Web server via the telephone system using ADSL, which was discussed in Chap. 2. (Alternatively, cable TV can be used, in which case the left-hand part of Fig. 8-14 is slightly different and the cable company is the ISP.) The user's computer breaks the data to be sent to the server into packets and sends these packets to the user's **ISP** (**Internet Service Provider**), a company that offers Internet access to its customers. The ISP has a high-speed (fiber-optic) connection to one of the regional or backbone networks that comprise the Internet. The user's packets are forwarded hop-by-hop across the Internet until they arrive at the Web server.

Most companies offering Web service have a specialized computer called a **firewall** that filters all incoming traffic in an attempt to remove unwanted packets (e.g., from hackers trying to break in). The firewall is connected to the local LAN, typically an Ethernet switch, which routes packets to the desired server. Of course, reality is a lot more complicated than we have shown, but the basic idea of Fig. 8-14 is still valid.

Network software consists of multiple **protocols**, each one being a set of formats, exchange sequences, and rules about what the packets mean. For example, when a user wants to fetch a Web page from a server, the user's browser sends a packet containing a *GET PAGE* request using the **HTTP** (**HyperText Transfer Protocol**) to the server, which understands how to process such requests. Many protocols are in use and often combined. In most situations, protocols are structured as a series of layers. Upper layers hand packets to lower layers for processing, with the bottom layer doing the actual transmission. At the receiving side, the packets work their way up the layers in the reverse order.

Since protocol processing is what network processors do for a living, it is necessary to explain a little bit about protocols before looking at the network processors themselves. Let us go back for a moment to the *GET PAGE* request. How is that sent to the Web server? The browser first establishes a connection to the Web server using a protocol called **TCP** (**Transmission Control Protocol**). The software that implements this protocol checks that all packets have been correctly received and in the proper order. If a packet gets lost, the TCP software assures that it is retransmitted as often as need be until it is received.

In practice, what happens is that the Web browser formats the *GET PAGE* request as a correct HTTP message and then hands it to the TCP software to transmit over the connection. The TCP software adds a header in front of the message containing a sequence number and other information. This header is naturally called the **TCP header**.

When it is done, the TCP software takes the TCP header and payload (containing the *GET PAGE* request) and passes it to another piece of software that implements the **IP protocol** (**Internet Protocol**). This software attaches an **IP header** to the front containing the source address (the machine the packet is coming from), the destination address (the machine the packet is supposed to go to), how many more hops the packet may live (to prevent lost packets from living forever), a checksum (to detect transmission and memory errors), and other fields.

Next the resulting packet (now consisting of the IP header, TCP header, and *GET PAGE* request) is passed down to the data link layer, where a data link header is attached to the front for actual transmission. The data link layer also adds a checksum to the end called a **CRC** (**Cyclic Redundancy Code**) used to detect transmission errors. It might seem that having checksums in both the data link layer and the IP layer is redundant, but it improves reliability. At each hop, the CRC is checked and the header and CRC stripped and regenerated, with a format being chosen that is appropriate for the outgoing link. Figure 8-15 shows what the packet looks like when on the Ethernet. On a telephone line (for ADSL) it is similar except with a "telephone-line header" instead of an Ethernet header. Header management is important and is one of the things network processors can do. Needless to say, we have only scratched the surface of the subject of computer networking. For a more comprehensive treatment, see Tanenbaum and Wetherall (2011).

Figure 8-15. A packet as it appears on the Ethernet.

Introduction to Network Processors

Many kinds of devices are connected to networks. End users have personal computers (desktop and notebook), of course, but increasingly also game machines, PDAs (palmtops), and smartphones. Companies have PCs and servers as end systems. However, there are also numerous devices that function as intermediate systems in networks, including routers, switches, firewalls, Web proxies, and load balancers. Interestingly enough, the intermediate systems are the most demanding, since they are expected to move the largest number of packets per second. Servers are also demanding but the user machines are not.

Depending on the network and the packet itself, an incoming packet may need various kinds of processing before being forwarded to either the outgoing line or the application program. This processing may include deciding where to send the packet, fragmenting it or reassembling its pieces, managing its quality of service (especially for audio and video streams), managing security (e.g., encryption or decryption), compression/decompression, and so on.

With LAN speeds approaching 40 gigabits/sec and 1-KB packets, a networked computer might have to process almost 5 million packets/sec. With 64-byte packets, the number of packets that have to be processed per second rises to nearly 80 million. Performing the various functions mentioned above in 12–200 nsec (in addition to making the multiple copies of the packet that are invariably needed) is just not doable in software. Hardware assistance is essential.

One kind of hardware solution for fast packet processing is to use a custom **ASIC** (**Application-Specific Integrated Circuit**). Such a chip is like a hardwired program that does whatever set of processing functions it was designed for. Many current routers uses ASICs. ASICs have many problems, however. First, they take a long time to design and manufacture. They are also rigid, so if new functionality is needed, a new chip is needed. Furthermore, bug management is a nightmare, since the only way to fix one is to design, manufacture, ship, and install new chips. They are also expensive unless the volume is so large as to allow amortizing the development effort over a substantial number of chips.

A second solution is the **FPGA** (**Field Programmable Gate Array**), a collection of gates that can be organized into the desired circuit by rewiring them in the field. These chips have a much shorter time to market than ASICs and can be rewired in the field by removing them from the system and inserting them into a special reprogramming device. On the other hand, they are complex, slow, and expensive, making them unattractive except for niche applications.

Finally, we come to **network processors**, programmable devices that can handle incoming and outgoing packets at wire speed (i.e., in real time). A common design is a plug-in board containing a network processor on a chip along with memory and support logic. One or more network lines connect to the board and are routed to the network processor. There packets are extracted, processed, and either sent out on a different network line (e.g., for a router) or are sent out onto the main system bus (e.g., the PCI bus) in the case of end-user device such as a PC. A typical network processor board and chip are illustrated in Fig. 8-16.

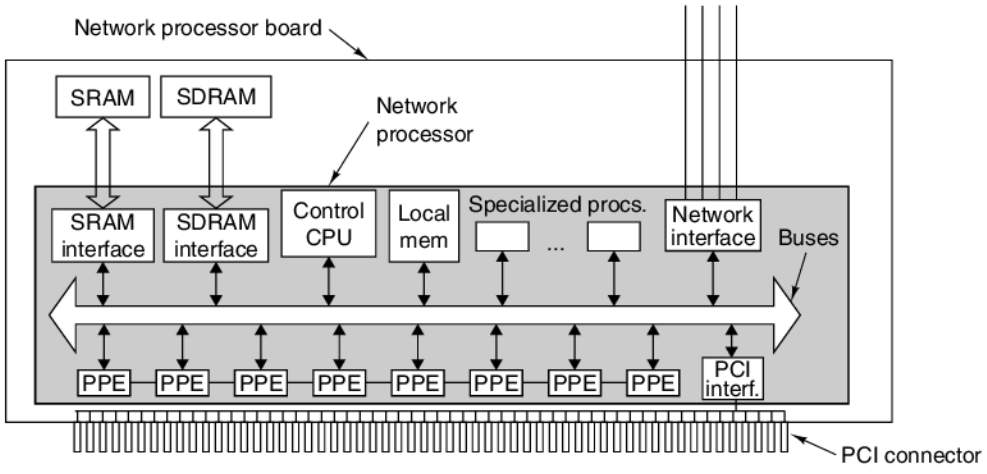


Figure 8-16. A typical network processor board and chip.

Both SRAM and SDRAM are provided on the board and typically used in different ways. SRAM is faster, but more expensive, than SDRAM, so there is only a relatively small amount of it. SRAM is used to hold routing tables and other key

data structures, whereas SDRAM holds the actual packets being processed. Making the SRAM and SDRAM external to the network processor chip gives the board designers the flexibility to determine how much of each to supply. In this way, low-end boards with a single network line (e.g., for a PC or server) can be equipped with a small amount of memory whereas a high-end board for a large router can be equipped with much more.

Network processor chips are optimized for quickly processing large numbers of incoming and outgoing packets. Millions of packets per second per network line is the norm and a router could easily have half a dozen lines. The only way to achieve such processing rates is to build network processors that are highly parallel inside. Indeed, all network processors consist of multiple **PPE**s, variously called **Protocol/Programmable/Packet Processing Engines**. Each one consists of a (possibly modified) RISC core and a small amount of internal memory for holding the program and some variables.

The PPEs can be organized in two ways. The simplest organization is having all the PPEs identical. When a packet arrives at the network processor, either an incoming packet from a network line or an outgoing packet from the bus, it is handed to an idle PPE for processing. If no PPE is idle, the packet is queued in the on-board SDRAM until a PPE frees up. When this organization is used, the horizontal connections shown between the PPEs in Fig. 8-16 do not exist because the PPEs have no need to communicate with one another.

The other PPE organization is the pipeline. In this one, each PPE performs one processing step and then feeds a pointer to its output packet to the next PPE in the pipeline. In this way, the PPE pipeline acts very much like the CPU pipelines we studied in Chap. 2. In both organizations, the PPEs are completely programmable.

In advanced designs, the PPEs have multithreading, meaning that they have multiple register sets and a hardware register indicating which one is currently in use. This feature is used to run multiple programs at the same time by allowing a program (i.e., thread) switch by just changing the "current register set" variable. Most commonly, when a PPE stalls, for example, when it references the SDRAM (which takes multiple clock cycles), it can instantaneously switch to a runnable thread. In this manner, a PPE can achieve a high utilization even when frequently blocking to access the SDRAM or perform some other slow external operation.

In addition to the PPEs, all network processors contain a control processor, usually just a standard general-purpose RISC CPU, for performing all work not related to packet processing, such as updating the routing tables. Its program and data are in the local on-chip memory. Furthermore, many network-processor chips also contain one or more specialized processors for doing pattern matching or other critical operations. These processors are really small ASICs that are good at one simple operation, such as looking up a destination address in the routing table. All the components of the network processor communicate over one or more on-chip, parallel buses that run at multigigabit/sec speeds.

Packet Processing

When a packet arrives, it goes through a number of processing stages, independent of whether the network processor has a parallel or pipeline organization. Some network processors divide these steps into operations performed on incoming packets (either from a network line or from the system bus), called **ingress processing**, and operations performed on outgoing packets, called **egress processing**. When this distinction is made, every packet goes first through ingress processing, then through egress processing. The boundary between ingress and egress processing is flexible because some steps can be done in either part (e.g., collecting traffic statistics).

Below we will discuss a potential ordering of the various steps, but note that not all packets need all steps and that many other orderings are equally valid

1. **Checksum verification.** If the incoming packet is arriving from the Ethernet, the CRC is recomputed so it can be compared with the one in the packet to make sure there was no transmission error. If the Ethernet CRC is correct or not present, the IP checksum is recomputed and compared to the one in the packet to make sure the IP packet was not damaged by a faulty bit in the sender's memory after the IP checksum was computed there. If all checksums are correct, the packet is accepted for further processing; otherwise, it is simply discarded.
2. **Field extraction.** The relevant header is parsed and key fields are extracted. In an Ethernet switch, only the Ethernet header is examined, whereas in an IP router, it is the IP header that is inspected. The key fields are stored in registers (parallel PPE organization) or SRAM (pipeline organization).
3. **Packet classification.** The packet is classified according to a series of programmable rules. The simplest classification is to distinguish data packets from control packets, but usually much finer distinctions are made.
4. **Path selection.** Most network processors have a special fast path optimized for handling plain old garden-variety data packets, with all other packets being treated differently, often by the control processor. Consequently, either the fast or the slow path has to be selected.
5. **Destination network determination.** IP packets contain a 32-bit destination address. It is not possible (or even desirable) to have a 2^{32} entry table to look up the destination of each IP packet, so the leftmost part of each IP address is the network number and the rest specifies a machine on that network. Network numbers can be of any

length, so determining the destination network number is nontrivial and made worse by the fact that multiple matches are possible and the longest one counts. Often a custom ASIC is used in this step.

6. **Route lookup.** Once the number of the destination network is known, the outgoing line to use can be looked up in a table in the SRAM. Again, a custom ASIC may be used in this step.
7. **Fragmentation and reassembly.** Programmers like to present large payloads to the TCP layer to reduce the number of system calls needed, but TCP, IP, and Ethernet all have maximum sizes for the packets they can handle. As a consequence of these limits, payloads and packets may have to be fragmented at the sending side and the pieces reassembled at the receiving side. These are tasks the network processor can perform.
8. **Computation.** Heavy-duty computation on the payload is sometimes required, for example, data compression/decompression and encryption/decryption. These are tasks a network processor can perform.
9. **Header management.** Sometimes headers have to be added, removed, or have some of their fields modified. For example, the IP header has a field that counts the number of hops the packet may yet make before being discarded. Every time it is retransmitted, this field must be decremented, something the network processor can do.
10. **Queue management.** Incoming and outgoing packets often have to be queued while waiting their turn at being processed. Multimedia applications may need a certain interpacket spacing in time to avoid jitter. A firewall or router may need to distribute the incoming load among multiple outgoing lines according to certain rules. All of these tasks can be done by the network processor.
11. **Checksum generation.** Outgoing packets need to be checksummed. The IP checksum can be generated by the network processor, but the Ethernet CRC is generally computed by hardware.
12. **Accounting.** In some cases, accounting for packet traffic is needed, especially when one network is forwarding traffic for other networks as a commercial service. The network processor can do the accounting.
13. **Statistics gathering.** Finally, many organizations like to collect statistics about their traffic. They want to know how many packets came and and how many went out, at what times of day, and more. The network processor is a good place to collect them.

Improving Performance

Performance is the name of the game for network processors. What can be done to improve it? But before improving it, we have to define what performance means. One metric is the number of packets forwarded per second. A second one is the number of bytes forwarded per second. These are different measures, and a scheme that works well with small packets may not work as well with large ones. In particular, with small packets, improving the number of destination lookups per second may help a lot, but with large packets it may not.

The most straightforward way to improve performance is to increase the speed of the network processor clock. Of course, performance is not linear with clock speed, since memory cycle time and other factors also influence it. Also, a faster clock means more heat must be dissipated.

Introducing more PPEs and parallelism is often a winner, especially with an organization consisting of parallel PPEs. A deeper pipeline can also help, but only if the job of processing a packet can be split up into smaller pieces.

Another technique is to add specialized processors or ASICs to handle specific, time-consuming operations that are performed repeatedly and that can be done faster in hardware than in software. Lookups, checksum computations, and cryptography are among the many candidates.

Adding more internal buses and widening existing buses may help gain speed by moving packets through the system faster. Finally, replacing SDRAM by SRAM can usually be counted to improve performance, but at a price, of course.

There is much more that can be said about network processors, of course. Some references are Freitas et al. (2009), Lin et al. (2010), and Yamamoto and Nakao (2011).

8.2.2 Graphics Processors

A second area in which coprocessors are used is for handling high-resolution graphics processing, such as 3D rendering. Ordinary CPUs are not especially good at the massive computations needed to process the large amounts of data required for these applications. For this reason, most PCs and many future processors will be equipped with **GPU**s (**Graphics Processing Units**) to which they can offload large portions of overall processing.

The NVIDIA Fermi GPU

We will study this increasingly important area by means of an example: the NVIDIA **Fermi GPU**, an architecture used in a family of graphics processing chips that are available at several speeds and sizes. The architecture of the Fermi GPU is shown in Fig. 8-17. It is organized into 16 **SM**s (**Streaming Multiprocessors**) having its own high-bandwidth private level-1 cache. Each of the streaming

multiprocessors contain 32 CUDA cores, for a total of 512 CUDA cores per Fermi GPU. A **CUDA** (**Compute Unified Device Architecture**) core is a simple processor supporting single-precision integer and floating-point computations. A single SM with 32 CUDA cores is illustrated in Fig. 2-7. The 16 SMs share access to a single unified 768-KB level 2 cache, which is connected to a multiported DRAM interface. The host processor interface provides a communication path between the host system and the GPU via a shared DRAM bus interface, typically through a PCI-Express interface.

