6

Distributed Shared Memory

In Chap. 1 we saw that two kinds of multiple-processor systems exist: multiprocessors and multicomputers. In a multiprocessor, two or more CPUs share a common main memory. Any process, on any processor, can read or write any word in the shared memory simply by moving data to or from the desired location. In a multicomputer, in contrast, each CPU has its own private memory. Nothing is shared.

To make an agricultural analogy, a multiprocessor is a system with a herd of pigs (processes) eating from a single feeding trough (shared memory). A multicomputer is a design in which each pig has its own private feeding trough. To make an educational analogy, a multiprocessor is a blackboard in the front of the room which all the students are looking at, whereas a multicomputer is each student looking at his or her own notebook. Although this difference may seem minor, it has far-reaching consequences.

The consequences affect both hardware and software. Let us first look at the implications for the hardware. Designing a machine in which many processors use the same memory simultaneously is surprisingly difficult. Bus-based multiprocessors, as described in Sec. 1.3.1, cannot be used with more than a few dozen processors because the bus tends to become a bottleneck. Switched multiprocessors, as described in Sec. 1.3.2, can be made to scale to large systems, but they are relatively expensive, slow, complex, and difficult to maintain.

In contrast, large multicomputers are easier to build. One can take an

almost unlimited number of single-board computers, each containing a CPU, memory, and a network interface, and connect them together. Multicomputers with thousands of processors are commercially available from various manufacturers. (Please note that throughout this chapter we use the terms "CPU" and "processor" interchangeably.) From a hardware designer's perspective, multicomputers are generally preferable to multiprocessors.

Now let us consider the software. Many techniques are known for programming multiprocessors. For communication, one process just writes data to memory, to be read by all the others. For synchronization, critical regions can be used, with semaphores or monitors providing the necessary mutual exclusion. There is an enormous body of literature available on interprocess communication and synchronization on shared-memory machines. Every operating systems textbook written in the past twenty years devotes one or more chapters to the subject. In short, a large amount of theoretical and practical knowledge exists about how to program a multiprocessor.

With multicomputers, the reverse is true. Communication generally has to use message passing, making input/output the central abstraction. Message passing brings with it many complicating issues, among them flow control, lost messages, buffering, and blocking. Although various solutions have been proposed, programming with message passing remains tricky.

To hide some of the difficulties associated with message passing, Birrell and Nelson (1984) proposed using remote procedure calls. In their scheme, now widely used, the actual communication is hidden away in library procedures. To use a remote service, a process just calls the appropriate library procedure, which packs the operation code and parameters into a message, sends it over the network, and waits for the reply. While this frequently works, it cannot easily be used to pass graphs and other complex data structures containing pointers. It also fails for programs that use global variables, and it makes passing large arrays expensive, since they must be passed by value rather than by reference.

In short, from a software designer's perspective, multiprocessors are definitely preferable to multicomputers. Herein lies the dilemma. Multicomputers are easier to build but harder to program. Multiprocessors are the opposite: harder to build but easier to program. What we need are systems that are both easy to build and easy to program. Attempts to build such systems form the subject of this chapter.

6.1. INTRODUCTION

In the early days of distributed computing, everyone implicitly assumed that programs on machines with no physically shared memory (i.e., multicomputers) obviously ran in different address spaces. Given this mindset, communication

was naturally viewed in terms of message passing between disjoint address spaces, as described above. In 1986, Li proposed a different scheme, now known under the name **distributed shared memory** (**DSM**) (Li, 1986; and Li and Hudak, 1989). Briefly summarized, Li and Hudak proposed having a collection of workstations connected by a LAN share a single paged, virtual address space. In the simplest variant, each page is present on exactly one machine. A reference to a *local* pages is done in hardware, at full memory speed. An attempt to reference a page on a different machine causes a hardware page fault, which traps to the operating system. The operating system then sends a message to the remote machine, which finds the needed page and sends it to the requesting processor. The faulting instruction is then restarted and can now complete.

In essence, this design is similar to traditional virtual memory systems: when a process touches a nonresident page, a trap occurs and the operating system fetches the page and maps it in. The difference here is that instead of getting the page from the disk, the operating system gets it from another processor over the network. To the user processes, however, the system looks very much like a traditional multiprocessor, with multiple processes free to read and write the shared memory at will. All communication and synchronization can be done via the memory, with no communication visible to the user processes. In effect, Li and Hudak devised a system that is both easy to program (logically shared memory) and easy to build (no physically shared memory).

Unfortunately, there is no such thing as a free lunch. While this system is indeed easy to program and easy to build, for many applications it exhibits poor performance, as pages are hurled back and forth across the network. This behavior is analogous to thrashing in single-processor virtual memory systems. In recent years, making these distributed shared memory systems more efficient has been an area of intense research, with numerous new techniques discovered. All of these have the goal of minimizing the network traffic and reducing the latency between the moment a memory request is made and the moment it is satisfied.

One approach is not to share the entire address space, only a selected portion of it, namely just those variables or data structures that need to be used by more than one process. In this model, one does not think of each machine as having direct access to an ordinary memory but rather, to a collection of shared variables, giving a higher level of abstraction. Not only does this strategy greatly reduce the amount of data that must be shared, but in most cases, considerable information about the shared data is available, such as their types, which can help optimize the implementation.

One possible optimization is to replicate the shared variables on multiple machines. By sharing replicated variables instead of entire pages, the problem of simulating a multiprocessor has been reduced to that of how to keep multiple

copies of a set of typed data structures consistent. Potentially, reads can be done locally without any network traffic, and writes can be done using a multicopy update protocol. Such protocols are widely used in distributed data base systems, so ideas from that field may be of use.

Going still further in the direction of structuring the address space, instead of just sharing variables we could share encapsulated data types, often called **objects**. These differ from shared variables in that each object has not only some data, but also procedures, called **methods**, that act on the data. Programs may only manipulate an object's data by invoking its methods. Direct access to the data is not permitted. By restricting access in this way, various new optimizations become possible.

Doing everything in software has a different set of advantages and disadvantages from using the paging hardware. In general, it tends to put more restrictions on the programmer but may achieve better performance. Many of these restrictions (e.g., working with objects) are considered good software engineering practice and are desirable in their own right. We will come back to this subject later.

Before getting into distributed shared memory in more detail, we must first take a few steps backward to see what shared memory really is and how shared-memory multiprocessors actually work. After that we will examine the semantics of sharing, since they are surprisingly subtle. Finally, we will come back to the design of distributed shared memory systems. Because distributed shared memory can be intimately related to computer architecture, operating systems, runtime systems, and even programming languages, all of these topics will come into play in this chapter.

6.2. WHAT IS SHARED MEMORY?

In this section we will examine several kinds of shared memory multiprocessors, ranging from simple ones that operate over a single bus, to advanced ones with highly sophisticated caching schemes. These machines are important for an understanding of distributed shared memory because much of the DSM work is being inspired by advances in multiprocessor architecture. Furthermore, many of the algorithms are so similar that it is sometimes difficult to tell whether an advanced machine is a multiprocessor or a multicomputer using a hardware implementation of distributed shared memory. We will conclude by comparing the various multiprocessor architectures to some distributed shared memory systems and discover that there is a spectrum of possible designs, from those entirely in hardware to those entirely in software. By examing the entire spectrum, we can get a better feel for where DSM fits in.

6.2.1. On-Chip Memory

Although most computers have an external memory, self-contained chips containing a CPU and all the memory also exist. Such chips are produced by the millions, and are widely used in cars, appliances, and even toys. In this design, the CPU portion of the chip has address and data lines that directly connect to the memory portion. Figure 6-1(a) shows a simplified diagram of such a chip.

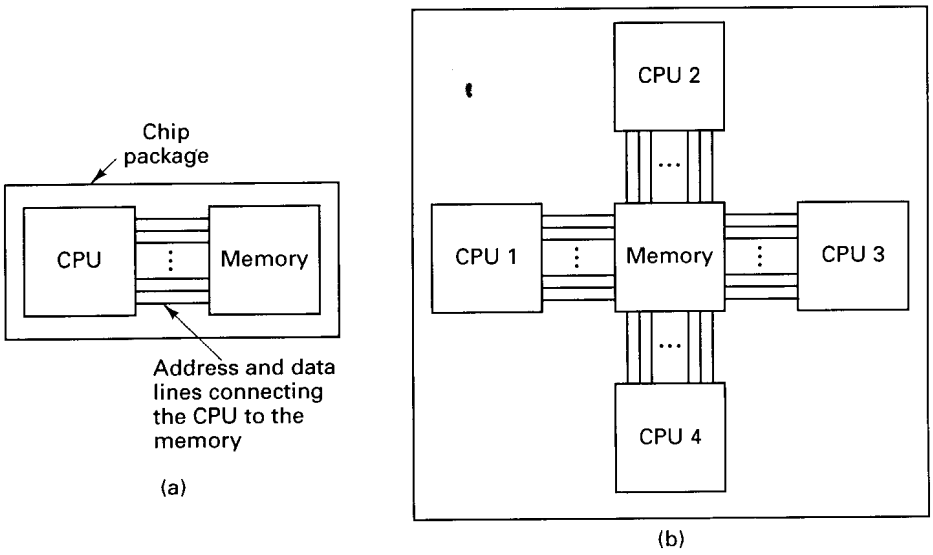


Fig. 6-1. (a) A single-chip computer. (b) A hypothetical shared-memory multiprocessor.

One could imagine a simple extension of this chip to have multiple CPUs directly sharing the same memory, as shown in Fig. 6-1(b). While it is possible to construct a chip like this, it would be complicated, expensive, and highly unusual. An attempt to construct a one-chip multiprocessor this way, with, say, 100 CPUs directly accessing the same memory would be impossible for engineering reasons. A different approach to sharing memory is needed.

6.2.2. Bus-Based Multiprocessors

If we look closely at Fig. 6-1(a), we see that the connection between the CPU and the memory is a collection of parallel wires, some holding the address the CPU wants to read or write, some for sending or receiving data, and the rest for controlling the transfers. Such a collection of wires is called a **bus**. This bus is on-chip, but in most systems, buses are external and are used to connect printed circuit boards containing CPUs, memories, and I/O controllers. On a

desktop computer, the bus is typically etched onto the main board (the parentboard), which holds the CPU and some of the memory, and into which I/O cards are plugged. On minicomputers the bus is sometimes a flat cable that wends its way among the processors, memories, and I/O controllers.

