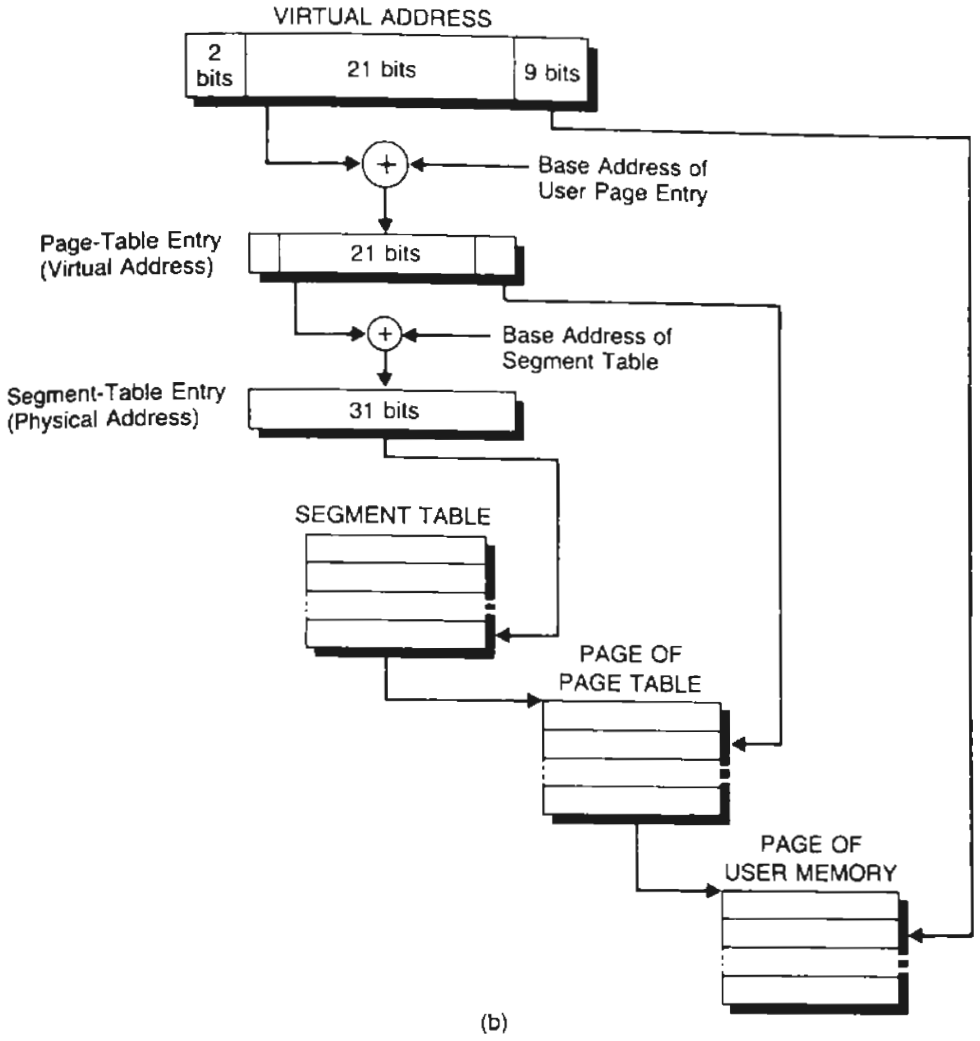

Fig. 2.31 continued

memory if they are not in active use. Hence, we need have in memory only the portions of the page table that are active.

The penalty for the two-level mapping is the second level of lookup. Moreover, both levels can have page faults during a lookup, although the Segment Table rarely faults because it is accessed relatively frequently. If we had a choice, we would prefer not to pay the penalty for the second access, but the enormous size of the resulting page table makes the alternative one-level mapping im-

practical under most cost measures. The scheme becomes practical only when the cost of tables of size 4M can be ignored.

There is a problem with the scheme in Fig. 2.31(a), resolved by the scheme shown in Fig. 2.31(b). The problem pertains to shared pages that are accessed by independent programs. Each program in a shared page produces virtual addresses that must be mapped to physical addresses. The virtual addresses produced by shared code may conflict with the virtual addresses selected by users or by other shared programs that are linked to run as part of the same job. It is necessary to ensure that shared programs generate virtual addresses that do not conflict with each other or with user addresses.

The scheme shown in Fig. 2.31(b) models the address-transformation mechanism used on the VAX architecture. In this scheme, virtual memory is divided into two regions, each with 2G bytes (31-bit addresses). The scheme uses two levels of mapping for addresses that lie in user virtual memory, but only one level of mapping for virtual addresses that lie in system virtual memory. The leading 2 bits of a 32-bit address uniquely identifies the virtual address as belonging to one of two user regions or to one of two system regions. Shared system programs reside in one of the system regions. The leading 2 bits of a virtual address thus determine which of four possible tables are to be used during the address mapping.

The VAX implementation treats the entire 21-bit field of an address with the leading 2 bits and 9-bit offset removed as a page number to be used in an address transformation. A page number is converted into an address by adding to it the contents of a base register identified as the User Page Table base address in Fig. 2.31(b). (The page number is multiplied by four before doing the addition to account for the length of each entry in the VAX-11 page tables, but this is not shown explicitly in the figure.)

The address produced by this transformation is a virtual address, not a physical address, which is one key difference between Fig. 2.31(a) and (b). This virtual address lies in system space, so that only one level of access is required to change it into a physical address of a page-table entry. After that transformation is completed, the physical address of the page-table entry goes through one additional level of access to produce the final physical address that corresponds to the original user virtual address.

To map the virtual address of the page-table entry, the processor extracts a 21-bit page number from the page-table entry address, and adds this to the base address of a table in system space to produce a true physical address. The system table in this case is identified in the figure as the Segment Table to be consistent with Fig. 2.31(a), but the VAX architecture documentation uses the name System Page Table for this table [Eckhouse and Levy 1980]. The physical address obtained from this mapping is an address in the Segment Table. The first memory access in the mapping process occurs at that address.

The Segment Table lookup produces the higher-order bits of the base address

of a page in the user page table. These bits, when concatenated with the 9-bit displacement field of the virtual address of the page-table entry produce a physical address of a single page-table entry, and the next access occurs at this location. From the page-table entry, the process finds the higher order bits of the base address of the user page. To these bits the processor concatenates the 9-bit displacement field of the original virtual address and produces the physical address of the item sought.

The two base addresses, User Page Table and Segment Table, both reside in processor registers. No memory access is required to obtain these base addresses, so the first access to physical memory is the access to the Segment Table.

References to shared pages of system programs are handled by placing the shared pages in system space, not in user space. System-space addresses are transformed to physical addresses by just a single level of mapping, as shown in the figure, and only the Segment Table is used for this mapping.