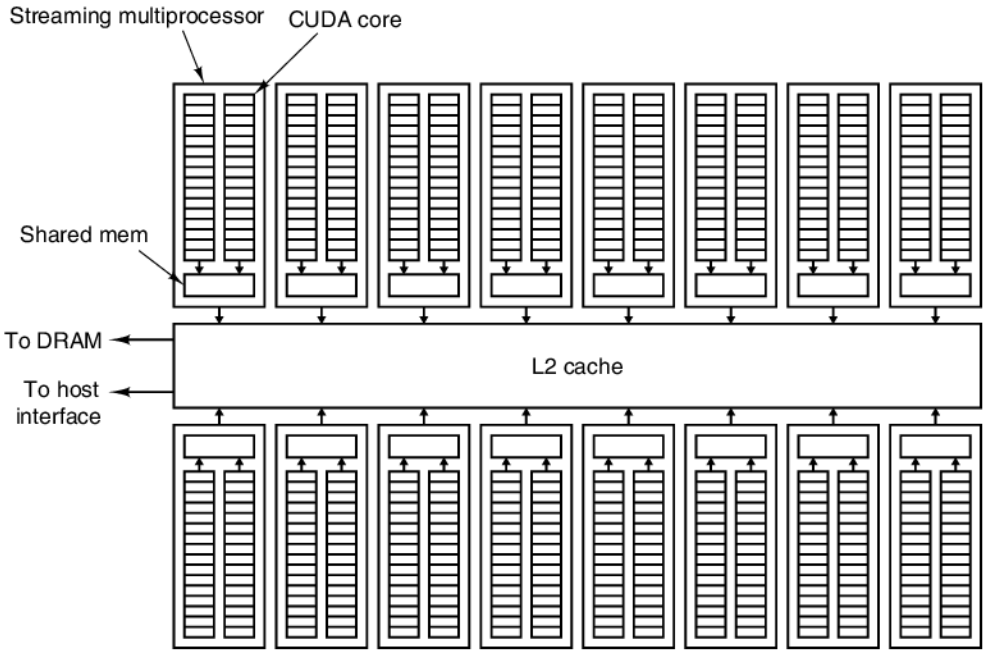


Figure 8-17. The Fermi GPU architecture.

The Fermi architecture is designed to efficiently execute graphics, video, and image processing codes, which typically have many redundant computations spread across many pixels. Because of this redundancy, the streaming multiprocessors, while capable of executing 16 operations at a time, require that all of the operations executed in a single cycle be identical. This style of processing is called **SIMD** (**Single-Instruction Multiple Data**) computation, and it has the important advantage that each SM fetches and decodes only a single instruction each cycle. Only by sharing the instruction processing across all of the cores in an SM can NVIDIA cram 512 cores onto a single silicon die. If programmers can harness all of the computation resources (always a very big and uncertain "if"), then the system provides significant computational advantages over traditional scalar architectures, such as the Core i7 or OMAP4430.

The SIMD processing requirements within the SMs place constraints on the kind of code programmers can run on these units. In fact, each CUDA core must be running the same code in lock-step to achieve 16 operations simultaneously. To ease this burden on programmer, NVIDIA developed the CUDA programming language. The CUDA language specifies the program parallelism using threads. Threads are then grouped into blocks, which are assigned to streaming processors. As long as every thread in a block executes exactly the same code sequence (that is, all branches make the same decision), up to 16 operations will execute simultaneously (assuming there are 16 threads ready to execute). When threads on an SM make different branch decisions, a performance-degrading effect called branch divergence will occur that forces threads with differing code paths to execute serially on the SM. Branch divergence reduces parallelism and slows GPU processing. Fortunately, there is a wide range of activities in graphics and image processing that can avoid branch divergence and achieve good speed-ups. Many other codes have also been shown to benefit from the SIMD-style architecture on graphics processors, such as medical imaging, proof solving, financial prediction, and graph analysis. This widening of potential applications for GPUs has earned them the new moniker of **GPGPU**s (**General-Purpose Graphics Processing Units**).

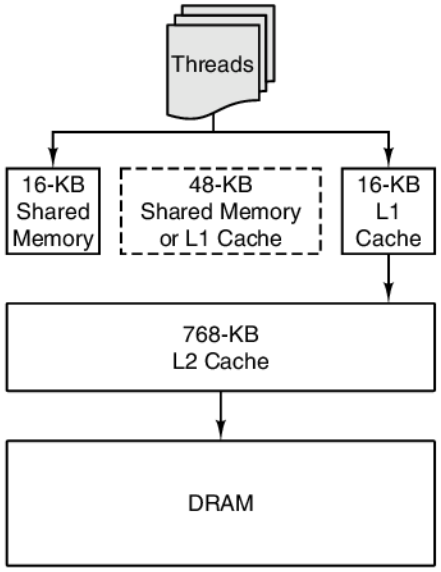


Figure 8-18. The Fermi GPU memory hierarchy.

With 512 CUDA cores, the Fermi GPU would grind to a halt without significant memory bandwidth. To provide this bandwidth, the Fermi GPU implements a modern memory hierarchy as illustrated in Fig. 8-18. Each SM has both a dedicated shared memory and a private level 1 data cache. The dedicated shared memory

is directly addressed by the CUDA cores, and it provides fast sharing of data between threads within a single SM. The level 1 cache speeds up accesses to DRAM data. To accommodate the wide range of program data usage, the SMs can be configured with 16-KB shared memory and 48-KB level 1 cache or 48-KB shared memory and 16-KB level 1 cache. All SMs share a single unified 768-KB level 2 cache. The level 2 cache provides faster access to DRAM data that do not fit in the level 1 caches. The level 2 cache also provides sharing between SMs, although this mode of sharing is much slower than the intra-SM sharing that occurs within an SM's shared memory. Beyond the level 2 cache is the DRAM, which holds the remaining data, imagery, and textures used by programs running on the Fermi GPU. Efficient programs will try to avoid accessing DRAM at all costs, as a single access can take hundreds of cycles to complete.

For a savvy programmer, the Fermi GPU represents one of the most computationally capable platforms ever created. A single Fermi-based GTX 580 GPU running at 772 MHz with 512 CUDA cores can achieve a sustained computational rate of 1.5 teraflops while consuming 250 watts of power. This statistic is even more impressive when one considers that the street price of a GTX 580 GPU is less than 600 U.S. dollars. By way of historical comparison, in 1990, the fastest computer in the world, the Cray-2, had a performance of 0.002 teraflops and a price tag (in inflation-adjusted dollars) of $30 million. It also filled a modest-sized room and came with its own liquid-cooling system to dissipate the 150 kW of power it consumed. The GTX 580 has 750 times more horsepower for 1/50000 of the price while consuming 1/600th as much energy. Not a bad deal.

8.2.3 Cryptoprocessors

A third area in which coprocessors are popular is security, especially network security. When a connection is established between a client and a server, in many cases they must first authenticate each other. Then a secure, encrypted connection has to be established between them so data can be transferred in a secure way to foil any snoopers who may tap the line.

The problem with security is that to achieve it, cryptography has to be used, and cryptography is very compute intensive. Cryptography comes in two general flavors, called **symmetric key cryptography** and **public-key cryptography**. The former is based on mixing up the bits very thoroughly, sort of the electronic equivalent of throwing a message into an electric blender. The latter is based on multiplication and exponentiation of large (e.g., 1024-bit) numbers and is extremely time consuming.

To handle the computation needed to encrypt data securely for transmission or storage and then decrypt them later, various companies have produced crypto coprocessors, sometimes as PCI bus plug-in cards. These coprocessors have special hardware that enables them to do the necessary cryptography much faster than an ordinary CPU can do it. Unfortunately, a detailed discussion of how they work

would first require explaining quite a bit about cryptography itself, which is beyond the scope of this book. For more information about crypto coprocessors, see Gaspar et al. (2010), Haghighizadeh et al. (2010), and Shoufan et al. (2011).

8.3 SHARED-MEMORY MULTIPROCESSORS

We have now seen how parallelism can be added to single chips and to individual systems by adding a coprocessor. The next step is to see how multiple full-blown CPUs can be combined into larger systems. Systems with multiple CPUs can be divided into multiprocessors and multicomputers. After taking a close look at what these terms actually mean, we will first study multiprocessors and then multicomputers.

8.3.1 Multiprocessors vs. Multicomputers

In any parallel computer system, CPUs working on different parts of the same job must communicate to exchange information. Precisely how they should do this is the subject of much debate in the architectural community. Two distinct designs, multiprocessors and multicomputers, have been proposed and implemented. The key difference between the two is the presence or absence of shared memory. This difference permeates how they are designed, built, and programmed, as well as their scale and price.

Multiprocessors

A parallel computer in which all the CPUs share a common memory is called a **multiprocessor**, as indicated symbolically in Fig. 8-19. All processes working together on a multiprocessor can share a single virtual address space mapped onto the common memory. Any process can read or write a word of memory by just executing a LOAD or STORE instruction. Nothing else is needed. The hardware does the rest. Two processes can communicate by simply having one of them write data to memory and having the other one read them back.

The ability for two (or more) processes to communicate by just reading and writing memory is the reason multiprocessors are popular. It is an easy model for programmers to understand and is applicable to a wide range of problems. Consider, for example, a program that inspects a bit-map image and lists all the objects in it. One copy of the image is kept in memory, as shown in Fig. 8-19(b). Each of the 16 CPUs runs a single process, which has been assigned one of the 16 sections to analyze. Nevertheless, each process has access to the entire image, which is essential, since some objects may occupy multiple sections. If a process discovers that one of its objects extends over a section boundary, it just follows the object

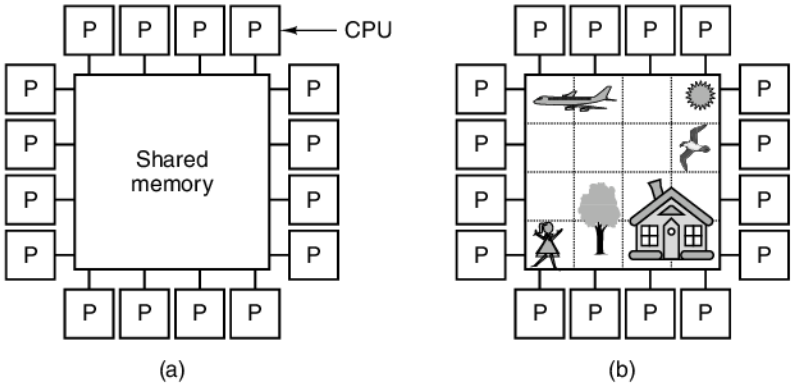


Figure 8-19. (a) A multiprocessor with 16 CPUs sharing a common memory. (b) An image partitioned into 16 sections, each being analyzed by a different CPU.

into the next section by reading the words of that section. In this example, some objects will be discovered by multiple processes, so some coordination is needed at the end to determine how many houses, trees, and airplanes there are.

Because all CPUs in a multiprocessor see the same memory image, there is only one copy of the operating system. Consequently, there is only one page map and one process table. When a process blocks, its CPU saves its state in the operating-system tables, then looks in those tables to find another process to run. It is this single-system image that distinguishes a multiprocessor from a multicomputer, in which each computer has its own copy of the operating system.

A multiprocessor, like all computers, must have I/O devices, such as disks, network adapters, and other equipment. In some multiprocessor systems, only certain CPUs have access to the I/O devices, and thus have a special I/O function. In other ones, every CPU has equal access to every I/O device. When every CPU has equal access to all the memory modules and all the I/O devices, and is treated as interchangeable with the others by the operating system, the system is called an **SMP** (**Symmetric MultiProcessor**).

Multicomputers

The second possible design for a parallel architecture is one in which each CPU has its own private memory, accessible only to itself and not to any other CPU. Such a design is called a **multicomputer**, or sometimes a **distributed memory system**, and is illustrated in Fig. 8-20(a). The key aspect of a multicomputer that distinguishes it from a multiprocessor is that each CPU in a multicomputer has its own private, local memory that it can access by just executing LOAD and STORE instructions but that no other CPU can access using LOAD and STORE instructions.

Thus multiprocessors have a single physical address space shared by all the CPUs, whereas multicomputers have one physical address space per CPU.

Since the CPUs on a multicomputer cannot communicate by just reading and writing the common memory, they need a different communication mechanism. What they do is pass messages back and forth using the interconnection network. Examples of multicomputers include the IBM BlueGene/L, Red Storm, and the Google cluster.

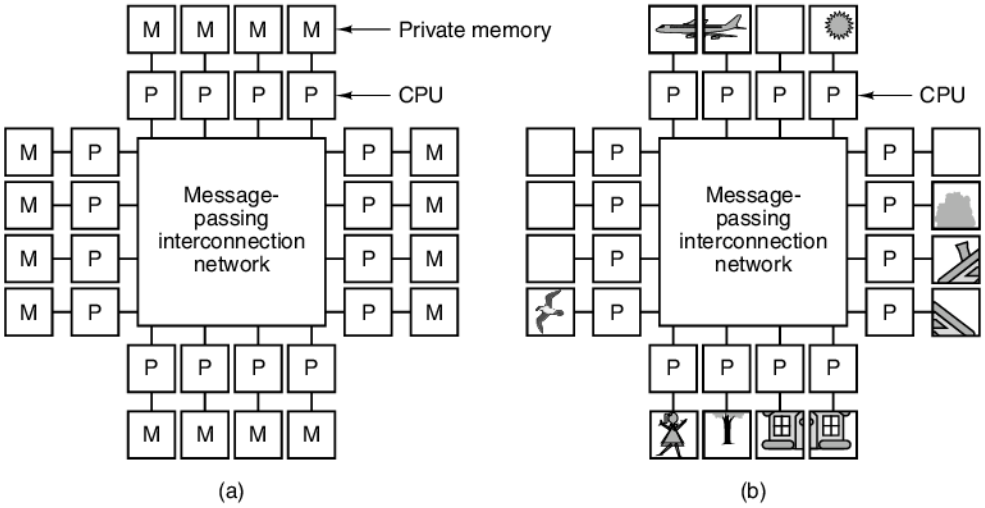


Figure 8-20. (a) A multicomputer with 16 CPUs, each with its own private memory. (b) The bit-map image of Fig. 8-19 split up among the 16 memories.

The absence of hardware shared memory on a multicomputer has important implications for the software structure. Having a single virtual address space with all processes being able to read and write all of memory by just executing LOAD and STORE instructions is impossible on a multicomputer. For example, if CPU 0 (the one in the upper left-hand corner) of Fig. 8-19(b) discovers that part of its object extends into the section assigned to CPU 1, it can nevertheless just continue reading memory to access the tail of the airplane. On the other hand, if CPU 0 in Fig. 8-20(b) makes the same discovery, it cannot just read CPU 1's memory. It has to do something quite different to get the data it needs.

In particular, it has to discover (somehow) which CPU has the data it needs and send that CPU a message requesting a copy of the data. Typically it will then block until the request is answered. When the message arrives at CPU 1, software there has to analyze it and send back the needed data. When the reply message gets back to CPU 0, the software is unblocked and can continue executing.

On a multicomputer, communication between processes often uses software primitives such as send and receive. This gives the software a different, and far more complicated, structure than on a multiprocessor. It also means that correctly

dividing up the data and placing them in the optimal locations is a major issue on a multicomputer. It is less of an issue on a multiprocessor since placement does not affect correctness or programmability although it may affect performance. In short, programming a multicomputer is much more difficult than programming a multiprocessor.

Under these conditions, why would anyone build multicomputers, when multiprocessors are easier to program? The answer is simple: large multicomputers are much simpler and cheaper to build than multiprocessors with the same number of CPUs. Implementing a memory shared by even a few hundred CPUs is a substantial undertaking, whereas building a multicomputer with 10,000 CPUs or more is straightforward. Later in this chapter we will study a multicomputer with over 50,000 CPUs.

Thus we have a dilemma: multiprocessors are hard to build but easy to program whereas multicomputers are easy to build but hard to program. This observation has led to a great deal of effort to construct hybrid systems that are relatively easy to build and relatively easy to program. This work has led to the realization that shared memory can be implemented in various ways, each with its own set of advantages and disadvantages. In fact, much research in parallel architectures these days relates to the convergence of multiprocessor and multicomputer architectures into hybrid forms that combine the strengths of each. The holy grail here is to find designs that are **scalable**, that is, continue to perform well as more and more CPUs are added.