A simple but practical way to build a multiprocessor is to base it on a bus to which more than one CPU is connected. Fig. 6-2(a) illustrates a system with three CPUs and a memory shared among all of them. When any of the CPUs wants to read a word from the memory, it puts the address of the word it wants on the bus and asserts (puts a signal on) a bus control line indicating that it wants to do a read. When the memory has fetched the requested word, it puts the word on the bus and asserts another control line to announce that it is ready. The CPU then reads in the word. Writes work in an analogous way.

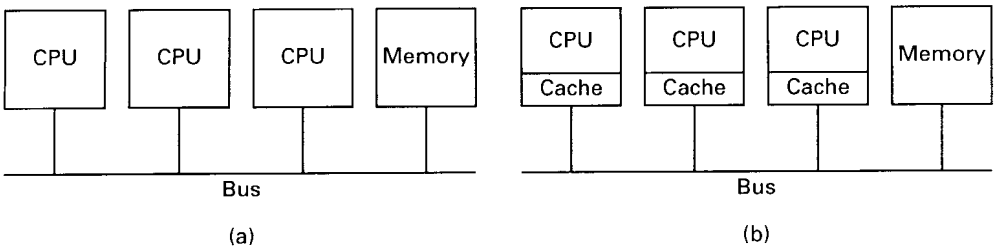


Fig. 6-2. (a) A multiprocessor. (b) A multiprocessor with caching.

To prevent two or more CPUs from trying to access the memory at the same time, some kind of bus arbitration is needed. Various schemes are in use. For example, to acquire the bus, a CPU might first have to request it by asserting a special request line. Only after receiving permission would it be allowed to use the bus. The granting of this permission can be done in a centralized way, using a bus arbitration device, or in a decentralized way, with the first requesting CPU along the bus winning any conflict.

The disadvantage of having a single bus is that with as few as three or four CPUs the bus is likely to become overloaded. The usual approach taken to reduce the bus load is to equip each CPU with a **snooping cache** (sometimes called a **snoopy cache),** so called because it "snoops" on the bus. Caches are shown in Fig. 6-2(b). They have been the subject of a large amount of research over the years (Agarwal et al., 1988; Agarwal and Cherian, 1989; Archibald and Baer, 1986; Cheong and Veidenbaum, 1988; Dahlgren et al., 1994; Eggers and Katz, 1989a, 1989b; Nayfeh and Olukotun, 1994; Przybylski et al., 1988; Scheurich and Dubois, 1987; Thekkath and Eggers, 1994; Vernon et al., 1988; and Weber and Gupta, 1989). All of these papers present slightly different **cache consistency protocols**, that is, rules for making sure that different caches do not contain different values for the same memory location.

One particularly simple and common protocol is called **write through**. When a CPU first reads a word from memory, that word is fetched over the bus and is stored in the cache of the CPU making the request. If that word is needed again later, the CPU can take it from the cache without making a memory request, thus reducing bus traffic. These two cases, read miss (word not cached) and read hit (word cached) are shown in Fig. 6-3 as the first two lines in the table. In simple systems, only the word requested is cached, but in most, a block of words of say, 16 or 32 words, is transferred and cached on the initial access and kept for possible future use.

Event	Action taken by a cache in response to its own CPU's operation	Action taken by a cache in response to a remote CPU's operation
Read miss	Fetch data from memory and store in cache	(No action)
Read hit	Fetch data from local cache	(No action)
Write miss	Update data in memory and store in cache	(No action)
Write hit	Update memory and cache	Invalidate cache entry

Fig. 6-3. The *write-through* cache consistency protocol. The entries for *hit* in the third column mean that the snooping CPU has the word in its cache, not that the requesting CPU has it.

Each CPU does its caching independent of the others. Consequently, it is possible for a particular word to be cached at two or more CPUs at the same time. Now let us consider what happens when a write is done. If no CPU has the word being written in its cache, the memory is just updated, as if caching were not being used. This operation requires a normal bus cycle. If the CPU doing the write has the only copy of the word, its cache is updated and memory is updated over the bus as well.

So far, so good. The trouble arises when a CPU wants to write a word that two or more CPUs have in their caches. If the word is currently in the cache of the CPU doing the write, the cache entry is updated. Whether it is or not, it is also written to the bus to update memory. All the other caches see the write (because they are snooping on the bus) and check to see if they are also holding the word being modified. If so, they invalidate their cache entries, so that after the write completes, memory is up-to-date and only one machine has the word in its cache.

An alternative to invalidating other cache entries is to update all of them. Updating is slower than invalidating in most cases, however. Invalidating requires supplying just the address to be invalidated, whereas updating needs to provide the new cache entry as well. If these two items must be presented on the bus consecutively, extra cycles will be required. Even if it is possible to put an address and a data word on the bus simultaneously, if the cache block size is

more than one word, multiple bus cycles will be needed to update the entire block. The issue of invalidate vs. update occurs in all cache protocols and also in DSM systems.

The complete protocol is summarized in Fig. 6-3. The first column lists the four basic events that can happen. The second one tells what a cache does in response to its *own* CPU's actions. The third one tells what happens when a cache sees (by snooping) that a *different* CPU has had a hit or miss. The only time cache S (the snooper) must do something is when it sees that another CPU has written a word that S has cached (a write hit from S's point of view). The action is for S to delete the word from its cache.

The *write-through* protocol is simple to understand and implement but has the serious disadvantage that all writes use the bus. While the protocol certainly reduces bus traffic to some extent, the number of CPUs that can be attached to a single bus is still too small to permit large-scale multiprocessors to be built using it.

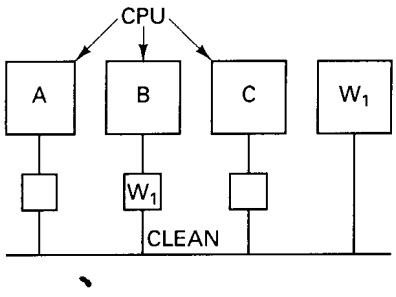
Fortunately, for many actual programs, once a CPU has written a word, that CPU is likely to need the word again, and it is unlikely that another CPU will use the word quickly. This situation suggests that if the CPU using the word could somehow be given temporary "ownership" of the word, it could avoid having to update memory on subsequent writes until a different CPU exhibited interest in the word. Such cache protocols exist. Goodman (1983) devised the first one, called **write once**. However, this protocol was designed to work with an existing bus and was therefore more complicated than is strictly necessary. Below we will describe a simplified version of it, which is typical of all ownership protocols. Other protocols are described and compared by Archibald and Baer (1986).

Our protocol manages cache blocks, each of which can be in one of the following three states:

1. INVALID — This cache block does not contain valid data.
2. CLEAN — Memory is up-to-date; the block may be in other caches.
3. DIRTY — Memory is incorrect; no other cache holds the block.

The basic idea is that a word that is being read by multiple CPUs is allowed to be present in all their caches. A word that is being heavily written by only one machine is kept in its cache and not written back to memory on every write to reduce bus traffic.

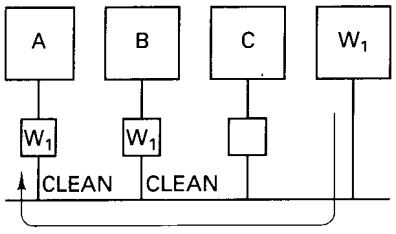
The operation of the protocol can best be illustrated by an example. For simplicity in this example, we will assume that each cache block consists of a single word. Initially, B has a cached copy of the word at address W, as illustrated in Fig. 6-4(a). The value is W_1. The memory also has a valid copy. In



Memory is correct

Initial state.
Word W containing value W_1 is in memory and is also cached by B.

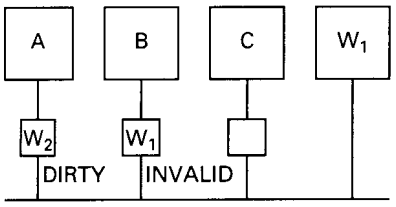
(a)



Memory is correct

A reads word W and gets W_1. B does not respond to the read, but the memory does.

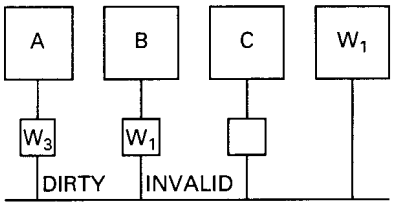
(b)



Memory is correct

A writes a value W_2. B snoops on the bus, sees the write, and invalidates its entry. A's copy is marked DIRTY.

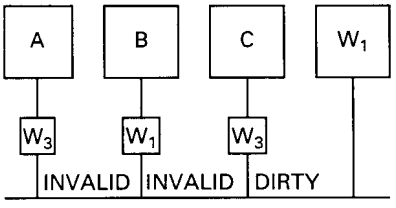
(c)



Memory is correct

A writes W again. This and subsequent writes by A are done locally, without any bus traffic.

(d)



Memory is correct

C reads or writes W. A sees the request by snooping on the bus, provides the value, and invalidates its own entry. C now has the only valid copy.

(e)

Fig. 6-4. An example of how a cache ownership protocol works.

Fig. 6-4(b), *A* requests and gets a copy of *W* from the memory. Although *B* sees the read request go by, it does not respond to it.

Now *A* writes a new value, W_2 to *W*. *B* sees the write request and responds by invalidating its cache entry. *A*'s state is changed to DIRTY, as shown in Fig. 6-4(c). The DIRTY state means that *A* has the only cached copy of *W* and that memory is out-of-date for *W*.

At this point, *A* overwrites the word again, as shown in Fig. 6-4(d). The write is done locally, in the cache, with no bus traffic. All subsequent writes also avoid updating memory.

Sooner or later, some other CPU, *C* in Fig. 6-4(e), accesses the word. *A* sees the request on the bus and asserts a signal that inhibits memory from responding. Instead, *A* provides the needed word and invalidates its own entry. *C* sees that the word is coming from another cache, not from memory, and that it is in DIRTY state, so it marks the entry accordingly. *C* is now the owner, which means that it can now read and write the word without making bus requests. However, it also has the responsibility of watching out for other CPUs that request the word, and servicing them itself. The word remains in DIRTY state until it is purged from the cache it is currently residing in for lack of space. At that time it disappears from all caches and is written back to memory.

Many small multiprocessors use a cache consistency protocol similar to this one, often with small variations. It has three important properties:

1. Consistency is achieved by having all the caches do bus snooping.
2. The protocol is built into the memory management unit.
3. The entire algorithm is performed in well under a memory cycle.

As we will see later, some of these do not hold for larger (switched) multiprocessors, and none of them hold for distributed shared memory.