Hence the page table used to access shared pages is the Segment Table, which is shared among all processes, whereas the page table used to access unshared user-pages resides in user memory and is private to each process. In this way, user addresses are distinct from addresses produced by shared programs, and shared programs can produce addresses without contention provided that they occupy disjoint regions of the system virtual memory.

Obviously, the system virtual memory must be large enough to accommodate all shared pages in distinct areas, which is possible only for large virtual memories. Although 2G bytes available for the VAX is very large by some standards, even this may not be enough for all possible shared programs. In actual practice, VAX uses both the scheme described here for sharing system functions, and conventional schemes for sharing other items in user memory by pointing to the shared items through user page tables. Thus, the commonly shared items are handled efficiently through a one-level mapping while the less commonly shared items require two levels of mapping.

Let us return momentarily to the translation-lookaside buffer that appears in Fig. 2.30. We described it as functionally similar to a cache memory, and indeed its design is very close to that of a cache. Clark and Emer [1985] describe the analysis of a translation-lookaside buffer for the VAX-11/780 architecture, and their paper is a model of the classic design and analysis techniques for cache. They use a trace-driven approach to simulate a variety of structures varying from 64 to 512 sets and with both one-way and two-way set associativity. Some aspects of their paper are different from cache studies and are worth commenting on here.

The Clark-Emer data suggest that misses occur in the translation-lookaside buffer at the rate of between 0.5 to 3.0 per 100 instructions. This is not the same as the miss and hit ratios described earlier because the ratios for cache references are developed on a *per-reference* basis, whereas the Clark-Emer data are on a *per-*

instruction basis. Since one instruction produces several references, including instruction fetch, indirect address fetch, operand fetch, and operand store, the equivalent miss ratio for translation lookaside buffers is probably a factor of 3 or 4 smaller than the misses per 100 instructions. So, indeed, misses in the translation-lookaside buffer are rather rare.

The penalty for a miss in the buffer is also quite different from the penalty for a cache miss. The cache miss is followed by an access to main memory, which is perhaps ten times slower than is the cache. But the cost of a buffer miss is an access to the cache, which is somewhat faster than an access to main memory.

Clark and Emer report a surprisingly small hit rate to the cache to retrieve items that miss in the lookaside buffer. The hit rate is only about 40 percent. Perhaps this is the case because the lookaside buffer may be very successful in handling most references to page table entries—so successful in fact that such references are quickly purged from cache once they are placed there.

If eventually a miss occurs in the lookaside buffer, then the likelihood of finding that reference in the cache apparently is very low, only 40 percent as opposed to over 90 percent for other references. Another possible explanation for the high miss rate is that a miss in the lookaside buffer occurs most frequently when a program changes its activity or a total change of process takes place. But these times are precisely the same times when a cache produces the bulk of its misses.

Another aspect of the lookaside buffer that is different from the cache is that the translation mapping is dependent on the process running. The lookaside buffer sees virtual addresses, not physical addresses. These addresses are not unique, so there must be some mechanism for identifying which virtual addresses go with which process in the lookaside buffer. This mechanism does not need to exist for a cache memory that stores physical addresses in its directory because physical addresses are unique.

One way to handle the problem of associating the correct mapping with an address reference is to place a process tag in the lookaside buffer with each entry. Then a match occurs only if the process tag in the buffer matches the process tag of the running process.

The approach used in the VAX-11/780 lookaside buffer is to flush the lookaside buffer of entries for private (per-process) mappings when a context switch occurs. This could be a fairly expensive process depending on the size of the lookaside buffer, the time it takes to purge the entries, and the frequency of context switches. Instead of selectively purging just the entries for private mappings, it may be faster to purge the entire buffer, and thereby purge the entries for shared as well as private mappings. In this case there is a penalty paid later for reloading the shared mappings.

We do not have specific advice on which approach is better because there

is no absolute answer. Here is a case in which the designer should perform a thorough analysis following the model of Clark and Emer to determine which approach yields both the best cost/performance for the technology to be used and the presumed workload for the architecture.

2.3.3 Improving Program Locality

The mapping transformations described thus far have presumed that large programs are broken into equal-sized pages, and the pages are managed by a virtual-memory operating system. The pages are arbitrary and need not have any relation to the logical structure of the program. Since the page is the atom to be used by the virtual-memory manager, a reference to any single item on a page results in the entire page being present in main memory.

If the contents of a single page are logically related, then bringing in a page when any item on that page is accessed makes available inexpensive accesses to the other related items. If the items on a page are unrelated, the page fetch may bring in unwanted items, resulting in poor use of both available memory bandwidth and resident memory.

Structuring programs so that related items are packed together on relatively few pages is definitely advantageous. In essence, this postulates a new structure in which programs and data are grouped together according to their logical relations rather than because of arbitrary factors. Virtual-memory systems that attempt to account for logical relationships within programs are sometimes called *segmented-memory systems*, as opposed to paged-memory systems. A *segment* is a collection of related programs and data that forms a subprogram unit. Segments can invoke other segments, and some commonly used segments can be shared among many users.

What makes segments different from pages is that segments are not fixed in size. They can be as large or small as the programmer chooses to make them. Because segments are not uniform in size, memory management is far more complex than for pages of a fixed, uniform size.

Although various techniques have been developed for memory allocation of variable-length structures, it is also possible to combine paging and segmentation in a single virtual-memory system. The idea here is to use segmentation to produce logical structures of program and data, and then move portions of segments in and out of memory by breaking segments into pages of fixed size. Techniques for paged virtual memories carry over directly to this scheme, and no significant added difficulty for handling variable-sized segments is imposed.

There are a few differences between segmented and paged systems, however, that should be brought to light. One difference is in the structures of a segmented-address space and a paged-address space. A paged-address space is a one-dimensional space in which all addresses lie in one contiguous region

in virtual memory. Given any address (except possibly the last address) in this memory, you obtain the address of the next item by increasing the current address by one.

A segmented memory is a two-dimensional space. Each address consists of two fields, a segment number and an offset within the segment. All addresses within a segment lie in one contiguous area of virtual memory. However, segments are not contiguous to each other; they are distinct.

When you increment the highest possible address of a segment, you do not obtain the address of an element in a new segment. You create a condition that is recognized as an attempt to access an out-of-bounds address.

For example, consider a virtual memory system with 48-bit addresses, of which 24 bits indicate an offset within a segment, and 24 bits indicate a segment number. In this system, a program can create references to up to 16M different segments, each of which has up to 16M addressable locations. If a program attempts to reference an item in Segment *i* and calculates an address whose offset exceeds 16M, the virtual address produced will not increment the segment portion of the address field when the offset overflows. Hence, the reference continues to be to Segment *i*, except that the overflow from the offset field is detected and produces a program exception.

Given this structure, we have an interesting problem in handling shared memory. Suppose a segment is shared by two programs, Program *A* and Program *B*. Shall we impose the restriction that both programs designate this segment as Segment 10? Or will we permit Program *A* to designate the segment as Segment 2, while Program *B* designates it as Segment 11? To restrict all shared segments to unique numbers is similar in spirit to the handling of shared segments in the VAX virtual memory, as we discussed earlier. This method makes sense for sharing system programs that are available essentially at all times.