One approach to building hybrid systems is based on the fact that modern computer systems are not monolithic but are constructed as a series of layers—the theme of this book. This insight opens the possibility of implementing the shared memory at any one of several layers, as shown in Fig. 8-21. In Fig. 8-21(a) we see the shared memory being implemented by the hardware as a true multiprocessor. In this design, there is a single copy of the operating system with a single set of tables, in particular, the memory allocation table. When a process needs more memory, it traps to the operating system, which then looks in its table for a free page and maps the page into the called's address space. As far as the operating system is concerned, there is a single memory and it keeps track of which process owns which page in software. There are many ways to implement hardware shared memory, as we will see later.

A second possibility is to use multicomputer hardware and have the operating system simulate shared memory by providing a single system-wide paged shared virtual address space. In this approach, called **DSM** (**Distributed Shared Memory**) (Li and Hudak, 1989), each page is located in one of the memories of Fig. 8-20(a). Each machine has its own virtual memory and its own page tables. When a CPU does a LOAD or STORE on a page it does not have, a trap to the operating system occurs. The operating system then locates the page and asks the CPU currently holding it in its memory to unmap the page and send it over the interconnection network. When it finally arrives, the page is mapped in and the faulting

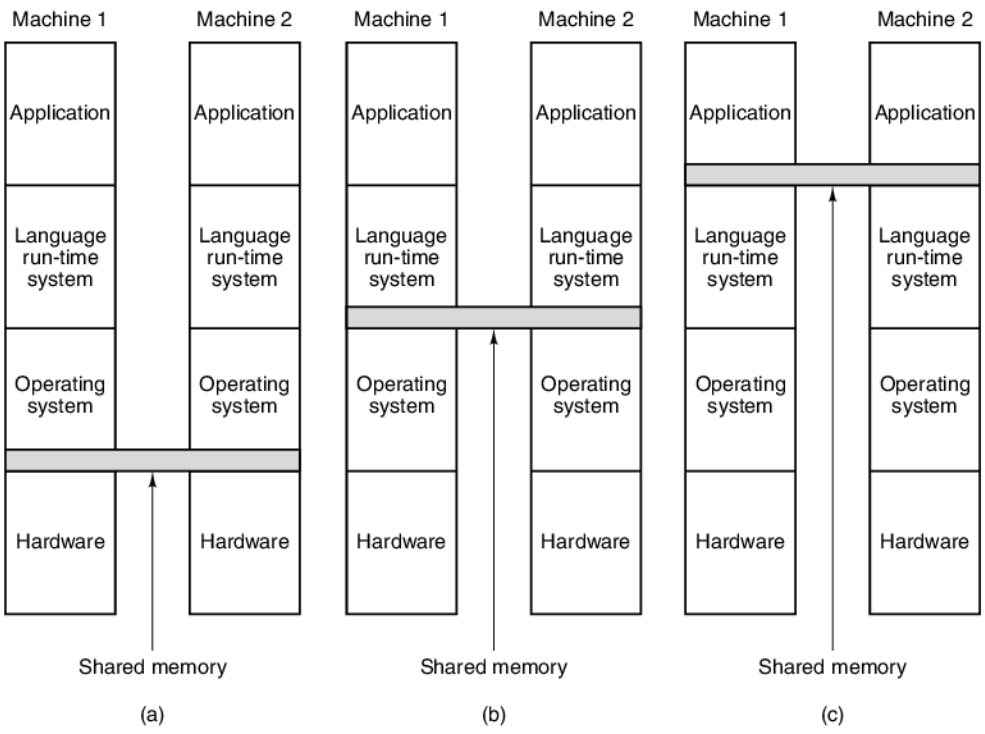


Figure 8-21. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) The language run-time system.

instruction restarted. In effect, the operating system is just satisfying page faults from remote memory instead of from disk. To the user, the machine looks as if it has shared memory. We will examine DSM later in this chapter.

A third possibility is to have a user-level run-time system implement a (possibly language-specific) form of shared memory. In this approach, the programming language provides some kind of shared-memory abstraction, which is then implemented by the compiler and run-time system. For example, the Linda model is based on the abstraction of a shared space of tuples (data records containing a collection of fields). Processes on any machine can input a tuple from the shared tuple space or output a tuple to the shared tuple space. Because access to the tuple space is controlled entirely in software (by the Linda run-time system), no special hardware or operating system support is needed.

Another example of a language-specific shared memory implemented by the run-time system is the Orca model of shared data objects. In Orca, processes share generic objects rather than just tuples and can execute object-specific methods on them. When a method call changes the internal state of an object, it is up to the run-time system to make sure all copies of the object on all machines are updated

simultaneously. Again, because objects are a strictly software concept, the implementation can be done by the run-time system without help from the operating system or hardware. We will look at both Linda and Orca later in this chapter.

Taxonomy of Parallel Computers

Now let us get back to our main topic, the architecture of parallel computers. Many kinds of parallel computers have been proposed and built over the years, so it is natural to ask if there is some way of categorizing them into a taxonomy. Many researchers have tried, with mixed results (Flynn, 1972, and Treleaven, 1985). Unfortunately, the Carolus Linnaeus† of parallel computing is yet to emerge. The only scheme that is used much is Flynn's, and even his is, at best, a very crude approximation. It is given in Fig. 8-22.

Instruction streams	Data streams	Name	Examples
1	1	SISD	Classical Von Neumann machine
1	Multiple	SIMD	Vector supercomputer, array processor
Multiple	1	MISD	Arguably none
Multiple	Multiple	MIMD	Multiprocessor, multicomputer

Figure 8-22. Flynn's taxonomy of parallel computers.

Flynn's classification is based on two concepts—instruction streams and data streams. An instruction stream corresponds to a program counter. A system with n CPUs has n program counters, hence n instruction streams.

A data stream consists of a set of operands. For example, in a weather-forecasting system, each of a large number of sensors might emit a stream of temperatures at regular intervals.

The instruction and data streams are, to some extent, independent, so four combinations exist, as listed in Fig. 8-22. SISD is just the classical, sequential von Neumann computer. It has one instruction stream, one data stream, and does one thing at a time. SIMD machines have a single control unit that issues one instruction at a time, but they have multiple ALUs to carry it out on multiple data sets simultaneously. The ILLIAC IV (Fig. 2-7) is the prototype of SIMD machines. Mainstream SIMD machines are increasingly rare, but conventional computers sometimes have some SIMD instructions for processing audiovisual material. The Core i7 SSE instructions are SIMD. Nevertheless, there is one new area in which some of the ideas from the SIMD world are playing a role: stream processors.

† Carolus Linnaeus (1707–1778) was the Swedish biologist who devised the system now used for classifying all plants and animals into kingdom, phylum, class, order, family, genus, and species.

These machines are specifically designed to handle the demands of multimedia rendering and may become important in the future (Kapasi et al., 2003).

MISD machines are a somewhat strange category, with multiple instructions operating on the same piece of data. It is not clear whether any such machines exist, although some people regard pipelined machines as MISD.

Finally, we have MIMD, which are just multiple independent CPUs operating as part of a larger system. Most parallel processors fall into this category. Both multiprocessors and multicomputers are MIMD machines.

Flynn's taxonomy stops here, but we have extended it in Fig. 8-23. SIMD has been split into two subgroups. The first one is for numeric supercomputers and other machines that operate on vectors, performing the same operation on each vector element. The second one is for parallel-type machines, such as the ILLIAC IV, in which a master control unit broadcasts instructions to many independent ALUs.

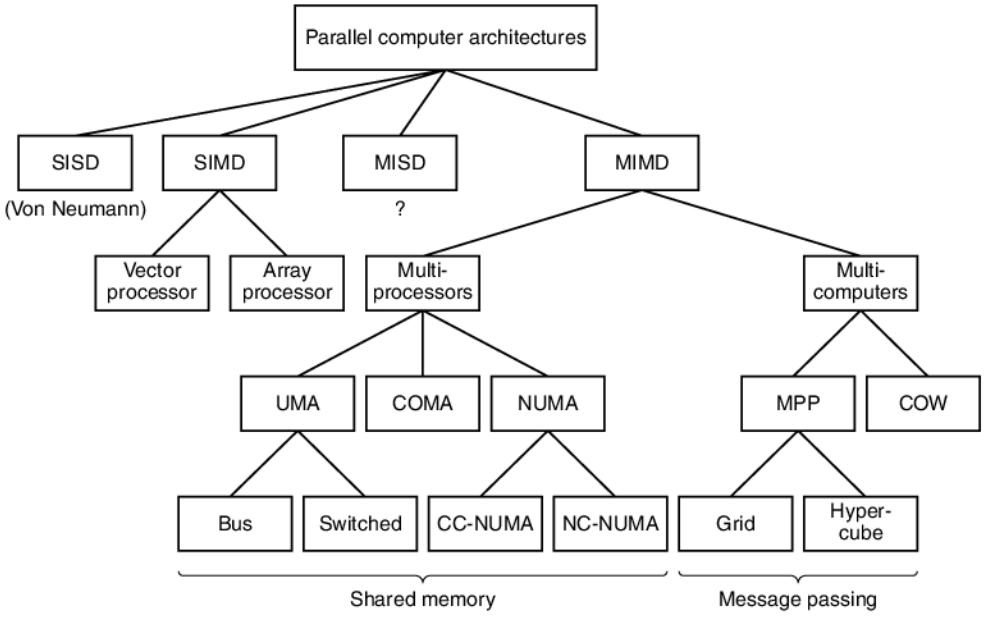


Figure 8-23. A taxonomy of parallel computers.

In our taxonomy, the MIMD category has been split into multiprocessors (shared-memory machines) and multicomputers (message-passing machines). Three kinds of multiprocessors exist, distinguished by the way the shared memory is implemented on them. They are called **UMA** (**Uniform Memory Access**), **NUMA** (**NonUniform Memory Access**), and **COMA** (**Cache Only Memory Access**). These categories exist because in large multiprocessors, the memory is usually split up into multiple modules. UMA machines have the property that each CPU

has the same access time to every memory module. In other words, every memory word can be read as fast as every other memory word. If this is technically impossible, the fastest references are slowed down to match the slowest ones, so programmers do not see the difference. This is what "uniform" means here. This uniformity makes the performance predictable, an important factor for writing efficient code.

In contrast, in a NUMA multiprocessor, this property does not hold. Often there is a memory module close to each CPU and accessing that memory module is much faster than accessing distant ones. The result is that for performance reasons, it matters where code and data are placed. COMA machines are also nonuniform, but in a different way. We will study each of these types and their subcategories in detail later.

The other main category of MIMD machines consists of the multicomputers, which, unlike the multiprocessors, do not have shared primary memory at the architectural level. In other words, the operating system on a multicomputer CPU cannot access memory attached to a different CPU by just executing a LOAD instruction. It has to send an explicit message and wait for an answer. The ability of the operating system to read a distant word by just doing a LOAD is what distinguishes multiprocessors from multicomputers. As we mentioned before, even on a multicomputer, user programs may have the ability to access remote memory by using LOAD and STORE instructions, but this illusion is supported by the operating system, not the hardware. This difference is subtle, but very important. Because multicomputers do not have direct access to remote memory, they are sometimes called **NORMA** (**NO Remote Memory Access**) machines.

Multicomputers can be roughly divided into two categories. The first contains the **MPP**s (**Massively Parallel Processors**), which are expensive supercomputers consisting of many CPUs tightly coupled by a high-speed proprietary interconnection network. The IBM SP/3 is a well-known commercial example.

The other category consists of regular PCs, workstations, or servers, possibly rack mounted, and connected by commercial off-the-shelf interconnection technology. Logically, there is not much difference, but huge supercomputers costing many millions of dollars are used differently than networks of PCs assembled by the users for a fraction of the price of an MPP. These home-brew machines go by various names, including **NOW** (**Network of Workstations**), **COW** (**Cluster of Workstations**), or sometimes just **cluster**.

8.3.2 Memory Semantics

Even though all multiprocessors present the CPUs with the image of a single shared address space, often many memory modules are present, each holding some portion of the physical memory. The CPUs and memories are often connected by a complex interconnection network, as discussed in Sec. 8.1.2. Several CPUs may be attempting to read a memory word at the same time several other CPUs are

attempting to write the same word, and some of the request messages may pass each other in transit and be delivered in a different order than they were issued. Add to this problem the existence of multiple copies of some blocks of memory (e.g., in caches), and the result can easily be chaos unless strict measures are taken to prevent it. In this section we will see what shared memory really means and look at how memories can reasonably respond under these circumstances.

One view of memory semantics is to see it as a contract between the software and the memory hardware (Adve and Hill, 1990). If the software agrees to abide by certain rules, the memory agrees to deliver certain results. The discussion then centers around what the rules are. These rules are called **consistency models**, and many different ones have been proposed and implemented (Sorin et al., 2011).

To give an idea of what the problem is, suppose that CPU 0 writes the value 1 to some memory word and a little later CPU 1 writes the value 2 to the same word. Now CPU 2 reads the word and gets the value 1. Should the computer owner bring the computer to the repair shop to get it fixed? That depends on what the memory promised (its contract).

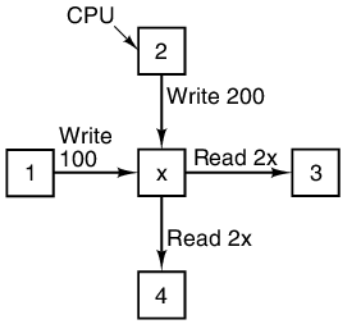
Strict Consistency

The simplest model is **strict consistency**. With this model, any read to a location x always returns the value of the most recent write to x. Programmers love this model, but it is effectively impossible to implement in any way other than having a single memory module that simply services all requests first-come, first-served, with no caching and no data replication. Such an implementation would make memory an enormous bottleneck and is thus not a serious candidate, unfortunately.

Sequential Consistency

Next best is a model called **sequential consistency** (Lamport, 1979). The idea here is that in the presence of multiple read and write requests, some interleaving of all the requests is chosen by the hardware (nondeterministically), but all CPUs see the same order.

To see what this means, consider an example. Suppose that CPU 1 writes the value 100 to word x, and 1 nsec later CPU 2 writes the value 200 to word x. Now suppose that 1 nsec after the second write was issued (but not necessarily completed yet) two other CPUs, 3 and 4, read word x twice in rapid succession, as shown in Fig. 8-24(a). Three possible orderings of the six events (two writes and four reads) are shown in Fig. 8-24(b)–(d), respectively. In Fig. 8-24(b), CPU 3 gets (200, 200) and CPU 4 gets (200, 200). In Fig. 8-24(c), they get (100, 200) and (200, 200), respectively. In Fig. 8-24(d), they get (100, 100) and (200, 100), respectively. All of these are legal, as well as some other possibilities that are not shown. Note that there is not single "correct" value.



(a)	(b)	(c)	(d)
	W100	W100	W200
	W200	R3 = 100	R4 = 200
	R3 = 200	W200	W100
	R3 = 200	R4 = 200	R3 = 100
	R4 = 200	R3 = 200	R4 = 100
	R4 = 200	R4 = 200	R3 = 100

Figure 8-24. (a) Two CPUs writing and two CPUs reading a common memory word. (b)–(d) Three possible ways the two writes and four reads might be interleaved in time.

However—and this is the essence of sequential consistency—no matter what, a sequentially consistent memory will never allow CPU 3 to get (100, 200) while CPU 4 gets (200, 100). If this were to occur, it would mean that according to CPU 3, the write of 100 by CPU 1 completed after the write of 200 by CPU 2. That is fine. But it would also mean that according to CPU 4, the write of 200 by CPU 2 completed before the write of 100 by CPU 1. By itself, this result is also possible. The problem is that sequential consistency guarantees that there is a single global ordering of all writes that is visible to all CPUs. If CPU 3 observes that 100 was written first, then CPU 4 must also see this order.

While sequential consistency is not as powerful a rule as strict consistency, it is still very useful. In effect, it says that when multiple events are happening concurrently, there is some true order in which they occur. Possibly it is determined by timing and chance, but a true ordering exists and all processors observe this same order. Although this statement may seem obvious, below we will discuss consistency models that do not guarantee even this much.

Processor Consistency

A looser consistency model, but one that is easier to implement on large multiprocessors, is **processor consistency** (Goodman, 1989). It has two properties:

1. Writes by any CPU are seen by all in the order they were issued.
2. For every memory word, all CPUs see writes to it in the same order.