6.2.3. Ring-Based Multiprocessors

The next step along the path toward distributed shared memory systems are ring-based multiprocessors, exemplified by **Memnet** (Delp, 1988; Delp et al., 1991; and Tam et al., 1990). In Memnet, a single address space is divided into a private part and a shared part. The private part is divided up into regions so that each machine has a piece for its stacks and other unshared data and code. The shared part is common to all machines (and distributed over them) and is kept consistent by a hardware protocol roughly similar to those used on bus-based multiprocessors. Shared memory is divided into 32-byte blocks, which is the unit in which transfers between machines take place.

All the machines in Memnet are connected together in a modified token-

passing ring. The ring consists of 20 parallel wires, which together allow 16 data bits and 4 control bits to be sent every 100 nsec, for a data rate of 160 Mbps. The ring is illustrated in Fig. 6-5(a). The ring interface, MMU (Memory Management Unit), cache, and part of the memory are integrated together in the **Memnet device**, which is shown in the top third of Fig. 6-5(b).

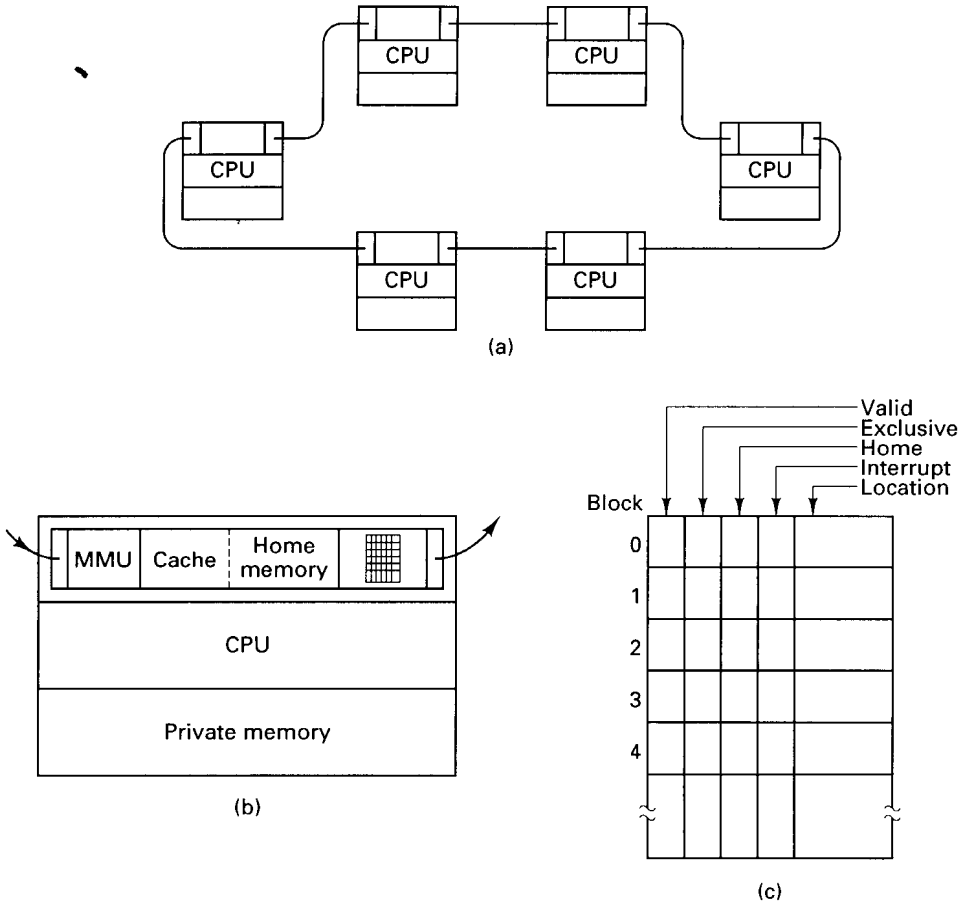


Fig. 6-5. (a) The Memnet ring. (b) A single machine. (c) The block table.

Unlike the bus-based multiprocessors of Fig. 6-2, in Memnet there is no centralized global memory. Instead, each 32-byte block in the shared address space has a home machine on which physical memory is always reserved for it, in the *Home memory* field of Fig. 6-5(b). A block may be cached on a machine other than its home machine. (The cache and home memory areas share the same buffer pool, but since they are used slightly differently, we treat them here as separate entities.) A read-only block may be present on multiple machines; a

read-write block may be present on only one machine. In both cases, a block need not be present on its home machine. All the home machine does is provide a guaranteed place to store the block if no other machine wants to cache it. This feature is needed because there is no global memory. In effect, the global memory has been spread out over all the machines.

The Memnet device on each machine contains a table, shown in Fig. 6-5(c), which contains an entry for each block in the shared address space, indexed by block number. Each entry contains a *Valid* bit telling whether the block is present in the cache and up to date, an *Exclusive* bit, specifying whether the local copy, if any, is the only one, a *Home* bit, which is set only if this is the block's home machine, an *Interrupt* bit, used for forcing interrupts, and a *Location* field that tells where the block is located in the cache if it is present and valid.

Having looked at the architecture of Memnet, let us now examine the protocols it uses. When the CPU wants to read a word from shared memory, the memory address to be read is passed to the Memnet device, which checks the block table to see if the block is present. If so, the request is satisfied immediately. If not, the Memnet device waits until it captures the circulating token, then puts a request packet onto the ring and suspends the CPU. The request packet contains the desired address and a 32-byte dummy field.

As the packet passes around the ring, each Memnet device along the way checks to see if it has the block needed. If so, it puts the block in the dummy field and modifies the packet header to inhibit subsequent machines from doing so. If the block's *Exclusive* bit is set, it is cleared. Because the block has to be somewhere, when the packet comes back to the sender, it is guaranteed to contain the block requested. The CPU sending the request then stores the block, satisfies the request, and releases the CPU.

A problem arises if the requesting machine has no free space in its cache to hold the incoming block. To make space, it picks a cached block at random and sends it home, thus freeing up a cache slot. Blocks whose *Home* bit are set are never chosen since they are already home.

Writes work slightly differently than reads. Three cases have to be distinguished. If the block containing the word to be written is present and is the only copy in the system (i.e., the *Exclusive* bit is set), the word is just written locally.

If the needed block is present but it is not the only copy, an invalidation packet is first sent around the ring to force all other machines to discard their copies of the block about to be written. When the invalidation packet arrives back at the sender, the *Exclusive* bit is set for that block and the write proceeds locally.

If the block is not present, a packet is sent out that combines a read request and an invalidation request. The first machine that has the block copies it into

the packet and discards its own copy. All subsequent machines just discard the block from their caches. When the packet comes back to the sender, it is stored there and written.

Memnet is similar to a bus-based multiprocessor in most ways. In both cases, read operations always return the value most recently written. Also, in both designs, a block may be absent from a cache, present in multiple caches for reading, or present in a single cache for writing. The protocols are similar, too; however, Memnet has no centralized global memory.

The biggest difference between bus-based multiprocessors and ring-based multiprocessors such as Memnet is that the former are tightly coupled, with the CPUs normally being in a single rack. In contrast, the machines in a ring-based multiprocessor can be much more loosely coupled, potentially even on desktops spread around a building, like machines on a LAN, although this loose coupling can adversely effect performance. Furthermore, unlike a bus-based multiprocessor, a ring-based multiprocessor like Memnet has no separate global memory. The caches are all there is. In both respects, ring-based multiprocessors are almost a hardware implementation of distributed shared memory.

One is tempted to say that a ring-based multiprocessor is like a duck-billed platypus—theoretically it ought not exist because it combines the properties of two categories said to be mutually exclusive (multiprocessors and distributed shared memory machines; mammals and birds, respectively). Nevertheless, it does exist, and shows that the two categories are not quite so distinct as one might think.

6.2.4. Switched Multiprocessors

Although bus-based multiprocessors and ring-based multiprocessors work fine for small systems (up to around 64 CPUs), they do not scale well to systems with hundreds or thousands of CPUs. As CPUs are added, at some point the bus or ring bandwidth saturates. Adding additional CPUs does not improve the system performance.

Two approaches can be taken to attack the problem of not enough bandwidth:

1. Reduce the amount of communication.
2. Increase the communication capacity.

We have already seen an example of an attempt to reduce the amount of communication by using caching. Additional work in this area might center on improving the caching protocol, optimizing the block size, reorganizing the program to increase locality of memory references, and so on.

Nevertheless, eventually there comes a time when every trick in the book

has been used, but the insatiable designers still want to add more CPUs and there is no bus bandwidth left. The only way out is to add more bus bandwidth. One approach is to change the topology, going, for example, from one bus to two buses or to a tree or grid. By changing the topology of the interconnection network, it is possible to add additional communication capacity.

A different method is to build the system as a hierarchy. Continue to put some number of CPUs on a single bus, but now regard this entire unit (CPUs plus bus) as a cluster. Build the system as multiple clusters and connect the clusters using an intercluster bus, as shown in Fig. 6-6(a). As long as most CPUs communicate primarily within their own cluster, there will be relatively little intercluster traffic. If one intercluster bus proves to be inadequate, add a second intercluster bus, or arrange the clusters in a tree or grid. If still more bandwidth is needed, collect a bus, tree, or grid of clusters together into a supercluster, and break the system into multiple superclusters. The superclusters can be connected by a bus, tree, or grid, and so on. Fig. 6-6(b) shows a system with three levels of buses.

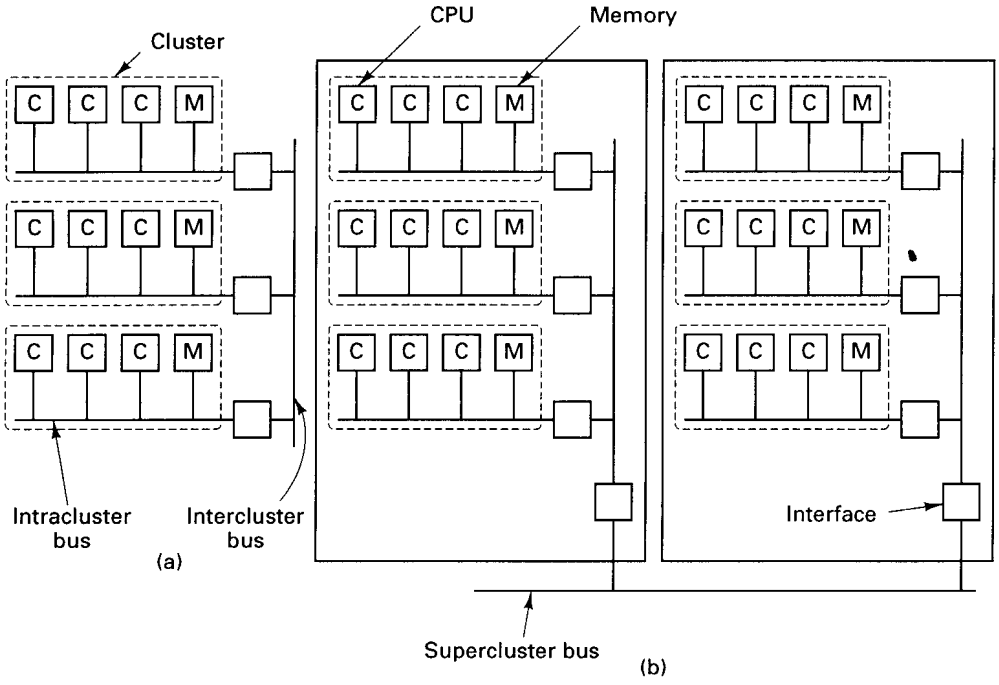


Fig. 6-6. (a) Three clusters connected by an intercluster bus to form one supercluster. (b) Two superclusters connected by a supercluster bus.

