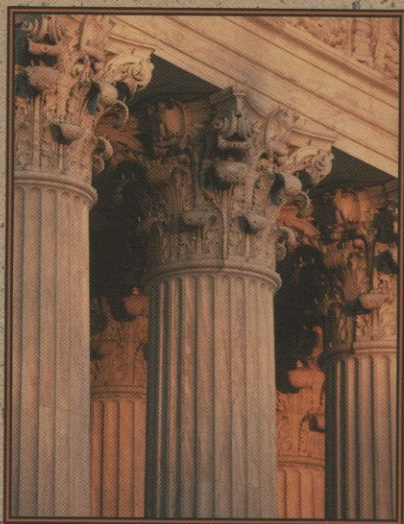


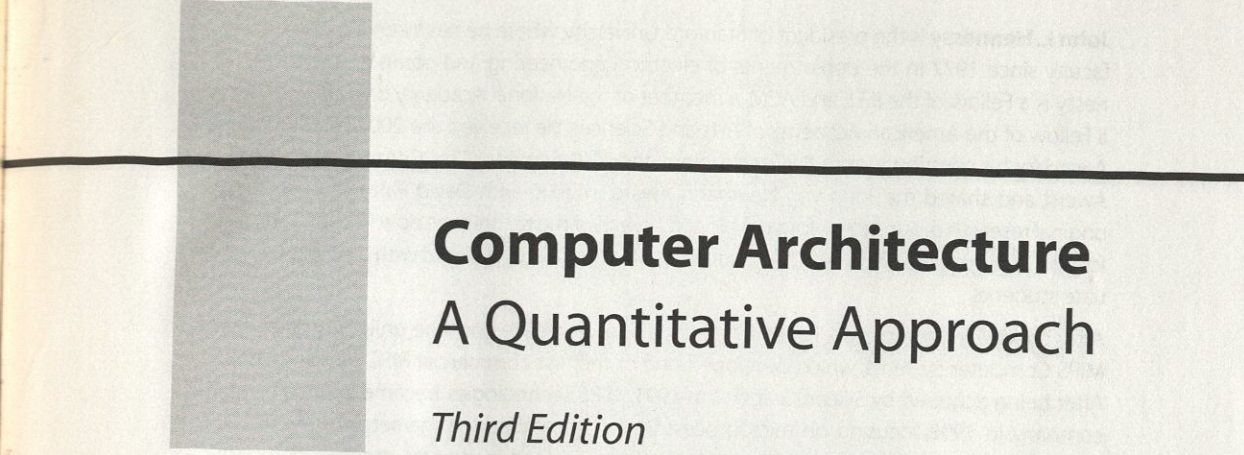
Computer Architecture

A Quantitative Approach

THIRD EDITION



John L. Hennessy
David A. Patterson



Computer Architecture A Quantitative Approach

Third Edition

John L. Hennessy is the president of Stanford University, where he has been a member of the faculty since 1977 in the departments of electrical engineering and computer science. Hennessy is a Fellow of the IEEE and ACM, a member of the National Academy of Engineering, and a Fellow of the American Academy of Arts and Sciences. He received the 2001 Eckert-Mauchly Award for his contributions to RISC technology, the 2001 Seymour Cray Computer Engineering Award, and shared the John von Neumann Award in 2000 with David Patterson. Hennessy's original research group at Stanford developed several of the techniques now in commercial use for optimizing compilers. In 1981, he started the MIPS project at Stanford with a handful of graduate students.

After completing the project in 1984, he took a one-year leave from the university to cofound MIPS Computer Systems, which developed one of the first commercial RISC microprocessors. After being acquired by Silicon Graphics in 1991, MIPS Technologies became an independent company in 1998, focusing on microprocessors for the embedded marketplace. As of 2001, over 200 million MIPS microprocessors have been shipped in devices ranging from video games and palmtop computers to laser printers and network switches.

Hennessy's more recent research at Stanford focuses on the area of designing and exploiting multiprocessors. He helped lead the design of the DASH multiprocessor architecture, the first distributed shared-memory multiprocessors supporting cache coherency and the basis for several commercial multiprocessor designs, including the Silicon Graphics Origin multiprocessors.