Both of these points are important. The first point says that if CPU 1 issues writes with values 1A, 1B, and 1C to some memory location in that sequence, then all

other processors see them in that order, too. In other words, any other processor in a tight loop observing 1A, 1B, and 1C by reading the words written will never see the value written by 1B and then see the value written by 1A, and so on. The second point is needed to require every memory word to have an unambiguous value after several CPUs write to it and finally stop. Everyone must agree on who went last.

Even with these constraints, the designer has a lot of flexibility. Consider what happens if CPU 2 issues writes 2A, 2B, and 2C concurrently with CPU 1's three writes. Other CPUs that are busily reading memory will observe some interleaving of the six writes, such as 1A, 1B, 2A, 2B, 1C, 2C or 2A, 1A, 2B, 2C, 1B, 1C or many others. Processor consistency does *not* guarantee that every CPU sees the same ordering (unlike sequential consistency, which does make this guarantee). Thus it is perfectly legitimate for the hardware to behave in such a way that some CPUs see the first ordering above, some see the second, and some see yet other ones. What *is* guaranteed is that no CPU will see a sequence in which 1B comes before 1A, and so on. The order each CPU does its writes is observed everywhere.

It is worth noting that some authors define processor consistency differently and do not require the second condition.

Weak Consistency

Our next model, **weak consistency**, does not even guarantee that writes from a single CPU are seen in order (Dubois et al., 1986). In a weakly consistent memory, one CPU might see 1A before 1B and another CPU might see 1A after 1B. However, to add some order to the chaos, weakly consistent memories have synchronization variables or a synchronization operation. When a synchronization is executed, all pending writes are finished and no new ones are started until all the old ones are done and the synchronization itself is done. In effect, a synchronization "flushes the pipeline" and brings the memory to a stable state with no operations pending. Synchronization operations are themselves sequentially consistent, that is, when multiple CPUs issue them, some order is chosen, but all CPUs see the same order.

In weak consistency, time is divided into well-defined epochs delimited by the (sequentially consistent) synchronizations, as illustrated in Fig. 8-25. No relative order is guaranteed for 1A and 1B, and different CPUs may see the two writes in different order, that is, one CPU may see 1A then 1B and another CPU may see 1B then 1A. This situation is permitted. However, all CPUs see 1B before 1C because the first synchronization operation forces 1A, 1B, and 2A to complete before 1C, 2B, 3A, or 3B is allowed to start. Thus by doing synchronization operations, software can force some order on the sequence of events, although not at zero cost since flushing the memory pipeline does take time and thus slows the machine down somewhat. Doing it too often can be a problem.

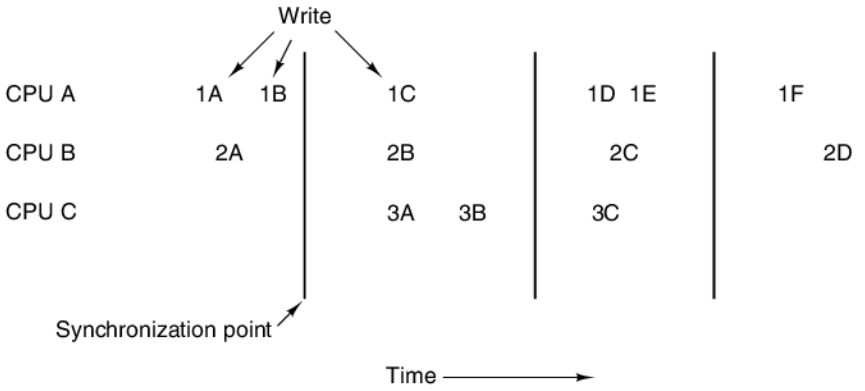


Figure 8-25. Weakly consistent memory uses synchronization operations to divide time into sequential epochs.

Release Consistency

Weak consistency has the problem that it is quite inefficient because it must finish off all pending memory operations and hold all new ones until the current ones are done. **Release consistency** improves matters by adopting a model akin to critical sections (Gharachorloo et al., 1990). The idea behind this model is that when a process exits a critical region it is not necessary to force all the writes to complete immediately. It is only necessary to make sure that they are done before any process enters that critical region again.

In this model, the synchronization operation offered by weak consistency is split into two different operations. To read or write a shared data variable, a CPU (i.e., its software) must first do an acquire operation on the synchronization variable to get exclusive access to the shared data. Then the CPU can use them as it wishes, reading and writing them at will. When it is done, the CPU does a release operation on the synchronization variable to indicate that it is finished. The release does not force pending writes to complete, but it itself does not complete until all previously issued writes are done. Furthermore, new memory operations are not prevented from starting immediately.

When the next acquire is issued, a check is made to see whether all previous release operations have completed. If not, the acquire is held up until they are all done (and hence all the writes done before them are all completed). In this way, if the next acquire occurs sufficiently long after the most recent release, it does not have to wait before starting and the critical region can be entered without delay. If it occurs too soon after a release, the acquire (and all the instructions following it) will be delayed until all pending releases are completed, thus guaranteeing that the variables in the critical section have been updated. This scheme is slightly more complicated than weak consistency, but it has the significant advantage of not delaying instructions as often in order to maintain consistency.

Memory consistency is not a done deal. Researchers are still proposing new models (Naeem et al., 2011, Sorin et al., 2011, and Tu et al., 2010).

8.3.3 UMA Symmetric Multiprocessor Architectures

The simplest multiprocessors are based on a single bus, as illustrated in Fig. 8-26(a). Two or more CPUs and one or more memory modules all use the same bus for communication. When a CPU wants to read a memory word, it first checks to see whether the bus is busy. If the bus is idle, the CPU puts the address of the word it wants on the bus, asserts a few control signals, and waits until the memory puts the desired word on the bus.

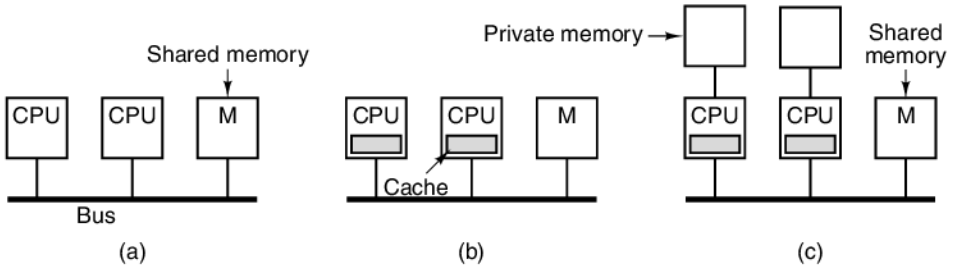


Figure 8-26. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

If the bus is busy when a CPU wants to read or write memory, the CPU just waits until the bus becomes idle. Herein lies the problem with this design. With two or three CPUs, contention for the bus will be manageable; with 32 or 64 it will be unbearable. The system will be totally limited by the bandwidth of the bus, and most of the CPUs will be idle most of the time.

The solution is to add a cache to each CPU, as depicted in Fig. 8-26(b). The cache can be inside the CPU chip, next to the CPU chip, on the processor board, or some combination of all three. Since many reads can now be satisfied out of the local cache, there will be much less bus traffic, and the system can support more CPUs. Thus caching is a big win here. However, as we shall see in a moment, keeping the caches consistent with one another is not trivial.

Yet another possibility is the design of Fig. 8-26(c), in which each CPU has not only a cache but also a local, private memory which it accesses over a dedicated (private) bus. To use this configuration optimally, the compiler should place all the program text, strings, constants and other read-only data, stacks, and local variables in the private memories. The shared memory is then used only for writable shared variables. In most cases, this careful placement will greatly reduce bus traffic, but it does require active cooperation from the compiler.

Snooping Caches

While the performance arguments given above are certainly true, we have glossed a bit too quickly over a fundamental problem. Suppose that memory is sequentially consistent. What happens if CPU 1 has a line in its cache, and then CPU 2 tries to read a word in the same cache line? In the absence of any special rules, it, too, would get a copy in its cache. In principle, having the same line cached twice is acceptable. Now suppose that CPU 1 modifies the line and then, immediately thereafter, CPU 2 reads its copy of the line from its cache. It will get **stale data**, thus violating the contract between the software and memory. The program running on CPU 2 will not be happy.

This problem, known in the literature as the **cache coherence** or **cache consistency** problem, is extremely serious. Without a solution, caching cannot be used, and bus-oriented multiprocessors would be limited to two or three CPUs. As a consequence of its importance, many solutions have been proposed over the years (e.g., Goodman, 1983, and Papamarcos and Patel, 1984). Although all these caching algorithms, called **cache coherence protocols**, differ in the details, all of them prevent different versions of the same cache line from appearing simultaneously in two or more caches.

In all solutions, the cache controller is specially designed to allow it to eavesdrop on the bus, monitoring all bus requests from other CPUs and caches and taking action in certain cases. These devices are called **snooping caches** or sometimes **snoopy caches** because they "snoop" on the bus. The set of rules implemented by the caches, CPUs, and memory for preventing different versions of the data from appearing in multiple caches forms the cache coherence protocol. The unit of transfer and storage for a cache is called a **cache line** and is typically 32 or 64 bytes.

The simplest cache coherence protocol is called **write through**. It can best be understood by distinguishing the four cases shown in Fig. 8-27. When a CPU tries to read a word that is not in its cache (i.e., a read miss), its cache controller loads the line containing that word into the cache. The line is supplied by the memory, which in this protocol is always up to date. Subsequent reads (i.e., read hits) can be satisfied out of the cache.

Action	Local request	Remote request
Read miss	Fetch data from memory	
Read hit	Use data from local cache	
Write miss	Update data in memory	
Write hit	Update cache and memory	Invalidate cache entry

Figure 8-27. The write-through cache coherence protocol. The empty boxes indicate that no action is taken.

On a write miss, the word that has been modified is written to main memory. The line containing the word referenced is *not* loaded into the cache. On a write hit, the cache is updated and the word is written through to main memory in addition. The essence of this protocol is that all write operations result in the word being written going through to memory to keep memory up to date at all times.

Now let us look at all these actions again, but this time from the snooper's point of view, shown in the right-hand column of Fig. 8-27. Let us call the cache performing the actions cache 1 and the snooping cache cache 2. When cache 1 misses on a read, it makes a bus request to fetch a line from memory. Cache 2 sees this but does nothing. When cache 1 has a read hit, the request is satisfied locally, and no bus request occurs, so cache 2 is not aware of cache 1's read hits.

Writes are more interesting. If CPU 1 does a write, cache 1 will make a write request on the bus, both on misses and on hits. On all writes, cache 2 checks to see whether it has the word being written. If not, from its point of view this is a remote request/write miss and it does nothing. (To clarify a subtle point, note that in Fig. 8-27 a remote miss means that the word is not present in the snooper's cache; it does not matter whether it was in the originator's cache or not. Thus a single request may be a hit locally and a miss at the snooper, or vice versa.)

Now suppose that cache 1 writes a word that *is* present in cache 2's cache (remote request/write hit). If cache 2 does nothing, it will have stale data, so it marks the cache entry containing the newly modified word as being invalid. In effect, it removes the item from the cache. Because all caches snoop on all bus requests, whenever a word is written, the net effect is to update it in the originator's cache, update it in memory, and purge it from all the other caches. In this way, inconsistent versions are prevented.

Of course, cache 2's CPU is free to read the same word on the very next cycle. In that case, cache 2 will read the word from memory, which is up to date. At that point, cache 1, cache 2, and the memory will all have identical copies of it. If either CPU does a write now, the other one's cache will be purged, and memory will be updated.

Many variations on this basic protocol are possible. For example, on a write hit, the snooping cache normally invalidates its entry containing the word being written. Alternatively, it could accept the new value and update its cache instead of marking it as invalid. Conceptually, updating the cache is the same as invalidating it followed by reading the word from memory. In all cache protocols, a choice must be made between an **update strategy** and an **invalidate strategy**. These protocols perform differently under different loads. Update messages carry payloads and are thus larger than invalidates but may prevent future cache misses.

Another variant is loading the snooping cache on write misses. The correctness of the algorithm is not affected by loading it, only the performance. The question is: "What is the probability that a word just written will be written again soon?" If it is high, there is something to be said for loading the cache on write misses, known as a **write-allocate policy**. If it is low, it is better not to update on

write misses. If the word is *read* soon, it will be loaded by the read miss anyway; little is gained by loading it on the write miss.

As with many simple solutions, this one is inefficient. Every write operation goes to memory over the bus, so with a modest number of CPUs, the bus will still become a bottleneck. To keep the bus traffic within bounds, other cache protocols have been devised. They all have the property that not all writes go directly through to memory. Instead, when a cache line is modified, a bit is set inside the cache noting that the cache line is correct but memory is not. Eventually, such a dirty line has to be written back to memory, but possibly after many writes have been made to it. This type of protocol is known as a **write-back protocol**.

The MESI Cache Coherence Protocol

One popular write-back cache coherence protocol is called **MESI**, after the initials of the names of the four states (M, E, S, and I) that it uses (Papamarcos and Patel, 1984). It is based on the earlier **write-once protocol** (Goodman, 1983). The MESI protocol is used by the Core i7 and many other CPUs for snooping on the bus. Each cache entry can be in one of the following four states:

1. Invalid – The cache entry does not contain valid data.
2. Shared – Multiple caches may hold the line; memory is up to date.
3. Exclusive– No other cache holds the line; memory is up to date.
4. Modified – The entry is valid; memory is invalid; no copies exist.

When the CPU is initially booted, all cache entries are marked invalid. The first time memory is read, the line referenced is fetched into the cache of the CPU reading memory and marked as being in the E (exclusive) state, since it is the only copy in a cache, as illustrated in Fig. 8-28(a) for the case of CPU 1 reading line *A*. Subsequent reads by that CPU use the cached entry and do not go over the bus. Another CPU may also fetch the same line and cache it, but by snooping, the original holder (CPU 1) sees that it is no longer alone and announces on the bus that it also has a copy. Both copies are marked as being in the S (shared) state, as shown in Fig. 8-28(b). In other words, the S state means that the line is in one or more caches for reading and memory is up to date. Subsequent reads by a CPU to a line it has cached in the S state do not use the bus and do not cause the state to change.

Now consider what happens if CPU 2 writes to the cache line it is holding in S state. It puts out an invalidate signal on the bus, telling all other CPUs to discard their copies. The cached copy now goes to M (modified) state, as shown in Fig. 8-28(c). The line is not written to memory. It is worth noting that if a line is in E state when it is written, no bus signal is needed to invalidate other caches because it is known that no other copies exist.