In this section we will look at a hierarchical design based on a grid of

clusters. The machine, called **Dash**, was built as a research project at Stanford University (Lenoski et al., 1992). Although many other researchers are doing similar work, this one is a typical example. In the remainder of this section we will focus on the 64-CPU prototype that was actually constructed, but the design principles have been chosen carefully so that one could equally well build a much larger version. The description given below has been simplified slightly in a few places to avoid going into unnecessary detail.

A simplified diagram of the Dash prototype is presented in Fig. 6-7(a). It consists of 16 clusters, each cluster containing a bus, four CPUs, 16M of the global memory, and some I/O equipment (disks, etc.). To avoid clutter in the figure, the I/O equipment and two of the CPUs have been omitted from each cluster. Each CPU is able to snoop on its local bus, as in Fig. 6-2(b), but not on other buses.

The total address space available in the prototype is 256M, divided up into 16 regions of 16M each. The global memory of cluster 0 holds addresses 0 to 16M. The global memory of cluster 1 holds addresses 16M to 32M, and so on. Memory is cached and transferred in units of 16-byte blocks, so each cluster has 1M memory blocks within its address space.

Directories

Each cluster has a **directory** that keeps track of which clusters currently have copies of its blocks. Since each cluster owns 1M memory blocks, it has 1M entries in its directory, one per block. Each entry holds a bit map with one bit per cluster telling whether or not that cluster has the block currently cached. The entry also has a 2-bit field telling the state of the block. The directories are essential to the operation of Dash, as we shall see. In fact, the name Dash comes from "Directory Architecture for Shared memory."

Having 1M entries of 18 bits each means that the total size of each directory is over 2M bytes. With 16 clusters, the total directory memory is just over 36M, or about 14 percent of the 256M. If the number of CPUs per cluster is increased, the amount of directory memory is not changed. Thus having more CPUs per cluster allows the directory cost to be amortized over a larger number of CPUs, reducing the cost per CPU. Also, the cost of the directory and bus controllers per CPU are reduced. In theory, the design works fine with one CPU per cluster, but the cost of the directory and bus hardware per CPU then becomes larger.

A bit map is not the only way to keep track of which cluster holds which cache block. An alternative approach is to organize each directory entry as an explicit list telling which clusters hold the corresponding cache block. If there is little sharing, the list approach will require fewer bits, but if there is substantial sharing, it will require more bits. Lists also have the disadvantage of being

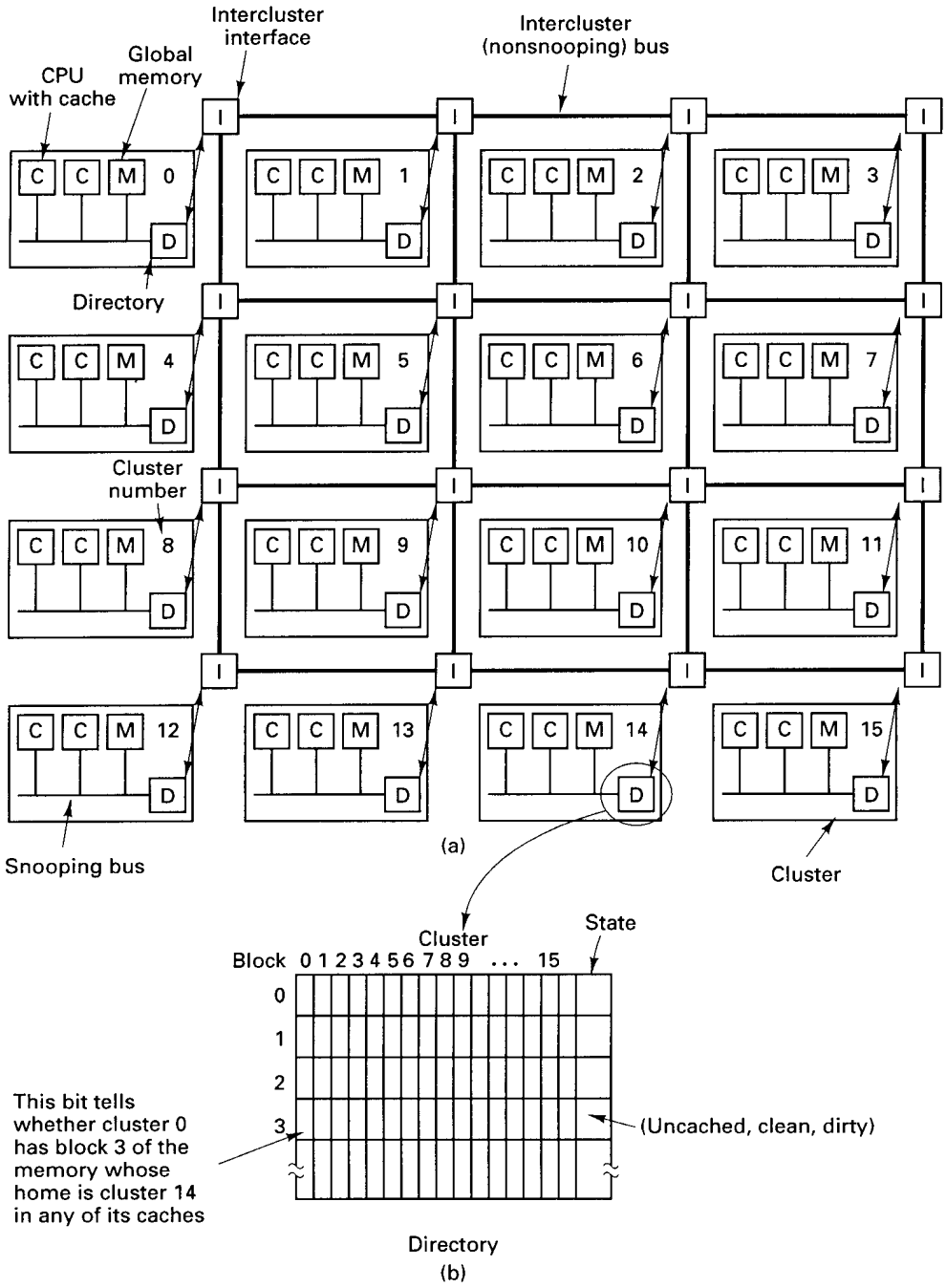


Fig. 6-7. (a) A simplified view of the Dash architecture. Each cluster actually has four CPUs, but only two are shown here. (b) A Dash directory.

variable-length data structures, but these problems can be solved. The M.I.T. Alewife multiprocessor (Agarwal et al., 1991; and Kranz et al., 1993), for example, is similar to Dash in many respects, although it uses lists instead of bit maps in its directories and handles directory overflows in software.

Each cluster in Dash is connected to an interface that allows the cluster to communicate with other clusters. The interfaces are connected by intercluster links (primitive buses) in a rectangular grid, as shown in Fig. 6-7(a). As more clusters are added to the system, more intercluster links are added, too, so the bandwidth increases and the system scales. The intercluster link system uses **wormhole routing**, which means that the first part of a packet can be forwarded even before the entire packet has been received, thus reducing the delay at each hop. Although not shown in the figure, there are actually two sets of intercluster links, one for request packets and one for reply packets. The intercluster links cannot be snooped upon.

Caching

Caching is done on two levels: a first-level cache and a larger second-level cache. The first-level cache is a subset of the second-level cache, so only the latter will concern us here. Each (second-level) cache monitors the local bus using a protocol somewhat similar to the cache ownership protocol of Fig. 6-4.

Each cache block can be in one of the following three states:

1. UNCACHED—The only copy of the block is in this memory.
2. CLEAN —Memory is up-to-date; the block may be in several caches.
3. DIRTY —Memory is incorrect; only one cache holds the block.

The state of each cache block is stored in the *State* field of its directory entry, as shown in Fig. 6-7(b).

Protocols

The Dash protocols are based on ownership and invalidation. At every instant, each cache block has a unique owner. For UNCACHED or CLEAN blocks, the block's home cluster is the owner. For DIRTY blocks, the cluster holding the one and only copy is the owner. Writing on a CLEAN block requires first finding and invalidating all existing copies. This is where the directories come in.

To see how this mechanism works, let us first consider how a CPU reads a memory word. It first checks its own caches. If neither cache has the word, a request is issued on the local cluster bus to see if another CPU in the cluster has the block containing it. If one does, a cache-to-cache transfer of the block is

executed to place the block in the requesting CPU's cache. If the block is CLEAN, a copy is made; if it is DIRTY, the home directory is informed that the block is now CLEAN and shared. Either way, a hit from one of the caches satisfies the instruction but does not affect any directory's bit map.

If the block is not present in any of the cluster's caches, a request packet is sent to the block's home cluster, which can be determined by examining the upper 4 bits of the memory address. The home cluster might well be the requester's cluster, in which case the message is not sent physically. The directory management hardware at the home cluster examines its tables to see what state the block is in. If it is UNCACHED or CLEAN, the hardware fetches the block from its global memory and sends it back to the requesting cluster. It then updates its directory, marking the block as cached in the requester's cluster (if it was not already so marked).

If, however, the needed block is DIRTY, the directory hardware looks up the identity of the cluster holding the block and forwards the request there. The cluster holding the DIRTY block then sends it to the requesting cluster and marks its own copy as CLEAN because it is now shared. It also sends a copy back to the home cluster so that memory can be updated and the block state changed to CLEAN. All these cases are summarized in Fig. 6-8(a). Where a block is marked as being in a new state, it is the home directory that is changed, as it is the home directory that keeps track of the state.