In a more general context, however, we may want all segments to be shareable, or we may have a huge collection of shared segments that exceeds the number of unique segment numbers available. So for one reason or another, we want to let Program *A* and Program *B* refer to a shared segment by their own respective indices for this segment.

One possible way to provide access to the shared data under this stipulation is to provide a segment table with each process. The segment table provides the information to translate a segment reference from Process *A* or from Process *B* into the correct physical reference to a shared segment. Figure 2.32 shows this scheme. Note that the shared segment is Segment 1 for *A* but is Segment 2 for *B*.

In this scheme, the virtual addresses for referring to the shared segment depend on which segment has issued the address. Then each process has a private segment table for accessing shared segments, as opposed to the common table used for the VAX architecture.

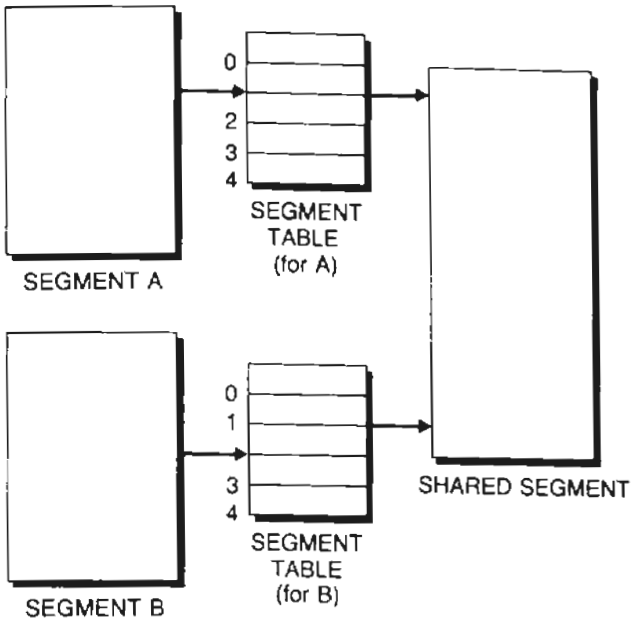

Fig. 2.32 Access to a shared segment through private segment tables.

Moreover, the addresses produced by the shared segments have to be mapped correctly for each context in which they operate. Thus, it may be necessary to force a shared segment to access a variable in Process *A* at Segment 10, but when running in a different context, that same access is made to a shared variable in Process *B* at Segment 15.

The segment number of a reference produced by a shared program in general is dependent on the context in which it runs, and the ability to produce segment numbers that depend on the context may have to be incorporated into the architecture. The VAX solution to sharing avoids the complication of the general solution and is satisfactory for shared system programs.

To eliminate the burden of consulting the segment table on each reference to an external segment, designers usually incorporate a translation-lookaside buffer or the equivalent to catch the majority of references without accessing the segment table. As control passes from one segment to another, it is rather important to purge the lookaside buffer so that its translation is correct for each context.

2.3.4 Replacement Algorithms

The obvious way to manage virtual memory is to manage it in the same way that cache is managed. In fact, Belady's work [1966] on optimal replacement strategies (cited in our cache discussion) was done in the context of virtual-memory systems. But, in general, virtual-memory systems are sufficiently different from cache memories as to require dramatically different techniques for management. The principal differences between virtual memory and cache memory are:

1. Page faults are very costly. There is a greater relative savings in reducing page faults than there is in cache misses.
2. While responding to a page fault, there is substantial time available for memory-management functions that might reduce future faults.
3. Virtual-memory systems may run competing programs when a program reaches a page fault. The competing programs may interfere with memory management and could grossly impair performance. No competing programs are run during the processing of cache misses.

Early implementations of virtual-memory systems examined various replacement policies, and soon the strengths of least-recently used replacement in predicting the future were recognized. However, when LRU replacement and other similar policies were implemented in early commercial virtual-memory systems, the systems occasionally entered periods of instability during which almost all machine cycles were devoted to handling page traffic and essentially no useful work could be accomplished. The problem was not the fault of the replacement policy *per se*, but rather a lack of understanding of the dynamics of a virtual-memory system. Detailed studies revealed that some critical factors had been overlooked.

One problem stemmed from trying to accomplish too much in a single virtual-memory system. The instabilities occurred at high loads; otherwise performance was acceptable.

Denning [1968a] termed the instability *thrashing* because the prime characteristic was a very high traffic between main memory and auxiliary memory of frequently used pages. A page might be brought to main memory, used a few times, then returned to auxiliary memory, only to be recalled to main memory.

Another related mode of instability occurred when a process lost some critical pages from main memory, but eventually recovered those pages, only to have lost other critical pages in the interim. Every program enters phases during which some subset of pages is used frequently. Denning [1968b] called this subset the *working set* of pages.

The working set is in essence the footprint of a program execution over a short period of time. If a program has its entire working set in main memory, it will have a very low page-fault rate as computation progresses. Page faults

increase dramatically when portions of the working set are not available. So one key principle is to run programs by striving to have the full working set in memory at the time the program is run.

A corollary of this principle recognizes that it is unrealistic to move an entire working set from memory to disk and back again between successive time slots allocated for program execution in a multiprogrammed, virtual-memory system. In fact, it is unrealistic to move any significant portion of a working set out of main memory and back in again under the same conditions.

In essence, if a program is to run effectively, its working set must be resident and must stay resident in main memory until the program terminates. This rule can be relaxed somewhat for programs that interact with humans because some displacement of the working set can be tolerated during the time that the human is thinking and reacting to prior output. Otherwise, as a general rule, main memory must hold the working sets of the active programs.

If the working sets exceed in total size the area reserved for them, then the system is likely to become unstable. Figure 2.21 tends to confirm the need to hold working sets in main memory. Virtual memories tend to behave like caches with large K (set-associativity) values. The figure shows a sharp drop in the reload transient as cache size increases. This drop occurs at the point where the cache is big enough to hold both footprints. If the cache is smaller than this critical size, the reload transient is very large, which means that one program completely overlays the other as they successively take control of the processor and cache.

Therefore, to eliminate thrashing, a reasonable approach is to estimate the size and content of working sets, and to load into main memory a collection of complete working sets whose total size does not exceed the memory available. Any additional requests for machine cycles should be deferred until some process or processes terminate and make sufficient memory available to hold the working set of the new process.

If this principle is to be used to manage memory, then we need a way to calculate the working set dynamically during program execution. Denning [1968b] provides some guidance by describing a mechanism for discovering the working set. He defines the function $W(t, w)$, the working set at time t for window w. This is the set of pages referenced in the last w seconds at time t.