Next consider what happens if CPU 3 reads the line. CPU 2, which now owns the line, knows that the copy in memory is not valid, so it asserts a signal on the

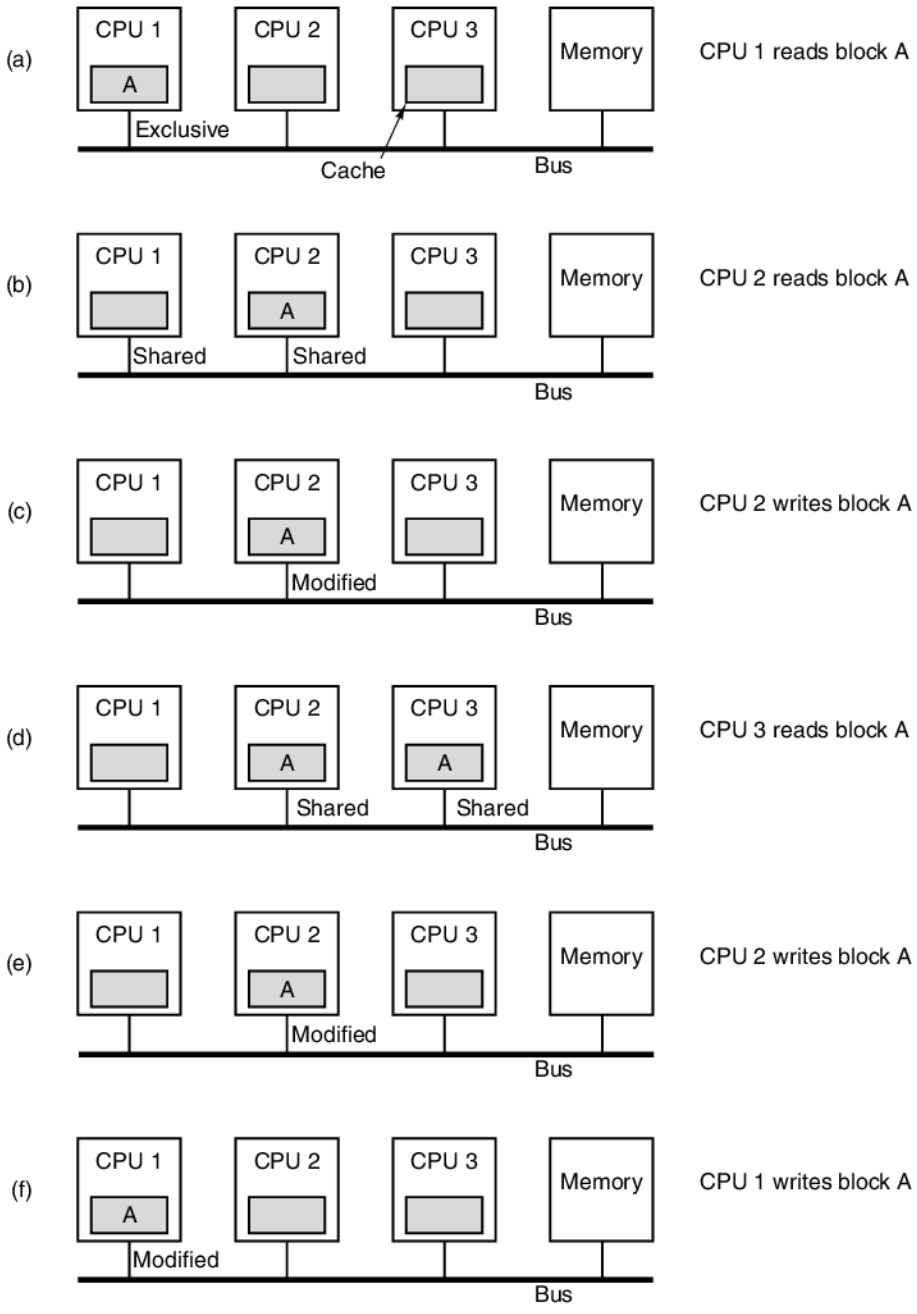


Figure 8-28. The MESI cache coherence protocol.

bus telling CPU 3 to please wait while it writes its line back to memory. When it is finished, CPU 3 fetches a copy, and the line is marked as shared in both caches, as shown in Fig. 8-28(d). After that, CPU 2 writes the line again, which invalidates the copy in CPU 3's cache, as shown in Fig. 8-28(e).

Finally, CPU 1 writes to a word in the line. CPU 2 sees that a write is being attempted and asserts a bus signal telling CPU 1 to please wait while it writes its line back to memory. When it is finished, it marks its own copy as invalid, since it knows another CPU is about to modify it. At this point we have the situation in which a CPU is writing to an uncached line. If the write-allocate policy is in use, the line will be loaded into the cache and marked as being in the M state, as shown in Fig. 8-28(f). If the write-allocate policy is not in use, the write will go directly to memory and the line will not be cached anywhere.

UMA Multiprocessors Using Crossbar Switches

Even with all possible optimizations, the use of a single bus limits the size of a UMA multiprocessor to about 16 or 32 CPUs. To go beyond that, a different kind of interconnection network is needed. The simplest circuit for connecting n CPUs to k memories is the **crossbar switch**, shown in Fig. 8-28. Crossbar switches have been used for decades within telephone switching exchanges to connect a group of incoming lines to a set of outgoing lines in an arbitrary way.

At each intersection of a horizontal (incoming) and vertical (outgoing) line is a **crosspoint**. A crosspoint is a small switch that can be electrically opened or closed, depending on whether the horizontal and vertical lines are to be connected or not. In Fig. 8-29(a) we see three crosspoints closed simultaneously, allowing connections between the (CPU, memory) pairs (001, 000), (101, 101), and (110, 010) at the same time. Many other combinations are also possible. In fact, the number of combinations is equal to the number of different ways eight rooks can be safely placed on a chess board.

One of the nicest properties of the crossbar switch is that it is a **nonblocking network**, meaning that no CPU is ever denied the connection it needs because some crosspoint or line is already occupied (assuming the memory module itself is available). Furthermore, no advance planning is needed. Even if seven arbitrary connections are already set up, it is always possible to connect the remaining CPU to the remaining memory. We will later see interconnection schemes that do not have these properties.

One of the worst properties of the crossbar switch is that the number of crosspoints grows as n^2. For medium-sized systems, a crossbar design is workable. We will discuss one such design, the Sun Fire E25K, later in this chapter. However, with 1000 CPUs and 1000 memory modules, we need a million crosspoints. Such a large crossbar switch is not feasible. We need something quite different.

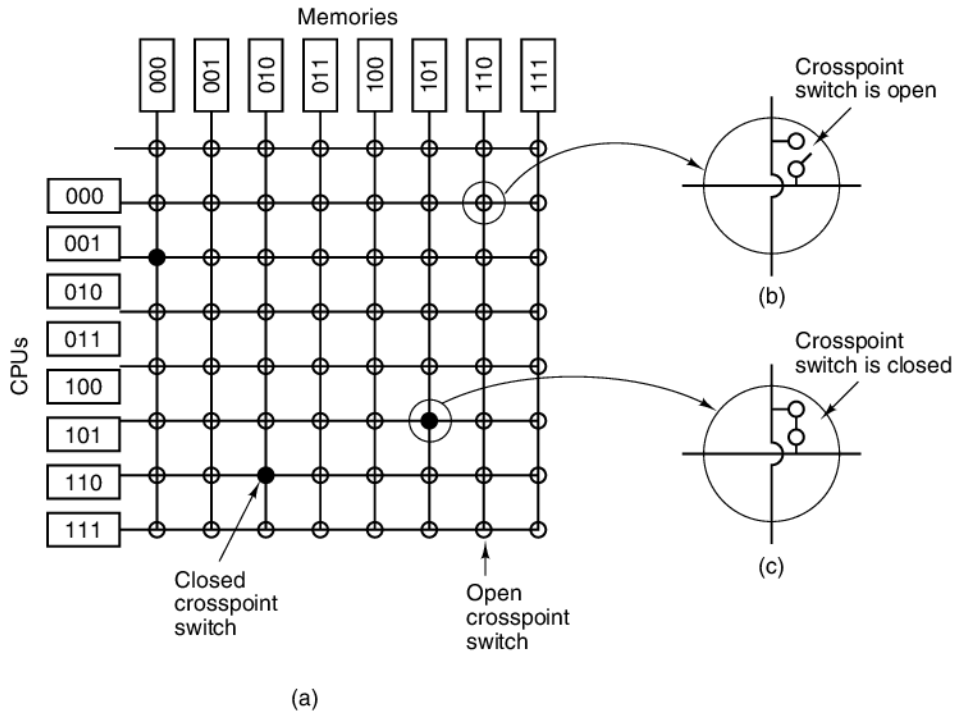


Figure 8-29. (a) An 8×8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

UMA Multiprocessors Using Multistage Switching Networks

That "something quite different" can be based on the humble 2×2 switch shown in Fig. 8-30(a). This switch has two inputs and two outputs. Messages arriving on either input line can be switched to either output line. For our purposes here, messages will contain up to four parts, as shown in Fig. 8-30(b). The *Module* field tells which memory to use. The *Address* specifies an address within a module. The *Opcode* gives the operation, such as READ or WRITE. Finally, the optional *Value* field may contain an operand, such as a 32-bit word to be written on a WRITE. The switch inspects the *Module* field and uses it to determine if the message should be sent on *X* or on *Y*.

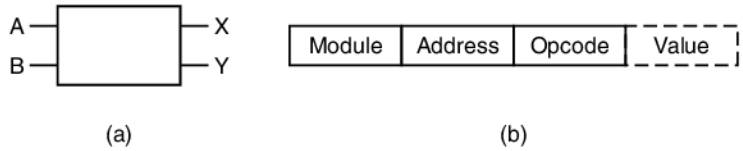


Figure 8-30. (a) A 2×2 switch. (b) A message format.

Our 2×2 switches can be arranged in many ways to build larger **multistage switching networks**. One option is the no-frills, economy class **omega network**,

illustrated in Fig. 8-31. Here we have connected eight CPUs to eight memories using 12 switches. More generally, for n CPUs and n memories we would need $\log_2 n$ stages, with $n/2$ switches per stage, for a total of $(n/2)\log_2 n$ switches, which is a lot better than n^2 crosspoints, especially for large values of n.

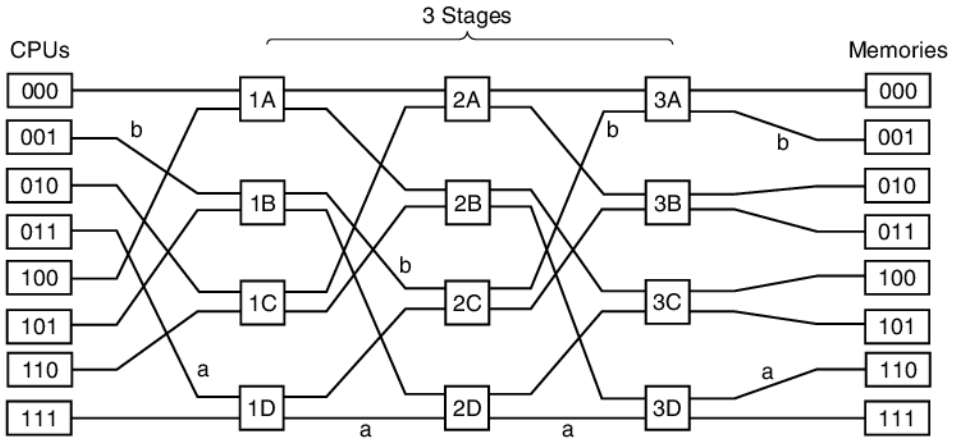


Figure 8-31. An omega switching network.

The wiring pattern of the omega network is often called the **perfect shuffle**, since the mixing of the signals at each stage resembles a deck of cards being cut in half and then mixed card-for-card. To see how the omega network works, suppose that CPU 011 wants to read a word from memory module 110. The CPU sends a READ message to switch 1D containing 110 in the *Module* field. The switch takes the first (i.e., leftmost) bit of 110 and uses it for routing. A 0 routes to the upper output and a 1 routes to the lower one. Since this bit is a 1, the message is routed via the lower output to 2D.

All the second-stage switches, including 2D, use the second bit for routing. This, too, is a 1, so the message is now forwarded via the lower output to 3D. Here the third bit is tested and found to be a 0. Consequently, the message goes out on the upper output and arrives at memory 110, as desired. The path followed by this message is marked in Fig. 8-31 by the letter *a*.

As the message moves through the switching network, the bits at the left-hand end of the module number are no longer needed. They can be put to good use by recording the incoming line number there, so the reply can find its way back. For path *a*, the incoming lines are 0 (upper input to 1D), 1 (lower input to 2D), and 1 (lower input to 3D), respectively. The reply is routed back using 011, only reading it from right to left this time.

While all this is going on, CPU 001 wants to write a word to memory module 001. An analogous process happens here, with the message routed via the upper, upper, and lower outputs, respectively, marked by the letter *b*. When it arrives, its

Module field reads 001, representing the path it took. Since these requests do not use any of the same switches, lines, or memory modules, they can go in parallel.

Now consider what would happen if CPU 000 simultaneously wanted to access memory module 000. Its request would come into conflict with CPU 001's request at switch 3A. One of them would have to wait. Unlike the crossbar switch, the omega network is a **blocking network**. Not every set of requests can be processed simultaneously. Conflicts can occur over the use of a wire or a switch, as well as between requests *to* memory and replies *from* memory.

It is clearly desirable to spread the memory references uniformly across the modules. One common technique is to use the low-order bits as the module number. Consider, for example, a byte-oriented address space for a computer that mostly accesses 32-bit words. The 2 low-order bits will usually be 00, but the next 3 bits will be uniformly distributed. By using these 3 bits as the module number, consecutively addressed words will be in consecutive modules. A memory system in which consecutive words are in different modules is said to be **interleaved**. Interleaved memories maximize parallelism because most memory references are to consecutive addresses. It is also possible to design switching networks that are nonblocking and that offer multiple paths from each CPU to each memory module, to spread the traffic better.

8.3.4 NUMA Multiprocessors

It should be clear by now that single-bus UMA multiprocessors are generally limited to no more than a few dozen CPUs and crossbar or switched multiprocessors need a lot of (expensive) hardware and are not that much bigger. To get to more than 100 CPUs, something has to give. Usually, what gives is the idea that all memory modules have the same access time. This concession leads to the idea of **NUMA** (**NonUniform Memory Access**) multiprocessors. Like their UMA cousins, they provide a single address space across all the CPUs, but unlike the UMA machines, access to local memory modules is faster than access to remote ones. Thus all UMA programs will run without change on NUMA machines, but the performance will be worse than on a UMA machine at the same clock speed.

All NUMA machines have three key characteristics that together distinguish them from other multiprocessors:

1. There is a single address space visible to all CPUs.
2. Access to remote memory done using LOAD and STORE instructions.
3. Access to remote memory is slower than access to local memory.

When the access time to remote memory is not hidden (because there is no caching), the system is called **NC-NUMA**. When coherent caches are present, the

system is called **CC-NUMA** (at least by the hardware people). The software people often call it **hardware DSM** because it is basically the same as software distributed shared memory but implemented by the hardware using a small page size.

One of the first NC-NUMA machines (although the name had not yet been coined) was the Carnegie-Mellon Cm*, illustrated in simplified form in Fig. 8-32 (Swan et al., 1977). It consisted of a collection of LSI-11 CPUs, each with some memory addressed over a local bus. (The LSI-11 was a single-chip version of the DEC PDP-11, a minicomputer popular in the 1970s.) In addition, the LSI-11 systems were connected by a system bus. When a memory request came into the (specially modified) MMU, a check was made to see if the word needed was in the local memory. If so, a request was sent over the local bus to get the word. If not, the request was routed over the system bus to the system containing the word, which then responded. Of course, the latter took much longer than the former. While a program could run happily out of remote memory, it took 10 times longer to execute than the same program running out of local memory.

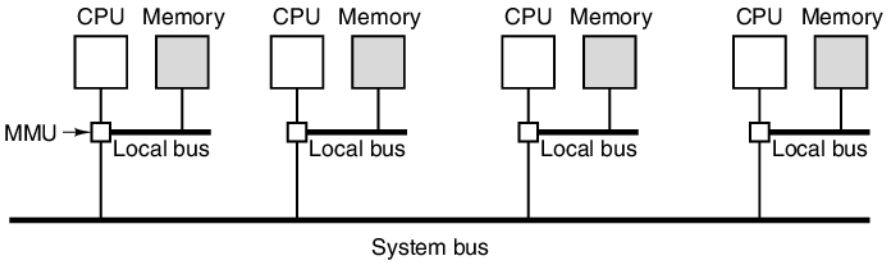


Figure 8-32. A NUMA machine based on two levels of buses. The Cm* was the first multiprocessor to use this design.

Memory coherence is guaranteed in an NC-NUMA machine because no caching is present. Each word of memory lives in exactly one location, so there is no danger of one copy having stale data: there are no copies. Of course, it now matters a great deal which page is in which memory because the performance penalty for being in the wrong place is so high. Consequently, NC-NUMA machines use elaborate software to move pages around to maximize performance.

Typically, a daemon process called a **page scanner** runs every few seconds. Its job is to examine the usage statistics and move pages around in an attempt to improve performance. If a page appears to be in the wrong place, the page scanner unmaps it so that the next reference to it will cause a page fault. When the fault occurs, a decision is made about where to place the page, possibly in a different memory. To prevent thrashing, usually there is some rule saying that once a page is placed, it is frozen in place for a time ΔT. Various algorithms have been studied, but the conclusion is that no one algorithm performs best under all circumstances (LaRowe and Ellis, 1991). Best performance depends on the application.

Cache Coherent NUMA Multiprocessors

Multiprocessor designs such as that of Fig. 8-32 do not scale well because they do not do caching. Having to go to the remote memory every time a nonlocal memory word is accessed is a major performance hit. However, if caching is added, then cache coherence must also be added. One way to provide cache coherence is to snoop on the system bus. Technically, doing this is not difficult, but beyond a certain number of CPUs, it becomes infeasible. To build really large multiprocessors, a fundamentally different approach is needed.