Writes work differently. Before a write can be done, the CPU doing the write must be sure that it is the owner of the only copy of the cache block in the system. If it already has the block in its on-board cache and the block is DIRTY, the write can proceed immediately. If it has the block but it is CLEAN, a packet is first sent to the home cluster requesting that all other copies be tracked down and invalidated.

If the requesting CPU does not have the cache block, it issues a request on the local bus to see if any of the neighbors have it. If so, a cache-to-cache (or memory-to-cache) transfer is done. If the block is CLEAN, all other copies, if any, must be invalidated by the home cluster.

If the local broadcast fails to turn up a copy and the block is homed elsewhere, a packet is sent to the home cluster. Three cases can be distinguished here. If the block is UNCACHED, it is marked DIRTY and sent to the requester. If it is CLEAN, all copies are invalidated and then the procedure for UNCACHED is followed. If it is DIRTY, the request is forwarded to the remote cluster currently owning the block (if needed). This cluster invalidates its own copy and satisfies the request. The various cases are shown in Fig. 6-8(b).

Obviously, maintaining memory consistency in Dash (or any large multiprocessor) is nothing at all like the simple model of Fig. 6-1(b). A single memory access may require a substantial number of packets to be sent. Furthermore, to keep memory consistent, the access usually cannot be completed until all the

Location where the block was found

Block state	R's cache	Neighbor's cache	Home cluster's memory	Some cluster's cache
UNCACHED			Send block to R; mark as CLEAN and cached only in R's cluster	
CLEAN	Use block	Copy block to R's cache	Copy block from memory to R; mark as also cached in R's cluster	
DIRTY	Use block	Send block to R and to home cluster; tell home to mark it as CLEAN and cached in R's cluster		Send block to R and to home cluster (if cached elsewhere); tell home to mark it as CLEAN and also cached in R's cluster

(a)

Location where the block was found

Block state	R's cache	Neighbor's cache	Home cluster's memory	Some cluster's cache
UNCACHED			Send block to R; mark as DIRTY and cached only in R's cluster	
CLEAN	Send message to home asking for exclusive ownership in DIRTY state; if granted, use block	Copy and invalidate block; send message to home asking for exclusive ownership in DIRTY state	Send block to R; invalidate all cached copies; mark it as DIRTY and cached only in R's cluster	
DIRTY	Use block	Cache-to-cache transfer to R; invalidate neighbor's copy		Send block directly to R; invalidate cached copy; home marks it as DIRTY and cached only in R's cluster

(b)

Fig. 6-8. Dash protocols. The columns show where the block was found. The rows show the state it was in. The contents of the boxes show the action taken. *R* refers to the requesting CPU. An empty box indicates an impossible situation. (a) Reads. (b) Writes.

packets have been acknowledged, which can have a serious effect on performance. To get around these problems, Dash uses a variety of special techniques, such as two sets of intercluster links, pipelined writes, and different memory semantics than one might expect. We will discuss some of these issues later. For the time being, the bottom line is that this implementation of "shared memory" requires a large data base (the directories), a considerable amount of computing power (the directory management hardware), and a potentially large number of packets that must be sent and acknowledged. We will see later that implementing distributed shared memory has precisely the same properties. The difference between the two lies much more in the implementation technique than in the ideas, architecture, or algorithms.

6.2.5. NUMA Multiprocessors

If nothing else, it should be abundantly clear by now that hardware caching in large multiprocessors is not simple. Complex data structures must be maintained by the hardware and intricate protocols, such as those of Fig. 6-8, must be built into the cache controller or MMU. The inevitable consequence is that large multiprocessors are expensive and not in widespread use.

However, researchers have spent a considerable amount of effort looking at alternative designs that do not require elaborate caching schemes. One such architecture is the **NUMA (NonUniform Memory Access)** multiprocessor. Like a traditional **UMA (Uniform Memory Access)** multiprocessor, a NUMA machine has a single virtual address space that is visible to all CPUs. When any CPU writes a value to location *a*, a subsequent read of *a* by a different processor will return the value just written.

The difference between UMA and NUMA machines lies not in the semantics but in the performance. On a NUMA machine, access to a remote memory is much slower than access to a local memory, and no attempt is made to hide this fact by hardware caching. The ratio of a remote access to a local access is typically 10:1, with a factor of two variation either way not being unusual. Thus a CPU can directly execute a program that resides in a remote memory, but the program may run an order of magnitude slower than it would have had it been in local memory.

Examples of NUMA Multiprocessors

To make the concept of a NUMA machine clearer, consider the example of Fig. 6-9(a), Cm*, the first NUMA machine (Jones et al., 1977). The machine consisted of a number of clusters, each consisting of a CPU, a microprogrammable MMU, a memory module, and possibly some I/O devices, all connected by a bus. No caches were present, and no bus snooping occurred. The clusters were connected by intercluster buses, one of which is shown in the figure.

When a CPU made a memory reference, the request went to the CPU's MMU, which then examined the upper bits of the address to see which memory was needed. If the address was local, the MMU just issued a request on the local bus. If it was to a distant memory, the MMU built a request packet containing the address (and for a write, the data word to be written), and sent it to the destination cluster over an intercluster bus. Upon receiving the packet, the destination MMU carried out the operation and returned the word (for a read) or an acknowledgement (for a write). Although it was possible for a CPU to run entirely from a remote memory, sending a packet for each word read and each word written slowed down operation by an order of magnitude.

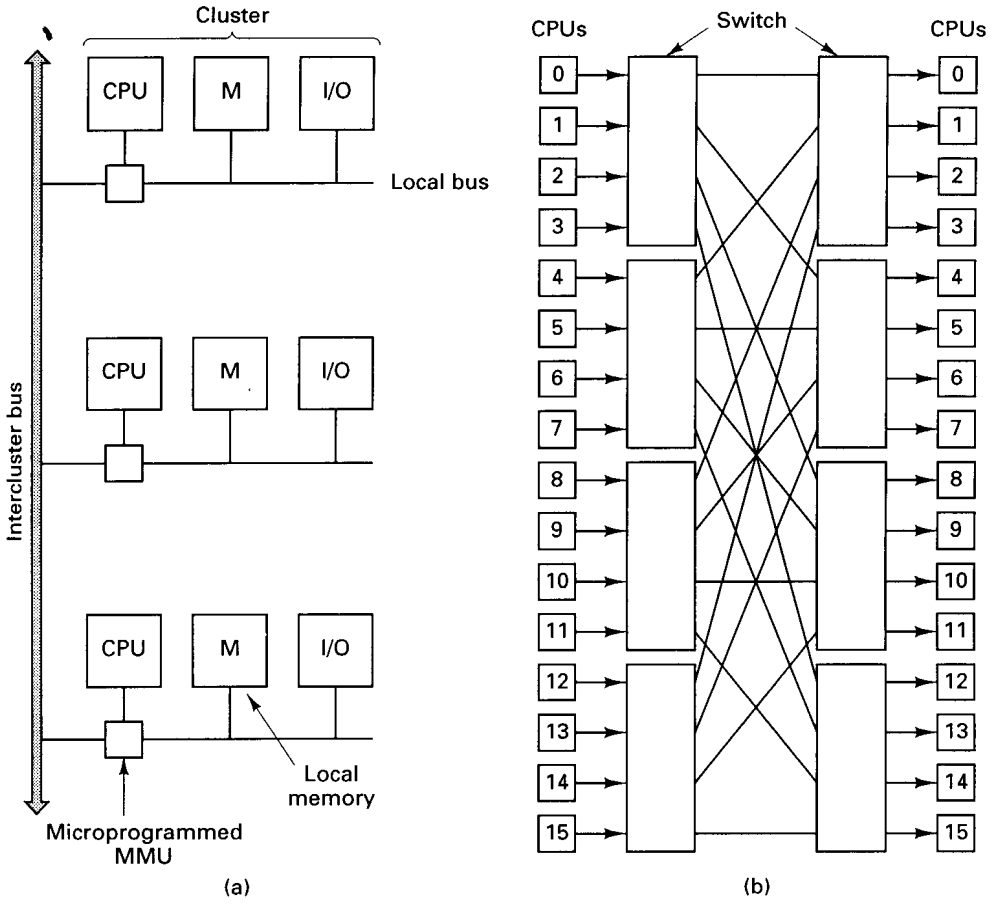


Fig. 6-9. (a) A simplified view of the Cm* system. (b) The BBN Butterfly. The CPUs on the right are the same as those on the left (i.e., the architecture is really a cylinder).

Figure 6-9(b) shows another NUMA machine, the BBN Butterfly. In this design, each CPU is coupled directly to one memory. Each of the small squares in Fig. 6-9(b) represents a CPU plus memory pair. The CPUs on the right-hand side of the figure are the same as those on the left. The CPUs are wired up via eight switches, each having four input ports and four output ports. Local memory requests are handled directly; remote requests are turned into request packets and sent to the appropriate memory via the switching network. Here, too, programs can run remotely, but at a tremendous penalty in performance.

Although neither of these examples has any global memory, NUMA machines can be equipped with memory that is not attached to any CPU.

Bolosky et al. (1989), for example, describe a bus-based NUMA machine that has a global memory that does not belong to any CPU but can be accessed by all of them (in addition to the local memories).

Properties of NUMA Multiprocessors

NUMA machines have three key properties that are of concern to us:

1. Access to remote memory is possible.
2. Accessing remote memory is slower than accessing local memory.
3. Remote access times are not hidden by caching.

The first two points are self explanatory. The third may require some clarification. In Dash and most other modern UMA multiprocessors, remote access is slower than local access as well. What makes this property bearable is the presence of caching. When a remote word is touched, a block of memory around it is fetched to the requesting processor's cache, so that subsequent references go at full speed. Although there is a slight delay to handle the cache fault, running out of remote memory can be only fractionally more expensive than running out of local memory. The consequence of this observation is that it does not matter so much which pages live in which memory: code and data are automatically moved by the hardware to wherever they are needed (although a bad choice of the home cluster for each page in Dash adds extra overhead).

NUMA machines do not have this property, so it matters a great deal which page is located in which memory (i.e., on which machine). The key issue in NUMA software is the decision of where to place each page to maximize performance. Below we will briefly summarize some ideas due to LaRowe and Ellis (1991). Other work is described in (Cox and Fowler, 1989; LaRowe et al., 1991; and Ramanathan and Ni, 1991).

When a program on a NUMA machine starts up, pages may or may not be manually prepositioned on certain processors' machines (their home processors). In either case, when a CPU tries to access a page that is not currently mapped into its address space, it causes a page fault. The operating system catches the fault and has to make a decision. If the page is read-only, the choice is to replicate the page (i.e., make a local copy without disturbing the original) or to map the virtual page onto the remote memory, thus forcing a remote access for all addresses on that page. If the page is read-write, the choice is to migrate the page to the faulting processor (invalidating the original page) or to map the virtual page onto the remote memory.