David A. Patterson has been teaching computer architecture at the University of California, Berkeley, since joining the faculty in 1977, and he holds the Pardee Chair of Computer Science. His teaching has been honored by the ACM and the University of California. In 2000 he won the James H. Mulligan, Jr. Education Medal from IEEE "for inspirational teaching through the development of creative curricula and teaching methodology, for important textbooks, and for effective integration of education and research missions." Patterson has also received the 1995 IEEE Technical Achievement Award for contributions to RISC and shared the 1999 IEEE Reynold B. Johnson Information Storage Award for contributions to RAID. In 2000 he shared the IEEE John von Neumann Medal with John Hennessy "for creating a revolution in computer architecture through their exploration, popularization, and commercialization of architectural innovations." Patterson is a member of the National Academy of Engineering and is a Fellow of both the ACM and the IEEE. In the past, he has been chair of the CS division in the EECS department at Berkeley, the ACM SIG in computer architecture, and the Computing Research Association.

At Berkeley, Patterson led the design and implementation of RISC I, likely the first VLSI reduced instruction set computer. This research became the foundation of the SPARC architecture, currently used by Sun Microsystems, Fujitsu, and others. He was a leader of the redundant arrays of inexpensive disks (RAID) project, which led to high-performance storage systems from many companies. He was also involved in the network of workstations (NOW) project, which led to cluster technology used by Internet companies. These projects earned three dissertation awards from the ACM. His current research project is called recovery oriented computing (ROC), which is developing techniques for building dependable, maintainable, and scalable Internet services.

Computer Architecture A Quantitative Approach

Third Edition

John L. Hennessy
Stanford University

David A. Patterson
University of California at Berkeley

With Contributions by

David Goldberg
Xerox Palo Alto Research Center

Krste Asanovic
*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology*



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

AMSTERDAM BOSTON LONDON NEW YORK
OXFORD PARIS SAN DIEGO SAN FRANCISCO
SINGAPORE SYDNEY TOKYO

Senior Editor Denise E. M. Penrose
Assistant Publishing Services Manager Edward Wade
Senior Production Editor Cheri Palmer
Editorial Coordinator Alyson Day
Cover Design Ross Carron Design
Cover Image Greg Pease / gettyimages
Text Design Rebecca Evans & Associates
Technical Illustration Lineworks, Inc.
Composition Nancy Logan
Copyeditor Ken DellaPenta
Proofreader Jennifer McClain
Indexer Ty Koontz
Printer Courier Corporation

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Morgan Kaufmann Publishers
An Imprint of Elsevier Science
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
www.mkp.com

© 1990, 1996, 2003 by Elsevier Science (USA)
All rights reserved

Published 1990. Third edition 2003
Printed in the United States of America
07 06 05 04 03 02 10 9 8 7 6 5 4 3 2

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, or otherwise—without the prior written permission of the publisher.

ADVICE, PRAISE, & ERRORS: Any correspondence related to this publication or intended for the authors should be addressed to ca3comments@mkp.com. Information regarding error sightings is also encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of \$1.00 (U.S.) per correction upon availability of the new printing. Bugs can be sent to cabugs@mkp.com. (Please include your full name and permanent mailing address.)

Library of Congress Control Number: 2001099789

ISBN 1-55860-596-7 (cloth)
ISBN 1-55860-724-2 (paper)

This book is printed on acid-free paper.