A memory manager based on the notion of the working set attempts to hold in memory only those pages that belong to the working set because references are most likely to be made to the working set and are very unlikely to be made to pages outside the working set, except during periods when the working set is changing because the program is moving into a new phase. Therefore the memory manager brings in a new page when a page fault occurs and adds the new page to the working set. The memory manager deletes from the working set those pages that have not been referenced within the last w seconds.

Here is one possible memory-management policy that takes advantage of the properties of the working set.

1. When a page fault occurs, add the new page to working set.
2. From the set of pages not referenced within a window of w seconds immediately prior to the page fault, select the least-recently used page, and discard it. If all pages have been referenced within the working-set window, then discard no page, and let the working set grow.
3. If two or more pages have not been referenced within the working-set window, then discard the two least-recently used pages.

Note that there must be some rule that diminishes the number of pages allocated; otherwise the size of memory allocated to a process grows until no free memory is available for references to new pages. That is the purpose of the third rule. This set of rules is slightly different but similar in intent to the rules proposed by Coffman and Denning [1973, p. 299].

It is very important that the working-set window be measured in *virtual* time, by counting clock ticks only while the process is executing. No clock ticks are counted during page faults or during other periods when the given process is inactive.

The size of the window can be determined experimentally. If it is too large, some pages will tend to be retained too long. If the window is too small, it may not span the times of references to the actual working set. The optimum window size is just large enough to cover the working sets of all programs. It is better to err by using too large a window than too small a window because the consequences of using the wrong window size are less severe when the window is too large.

The working-set concept is an intuitively appealing way of handling page replacement, but measuring the working set is somewhat difficult, even with special hardware. One possible way to approximate the working set is to identify which pages are accessed during the brief execution of a program in a system that grants the processor in round-robin fashion to a collection of programs.

We assume that each program is granted some fixed quantum of time during which the processor executes that program. Before the quantum begins, access tables are initialized to show that no page has been touched. These tables are best kept in hardware such as a translation-lookaside buffer or a special memory devoted to tracking page accesses. As each page is touched, the hardware automatically sets an activity bit in the access table to record this occurrence. At the completion of the quantum, the working set is deemed to be the pages whose activity bits have been set, and the remaining pages may be removed from main memory.

This form of the memory-management algorithm is quite workable if supported by hardware that can turn off all activity bits for a process and can turn them on selectively on access. Other solutions are possible as well, and the

architect has a great deal of freedom in trading off the cost of implementation against approximations to the working set. The working set itself is an approximation to a perfect predictor of the future.

Chu and Opderbeck [1976] proposed an alternative approach to the working-set approach that has the advantage of using directly measurable variables for guiding the replacement decision. The method is called the *page-fault-frequency method*. It exhibits different policies when the frequency of page faults is above or below a fixed threshold. The reasoning behind this method is that programs tend to operate in phases, accessing one working set consistently while in one phase, and then moving to a new working set in the next phase. The page faults tend to occur during a change of phase.

Our earlier discussion of the cache-reload transient examines the corresponding phenomenon for cache memories. A high frequency of page faults is a signal that the program is entering a new phase and that the current working set may have to be replaced. It is clear that as new pages are touched, they should enter the new working set. All pages in the former working set are candidates for replacement.

One way of implementing a replacement policy based on the notion of page-fault frequency is the following, which is similar to the method proposed by Chu and Opderbeck.

1. Assume a threshold θ for page-fault frequencies.
2. When a page fault occurs, estimate the page-fault frequency for the given program. A crude estimate is $1/(t_1 - t_0)$, where t_1 is the virtual time of the present fault, and t_0 is the virtual time of the last fault. A better estimate is an average taken over the last few faults.
3. If the estimated frequency exceeds θ, then assume that the program has entered a transient phase or that there is presently insufficient memory allocated to the process to hold its full working set. Add the newly referenced page to the working set and increase the amount of memory allocated to the program by one page.
4. If the estimated frequency does not exceed θ, then assume that the program is in a stable pattern of memory references. Add the new page to the working set and remove some page not referenced since the last page fault, preferably the least-recently used page.
5. If the estimated frequency does not exceed θ over some fixed time period then assume that the program has entered a stable phase and that it may have some dead pages within its present allocation. If there are pages that are currently allocated to the program and that have not been referenced recently, then decrease the number of pages allocated to the program and discard a corresponding number of unreferenced pages. We presume that the pages are discarded in the order of least-recently used, if this is possible.

The role of the last rule is to provide a means for decreasing the allocation of pages to a process. Without this rule, the number of pages allocated would grow until no memory remained for allocation to new pages. Another way to implement this rule is to establish a different lower threshold on page-fault frequency, below which a process has pages removed from its working set.

One can easily measure the frequency of page faults by means of a process timer that is normally present in an architecture. The memory manager takes note of the length of execution time between faults each time a page fault occurs. This is easy for the memory manager to do because it is invoked when a fault occurs and has access to the process timer.

If recent history for a process shows that faults are occurring at a rate that exceeds a system threshold, the memory manager increases the allocation of pages for the program, using a decision criterion similar to the working-set criterion. If the fault rate is below the threshold, the memory manager either performs a one-for-one replacement or reduces the allocation by discarding one or more pages that have not been referenced recently, in addition to performing one-for-one replacement. The latter strategy may occasionally produce costly page faults if too much is discarded, but the strategy may be useful to invoke when the pages used to hold active working sets occupy nearly all available memory. Also, by attempting to replace pages when page faults are low, the probability of finding a dead page is somewhat higher than when page faults are high.

An interesting question regarding memory allocation is the question of what is an optimal allocation? An optimal allocation of memory to processes is one that produces the least page-fault rate. Assume that each process has a fault rate that is a function only of the number of pages of the current memory allocation, and let the fault rate for Process i be $R_i(x_i)$, where x_i is the memory allocation for process i. The allocation that produces the least fault rate globally over all processes is the one that makes all the fault rate derivatives equal in the following sense:

$$\left.\frac{dR_i(x)}{dx}\right|_{x=x_i} = \left.\frac{dR_j(x)}{dx}\right|_{x=x_j} \tag{2.16}$$

Equation (2.16) says that the optimum is reached by adjusting the allocation of each process so that the fault rates of each process for their respective allocations are equal. There is a simple intuitive explanation of the formula. When a page fault occurs, and a page is taken away from some process, and given to another, how much does the global fault-rate change? If Process i loses a page, its fault rate increases by $R_i(x_i - 1) - R_i(x_i)$. The increase is approximately equal to the negative of the derivative of the fault-rate function. If Process j is the process that produces the fault, then its fault rate decreases by $R_j(x_j - 1) - R_j(x_j)$ when

it receives room to add a new page to its memory allocation. Again, this is a negative of a derivative function. There is a net improvement if the increase in fault rate for the process that loses a page is smaller than the improvement in fault rate for the process that gains a page. The replacement policy should take away a page from the process that has the smallest fault-rate derivative. When this policy is followed over a period of time, the system reaches an allocation state in which all fault-rate derivatives are equal and no further improvement can be made. This is a state in which the global fault rate is minimal.