The trade-offs involved here are simple. If a local copy is made (replication or migration) and the page is not reused much, considerable time will have been

wasted fetching it for nothing. On the other hand, if no copy is made, the page is mapped remote, and many accesses follow, they will all be slow. In essence, the operating system has to guess if the page will be heavily used in the future. If it guesses wrong, a performance penalty will be extracted.

Whichever decision is made, the page is mapped in, either local or remote, and the faulting instruction restarted. Subsequent references to that page are done in hardware, with no software intervention. If no other action were taken, then a wrong decision once made could never be reversed.

NUMA Algorithms

To allow mistakes to be corrected and to allow the system to adapt to changes in reference patterns, NUMA systems usually have a daemon process, called the **page scanner**, running in the background. Periodically (e.g., every 4 sec), the page scanner gathers usage statistics about local and remote references, which are maintained with help from the hardware. Every n times it runs, the page scanner reevaluates earlier decisions to copy pages or map them to remote memories. If the usage statistics indicate that a page is in the wrong place, the page scanner unmaps the page so that the next reference causes a page fault, allowing a new placement decision to be made. If a page is moved too often within a short interval, the page scanner can mark the page as **frozen**, which inhibits further movement until some specified event happens (e.g., some number of seconds have elapsed).

Numerous strategies have been proposed for NUMA machines, differing in the algorithm used by the scanner to invalidate pages and the algorithm used to make placement decisions after a page fault. One possible scanner algorithm is to invalidate any page for which there have been more remote references than local ones. A stronger test is to invalidate a page only if the remote reference count has been greater than the local one the last k times the scanner has run. Other possibilities are to defrost frozen pages after t seconds have elapsed or if the remote references exceed the local ones by some amount or for some time interval.

When a page fault occurs, various algorithms are possible, always including replicate/migrate and never including replicate/migrate. A more sophisticated one is to replicate or migrate unless the page is frozen. Recent usage patterns can also be taken into account, as can the fact that the page is or is not on its "home" machine.

LaRowe and Ellis (1991) have compared a large number of algorithms and concluded that no single policy is best. The machine architecture, the size of the penalty for a remote access, and the reference pattern of the program in question all play a large role in determining which algorithm is best.

6.2.6. Comparison of Shared Memory Systems

Shared memory systems cover a broad spectrum, from systems that maintain consistency entirely in hardware to those that do it entirely in software. We have studied the hardware end of the spectrum in some detail and have given a brief summary of the software end (page-based distributed shared memory and object-based distributed shared memory). In Fig. 6-10 the spectrum is shown explicitly.

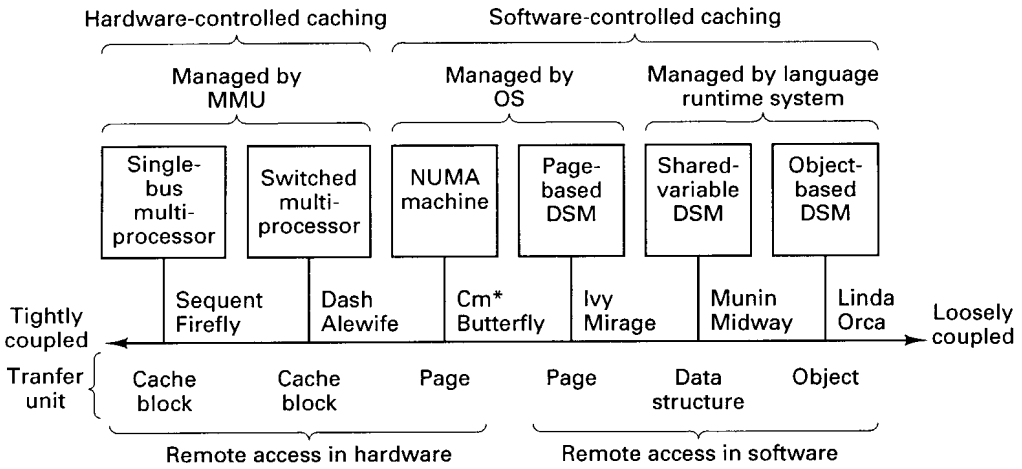


Fig. 6-10. The spectrum of shared memory machines.

On the left-hand side of Fig. 6-10 we have the single-bus multiprocessors that have hardware caches and keep them consistent by snooping on the bus. These are the simplest shared-memory machines and operate entirely in hardware. Various machines made by Sequent and other vendors and the experimental DEC Firefly workstation (five VAXes on a common bus) fall into this category. This design works fine for a small or medium number of CPUs, but degrades rapidly when the bus saturates.

Next come the switched multiprocessors, such as the Stanford Dash machine and the M.I.T. Alewife machine. These also have hardware caching but use directories and other data structures to keep track of which CPUs or clusters have which cache blocks. Complex algorithms are used to maintain consistency, but since they are stored primarily in MMU microcode (with exceptions potentially handled in software), they count as mostly "hardware" implementations.

Next come the NUMA machines. These are hybrids between hardware and software control. As in a multiprocessor, each NUMA CPU can access each

word of the common virtual address space just by reading or writing it. Unlike in a multiprocessor, however, caching (i.e., page placement and migration) is controlled by software (the operating system), not by hardware (the MMUs). Cm* (Jones et al., 1977) and the BBN Butterfly are examples of NUMA machines.

Continuing along the spectrum, we come to the machines running a page-based distributed shared memory system such as IVY (Li, 1986) and Mirage (Fleisch and Popek, 1989). Each of the CPUs in such a system has its own private memory and, unlike the NUMA machines and UMA multiprocessors, cannot reference remote memory directly. When a CPU addresses a word in the address space that is backed by a page currently located on a different machine, a trap to the operating system occurs and the required page must be fetched by software. The operating system acquires the necessary page by sending a message to the machine where the page is currently residing and asking for it. Thus both placement and access are done in software here.

Then we come to machines that share only a selected portion of their address spaces, namely shared variables and other data structures. The Munin (Bennett et al., 1990) and Midway (Bershad et al., 1990) systems work this way. User-supplied information is required to determine which variables are shared and which are not. In these systems, the focus changes from trying to pretend that there is a single common memory to how to maintain a set of replicated distributed data structures consistent in the face of updates, potentially from all the machines using the shared data. In some cases the paging hardware detects writes, which may help maintain consistency efficiently. In other cases, the paging hardware is not used for consistency management.

Finally, we have systems running object-based distributed shared memory. Unlike all the others, programs here cannot just access the shared data. They have to go through protected methods, which means that the runtime system can always get control on every access to help maintain consistency. Everything is done in software here, with no hardware support at all. Orca (Bal, 1991) is an example of this design, and Linda (Carriero and Gelernter, 1989) is similar to it in some important ways.

The differences between these six types of systems are summarized in Fig. 6-11, which shows them from tightly coupled hardware on the left to loosely coupled software on the right. The first four types offer a memory model consisting of a standard, paged, linear virtual address space. The first two are regular multiprocessors and the next two do their best to simulate them. Since the first four types act like multiprocessors, the only operations possible are reading and writing memory words. In the fifth column, the shared variables are special, but they are still accessed only by normal reads and writes. The object-based systems, with their encapsulated data and methods, can offer more general operations and represent a higher level of abstraction than raw memory.

	Multiprocessors			DSM		
Item	Single bus	Switched	NUMA	Page based	Shared variable	Object based
Linear, shared virtual address space?	Yes	Yes	Yes	Yes	No	No
Possible operations	R/W	R/W	R/W	R/W	R/W	General
Encapsulation and methods?	No	No	No	No	No	Yes
Is remote access possible in hardware?	Yes	Yes	Yes	No	No	No
Is unattached memory possible?	Yes	Yes	Yes	No	No	No
Who converts remote memory accesses to messages?	MMU	MMU	MMU	OS	Runtime system	Runtime system
Transfer medium	Bus	Bus	Bus	Network	Network	Network
Data migration done by	Hardware	Hardware	Software	Software	Software	Software
Transfer unit	Block	Block	Page	Page	Shared variable	Object

Fig. 6-11. Comparison of six kinds of shared memory systems.

The real difference between the multiprocessors and the DSM systems is whether or not remote data can be accessed just by referring to their addresses. On all the multiprocessors the answer is yes. On the DSM systems it is no: software intervention is always needed. Similarly, unattached global memory, that is, memory not associated with any particular CPU, is possible on the multiprocessors but not on the DSM systems (because the latter are collections of separate computers connected by a network).

In the multiprocessors, when a remote access is detected, a message is sent to the remote memory by the cache controller or MMU. In the DSM systems it is sent by the operating system or runtime system. The medium used is also different, being a high-speed bus (or collection of buses) for the multiprocessors and a conventional LAN (usually) for the DSM systems (although sometimes the difference between a "bus" and a "network" is arguable, having mainly to do with the number of wires).

The next point relates to who does data migration when it is needed. Here the NUMA machines are like the DSM systems: in both cases it is the software, not the hardware, which is responsible for moving data around from machine to machine. Finally, the unit of data transfer differs for the six systems, being a cache block for the UMA multiprocessors, a page for the NUMA machines and page-based DSM systems, and a variable or object for the last two.

6.3. CONSISTENCY MODELS

Although modern multiprocessors have a great deal in common with distributed shared memory systems, it is time to leave the subject of multiprocessors and move on. In our brief introduction to DSM systems at the start of this chapter, we said that they have one or more copies of each read-only page and one copy of each writable page. In the simplest implementation, when a writable page is referenced by a remote machine, a trap occurs and the page is fetched. However, if some writable pages are heavily shared, having only a single copy of each one can be a serious performance bottleneck.

Allowing multiple copies eases the performance problem, since it is then sufficient to update any copy, but doing so introduces a new problem: how to keep all the copies consistent. Maintaining perfect consistency is especially painful when the various copies are on different machines that can only communicate by sending messages over a slow (compared to memory speeds) network. In some DSM (and multiprocessor) systems, the solution is to accept less than perfect consistency as the price for better performance. Precisely what consistency means and how it can be relaxed without making programming unbearable is a major issue among DSM researchers.

A **consistency model** is essentially a contract between the software and the memory (Adve and Hill, 1990). It says that if the software agrees to obey certain rules, the memory promises to work correctly. If the software violates these rules, all bets are off and correctness of memory operation is no longer guaranteed. A wide spectrum of contracts have been devised, ranging from contracts that place only minor restrictions on the software to those that make normal programming nearly impossible. As you probably already guessed, the ones with minor restrictions do not perform nearly as well as the ones with major restrictions. Such is life. In this section we will study a variety of consistency models used in DSM systems. For additional information, see the paper by Mosberger (1993).