To Andrea, Linda, and our four sons

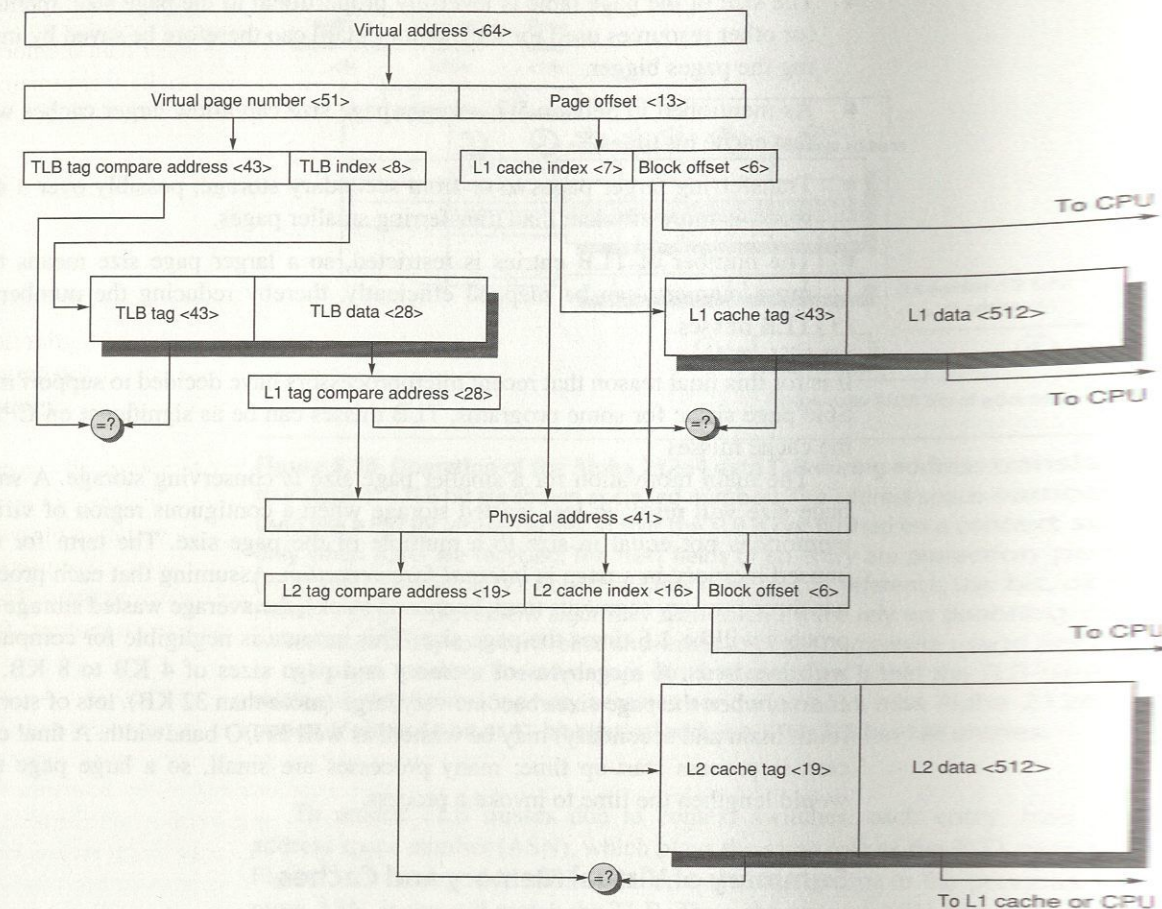


Figure 5.37 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB, and the L2 cache is a direct-mapped 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache, as in Figure 5.43, is replication of pieces of this figure.

and the data are sent to the CPU, which uses the block offset to select the desired word. On an L2 miss, the physical address is then used to get the block from memory.

Although this is a simple example, the major difference between this drawing and a real cache is replication. First, there is only one L1 cache. When there are two L1 caches, the top half of the diagram is duplicated. Note this would lead to two TLBs, which is typical. Hence, one cache and TLB is for instructions, driven from the PC, and one cache and TLB is for data, driven from the effective address. The second simplification is that all the caches and TLBs are direct

mapped. If any were n -way set associative, then we would replicate each set of tag memory, comparators, and data memory n times and connect data memories with an $n:1$ multiplexor to select a hit. Of course, if the total cache size remained the same, the cache index would also shrink by n bits according to the formula in Figure 5.9 on page 412.

5.11

Protection and Examples of Virtual Memory

The invention of multiprogramming, where a computer would be shared by several programs running concurrently, led to new demands for protection and sharing among programs. These demands are closely tied to virtual memory in computers today, and so we cover the topic here along with two examples of virtual memory.

Multiprogramming leads to the concept of a *process*. Metaphorically, a process is a program's breathing air and living space—that is, a running program plus any state needed to continue running it. Time-sharing is a variation of multiprogramming that shares the CPU and memory with several interactive users at the same time, giving the illusion that all users have their own computers. Thus, at any instant it must be possible to switch from one process to another. This exchange is called a *process switch* or *context switch*.

A process must operate correctly whether it executes continuously from start to finish, or it is interrupted repeatedly and switched with other processes. The responsibility for maintaining correct process behavior is shared by designers of the computer and the operating system. The computer designer must ensure that the CPU portion of the process state can be saved and restored. The operating system designer must guarantee that processes do not interfere with each others' computations.

The safest way to protect the state of one process from another would be to copy the current information to disk. However, a process switch would then take seconds—far too long for a time-sharing environment.

This problem is solved by operating systems partitioning main memory so that several different processes have their state in memory at the same time. This division means that the operating system designer needs help from the computer designer to provide protection so that one process cannot modify another. Besides protection, the computers also provide for sharing of code and data between processes, to allow communication between processes or to save memory by reducing the number of copies of identical information.