Equation (2.16) assumes that all processes are executed equally often. If not, then the derivatives have to be multiplied by a weighting factor equal to the fraction of execution of time allocated to a process. If, for example, Process i receives 60 percent of processor cycles and Process j receives the remaining cycles, then the weighting factor that multiplies Process i's derivative is 0.6 and the factor for Process j is 0.4. Equation (2.16) can be derived by expressing the global fault rate as the sum of weighted fault rates of individual processes. To find an optimum point, take the derivative of the global fault rate with respect to memory allocation and set the resulting function equal to zero. Ghanem [1975] derived this equation in the context of virtual memory systems. It has also been used to explore allocation of cache memory in processors [Stone, Turek, Wolf, 1992] and in disk caches [Thiebaut, Stone, Wolf, 1992].

Because the optimum allocation depends on fault-rate derivatives, not on the fault rates themselves, it is clear the page-fault frequency replacement policy is not optimum. At an optimum allocation the fault rates of the various processes need not be equal. But the page-fault frequency policy attempts to produce fault rates that are all close to each other. In reality, if all processes have similar fault-rate functions, then the allocation that produces equal fault rates for all processes will also produce approximately equal fault-rate derivatives. If the fault-rate functions differ drastically from process to process, page-fault frequency could be quite far from optimal.

Optimal replacement is not easy to implement because the fault-rate derivatives are difficult to obtain. Ease of implementation is a strength of page-fault frequency replacement because the fault rates are directly measurable. To measure the fault-rate derivative, an operating system can, in principle, construct a table of fault rates as a function of memory allocation for each process. The derivative is the difference between adjacent entries. Derivative data collected this way tend to be very noisy, and require smoothing over multiple observations to be useful.

Another challenge in implementing an optimal allocation scheme is the problem of finding derivatives of fault rates for allocations larger than the present allocation. One might believe that the operating system has to allocate more memory than desirable in order to discover that the fault-rate derivatives for those greater allocations are too small. In fact, a shadow directory for pages will help obtain that information without requiring the allocation to be made. As

each page is removed from a process, its identity is retained in a shadow directory of recently removed pages. When a fault occurs, the operating system can consult the shadow directory to look for a match. If there is a match, say five page replacements earlier, then the operating system can infer that the page fault would not have occurred if there were five additional pages allocated. At that allocation the page-fault rate is reduced to the lower fault rate obtained by ignoring the last five faults.

The working-set and page-fault frequency algorithms ultimately retain only the pages in the working set while programs are executing continuously in one phase of computation. There is a difference in the behavior of these algorithms during transients. The page-fault frequency algorithm anchors its observation point at a page fault, and by doing so, the memory manager can fix its observation at a time when a transient appears to have begun. Then the manager can observe all of the pages touched since that fixed time.

In a sense, this is a working-set algorithm in which the window size varies dynamically, depending on the observed fault rate. It provides for narrowing that window during transients and widening the window when transients have ended. This will tend to discard old pages when they are no longer needed. A pure working-set algorithm has a fixed window size, but this is very difficult to implement. In reality, the working-set window used by typical memory managers begins each time the memory manager takes control.

We have purposely avoided a detailed specification of the page-fault-frequency and working-set-replacement algorithms because an implementer is free to adjust and modify a replacement algorithm to fit the characteristics of the architecture and the workload. There is no single implementation of either algorithm that is preferred or standard. The general idea behind the algorithms is what is important.

The working-set concept is based on the assumption that the immediate future will be something like the recent past. The page-fault-frequency algorithm is based on the notion that a transient between two program phases is signaled by a higher-than-normal page-fault rate.

The working-set algorithm in its purest form is difficult to implement because a sliding window of fixed size is not easily incorporated into hardware or software. The page-fault-frequency algorithm provides for policies that depend on more readily observable quantities and on hardware logging of accesses, which is easier to implement than is a working-set window.

In spite of these apparent differences, the practical implementations of working-set-replacement policies use the same hardware and the same observations as the page-fault-frequency algorithms, and the actual replacement policy for a working-set algorithm may be implemented almost identically to a page-fault-frequency algorithm. In fact, Coffman and Denning [1973, p. 289] describe a working-set-replacement policy that is essentially a page-fault-frequency model as we have described previously.

2.3.5 Buffering Effects in Virtual-Memory Systems

The preceding section describes how to exploit the characteristic reference patterns of almost all programs to hold the frequently used data in the fastest memory. This section describes another characteristic of virtual-memory systems that is not widely recognized. The characteristic is that a certain amount of space must be allocated permanently to buffering disk operations, and the amount of space to use grows proportionally with the access delay to disk data. We examine the implications of this characteristic and suggest that the amount of buffer space in future systems is likely to grow because access delays are likely to be relatively longer.

We indicated earlier that accesses to data on rotating memories suffer a delay anywhere from 10 to 100 ms or more. If we break this access delay into components, part is due to the time required for a read/write head to position itself over the track that contains the data requested. An additional delay stems from the rotational delay while waiting for an item to reach the read/write head. Yet another delay is the time required to transmit data from the auxiliary memory through an input/output port to main memory. The rotational delay averages a half revolution when access requests are honored in a first-come, first-serve order and the requests are randomly generated. By batching requests and carefully reordering them to reduce waiting time, the average delay due to rotational latency can be reduced, but this is partially offset by an increase in the average time due to time lost by reordering or other aspects of contention for disk resources.

Obviously, mechanical limitations prevent the rotational speed of disk drives to be so fast that the average rotational latency is comparable to the time needed to access random-access memory. Moreover, very large memories will inevitably suffer from access delays to individual data, regardless of the storage technology, simply because an effective means to reduce the cost per bit of large memories is to share access circuitry over many bits.

In present technology, the single read/write mechanism of a disk drive is shared by all bits on the disk, and it takes mechanical motion to position the read/write mechanism over any designated position on the disk. In future technologies, the motion might be nonmechanical by, for example, deflecting a laser beam to a particular physical position on a storage surface. Nevertheless, the time required to redirect a laser beam may be long compared to the cycle times of very high speed memory devices, even though the time is much shorter than the time would be if mechanical motion were required.

Now consider how a long latency time affects a computer system. In virtual-memory systems, the long latency experienced after a page fault requires that the system be used for other purposes. There is typically a queue of processes ready to use the machine when a running process faults. Let us assume momentarily that we have as much capacity as required in available memory and

input/output bandwidth to ensure a large queue of ready processes. Under these conditions disk latency does not necessarily lead to idle processor time, although in actual practice these ideal conditions are not realized, and latency could lead to substantial idle time.

Consider what happens when a program experiences a page fault. The portion of the program resident in main memory remains inactive in main memory while the missing page is retrieved. The longer the access time to the missing page, the longer that the resident program occupies main memory without doing useful work. The effect of latency is to create certain regions in memory that are inactive. In a sense, they have become buffer regions awaiting pages arriving from the disk and holding pages awaiting transfer back to the disk.