The most popular approach for building large **CC-NUMA** (**Cache Coherent NUMA**) multiprocessors currently is the **directory-based multiprocessor**. The idea is to maintain a database telling where each cache line is and what its status is. When a cache line is referenced, the database is queried to find out where it is and whether it is clean or dirty (modified). Since this database must be queried on every single instruction that references memory, it must be kept in extremely fast special-purpose hardware that can respond in a fraction of a bus cycle.

To make the idea of a directory-based multiprocessor somewhat more concrete, let us consider a simple (hypothetical) example, a 256-node system, each node consisting of one CPU and 16 MB of RAM connected to the CPU via a local bus. The total memory is 2^{32} bytes, divided up into 2^{26} cache lines of 64 bytes each. The memory is statically allocated among the nodes, with 0–16M in node 0, 16–32M in node 1, and so on. The nodes are connected by an interconnection network, as shown in Fig. 8-33(a). This network could be a grid, hypercube, or other topology. Each node also holds the directory entries for the 2^{18} 64-byte cache lines comprising its 2^{24}-byte memory. For the moment, we will assume that a line can be held in at most one cache.

To see how the directory works, let us trace a LOAD instruction from CPU 20 that references a cached line. First the CPU issuing the instruction presents it to its MMU, which translates it to a physical address, say, 0x24000108. The MMU splits this address into the three parts shown in Fig. 8-33(b). In decimal, the three parts are node 36, line 4, and offset 8. The MMU sees that the memory word referenced is from node 36, not node 20, so it sends a request message through the interconnection network to the line's home node, 36, asking whether its line 4 is cached, and if so, where.

When the request arrives at node 36 over the interconnection network, it is routed to the directory hardware. The hardware indexes into its table of 2^{18} entries, one for each of its cache lines and extracts entry 4. From Fig. 8-33(c) we see that the line is not cached, so the hardware fetches line 4 from the local RAM, sends it back to node 20, and updates directory entry 4 to indicate that the line is now cached at node 20.

Now let us consider a second request, this time asking about node 36's line 2. From Fig. 8-33(c) we see that this line is cached at node 82. At this point the hardware could update directory entry 2 to say that the line is now at node 20 and then

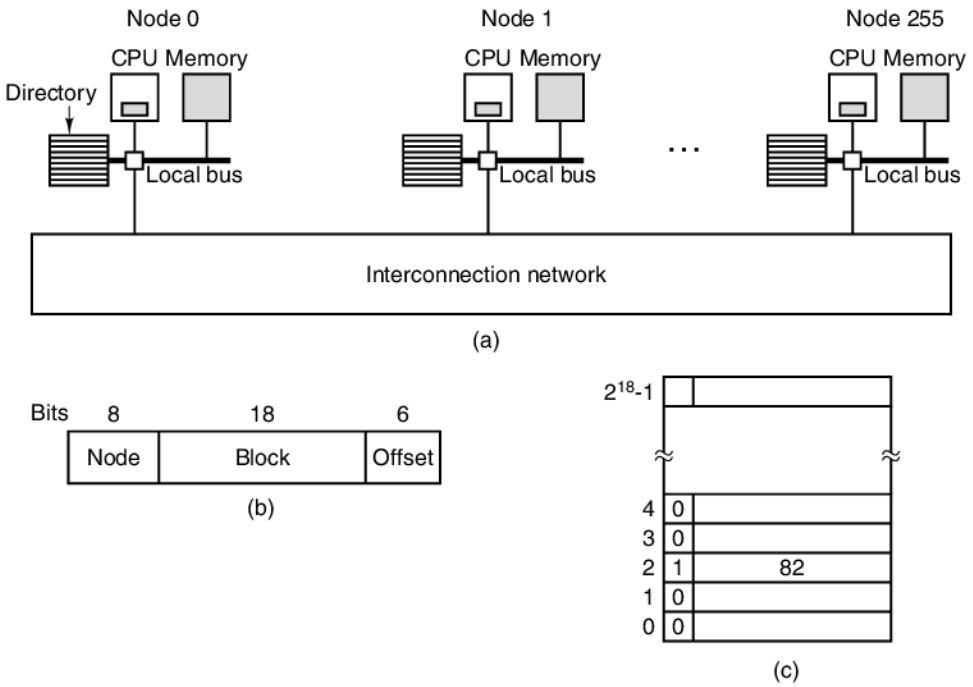


Figure 8-33. (a) A 256-node directory-based multiprocessor. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

send a message to node 82 instructing it to pass the line to node 20 and invalidate its cache. Note that even a so-called "shared-memory multiprocessor" has a lot of message passing going on under the hood.

As a quick aside, let us calculate how much memory is being taken up by the directories. Each node has 16 MB of RAM and 2^{18} 9-bit entries to keep track of that RAM. Thus the directory overhead is about 9×2^{18} bits divided by 16 MB or about 1.76 percent, which is generally acceptable (although it has to be high-speed memory, which increases its cost). Even with 32-byte cache lines the overhead would only be 4 percent. With 128-byte cache lines, it would be under 1 percent.

An obvious limitation of this design is that a line can be cached at only one node. To allow lines to be cached at multiple nodes, we would need some way of locating all of them, for example, to invalidate or update them on a write. Various options are possible to allow caching at several nodes at the same time.

One possibility is to give each directory entry k fields for specifying other nodes, thus allowing each line to be cached at up to k nodes. A second possibility is to replace the node number in our simple design with a bit map, with one bit per node. In this option there is no limit on how many copies there can be, but there is a substantial increase in overhead. Having a directory with 256 bits for each

64-byte (512-bit) cache line implies an overhead of over 50 percent. A third possibility is to keep one 8-bit field in each directory entry and use it as the head of a linked list that threads all the copies of the cache line together. This strategy requires extra storage at each node for the linked list pointers, and it also requires following a linked list to find all the copies when that is needed. Each possibility has its own advantages and disadvantages, and all three have been used in real systems.

Another improvement to the directory design is to keep track of whether the cache line is clean (home memory is up to date) or dirty (home memory is not up to date). If a read request comes in for a clean cache line, the home node can satisfy the request from memory, without having to forward it to a cache. A read request for a dirty cache line, however, must be forwarded to the node holding the cache line because only it has a valid copy. If only one cache copy is allowed, as in Fig. 8-33, there is no real advantage to keeping track of its cleanliness, because any new request requires a message to be sent to the existing copy to invalidate it.

Of course, keeping track of whether each cache line is clean or dirty implies that when a cache line is modified, the home node has to be informed, even if only one cache copy exists. If multiple copies exist, modifying one of them requires the rest to be invalidated, so some protocol is needed to avoid race conditions. For example, to modify a shared cache line, one of the holders might have to request exclusive access *before* modifying it. Such a request would cause all other copies to be invalidated before permission was granted. Other performance optimizations for CC-NUMA machines are discussed in Cheng and Carter (2008).

The Sun Fire E25K NUMA Multiprocessor

As an example of a shared-memory NUMA multiprocessor, let us study the Sun Microsystems Sun Fire family. Although it contains various models, we will focus on the E25K, which has 72 UltraSPARC IV CPU chips. An UltraSPARC IV is essentially a pair of UltraSPARC III processors that share a common cache and memory. The E15K is essentially the same system except with uniprocessor instead of dual-processor CPU chips. Smaller members exist as well, but from our point of view, what is interesting is how the one with the most CPUs works.

The E25K system consists of up to 18 boardsets, each boardset consisting of a CPU-memory board, an I/O board with four PCI slots, and an expander board that couples the CPU-memory board with the I/O board and joins the pair to the centerplane, which holds the boards and contains the switching logic. Each CPU-memory board contains four CPU chips and four 8-GB RAM modules. Consequently, each CPU-memory board on the E25K holds eight CPUs and 32 GB of RAM (four CPUs and four 32 GB of RAM on the E15K). A full E25K thus contains 144 CPUs, 576 GB of RAM, and 72 PCI slots. It is illustrated in Fig. 8-34. Interestingly enough, the number 18 was chosen due to packaging constraints: a system with 18 boardsets was the largest one that could fit through a doorway in one piece.

While programmers just think about 0s and 1s, engineers have to worry about things like how the customer will get the product through the door and into the building.

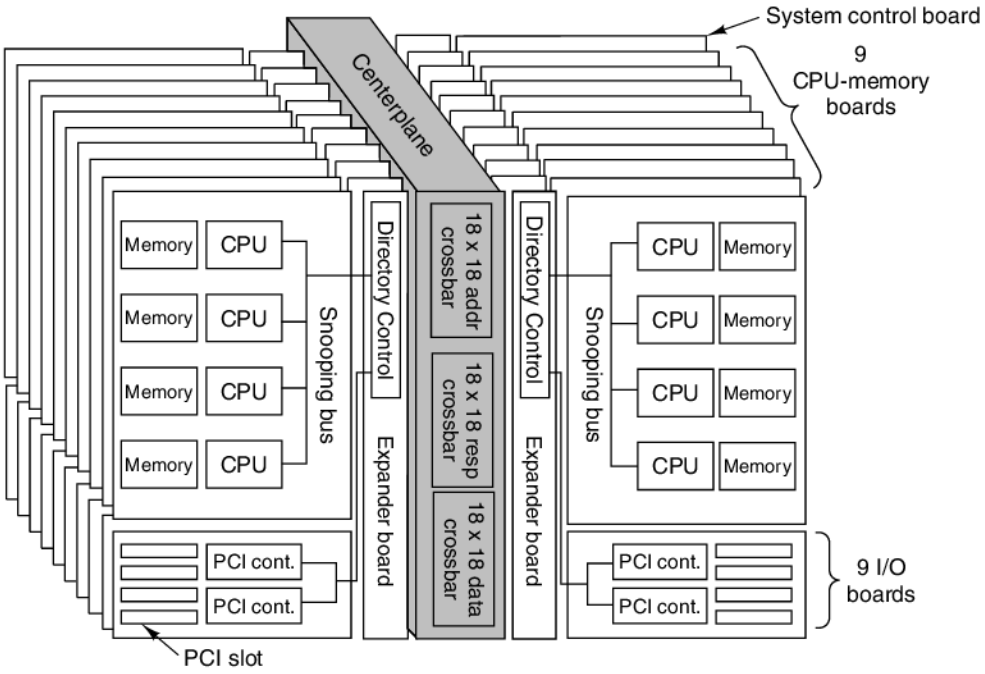


Figure 8-34. The Sun Microsystems E25K multiprocessor.

The centerplane is composed of a set of three 18 × 18 crossbar switches for connecting the 18 boardsets. One crossbar is for the address lines, one is for responses, and one is for data transfer. In addition to the 18 expander boards, the centerplane also has a system control boardset plugged into it. This boardset has a single CPU but also interfaces to the CD-ROM, tape, serial lines, and other peripheral devices needed for booting, maintaining, and controlling the system.

The heart of any multiprocessor is the memory subsystem. How does one connect 144 CPUs to the distributed memory? The straightforward ways—a big shared snooping bus or a 144 × 72 crossbar switch—do not work well. The former fails due to the bus being a bottleneck and the latter fails because the switch is too difficult and expensive to build. Thus large multiprocessors such as the E25K are forced to use a more complex memory subsystem.

At the boardset level, snooping logic is used so all local CPUs can check all memory requests coming from the boardset to references to blocks they currently have cached. Thus when a CPU needs a word from memory, it first converts the virtual address to a physical address and checks its cache. (Physical addresses are 43 bits, but packaging restrictions limit memory to 576 GB.) If the cache block it

needs is in its own cache, the word is returned. Otherwise, the snooping logic checks if a copy of that word is available somewhere else on the boardset. If so, the request is satisfied. If not, the request is passed on via the 18×18 address crossbar switch as described below. The snooping logic can do one snoop per clock cycle. The system clock runs at 150 MHz, so it is possible to perform 150 million snoops/sec per boardset or 2.7 billion snoops/sec system wide.

Although the snooping logic is logically a bus, as portrayed in Fig. 8-34, physically it is a device tree, with commands being relayed up and down the tree. When a CPU or PCI board puts out an address, it goes to an address repeater via a point-to-point connection, as shown in Fig. 8-35. The two address repeaters converge on the expander board, where the addresses are sent back down the tree for each device to check for hits. This arrangement is used to avoid having a bus that involves three boards.

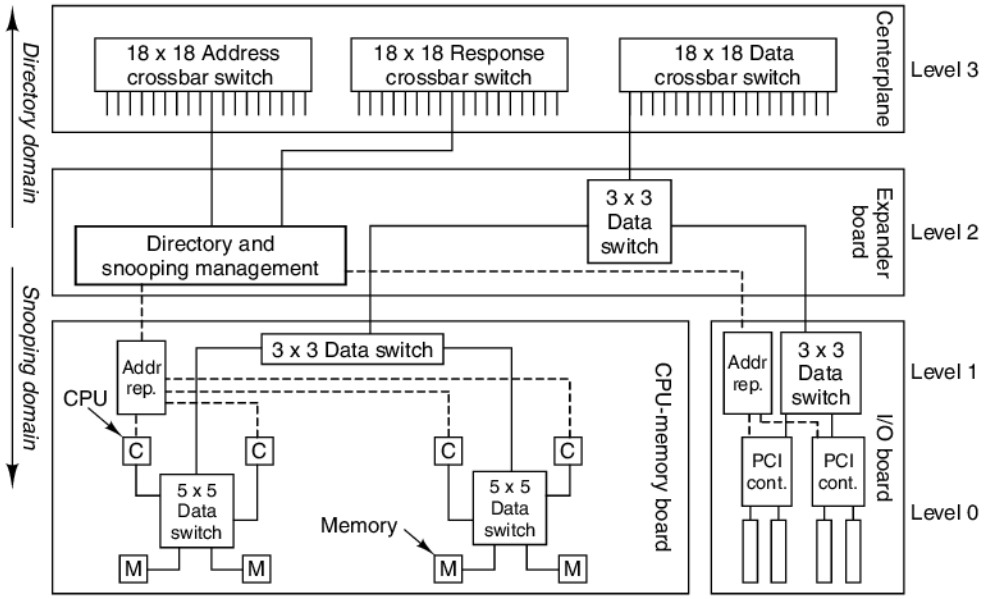


Figure 8-35. The Sun Fire E25K uses a four-level interconnect. Dashed lines are address paths. Solid lines are data paths.

Data transfers use a four-level interconnect as depicted in Fig. 8-35. This design was chosen for high performance. At level 0, pairs of CPU chips and memories are connected by a small crossbar switch that also has a connection to level 1. The two groups of CPU-memory pairs are connected by a second crossbar switch at level 1. The crossbar switches are custom ASICs. For all of them, all the inputs are available on both the rows and the columns, although not all combinations are used (or even make sense). All the switching logic on the boards is built from 3×3 crossbars.

Each boardset consists of three boards: the CPU-memory board, the I/O board, and the expander board, which connects the other two. The level 2 interconnect is another 3×3 crossbar switch (on the expander board) that joins the actual memory to the I/O ports (which are memory mapped on all UltraSPARCs). All data transfers to or from the boardset, whether to memory or to an I/O port, pass through the level 2 switch. Finally, data that have to be transferred to or from a remote board pass through an 18×18 data crossbar switch at level 3. Data transfers are done 32 bytes at a time, so it takes two clock cycles to transfer 64 bytes, the usual transfer unit.

Having looked at how the components are arranged, let us now consider how the shared memory operates. At the bottom level, the 576 GB of memory is split into 2^{29} blocks of 64 bytes each. These blocks are the atomic units of the memory system. Each block has a home board where it lives when not in use elsewhere. Most blocks are on their home board most of the time. However, when a CPU needs a memory block, either from its own board or one of the 17 remote ones, it first requests a copy for its own cache, then accesses the cached copy. Although each CPU chip on the E25K contains two CPUs, they share a single physical cache and thus share all the blocks contained in it.

Each memory block and cache line of each CPU chip can be in one of three states:

1. Exclusive access (for writing).
2. Shared access (for reading).
3. Invalid (i.e., empty).

When a CPU needs to read or write a memory word, it first checks its own cache. Failing to find the word there, it issues a local request for the physical address that is broadcast only on its own boardset. If a cache on the boardset has the needed line, the snooping logic detects the hit and responds to the request. If the line is in exclusive mode, it is transferred to the requester and the original copy marked invalid. If it is in shared mode, the cache does not respond since memory always responds when a cache line is clean.