Protecting Processes

The simplest protection mechanism is a pair of registers that checks every address to be sure that it falls between the two limits, traditionally called *base* and *bound*. An address is valid if

$$\text{Base} \leq \text{Address} \leq \text{Bound}$$

6.1	Introduction	518
6.2	Characteristics of Application Domains	520
6.3	Symmetric Shared-Memory Architectures	521
6.4	Performance of Symmetric Shared-Memory Multiprocessors	522
6.5	Distributed Shared-Memory Architectures	523
6.6	Performance of Distributed Shared-Memory Multiprocessors	524
6.7	Synchronization	525
6.8	Models of Memory Consistency: An Introduction	526
6.9	Multithreading: Exploiting Thread-Level Parallelism within a Processor	527
6.10	Crosscutting Issues	528
6.11	Putting It All Together: Sun's Wildfire Prototype	529
6.12	Another View: Multithreading in a Commercial Server	530
6.13	Another View: Embedded Multiprocessors	531
6.14	Fallacies and Pitfalls	532
6.15	Concluding Remarks	533
6.16	Historical Perspective and References	534
	Exercises	535

6

Multiprocessors and Thread-Level Parallelism

The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer....Electronic circuits are ultimately limited in their speed of operation by the speed of light ...and many of the circuits were already operating in the nanosecond range.

W. Jack Bouknight et al.
The Illiac IV System (1972)

...sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light ...

Angel L. DeCegama
The Technology of Parallel Processing, Volume I (1989)

...today's multiprocessors ...are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.

David Mitchell
The Transputer: The Time Is Now (1989)

6.1 Introduction

As the quotations that open this chapter show, the view that advances in uniprocessor architecture were nearing an end has been widely held at varying times. To counter this view, we observe that during the period 1985–2000, uniprocessor performance growth, driven by the microprocessor, was at its highest rate since the first transistorized computers in the late 1950s and early 1960s.

On balance, though, we believe that parallel processors will definitely have a bigger role in the future. This view is driven by three observations. First, since microprocessors are likely to remain the dominant uniprocessor technology, the logical way to improve performance beyond a single processor is by connecting multiple microprocessors together. This combination is likely to be more cost-effective than designing a custom processor. Second, it is unclear whether the pace of architectural innovation that has been based for more than 15 years on increased exploitation of instruction-level parallelism can be sustained indefinitely. As we saw in Chapters 3 and 4, modern multiple-issue processors have become incredibly complex, and the performance increases achieved through increasing complexity, increasing silicon, and increasing power seem to be diminishing. Third, there appears to be slow but steady progress on the major obstacle to widespread use of parallel processors, namely, software. This progress is probably faster in the server and embedded markets, as we discussed in Chapters 3 and 4. Server and embedded applications exhibit natural parallelism that can be exploited without some of the burdens of rewriting a gigantic software base. This is more of a challenge in the desktop space.

We, however, are extremely reluctant to predict the death of advances in uniprocessor architecture. Indeed, we believe that the rapid rate of performance growth will continue at least for the next five years. Whether this pace of innovation can be sustained longer is difficult to predict but hard to bet against. Nonetheless, if the pace of progress in uniprocessors does slow down, multiprocessor architectures will become increasingly attractive.

That said, we are left with two problems. First, multiprocessor architecture is a large and diverse field, and much of the field is in its youth, with ideas coming and going and, until very recently, more architectures failing than succeeding. Given that we are already on page 528, full coverage of the multiprocessor design space and its trade-offs would require another volume. (Indeed, Culler, Singh, and Gupta [1999] cover *only* multiprocessors in their 1000-page book!) Second, such coverage would necessarily entail discussing approaches that may not stand the test of time, something we have largely avoided to this point. For these reasons, we have chosen to focus on the mainstream of multiprocessor design: multiprocessors with small to medium numbers of processors (≤ 128). Such designs vastly dominate in terms of both units and dollars. We will pay only slight attention to the larger-scale multiprocessor design space (≥ 128 processors). At the present, the future architecture of such multiprocessors is unsettled, and even the viability of that marketplace is in doubt. We will return to this topic briefly at the end of the chapter, in Section 6.15.

A Taxonomy of Parallel Architectures

We begin this chapter with a taxonomy so that you can appreciate both the breadth of design alternatives for multiprocessors and the context that has led to the development of the dominant form of multiprocessors. We briefly describe the alternatives and the rationale behind them; a longer description of how these different models were born (and often died) can be found in the historical perspective at the end of the chapter.