If latency is truly very large, then it may be reasonable to remove inactive data from main memory when a page fault occurs and reload them at a later time. Even so, some physical pages of main memory are still being used as buffer memory. These pages buffer the data moving out of main memory immediately after a page fault. They also buffer the data of a new process being moved into main memory. Once a new process becomes resident, it can become active, and the physical pages are again activated after being used solely for buffering.

To give some idea how much memory has to be dedicated to buffering in a steady state, consider a simple model of a program that experiences page faults. Suppose that on the average a program with a working-set size of W pages can execute for N seconds between faults. Let the delay due to disk latency be D seconds. Then for D seconds out of every $N + D$ seconds, the working set is idle, and we have in a sense a buffer of an average size $WD/(N + D)$ if the program executes without any other programs in memory.

Since we assume that the processor is to be fully utilized by other programs waiting to run, we need to determine how many such programs should be available. The given program has to wait D seconds when it faults, and during this time we can run roughly D/N other programs, each faulting after N seconds, to use up a total of D seconds. At the end of D seconds, the original page fault should be cleared, and the initial program can be restarted. Since this ideal system has $(N + D)/N$ programs running concurrently, each of which is acting like a buffer with an effective size of $WD/(N + D)$, the total buffer storage is something on the order of WD/N. Hence, the amount of main memory dedicated to buffering page faults increases linearly with D.

In this analysis we have been counting the space occupied by programs as buffer space, but the space is quite distinct from the regions that are set aside as disk buffers. These regions, too, must grow in size in proportion to latency.

When a record or a page is transmitted to a buffer to await transfer to a disk, the relative time spent in that buffer is a function of the latency. Together, the memory requirements for input/output buffers and programs delayed by page faults must grow proportionately to disk latency. Since the timing factors

are relative, we discover that the growth must take place if we hold disk latency fixed and double processor performance. This particular event happens to be a likely one in an era when advances in semiconductors improve processor performance by larger factors than advances in mechanical technology can reduce access delays in auxiliary memories.

There are several implications of this observation. Suppose, for example, that we have a high-performance virtual-memory system with 100 M-bytes of main memory, and the system is very efficient in its current implementation. Now suppose that we obtain a new disk with double the capacity of the present disk, but with twice the average access time. When the new disk replaces the old disk, we should also increase the size of main memory to compensate for longer access time or reduce the number of concurrently running processes by a factor of 2. If we do not compensate for the longer access times, the longer latency will degrade performance.

A second implication is that the page-replacement algorithms are somewhat sensitive to the physical characteristics of the rotating memory. As the amount of memory dedicated to buffering disk operations increases, the amount of memory left available to hold working sets decreases. In other words, the page-replacement algorithm must relate disk latency to the amount of memory that can be allocated among requesting programs. If new disks replace old ones, the page-replacement policy has to alter its estimate of memory available for user programs.

A third implication is that it becomes reasonable to consider where that buffer should be located. In fact, the buffer might well be located in the auxiliary memory rather than in the main processor. Such a scheme is shown in Fig. 2.33. The disk buffers (sometimes called *disk caches*) first appeared in volume in the mid-1970s as the costs of memory diminished.

When memory is very expensive, one can argue that the wisest way to design a memory system is to place all the memory in one unit, the central processor, so that it can be allocated freely as necessary. As memory costs decrease, the need to conserve memory by using a single pool is diminished. Other factors dictate that there are benefits to breaking memory into two or more pools that are preallocated to specific purposes.

Figure 2.33 shows a system with a disk buffer contained within the disk system that is distinct from the memory associated within the central processor. The disk buffer is dedicated to disk operations and cannot be used as executable memory.

The purpose of the disk buffer is, in effect, to create one more level in the memory hierarchy. The disk buffer acts as an auxiliary memory with a very short latency, thereby reducing the buffer requirements in main memory.

In essence, the buffers that we have observed earlier have been moved out of main memory and now reside at the other end of the input/output channel. There will still be some buffering of pages in main memory because latency is

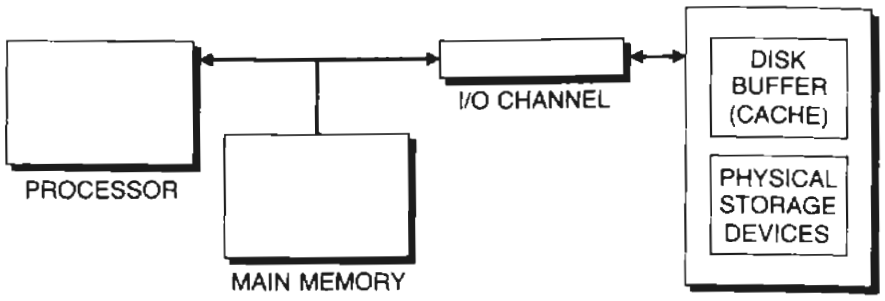

Fig. 2.33 A storage system with a disk buffer.

small but nonzero because of transmission delays in the input/output channel and page faults in the disk buffer. If the system of Fig. 2.33 is well designed, the page faults will occur mostly in the disk buffer and seldom in main memory.

There are several performance gains that can be achieved by moving the buffer to the disk. One advantage is that the data-management algorithms for the buffer can be optimized to the specific characteristics of the disk system. Thus, when a disk system is replaced with a new system, the memory-management algorithms in the central processor need not be altered because the device-dependent characteristics are being treated within the disk system. The disk buffer creates an auxiliary memory whose performance characteristics as seen by the central processor tend to be independent of the true physical characteristics of the disk.

Earlier we discussed how changes in latency result in changes in the amount of memory serving as a buffer for pages. In the configuration of Fig. 2.33, almost all effects of latency can be absorbed within the disk-system buffer, making such changes completely invisible to the central processor memory-management algorithm. Thus, when new equipment imposes a need for additional buffer capacity, that buffer capacity can be added in the disk subsystem where it is of direct use.

To obtain full benefit of the disk buffer, the buffer controller must have information regarding the requests for data because various types of requests have different management algorithms that work best. We have been discussing page traffic, and for such requests a memory-management algorithm should attempt to identify current working sets.

The disk system also receives requests for data files, and for such requests the type of access is a major consideration. For example, the manager should prefetch records associated with sequential files and deallocate space for sequential records immediately after their first use. Database management treats indices differently from data records, so a disk-buffer manager should manage

indices differently than records. Therefore, the disk buffer in Fig. 2.33 is ideally implemented as an "intelligent" disk buffer that can manage the buffer memory in a manner that takes best advantage of each identifiable type of access.

There is a performance benefit in Fig. 2.33 over a system that has no disk buffer. The performance enhancement is due to the ability to hold data in fast memory without burdening the input/output channel. The disk buffer may actually access and hold many records that are never requested if the buffer-management algorithms attempt to bring data that are likely to be accessed into the buffer memory. This incurs the costs of wasted accesses and additional hardware for the buffer memory to hold the unused requests, but it does not load the input/output channel.