If the snooping logic cannot find the cache line or it is present and shared, it sends a request over the centerplane to the home board asking where the memory block is. The state of each memory block is stored in the block's ECC bits, so the home board can immediately determine its state. If the block is either unshared or shared with one or more remote boards, the home memory will be up to date, and the request can be satisfied from the home board's memory. In this case, a copy of the case line is transmitted over the data crossbar switch in two clock cycles, eventually arriving at the requesting CPU.

If the request was for reading, an entry is made in the directory at the home board noting that a new customer is sharing the cache line and the transaction is

finished. However, if the request was for writing, an invalidation message must be sent to all other boards (if any) holding a copy of it. In this way, the board making the write request ends up with the only copy.

Now consider the case in which the requested block is in exclusive state located on a different board. When the home board gets the request, it looks up the location of the remote board in the directory and sends the requester a message telling where the cache line is. The requester now sends the request to the correct boardset. When the request arrives, the board sends back the cache line. If it was a read request, the line is marked shared and a copy sent back to the home board. If it was a write request, the responder invalidates its copy so the new requester has an exclusive copy.

Since each board has 2^{29} memory blocks, it would take a directory with 2^{29} entries to keep track of them all in the worst case. Since the directory is much smaller than 2^{29}, it could happen that there is no room in the directory (which is searched associatively) for some entries. In this case, the home directory has to locate the block by broadcasting a request for it to all the other 17 boards. The response crossbar switch plays a role in the directory coherence and update protocol by handling much of the reverse traffic back to the sender. Splitting the protocol traffic over two buses (address and response) and the data over a third bus increases the throughput of the system.

By distributing the load over multiple devices on different boards, the Sun Fire E25K is able to achieve very high performance. In addition to the 2.7 billion snoops/sec mentioned above, the centerplane can handle up to nine simultaneous transfers, with nine boards sending and nine boards receiving. Since the data crossbar is 32 bytes wide, on every clock cycle 288 bytes can be moved through the centerplane. At a clock rate of 150 MHz, this gives a peak aggregate bandwidth of 40 GB/sec when all accesses are remote. If the software can place pages in such a way to ensure that most accesses are local, then the system bandwidth can be appreciably higher than 40 GB/sec.

For more technical information about the Sun Fire, see Charlesworth (2002) and Charlesworth (2001).

In 2009 Oracle purchased Sun Microsystems, and they have continued development of SPARC-based servers. The SPARC Enterprise M9000 is the successor to the E25K. The M9000 incorporates faster quad-core SPARC processors, plus additional memory and PCIe slots. A fully equipped M9000 server contains 256 SPARC processors, 4 TB of DRAM, and 128 PCIe I/O interfaces.

8.3.5 COMA Multiprocessors

NUMA and CC-NUMA machines have the disadvantage that references to remote memory are much slower than those to local memory. In CC-NUMA, this performance difference is hidden to some extent by the caching. Nevertheless, if

the amount of remote data needed greatly exceeds the cache capacity, cache misses will occur constantly and performance will be poor.

Thus we have a situation that UMA machines have excellent performance but are limited in size and are quite expensive. NC-NUMA machines scale to somewhat larger sizes but require manual or semi-automated placement of pages, often with mixed results. The problem is that it is hard to predict which pages will be needed where, and in any case, a page is often too large a unit to move around. CC-NUMA machines, such as the Sun Fire E25K, may experience poor performance if many CPUs need a lot of remote data. All in all, each of these designs has serious limitations.

An alternative kind of multiprocessor tries to get around all these problems by using each CPU's main memory as a cache. In this design, called **COMA** (**Cache Only Memory Access**), pages do not have fixed home machines, as they do in NUMA and CC-NUMA machines. In fact, pages are not significant at all.

Instead, the physical address space is split into cache lines, which migrate around the system on demand. Blocks do not have home machines. Like nomads in some Third World countries, home is where you are right now. A memory that just attracts lines as needed is called an **attraction memory**. Using the main RAM as a big cache greatly increases the hit rate, hence the performance.

Unfortunately, as usual, there is no such thing as a free lunch. COMA systems introduce two new problems:

1. How are cache lines located?
2. When a line is purged, what happens if it is the last copy?

The first problem relates to the fact that after the MMU has translated a virtual address to a physical address, if the line is not in the true hardware cache, there is no easy way to tell if it is in main memory at all. The paging hardware does not help here at all because each page is made up of many individual cache lines that wander around independently. Furthermore, even if it is known that a line is not in main memory, where is it then? It is not possible to just ask the home machine, because there is no home machine.

Some solutions to the location problem have been proposed. To see if a cache line is in main memory, new hardware could be added to keep track of the tag for each cached line. The MMU could then compare the tag for the line needed to the tags for all the cache lines in memory to look for a hit. This solution needs additional hardware.

A somewhat different solution is to map entire pages in but not require that all the cache lines be present. In this solution, the hardware would need a bit map per page, giving one bit per cache line indicating the line's presence or absence. In this design, called **simple COMA** if a cache line is present, it must be in the right position in its page, but if it is not present, any attempt to use it causes a trap to allow the software to go find it and bring it in.

This leads us to finding lines that are really remote. One solution is to give each page a home machine in terms of where its directory entry is, but not where the data are. Then a message can be sent to the home machine to at least locate the cache line. Other schemes involve organizing memory as a tree and searching upward until the line is found.

The second problem in the list above relates to not purging the last copy. As in CC-NUMA, a cache line may be at multiple nodes at once. When a cache miss occurs, a line must be fetched, which usually means a line must be thrown out. What happens if the line chosen happens to be the last copy? In that case, it cannot be thrown out.

One solution is to go back to the directory and check to see if there are other copies. If so, the line can be safely thrown out. Otherwise, it has to be migrated somewhere else. Another solution is to label one copy of each cache line as the master copy and never throw it out. This solution avoids the need to check with the directory. All in all, COMA offers promise to provide better performance than CC-NUMA, but few COMA machines have been built, so more experience is needed. The first two COMA machines built were the KSR-1 (Burkhardt et al., 1992) and the Data Diffusion Machine (Hagersten et al., 1992). More recent papers on COMA are Vu et al. (2008) and Zhang and Jesshope (2008).

8.4 MESSAGE-PASSING MULTICOMPUTERS

As we saw in Fig. 8-23, the two kinds of MIMD parallel processors are multiprocessors and multicomputers. In the previous section we studied multiprocessors. We saw that they appear to the operating system as having shared memory that can be accessed using ordinary LOAD and STORE instructions. This shared memory can be implemented in many ways as we have seen, including snooping buses, data crossbars, multistage switching networks, and various directory-based schemes. Nevertheless, programs written for a multiprocessor can just access any location in memory without knowing anything about the internal topology or implementation scheme. This illusion is what makes multiprocessors so attractive and why programmers like this programming model.

On the other hand, multiprocessors also have their limitations, which is why multicomputers are important, too. First and foremost, multiprocessors do not scale to large sizes. We saw the enormous amount of hardware Sun had to use to get the E25K to scale to 72 CPUs. In contrast, we will study a multicomputer below that has 65,536 CPUs. It will be years before anyone builds a commercial 65,536-node multiprocessor. By then million-node multicomputers will be in use.

In addition, memory contention in a multiprocessor can severely affect performance. If 100 CPUs are all trying to read and write the same variables constantly, contention for the various memories, buses, and directories can cause an enormous performance hit.

As a consequence of these and other factors, there is a great deal of interest in building and using parallel computers in which each CPU has its own private memory, not directly accessible to any other CPU. These are the multicomputers. Programs on multicomputer CPUs interact using primitives like send and receive to explicitly pass messages because they cannot get at each other's memory with LOAD and STORE instructions. This difference completely changes the programming model.

Each node in a multicomputer consists of one or a few CPUs, some RAM (conceivably shared among the CPUs at that node only), a disk and/or other I/O devices, and a communication processor. The communication processors are connected by a high-speed interconnection network of the types we discussed in Sec. 8.3.3. Many different topologies, switching schemes, and routing algorithms are used. What all multicomputers have in common is that when an application program executes the send primitive, the communication processor is notified and transmits a block of user data to the destination machine (possibly after first asking for and getting permission). A generic multicomputer is shown in Fig. 8-36.

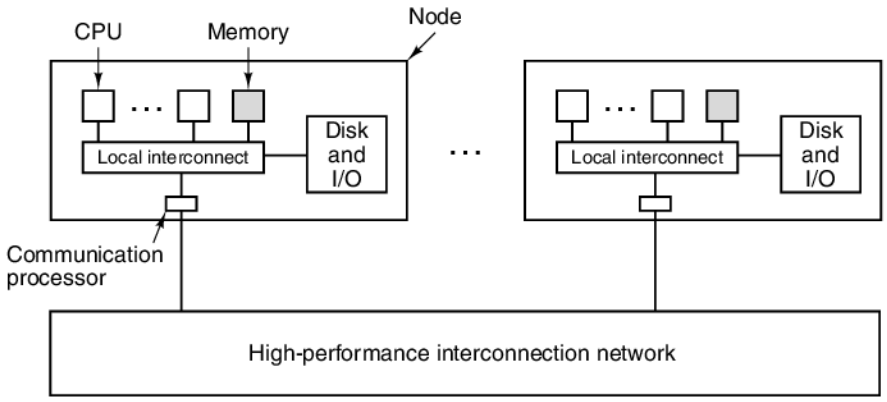


Figure 8-36. A generic multicomputer.

8.4.1 Interconnection Networks

In Fig. 8-36 we see that multicomputers are held together by interconnection networks. Now it is time to look more closely at these interconnection networks. Interestingly enough, multiprocessors and multicomputers are surprisingly similar in this respect because multiprocessors often have multiple memory modules that must also be interconnected with one another and with the CPUs. Thus the material in this section frequently applies to both kinds of systems.

The fundamental reason why multiprocessor and multicomputer interconnection networks are similar is that at the very bottom both of them use message

passing. Even on a single-CPU machine, when the processor wants to read or write a word, what it typically does is assert certain lines on the bus and wait for a reply. This action is fundamentally like message passing: the initiator sends a request and waits for a response. In large multiprocessors, communication between CPUs and remote memory almost always consists of the CPU sending an explicit message, called a **packet**, to memory requesting some data, and the memory sending back a reply packet.

Topology

The topology of an interconnection network describes how the links and switches are arranged, for example, as a ring or as a grid. Topological designs can be modeled as graphs, with the links as arcs and the switches as nodes, as shown in Fig. 8-37. Each node in an interconnection network (or its graph) has some number of links connected to it. Mathematicians call the number of links the **degree** of the node; engineers call it the **fanout**. In general, the greater the fanout, the more routing choices there are and the greater the fault tolerance, that is, the ability to continue functioning even if a link fails by routing around it. If every node has k arcs and the wiring is done right, it is possible to design the network so that it remains fully connected even if $k - 1$ links fail.

Another property of an interconnection network (or its graph) is its **diameter**. If we measure the distance between two nodes by the number of arcs that have to be traversed to get from one to the other, then the diameter of a graph is the distance between the two nodes that are the farthest apart (i.e., have the greatest distance between them). The diameter of an interconnection network is related to the worst-case delay when sending packets from CPU to CPU or from CPU to memory because each hop across a link takes a finite amount of time. The smaller the diameter, the better the worst-case performance. Also important is the average distance between two nodes, since this relates to the average packet transit time.

Yet another important property of an interconnection network is its transmission capacity, that is, how much data it can move per second. One useful measure of this capacity is the **bisection bandwidth**. To compute this quantity, we first have to (conceptually) partition the network into two equal (in terms of number of nodes) but unconnected parts by removing a set of arcs from its graph. Then we compute the total bandwidth of the arcs that have been removed. There may be many different ways to partition the network into two equal parts. The bisection bandwidth is the minimum of all the possible partitions. The significance of this number is that if the bisection bandwidth is, say, 800 bits/sec, then if there is a lot of communication between the two halves, the total throughput may be limited to only 800 bits/sec, in the worst case. Many designers believe bisection bandwidth is the most important metric of an interconnection network. Many interconnection networks are designed with the goal of maximizing the bisection bandwidth.

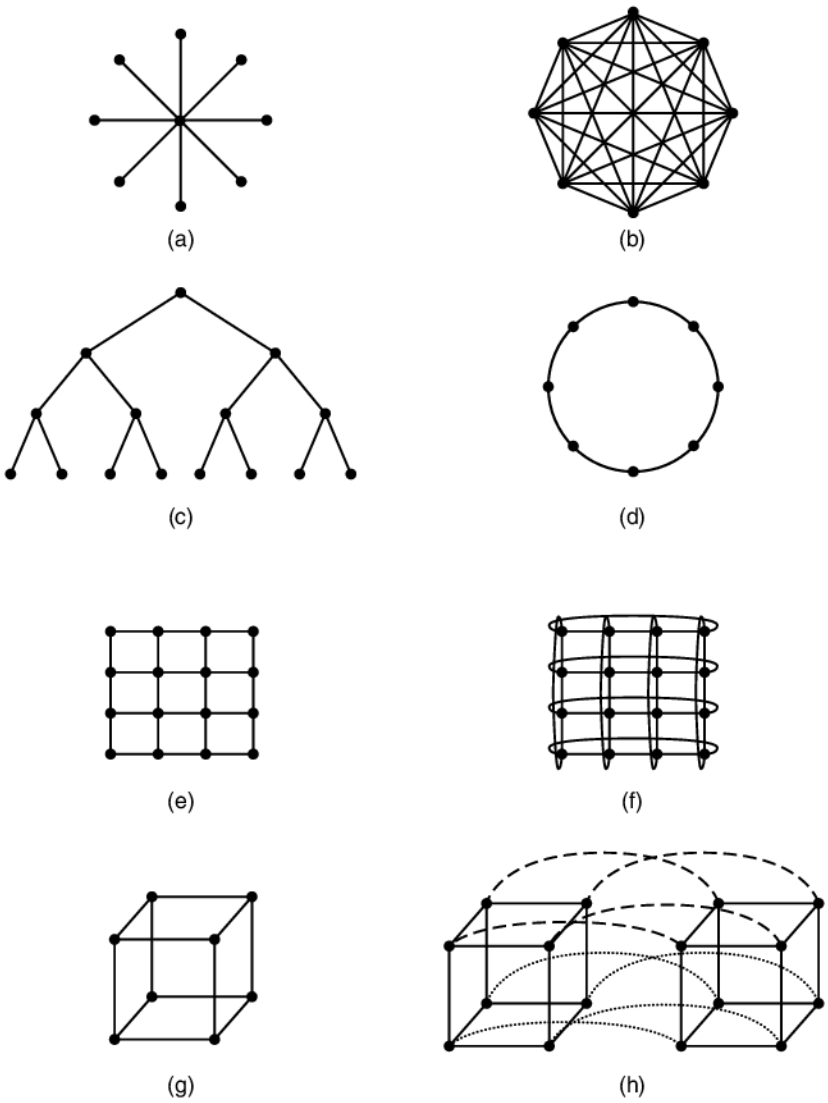


Figure 8-37. Various topologies. The heavy dots represent switches. The CPUs and memories are not shown. (a) A star. (b) A complete interconnect. (c) A tree. (d) A ring. (e) A grid. (f) A double torus. (g) A cube. (h) A 4D hypercube.

Interconnection networks can be characterized by their **dimensionality**. For our purposes, the dimensionality is determined by the number of choices there are to get from the source to the destination. If there is never any choice (i.e., there is only one path from each source to each destination), the network is zero dimensional. If there is one dimension in which a choice can be made, for example, go

east or go west, the network is one dimensional. If there are two axes, so a packet can go east or west, or alternatively, go north or south, the network is two dimensional, and so on.

Several topologies are shown in Fig. 8-37. Only the links (lines) and switches (dots) are shown here. The memories and CPUs (not shown) would typically be attached to the switches by interfaces. In Fig. 8-37(a), we have a zero-dimensional **star** configuration, in which the CPUs and memories would be attached to the outer nodes, with the central one just doing switching. Although a simple design, for a large system, the central switch is likely to be a major bottleneck. Also, from a fault-tolerance perspective, this is a poor design since a single failure at the central switch completely destroys the system.