6.3.1. Strict Consistency

The most stringent consistency model is called **strict consistency**. It is defined by the following condition:

> *Any read to a memory location* x *returns the value stored by the most recent write operation to* x.

This definition is natural and obvious, although it implicitly assumes the existence of absolute global time (as in Newtonian physics) so that the determination of "most recent" is unambiguous. Uniprocessors have traditionally

observed strict consistency and uniprocessor programmers have come to expect such behavior as a matter of course. A system on which the program

```
a = 1; a = 2; print(a);
```

printed 1 or any value other than 2 would quickly lead to a lot of very agitated programmers (in this chapter, *print* is a procedure that prints its parameter or parameters).

In a DSM system, the matter is more complicated. Suppose x is a variable stored only on machine B. Imagine that a process on machine A reads x at time T_1, which means that a message is then sent to B to get x. Slightly later, at T_2, a process on B does a write to x. If strict consistency holds, the read should always return the old value regardless of where the machines are and how close T_2 is to T_1. However, if $T_2 - T_1$ is, say, 1 nanosecond, and the machines are 3 meters apart, in order to propagate the read request from A to B to get there before the write, the signal would have to travel at 10 times the speed of light, something forbidden by Einstein's special theory of relativity. Is it reasonable for programmers to demand that the system be strictly consistent, even if this requires violating the laws of physics?

This brings us to the matter of the contract between the software and the memory. If the contract implicitly or explicitly promises strict consistency, then the memory had better deliver it. On the other hand, a programmer who really expects strict consistency, and whose program fails if it is not present, is living dangerously. Even on a small multiprocessor, if one processor starts to write memory location a, and a nanosecond later another processor starts to read a the reader will probably get the old value from its local cache. Anyone who writes programs that fail under these circumstances should be made to stay after school and write a program to print 100 times: "I will avoid race conditions."

As a more realistic example, one could imagine a system to provide sports fans with up-to-the-minute (but perhaps not up-to-the-nanosecond) scores for sporting events worldwide. Answering a query as if it had been made 2 nanoseconds earlier or later might well be acceptable in this case, especially if it gave much better performance by allowing multiple copies of the data to be stored. In this case strict consistency is not promised, delivered, or needed.

To study consistency in detail, we will give numerous examples. To make these examples precise, we need a special notation. In this notation, several processes, P_1, P_2, and so on can be shown at different heights in the figure. The operations done by each process are shown horizontally, with time increasing to the right. Straight lines separate the processes. The symbols

$W(x)a$ and $R(y)b$

mean that a write to x with the value a and a read from y returning b have been

done, respectively. The initial value of all variables in these diagrams throughout this chapter is assumed to be 0. As an example, in Fig. 6-12(a) P_1 does a write to location x, storing the value 1. Later, P_2 reads x and sees the 1. This behavior is correct for a strictly consistent memory.

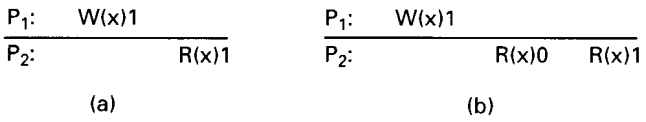


Fig. 6-12. Behavior of two processes. The horizontal axis is time. (a) Strictly consistent memory. (b) Memory that is not strictly consistent.

In contrast, in Fig. 6-12(b), P_2 does a read after the write (possibly only a nanosecond after it, but still after it), and gets 0. A subsequent read gives 1. Such behavior is incorrect for a strictly consistent memory.

In summary, when memory is strictly consistent, all writes are instantaneously visible to all processes and an absolute global time order is maintained. If a memory location is changed, all subsequent reads from that location see the new value, no matter how soon after the change the reads are done and no matter which processes are doing the reading and where they are located. Similarly, if a read is done, it gets the then-current value, no matter how quickly the next write is done.

6.3.2. Sequential Consistency

While strict consistency is the ideal programming model, it is nearly impossible to implement in a distributed system. Furthermore, experience shows that programmers can often manage quite well with weaker models. For example, all textbooks on operating systems discuss critical sections and the mutual exclusion problem. This discussion always includes the caveat that properly-written parallel programs (such as the producer-consumer problem) should not make any assumptions about the relative speeds of the processes or how their statements will interleave in time. Counting on two events within one process to happen so quickly that the other process will not be able to do something in between is looking for trouble. Instead, the reader is taught to program in such a way that the exact order of statement execution (in fact, memory references) does not matter. When the order of events is essential, semaphores or other synchronization operations should be used. Accepting this argument in fact means learning to live with a weaker memory model. With some practice, most parallel programmers are able to adapt to it.

Sequential consistency is a slightly weaker memory model than strict

consistency. It was first defined by Lamport (1979), who said that a sequentially consistent memory is one that satisfies the following condition:

> *The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

What this definition means is that when processes run in parallel on different machines (or even in pseudoparallel on a timesharing system), any valid interleaving is acceptable behavior, but *all processes must see the same sequence of memory references*. A memory in which one process (or processor) sees one interleaving and another process sees a different one is not a sequentially consistent memory. Note that nothing is said about time; that is, there is no reference to the "most recent" store. Note that in this context, a process "sees" writes from all processes but only its own reads.

That time does not play a role can be seen from Fig. 6-13. A memory behaving as shown in Fig. 6-13(a) is sequentially consistent even though the first read done by P_2 returns the initial value of 0 instead of the new value of 1.

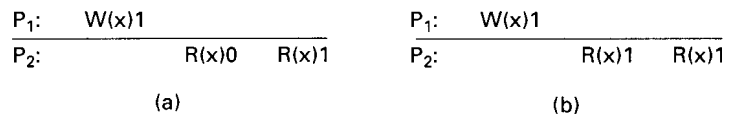


Fig. 6-13. Two possible results of running the same program.

Sequentially consistent memory does not guarantee that a read returns the value written by another process a nanosecond earlier, or a microsecond earlier, or even a minute earlier. It merely guarantees that all processes see all memory references in the same order. If the program that generated Fig. 6-13(a) is run again, it might give the result of Fig. 6-13(b). The results are not deterministic. Running a program again may not give the same result in the absence of explicit synchronization operations.

```
a = 1;          b = 1;          c = 1;
print (b, c);   print (a, c);   print (a, b);

   (a)             (b)             (c)
```

Fig. 6-14. Three parallel processes.

To make this point more explicit, let us consider the example of Fig. 6-14 (Dubois et al., 1988). Here we see the code for three processes that run in parallel on three different processors. All three processes share the same sequentially consistent distributed shared memory, and all have access to the variables *a*, *b*,

and c. From a memory reference point of view, an assignment should be seen as a write, and a print statement should be seen as a simultaneous read of its two parameters. All statements are assumed to be atomic.

Various interleaved execution sequences are possible. With six independent statements, there are potentially 720 (6!) possible execution sequences, although some of these violate program order. Consider the 120 (5!) sequences that begin with $a = 1$. Half of these have *print* (a, c) before $b = 1$ and thus violate program order. Half also have *print* (a, b) before $c = 1$ and also violate program order. Only 1/4 of the 120 sequences or 30 are valid. Another 30 valid sequences are possible starting with $b = 1$ and another 30 can begin with $c = 1$, for a total of 90 valid execution sequences. Four of these are shown in Fig. 6-15.

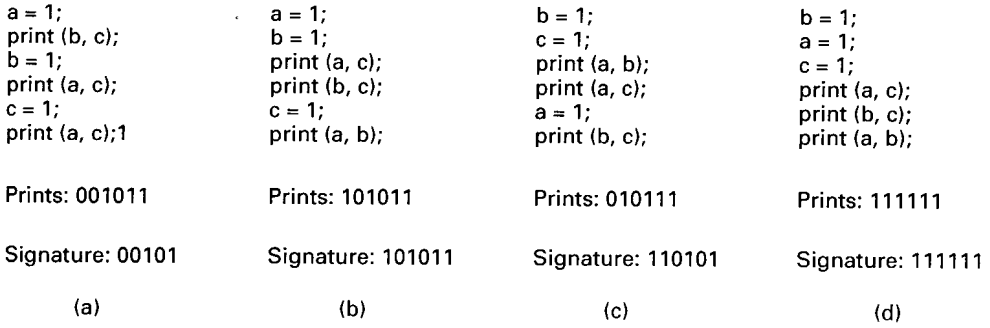


Fig. 6-15. Four valid execution sequences for the program of Fig. 6-14. The vertical axis is time, increasing downward.

In Fig. 6-15(a), the three processes are run in order, first P_1, then P_2, then P_3. The other three examples demonstrate different, but equally valid, interleavings of the statements in time. Each of the three processes prints two variables. Since the only values each variable can take on are the initial value (0), or the assigned value (1), each process produces a 2-bit string. The numbers after *Prints* are the actual outputs that appear on the output device.

If we concatenate the output of P_1, P_2, and P_3 in that order, we get a 6-bit string that characterizes a particular interleaving of statements (and thus memory references). This is the string listed as the *Signature* in Fig. 6-15. Below we will characterize each ordering by its signature rather than by its printout.

Not all 64 signature patterns are allowed. As a trivial example, 000000 is not permitted, because that would imply that the print statements ran before the assignment statements, violating Lamport's requirement that statements are executed in program order. A more subtle example is 001001. The first two bits, 00, mean that b and c were both 0 when P_1 did its printing. This situation occurs only when P_1 executes both statements before P_2 or P_3 starts. The next

two bits, 10, mean that P_2 must run after P_1 has started but before P_3 has started. The last two bits, 01, mean that P_3 must complete before P_1 starts, but we have already seen that P_1 must go first. Therefore, 001001 is not allowed.

In short, the 90 different valid statement orderings produce a variety of different program results (less than 64, though) that are allowed under the assumption of sequential consistency. The contract between the software and memory here is that the software must accept all of these as valid results. In other words, the software must accept the four results shown in Fig. 6-15 and all the other valid results as proper answers, and must work correctly if any of them occurs. A program that works for some of these results and not for others violates the contract with the memory and is incorrect.

A sequentially consistent memory can be implemented on a DSM or multiprocessor system that replicates writable pages by ensuring that no memory operation is started until all the previous ones have been completed. In a system with an efficient, totally-ordered reliable broadcast mechanism, for example, all shared variables could be grouped together on one or more pages, and operations to the shared pages could be broadcast. The exact order in which the operations are interleaved does not matter as long as all processes agree on the order of all operations on the shared memory.