There are situations in which the input/output channel is the primary bottleneck, and the disk system has spare capacity to read data into its buffer. Under these conditions, the ability to have new data in the buffer can reduce average access time without contributing to the channel overload.

Exercises

2.1 The object of this exercise is to work through the design of a cache.

a) The instruction set for your architecture has 44-bit addresses, with each addressable item being a byte. You elect to design a four-way set associative cache with each of the four lines in a set containing 64 bytes. Assume that you have 256 sets in the cache. Show how the 44-bit physical address is treated in performing a cache reference.

b) Consider the following sequence of addresses. (All are hex numbers.)

0E1B01AA050 0E1B01AA073 0E1B2FE3057 0E1B4FFD85F 0E1B01AA04E

In your cache, what will be the tags in the set(s) that contain these references at the end of the sequence? Assume some initial state. Show the initial and final states.

c) The cost of your cache is roughly proportional to the number of bits of storage required. The purpose of this question is to determine how many bits are used for each function.

How many bits are required to hold the cache data? How did you get this number?

How many bits are required to store address tags? How did you get this number?

The cache is four-way set associative. What is the fewest number of bits per set required to keep track of which line to replace in the set when replacement is LRU? How did you get this number?

d) We can construct a cache with the identical number of data bytes by doubling the line size and by reducing set associativity to 2. How does this change the cost of the cache as measured in total bits?

2.2 This problem asks you to consider the performance effects of doubling the line size of a cache as compared to doubling the number of sets in a cache.

Suppose you have a basic cache design such as that given in the first problem. You wish to increase the size of the cache because memory prices have dropped since the last design was completed. You have two options—double the line size of the cache or double the number of sets. You want to estimate the performance difference of the two designs. Make any assumptions that you wish about the basic design.

a) Find a sequence of address references that produces more misses in the cache with longer line size than in the cache with more sets.

b) Find a sequence of address references that produces more misses in the cache with more sets than in the cache with longer line size.

c) Produce a sequence of references to data in a scientific program that performs a matrix multiply of two matrices. Neither matrix fits in the cache. Show your matrix multiply algorithm in a high-level language, then show the sequence of data references it produces to the two operand matrices and to the result matrix. Then describe which of the two means of doubling cache produces fewer misses, and explain why. You may ignore instruction accesses and focus attention solely on data fetches when answering this question.

d) Given your answers to the prior parts of this question, describe qualitatively the characteristics of an address trace that determine which of the two ways to double cache size will perform better.

2.3 After some careful experimentation, you discover that each time you double the size of a cache, you reduce the absolute number of cache misses by a factor of $1 - k$ where $k < 1$. That is,

$$Misses(2N) = k\ Misses(N)$$

Find a general solution to this recurrence equation in the form

$$Misses(N) = A\ N^B$$

where A and B are constants.

2.4 The object of this exercise is to work through the design of a cache together with a virtual memory mapping.

a) The instruction set for your architecture has a virtual address 32 bits long, and the virtual address is mapped into a physical address that is 28 bits long. Suppose pages have P bytes, and suppose p is the base 2 logarithm of P. Then the page mapping is done by stripping the page displacement (the least-significant p bits of the address) from the full virtual address, using the remaining bits as the index of a very large page table. The page table produces $28 - p$ bits, which are concatenated with the p-bit page displacement to form a physical address of 28 bits. This is used for the cache reference.

Suppose that $p = 11$. Work out a scheme that permits you to overlap in time as much as possible the virtual-memory function and the cache lookup. Explain your scheme and show the relative timing of the operations. Explain how you selected the number of sets in the cache to enable the overlapping of operations.

b) Suppose that $p = 10$. Explain how you can modify the scheme for $p = 11$ to continue to permit the maximum amount of overlapping of cache lookup and virtual-address mapping in this case.

c) Now let $p = 12$. Indicate how you can modify the cache scheme for $p = 11$ for this case.

d) What can you say is the general rule that you should use for the relationship of page size and the cache design parameters?

2.5 The object of this exercise is to practice cache analysis. This exercise and the ones that follow refer to the trace files on the disk available from the publisher as supplementary material for adopters of the text. If you do not have access to these files, you are invited to obtain traces from any source available to you, and to repeat the experiments on those traces.

a) Simulate a cache with 32 sets, one-way set associativity, and 8-byte line size on TRACE1, and verify that there are no hits on the trace. This is the size of the cache on which the original trace was stripped.

b) Write your cache simulator so that you can detect the first miss of each line of the cache that you simulate. Let the first miss for each line be called a "simulation transient-miss." Simulate the following caches, and record the total number of misses observed (including simulation transient-misses), the number of misses if all simulation transient-misses are treated as hits, and the estimate for the number of misses if the simulation transient accesses had the same hit ratio as the other accesses on the trace. Also record at the end of the simulation the number of sets that are fully initialized.

Simulate using both TRACE1 and TRACE2 the following cache structures:

i) one-way associative: 32, 64, 128, 256, and 512 sets.
ii) two-way associative: 32, 64, 128, 256, and 512 sets.
iii) four-way associative: 32, 64, 128, 256, and 512 sets.

c) Plot the log (base 2) of the estimated number of misses as a function of the log (base 2) of the cache size in bytes. Plot three different curves on the same graph, one each for one-way, two-way, and four-way associativity. Do you see any regularity in this graph? Comment on what you see.

d) You can double the size of the cache by doubling the number of sets, doubling the set associativity, or doubling the line size. Your data does not give you information on the effect of doubling the line size. However, for this trace, it gives a great deal of information on the relative performance attained from doubling the number of sets versus doubling set associativity. What do you observe?

2.6 The purpose of this question is to explore cost-performance trade-offs in regard to the structure of caches.

a) The stripped trace TRACE1 on the floppy disk containing trace information is assumed to be exactly 10 percent of the length of the original trace for the purposes of this question. Cache access time is one cycle. Main memory access time is four cycles. Each instruction causes exactly one instruction fetch and one operand fetch or store. Compute the cost-performance ratio for each of the 15 caches studied in Exercise 2.5, using the best estimate for your hit ratios and your cost estimates from Exercise 2.1 for costs.

b) Which cache structure (or structures) are most reasonable to build, according to your interpretation of the cost-performance data?

2.7 The purpose of this exercise is to explore how to do set sampling and to examine its accuracy. For this exercise use the traces supplied on the floppy disk, or use a source of traces available to you. Also use your simulation program that was created in Exercise 2.5.

a) Alter your cache simulator to examine only the addresses whose set number ends with the 3-bit pattern (0 1 0). This should cause your simulator to sample exactly 12.5 percent of the sets. Then with sampling in place, simulate the caches that you simulated in Exercise 2.5 on TRACE1 and TRACE2 data.

b) From your data, estimate as closely as you can how many misses would be observed on the full trace. Compare your estimates with the answers in Exercise 2.5. Calculate the relative error in the estimate of performance produced by set sampling as compared to performance as calculated in Exercise 2.6. The relative error is defined to be

$$\text{Rel error} = \text{abs}(\text{sampling answer} - \text{true answer})/\text{sampling answer}$$

Plot the relative error in percent as a function of cache size, using 3 curves, one each for set associativity of 1, 2, and 4.

c) From the plots of relative error, comment on the ranges of cache parameters for which the sampling technique is accurate enough to be useful.