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 30 years ago, Flynn [1966] proposed a simple model of categorizing all computers that is still useful today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor, and placed all computers into one of four categories:

1. *Single instruction stream, single data stream (SISD)*—This category is the uniprocessor.
2. *Single instruction stream, multiple data streams (SIMD)*—The same instruction is executed by multiple processors using different data streams. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. The multimedia extensions we considered in Chapter 2 are a limited form of SIMD parallelism. Vector architectures are the largest class of processors of this type.
3. *Multiple instruction streams, single data stream (MISD)*—No commercial multiprocessor of this type has been built to date, but may be in the future. Some special-purpose stream processors approximate a limited form of this (there is only a single data stream that is operated on by successive functional units).
4. *Multiple instruction streams, multiple data streams (MIMD)*—Each processor fetches its own instructions and operates on its own data. The processors are often off-the-shelf microprocessors.

This is a coarse model, as some multiprocessors are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.

As discussed in the historical perspectives, many of the early multiprocessors were SIMD, and the SIMD model received renewed attention in the 1980s, and except for vector processors, was gone by the mid-1990s. MIMD has clearly emerged as the architecture of choice for general-purpose multiprocessors. Two factors are primarily responsible for the rise of the MIMD multiprocessors:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.

2. MIMDs can build on the cost-performance advantages of off-the-shelf microprocessors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers.

With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. Recall from the last chapter that a process is a segment of code that may be run independently, and that the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of the processes on other processors.

It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called *threads*. Today, the term *thread* is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space.

To take advantage of an MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute. The independent threads are typically identified by the programmer or created by the compiler. Since the parallelism in this situation is contained in the threads, it is called *thread-level parallelism*.

Threads may vary from large-scale, independent processes—for example, independent programs running in a multiprogrammed fashion on different processors—to parallel iterations of a loop, automatically generated by a compiler and each executing for perhaps less than a thousand instructions. Although the size of a thread is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction is that such parallelism is identified at a high level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel. In contrast, instruction-level parallelism is identified primarily by the hardware, although with software help in some cases, and is found and exploited one instruction at a time.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictate a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization because what constitutes a small or large number of processors is likely to change over time.

The first group, which we call *centralized shared-memory architectures*, has at most a few dozen processors in 2000. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By replacing a single bus with multiple buses, or even a switch, a centralized shared-memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable, sharing a centralized memory, even organized as multiple banks, becomes less attractive as the number of processors sharing it increases.

Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are often called *symmetric (shared-memory) multiprocessors (SMPs)*, and this style of architecture is sometimes called *uniform memory access (UMA)*. This type of centralized shared-memory architecture is currently by far the most popular organization. Figure 6.1 shows what these multiprocessors look like. The architecture of such multiprocessors is the topic of Section 6.3.

The second group consists of multiprocessors with physically distributed memory. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the scale of multiprocessor for which distributed memory is preferred over a single, centralized memory continues to decrease in number (which is another reason not to use small and large scale). Of course, the larger number of processors raises the need for a high bandwidth interconnect, of which we will see examples in Chapter 8. Both direct interconnection networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used. Figure 6.2 shows what these multiprocessors look like.

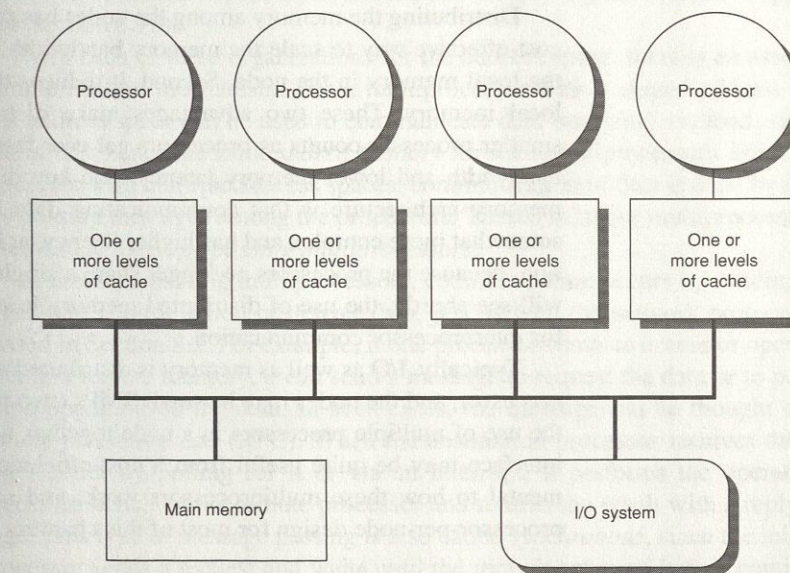


Figure 6.1 Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by a bus. In larger designs, multiple buses, or even a switch may be used, but the key architectural property—uniform access time to all memory from all processors—remains.