In Fig. 8-37(b), we have another zero-dimensional design that is at the other end of the spectrum, a **full interconnect**. Here every node has a direct connection to every other node. This design maximizes the bisection bandwidth, minimizes the diameter, and is exceedingly fault tolerant (it can lose any six links and still be fully connected). Unfortunately, the number of links required for k nodes is $k(k-1)/2$, which quickly gets out of hand for large k.

Another topology is the **tree**, illustrated in Fig. 8-37(c). A problem with this design is that the bisection bandwidth is equal to the link capacity. Since there will normally be a lot of traffic near the top of the tree, the top few nodes will become bottlenecks. One way around this problem is to increase the bisection bandwidth by giving the upper links more bandwidth. For example, the lowest-level links might have a capacity b, those at the next level might have a capacity $2b$ and the top-level links might each have $4b$. Such a design is called a **fat tree** and has been used in commercial multicomputers, such as the (now-defunct) Thinking Machines' CM-5.

The **ring** of Fig. 8-37(d) is a one-dimensional topology by our definition because every packet sent has a choice of going left or going right. The **grid** or **mesh** of Fig. 8-37(e) is a two-dimensional design that has been used in many commercial systems. It is highly regular, easy to scale up to large sizes, and has a diameter that increases only as the square root of the number of nodes. A variant on the grid is the **double torus** of Fig. 8-37(f), which is a grid with the edges connected. Not only is it more fault tolerant than the grid, but the diameter is also less because the opposite corners can now communicate in only two hops.

Yet another popular topology is the three-dimensional torus. It consists of a 3D-structure with nodes at the points (i, j, k) where all coordinates are integers in the range from (1, 1, 1) to (l, m, n). Each node has six neighbors, two along each axis. The nodes at the edges have links that wrap around to the opposite edge, just as with the 2D torus.

The **cube** of Fig. 8-37(g) is a regular three-dimensional topology. We have illustrated a $2 \times 2 \times 2$ cube, but in the general case it could be a $k \times k \times k$ cube. In Fig. 8-37(h) we have a four-dimensional cube constructed from two three-dimensional cubes with the corresponding nodes connected. We could make a five-

dimensional cube by cloning the structure of Fig. 8-37(h) and connecting the corresponding nodes to form a block of four cubes. To go to six dimensions, we could replicate the block of four cubes and interconnect the corresponding nodes, and so on. An n-dimensional cube formed this way is called a **hypercube**. Many parallel computers use this topology because the diameter grows linearly with the dimensionality. Put in other words, the diameter is the base 2 logarithm of the number of nodes, so, for example, a 10-dimensional hypercube has 1024 nodes but a diameter of only 10, giving excellent delay properties. Note that in contrast, 1024 nodes arranged as a 32×32 grid has a diameter of 62, more than six times worse than the hypercube. The price paid for the smaller diameter is that the fanout and thus the number of links (and the cost) is much larger for the hypercube. Nevertheless, the hypercube is a common choice for high-performance systems.

Multicomputers come in all shapes and sizes, so it is hard to give a clean taxonomy of them. Nevertheless, two general "styles" stand out: the MPPs and the clusters. We will now study each of these in turn.

8.4.2 MPPs—Massively Parallel Processors

The first category consists of the **MPP**s (**Massively Parallel Processors**), which are huge multimillion-dollar supercomputers. These are used in science, in engineering, and in industry for very large calculations, for handling very large numbers of transactions per second, or for data warehousing (storing and managing immense databases). Initially, MPPs were primarily used as scientific supercomputers, but now most of them are used in commercial environments. In a sense, these machines are the successors to the mighty mainframes of the 1960s (but the connection is tenuous, sort of like a paleontologist claiming that a flock of sparrows is the successor to the *Tyrannosaurus Rex*). To a large extent, the MPPs have displaced SIMD machines, vector supercomputers, and array processors at the top of the digital food chain.

Most of these machines use standard CPUs as their processors. Popular choices are Intel processors, the Sun UltraSPARC, and the IBM PowerPC. What sets the MPPs apart is their use of a very high-performance proprietary interconnection network designed to move messages with low latency and at high bandwidth. Both of these are important because the vast majority of all messages are small (well under 256 bytes), but most of the total traffic is caused by large messages (more than 8 KB). MPPs also come with extensive proprietary software and libraries.

Another point that characterizes MPPs is their enormous I/O capacity. Problems big enough to warrant using MPPs invariably have massive amounts of data to be processed, often terabytes. These data must be distributed among many disks and need to be moved around the machine at great speed.

Finally, another issue specific to MPPs is their attention to fault tolerance. With thousands of CPUs, several failures per week are just inevitable. Having an

18-hour run aborted because one CPU crashed is unacceptable, especially when one such failure is to be expected every week. Thus large MPPs always have special hardware and software for monitoring the system, detecting failures, and recovering from them smoothly.

While it would be nice to study the general principles of MPP design now, in truth, there are not many principles. When you come right down to it, an MPP is a collection of more-or-less standard computing nodes connected by a very fast interconnect of the types we have already examined. So instead, we will now look at two examples of MPPs: BlueGene/P and Red Storm.

BlueGene

As a first example of a massively parallel processor, we will now examine the IBM BlueGene system. IBM conceived this project in 1999 as a massively parallel supercomputer for solving computationally intensive problems in, among other fields, the life sciences. For example, biologists believe that the three-dimensional structure of a protein determines its functionality, yet computing the 3D structure of one small protein from the laws of physics took years on the supercomputers of that period. The number of proteins found in human beings is over half a million. Many of them are extremely large and their misfolding is known to be responsible for certain diseases (e.g., cystic fibrosis). Clearly, determining the 3D structure of all the human proteins would require increasing the world's computing power by many orders of magnitude, and modeling protein folding is only one problem that BlueGene was designed to handle. Equally complex challenges in molecular dynamics, climate modeling, astronomy, and even financial modeling also require orders of magnitude improvement in supercomputing.

IBM felt that there was enough of a market for massive supercomputing that it invested $100 million to design and build BlueGene. In November 2001, Livermore National Laboratory, run by the U.S. Department of Energy, signed up as a partner and first customer for the first version of the BlueGene family, called **BlueGene/L**. In 2007, IBM deployed the second generation of the BlueGene supercomputer, called the **BlueGene/P**, which we detail here.

The goal of the BlueGene project was not just to produce the world's fastest MPP, but to also to produce the most efficient one in terms of teraflops/dollar, teraflops/watt, and teraflops/m^3. For this reason, IBM rejected the philosophy behind previous MPPs, which was to use the fastest components money could buy. Instead, a decision was made to produce a custom system-on-a-chip component that was to run at a modest speed and low power in order to produce a very large machine with a high packing density. The first BlueGene/P was delivered to a German university in November 2007. The system contained 65,536 processors, and it was capable of 167 teraflops/sec. When deployed it was the fastest computer in Europe, and the sixth fastest computer in the world. The system was also regarded as one of the most computationally power-efficient supercomputers in the world,

able to produce 371 megaflops/W, making it nearly twice as power efficient as its predecessor the BlueGene/L. This first BlueGene/P deployment was upgraded in 2009 to include 294,912 processors, giving it a computational punch of 1 petaflop/sec.

The heart of the BlueGene/P system is the custom node chip illustrated in Fig. 8-38. It consists of four PowerPC 450 cores running at 850 MHz. The PowerPC 450 is a pipelined dual-issue superscalar processor popular in embedded systems. Each core has a pair of dual-issue floating-point units, which together can issue four floating-point instructions per clock cycle. The floating-point units have been augmented with a number of SIMD-type instructions sometimes useful in scientific computations on arrays. While no performance slouch, this chip is clearly not a top-of-the-line multiprocessor.

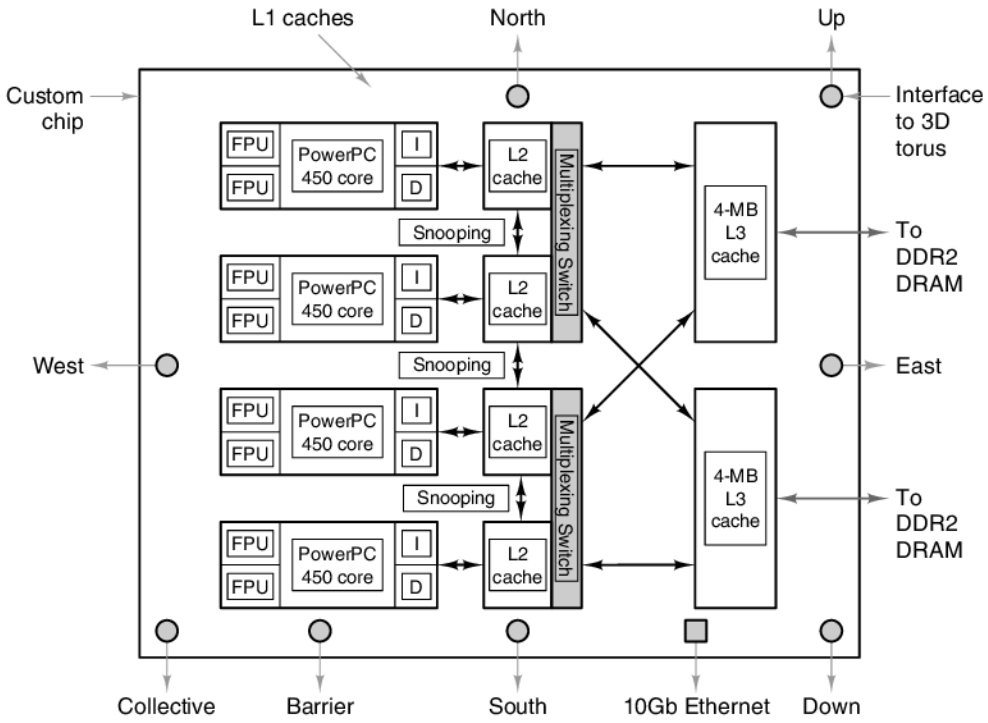


Figure 8-38. The BlueGene/P custom processor chip.

Three levels of cache are present on the chip. The first consists of a split L1 cache with 32 KB for instructions and 32 KB for data. The second is a unified cache consisting of a unified 2-KB cache. The L2 caches are really prefetch buffers rather than true caches. They snoop on each other and are cache consistent. The third level is a unified 4-MB shared cache that feeds data to the L2 caches. The four processors share access to two 4-MB L3 cache modules. There is cache

coherency between the L1 caches on the four CPUs. Thus when a shared piece of memory resides in more than one cache, accesses to that storage by one processor will be immediately visible to the other three processors. A memory reference that misses on the L1 cache but hits on the L2 cache takes about 11 clock cycles. A miss on L2 that hits on L3 takes about 28 cycles. Finally, a miss on L3 that has to go to the main DRAM takes about 75 cycles.

The four CPUs are connected via a high-bandwidth bus to a 3D torus network, which requires six connections: up, down, north, south, east, and west. In addition, each processor has a port to the collective network, used for broadcasting data to all processors. The barrier port is used to speed up synchronization operations, giving each processor fast access to a specialized synchronization network.

At the next level up, IBM designed a custom card that holds one of the chips shown in Fig. 8-38 along with 2 GB of DDR2 DRAM. The chip and the card are shown in Fig. 8-39(a)–(b) respectively.

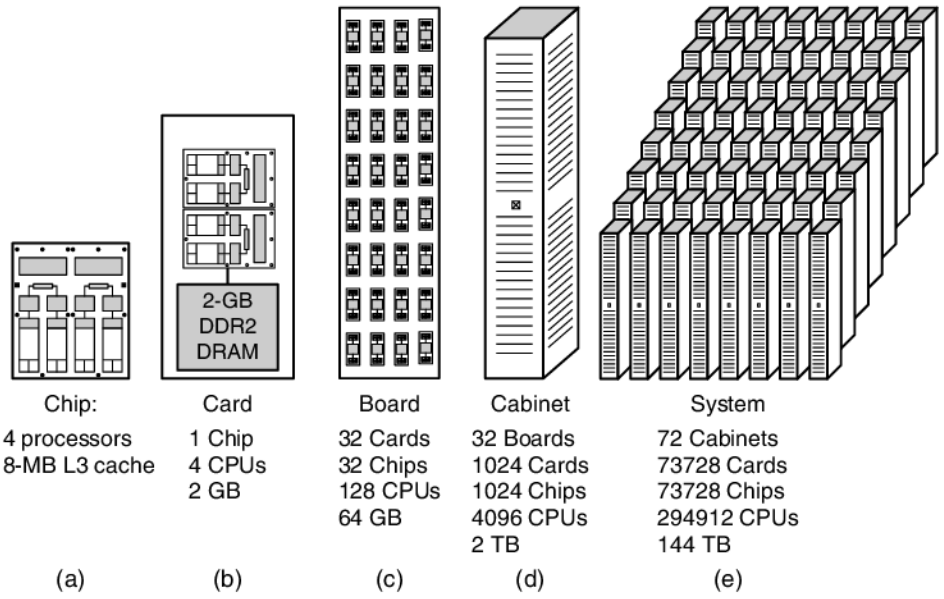


Figure 8-39. The BlueGene/P: (a) chip. (b) card. (c) board. (d) cabinet. (e) system.

The cards are mounted on plug-in boards, with 32 cards per board for a total of 32 chips (and thus 128 CPUs) per board. Since each card contains 2 GB of DRAM, the boards contain 64 GB apiece. One board is illustrated in Fig. 8-39(c). At the next level, 32 of these boards are plugged into a cabinet, packing 4096 CPUs into a single cabinet. A cabinet is illustrated in Fig. 8-39(d).

Finally, a full system, consisting of up to 72 cabinets with 294,912 CPUs, is depicted in Fig. 8-39(e). A PowerPC 450 can issue up to 6 instructions/cycle, thus

a full BlueGene/P system could conceivably issue up to 1,769,472 instructions per cycle. At 850 MHz, this gives the system a possible performance of 1.504 petaflops/sec. However, data hazards, memory latency, and lack of parallelism work together to ensure that the actual throughput of the system is much less. Real programs running on the BlueGene/P have demonstrated performance rates of up to 1 petaflop/sec.

The system is a multicomputer in the sense that no CPU has direct access to any memory except the 2 GB on its own card. While CPUs within a processor chip have shared memory, processors at the board, rack, and system level do not share the same memory. In addition, there is no demand paging because there are no local disks to page off. Instead, the system has 1152 I/O nodes, which are connected to disks and the other peripheral devices.

All in all, while the system is extremely large, it is also quite straightforward with little new technology except in the area of high-density packaging. The decision to keep it simple was no accident since a major goal was high reliability and availability. Consequently, a great deal of careful engineering went into the power supplies, fans, cooling, and cabling with the goal of a mean-time-to-failure of at least 10 days.

To connect all the chips, a scalable, high-performance interconnect is needed. The design used is a three-dimensional torus measuring $72 \times 32 \times 32$. As a consequence, each CPU needs only six connections to the torus network, two to other CPUs logically above and below it, north and south of it, and east and west of it. These six connections are labeled east, west, north, south, up, and down, respectively in Fig. 8-38. Physically, each 1024-node cabinet is an $8 \times 8 \times 16$ torus. Pairs of neighboring cabinets form an $8 \times 8 \times 32$ torus. Four pairs of cabinets in the same row form an $8 \times 32 \times 32$ torus. Finally, all 9 rows form a $72 \times 32 \times 32$ torus.

All links are thus point-to-point and operate at 3.4 Gbps. Since each of the 73,728 nodes has three links to "higher" numbered nodes, one in each dimension, the total bandwidth of the system is 752 terabits/sec. The information content of this book is about 300 million bits, including all the art in encapsulated PostScript format, so BlueGene/P could move 2.5 million copies of this book per second. Where they would go and who would want them is left as an exercise for the reader.

Communication on the 3D torus is done in the form of **virtual cut through** routing. This technique is somewhat akin to store-and-forward packet switching, except that entire packets are not stored before being forwarded. As soon as a byte has arrived at one node, it can be forwarded to the next one along the path, even before the entire packet has arrived. Both dynamic (adaptive) and deterministic (fixed) routing are possible. A small amount of special-purpose hardware on the chip is used to implement the virtual cut through.

In addition to the main 3D torus used for data transport, four other communication networks are present. The second one is the collective network in the form