2.8 This problem concerns the theory behind trace stripping.

a) Suppose you produce an address trace for a computer system, and you process that trace in the following way. You simulate the behavior of a cache with 64 sets, one-way set associativity (direct mapping), and lines of size 32 bytes. Addresses are byte addresses. As you process the trace, you produce an output trace that contains just the addresses of the cache misses.

Now suppose that you use the output trace as the input trace to a cache simulator. Assume that the cache you simulate has 64 sets, one-way set associativity, and 64-byte lines. Will this trace generate more, equal, or fewer cache misses than the original trace? Prove that your answer is correct.

b) Assume that you use the same reduced trace and use it as input for the simulation of a cache with 32 sets, one-way set associativity, and 32-byte lines. Will this trace generate more, equal, or fewer cache misses than the original trace? Prove that your answer is correct.

2.9 The object of this exercise is to explore trace stripping more deeply.

a) The chapter states that if a trace is stripped by simulating an N-set, one-way set-associative cache, the stripped trace can be used for any cache with a multiple of N sets and K-way set associativity for any K greater than or equal to 1. Consider such a trace and prove that it can be used to evaluate a $2N$-set four-way cache. Specifically, prove that every miss on the full trace is a miss on the reduced trace for the new cache, and conversely, every miss on the reduced trace is a miss on the full trace for the new cache.

b) Suppose that a trace is stripped by simulating an N-set four-way cache. The reduced trace contains just the misses produced by this cache. Prove or disprove the following statement:

If a $2N$-set cache is simulated on the full trace and on the reduced trace, the number of misses observed in both cases is equal.

2.10 The object of this exercise is to work through the design of a virtual-memory system. In this problem, you are given four sets of design parameters and are asked to design a virtual-memory system to meet each specification. There are many possible designs for each set of constraints. As you explore the design options, your main goal is to estimate the cost and performance of competing alternatives. In considering performance, the number of levels of mapping is the main criterion. Fewer levels give better performance, and one level is best. In considering cost, determine how much memory space is committed to tables for mapping each level. You should be able to narrow the design choices to a few different possibilities, and you will probably not be able to select a best design from the alternatives with the information given. If you can narrow the choices to one, do so, and if not select a range of satisfactory designs. Discuss your selections and why you chose them.

a) Consider the design of a mapping device that maps virtual-memory addresses into physical addresses. The size of physical memory is 64K bytes, virtual addresses are 24 bits, and the working-set size of processes for which this machine is designed is approximately 40 K-bytes. Work out the design parameters for a virtual-memory mapper, including a translation-lookaside buffer. Discuss your reasoning that led you to select a particular design or set of designs.

b) Repeat *a* for a physical memory of size 64 M-bytes, a virtual address of 36 bits, and a typical working-set size of 4 M-bytes.

c) Repeat *a* for a physical memory of size 1 M-bytes, a virtual address of 22 bits, and a typical working-set size of 64K bytes.

d) Repeat *a* for a physical memory of size 512 M-bytes, a virtual address of 48 bits, and a typical working-set size of 32 M-bytes.

2.11 This problem concerns replacement algorithms for virtual-memory systems.

a) Page-fault-frequency (PFF) and working-set-replacement algorithms have similar behavior except during the transient as a program changes from one phase to another and thereby changes its working set. Assume that if a working set is entirely resident within main memory, the expected fault rate is about one fault per 1000 instructions, and the number of instructions between faults increases by ten percent for each additional page in main memory over and above the pages that hold the working set.

If the working set is not contained in memory, the fault rate is one fault per 25 instructions, and this improves by 25 instructions between faults for every page of the working set that is added, up to the point that the working set of 20 pages is resident in main memory. Given this information, what is the threshold that you would set for PFF, how many faults would occur, and what would be the maximum number of pages resident as you change from one phase of the program to another phase of the program with identical page-fault characteristics? Discuss how you selected the threshold and show how you arrived at your answers.

b) Repeat the first part of this exercise for the case in which the replacement al-

gorithm is working set, but in this case indicate how you selected the working-set window instead of explaining the choice of a threshold.

c) Describe a practical mechanism for determining whether or not a page is in a working set for a program. Your mechanism does not have to use Denning's definition exactly as stated in the text, but it should yield a reasonable approximation to the working set. What aspect of your solution, if any, is the most costly in time expended? What aspect is likely to be the most costly to implement assuming current cost conditions?

2.12 The object of this exercise is to examine performance characteristics of virtual memory.

a) Consider a physical disk system that is capable of performing an average of 50 accesses per second. Assume that an average working set is 50 K-bytes, that the mean number of instructions between page faults is 100 when less than the full working set is present in memory and is 5000 when the 50K working set is wholly contained in memory, and the page-fault rate drops off by 30 percent for each doubling of the number of pages in memory in excess of 50K. Assume that each instruction takes 1 μs on the average to execute. Plot the throughput (completed instructions per second) as a function of memory size for a single program being executed.

b) Repeat the first part of this exercise for the case in which two programs share memory equally.

c) How should you partition memory to obtain maximum throughput for the statistics given when memory contains 50 K-bytes? 100 K-bytes? 250 K-bytes? An arbitrarily large number of bytes? Describe how you obtained these answers.

2.13 Modern disks incorporate disk cache, which is a high-speed semiconductor memory that buffers disk accesses. Assume that the disk controller understands how data are being requested from disk and that it has the ability to treat executable programs, sequential files, and pages of virtual memory that have been swapped out of main memory. Describe how you would manage the memory in the disk cache to do a reasonable job of avoiding accesses to the rotating physical disk. Make reasonable assumptions concerning the frequency of accesses, the size of disk cache, and the fault rate as a function of data size.

2.14 This problem concerns buffer requirements for virtual-memory systems. Assume the performance data for programs given in Exercise 2.12. For the 250K memory example at maximum performance, how much of the memory system for your answer to Exercise 2.12 is serving as buffer memory on the average? Show how you obtained this answer. Assume that the disk-access time increases by a factor of 2, and you want to obtain equal throughput as for a system with the faster disk-access time. Determine how to obtain increased throughput by adding additional memory, using as little extra memory as possible. How will you allocate memory in the new system to achieve the necessary throughput? How much of the memory on the average is serving as buffer?

2.15 This problem addresses the design of a disk cache.

a) Assume that a program has to access a sequential file. What should the disk cache do in managing the records in this file? Describe the commands and replies