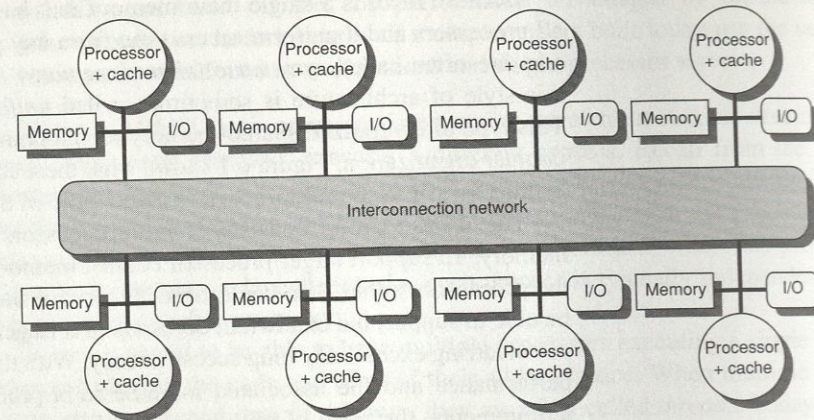


Figure 6.2 The basic architecture of a distributed-memory multiprocessor consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes. Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency. The key disadvantage for a distributed-memory architecture is that communicating data between processors becomes somewhat more complex and has higher latency, at least when there is no contention, because the processors no longer share a single, centralized memory. As we will see shortly, the use of distributed memory leads to two different paradigms for interprocessor communication.

Typically, I/O as well as memory is distributed among the nodes of the multiprocessor, and the nodes may be small SMPs (two to eight processors). Although the use of multiple processors in a node together with a memory and a network interface may be quite useful from a cost-efficiency viewpoint, it is not fundamental to how these multiprocessors work, and so we will focus on the one-processor-per-node design for most of this chapter.

Models for Communication and Memory Architecture

As discussed earlier, any large-scale multiprocessor must use multiple memories that are physically distributed with the processors. There are two alternative architectural approaches that differ in the method used for communicating data among processors.

In the first method, communication occurs through a shared address space. That is, the physically separate memories can be addressed as one logically shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. These multiprocessors are called *distributed shared-memory* (DSM) architectures. The term *shared memory* refers to the fact that the *address space* is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does *not* mean that there is a single, centralized memory. In contrast to the symmetric shared-memory multiprocessors, also known as UMAs (uniform memory access), the DSM multiprocessors are also called NUMAs (nonuniform memory access), since the access time depends on the location of a data word in memory.

Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor. In such multiprocessors, the same physical address on two different processors refers to two different locations in two different memories. Each processor-memory module is essentially a separate computer; therefore, these parallel processors have been called *multicomputers*. A multicomputer can even consist of completely separate computers connected on a local area network, which today are popularly called *clusters*. For applications that require little or no communication and can make use of separate memories, such clusters of processors, whether using a standardized or customized interconnect, can form a very cost-effective approach (see Section 8.10).

With each of these organizations for the address space, there is an associated communication mechanism. For a multiprocessor with a shared address space, that address space can be used to communicate data implicitly via load and store operations; hence the name *shared memory* for such multiprocessors. For a multiprocessor with multiple address spaces, communication of data is done by explicitly passing messages among the processors. Therefore, these multiprocessors are often called *message-passing multiprocessors*.

In message-passing multiprocessors, communication occurs by sending messages that request action or deliver data, just as with the network protocols discussed in Section 8.2. For example, if one processor wants to access or operate on data in a remote memory, it can send a message to request the data or to perform some operation on the data. In such cases, the message can be thought of as a *remote procedure call* (RPC). When the destination processor receives the message, either by polling for it or via an interrupt, it performs the operation or access on behalf of the remote processor and returns the result with a reply message. This type of message passing is also called *synchronous*, since the initiating processor sends a request and waits until the reply is returned before continuing. Software systems have been constructed to encapsulate the details of sending and receiving messages, including passing complex arguments or return values, presenting a clean RPC facility to the programmer.

Communication can also occur from the viewpoint of the writer of data rather than the reader, and this can be more efficient when the processor producing data