Various formal systems have been proposed for expressing sequential consistency (and other models). Let us briefly consider the system of Ahamad et al. (1993). In their method, the sequence of read and write operations of process i is designated by H_i (the history of P_i). Figure 6-12(b) shows two such sequences, H_1 and H_2 for P_1 and P_2, respectively, as follows:

$$H_1 = W(x)1$$

$$H_2 = R(x)0\ \ R(x)1$$

The set of all such sequences is called H.

To get the relative order in which the operations appear to be executed, we must merge the operation strings in H into a single string, S, in which each operation appearing in H appears in S once. Intuitively, S gives the order that the operations would have been carried out had there been a single centralized memory. All legal values for S must obey two constraints:

1. Program order must be maintained.
2. Memory coherence must be respected.

The first constraint means that if a read or write access, A, appears before another access, B, in one of the strings in H, A must also appear before B in S. If this constraint is true for all pairs of operations, the resulting S will not show any operations in an order that violates any of the programs.

The second constraint, called **memory coherence**, means that a read to some location, x, must always return the value most recently written to x; that is, the value v written by the most recent $W(x)v$ before the $R(x)$. Memory coherence examines in isolation each location and the sequence of operations on it, without regard to other locations. Consistency, in contrast, deals with writes to *different* locations and their ordering.

For Fig. 6-12(b) there is only one legal value of S:

$$S = R(x)0\ \ W(x)1\ \ R(x)1,$$

For more complicated examples there might be several legal values of S. The behavior of a program is said to be correct if its operation sequence corresponds to some legal value of S.

Although sequential consistency is a programmer-friendly model, it has a serious performance problem. Lipton and Sandberg (1988) proved that if the read time is r, the write time is w, and the minimal packet transfer time between nodes is t, then it is always true that $r + w \geq t$. In other words, for any sequentially consistent memory, changing the protocol to improve the read performance makes the write performance worse, and vice versa. For this reason, researchers have investigated other (weaker) models. In the following sections we will discuss some of them.

6.3.3. Causal Consistency

The **causal consistency** model (Hutto and Ahamad, 1990) represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not.

To see what causality is all about, consider an example from daily life (of a computer scientist). During a discussion on the relative merits of different programming languages in one of the USENET newsgroups, some hothead posts the message: "Anybody caught programming in FORTRAN should be shot." Very shortly thereafter, a cooler individual writes: "I am against capital punishment, even for major offenses against good taste." Due to varying delays along the message propagation paths, a third subscriber may get the reply first and become quite confused upon seeing it. The problem here is that causality has been violated. If event B is caused or influenced by an earlier event, A, causality requires that everyone else first see A, then see B.

Now consider a memory example. Suppose that process P_1 writes a variable x. Then P_2 reads x and writes y. Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x read by P_2 (i.e., the value written by P_1). On the other hand, if two processes spontaneously and simultaneously write two variables, these are not causally related. When there is a read followed later by a write, the two

events are potentially causally related. Similarly, a read is causally related to the write that provided the data the read got. Operations that are not causally related are said to be **concurrent**.

For a memory to be considered causally consistent, it is necessary that the memory obey the following condition:

> *Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.*

As an example of causal consistency, consider Fig. 6-16. Here we have an event sequence that is allowed with a causally consistent memory, but which is forbidden with a sequentially consistent memory or a strictly consistent memory. The thing to note is that the writes $W(x)2$ and $W(x)3$ are concurrent, so it is not required that all processes see them in the same order. If the software fails when different processes see concurrent events in a different order, it has violated the memory contract offered by causal memory.

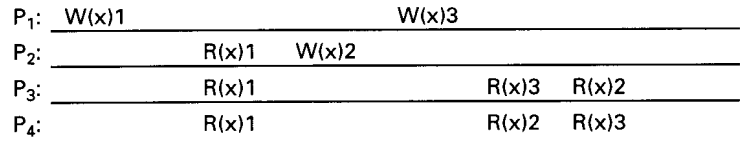


Fig. 6-16. This sequence is allowed with causally consistent memory, but not with sequentially consistent memory or strictly consistent memory.

Now consider a second example. In Fig. 6-17(a) we have $W(x)2$ potentially depending on $W(x)1$ because the 2 may be a result of a computation involving the value read by $R(x)1$. The two writes are causally related, so all processes must see them in the same order. Therefore, Fig. 6-17(a) is incorrect. On the other hand, in Fig. 6-17(b) the read has been removed, so $W(x)1$ and $W(x)2$ are now concurrent writes. Causal memory does not require concurrent writes to be globally ordered, so Fig. 6-17(b) is correct.

Implementing causal consistency requires keeping track of which processes have seen which writes. It effectively means that a dependency graph of which operation is dependent on which other operations must be constructed and maintained. Doing so involves some overhead.

6.3.4. PRAM Consistency and Processor Consistency

In causal consistency, it is permitted that concurrent writes be seen in a different order on different machines, although causally-related ones must be seen in the same order by all machines. The next step in relaxing memory is to drop

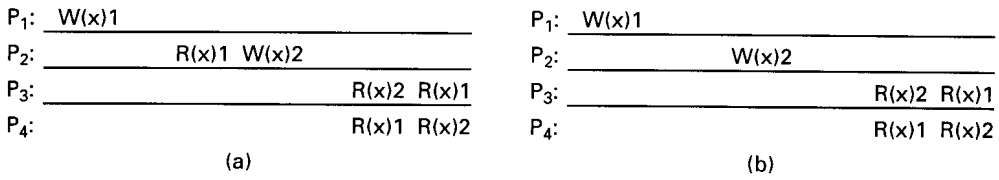


Fig. 6-17. (a) A violation of causal memory. (b) A correct sequence of events in causal memory.

the latter requirement. Doing so gives **PRAM consistency** (Pipelined RAM), which is subject to the condition:

> *Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.*

PRAM consistency is due to Lipton and Sandberg (1988). PRAM stands for Pipelined RAM, because writes by a single process can be pipelined, that is, the process does not have to stall waiting for each one to complete before starting the next one. PRAM consistency is contrasted with causal consistency in Fig. 6-18. The sequence of events shown here is allowed with PRAM consistent memory but not with any of the stronger models we have studied so far.

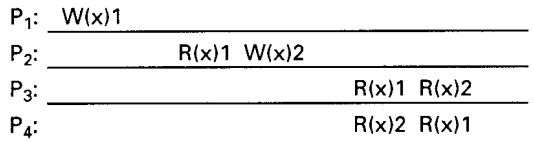


Fig. 6-18. A valid sequence of events for PRAM consistency.

PRAM consistency is interesting because it is easy to implement. In effect it says that there are no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order, as though they were in a pipeline. Put in other terms, in this model all writes generated by different processes are concurrent.

Let us now reconsider the three processes of Fig. 6-14, but this time using PRAM consistency instead of sequential consistency. Under PRAM consistency, different processes may see the statements executed in a different order. For example, Fig. 6-19(a) shows how P_1 might see the events, whereas Fig. 6-19(b) shows how P_2 might see them and Fig. 6-19(c) shows P_3's view. For a sequentially consistent memory, three different views would not be allowed.

If we concatenate the output of the three processes, we get a result of

```
a = 1;                  a = 1;                  b = 1;
* print (b, c);         b = 1;                  print (a, c);
b = 1;                  * print (a, c);         c = 1;
print (a, c);           print (b, c);           * print (a, b);
c = 1;                  c = 1;                  a = 1;
print (a, b);           print (a, b);           print (b, c);

Prints: 00              Prints: 10              Prints: 01

   (a)                     (b)                     (b)
```

Fig. 6-19. Statement execution as seen by three processes. The statements marked with asterisks are the ones that actually generate output.

001001, which, as we saw earlier, is impossible with sequential consistency. The key difference between sequential consistency and PRAM consistency is that with the former, although the order of statement execution (and memory references) is nondeterministic, at least all processes agree what it is. With the latter, they do not agree. Different processes can see the operations in a different order.

Sometimes PRAM consistency can lead to results that may be counterintuitive. The following example, due to Goodman (1989), was devised for a slightly different memory model (discussed below), but also holds for PRAM consistency. In Fig. 6-20 one might naively expect one of three possible outcomes: P_1 is killed, P_2 is killed, or neither is killed (if the two assignments go first). With PRAM consistency, however, both processes can be killed. This result can occur if P_1 reads b before it sees P_2's store into b, and P_2 reads a before it sees P_1's store into a. With a sequentially consistent memory, there are six possible statement interleavings, and none of them results in both processes being killed.

```
a = 1;                          b = 1;
if (b == 0) kill (P2);          if (a == 0) kill (P1);

   (a)                             (b)
```

Fig. 6-20. Two parallel processes. (a) P_1. (b) P_2.

Goodman's (1989) model, called **processor consistency**, is close enough to PRAM consistency that some authors have regarded them as being effectively the same (e.g., Attiya and Friedman, 1992; and Bitar, 1990). However, Goodman gave an example that suggests he intended that there be an additional condition imposed on processor consistent memory, namely memory coherence, as described above: in other words, for every memory location, x, there be global agreement about the order of writes to x. Writes to different locations need not be viewed in the same order by different processes. Gharachorloo et al. (1990) describe using processor consistency in the Dash multiprocessor, but use a

slightly different definition than Goodman. The differences between PRAM and the two processor consistency models are subtle, and are discussed by Ahamad et al. (1993).

6.3.5. Weak Consistency

Although PRAM consistency and processor consistency can give better performance than the stronger models, they are still unnecessarily restrictive for many applications because they require that writes originating in a single process be seen everywhere in order. Not all applications require even seeing all writes, let alone seeing them in order. Consider the case of a process inside a critical section reading and writing some variables in a tight loop. Even though other processes are not supposed to touch the variables until the first process has left its critical section, the memory has no way of knowing when a process is in a critical section and when it is not, so it has to propagate all writes to all memories in the usual way.

A better solution would be to let the process finish its critical section and then make sure that the final results were sent everywhere, not worrying too much whether all intermediate results had also been propagated to all memories in order, or even at all. This can be done by introducing a new kind of variable, a **synchronization variable**, that is used for synchronization purposes. The operations on it are used to synchronize memory. When a synchronization completes, all writes done on that machine are propagated outward and all writes done on other machines are brought in. In other words, all of (shared) memory is synchronized.

Dubois et al. (1986) define this model, called **weak consistency**, by saying that it has three properties:

1. *Accesses to synchronization variables are sequentially consistent.*
2. *No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
3. *No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.*

Point 1 says that all processes see all accesses to synchronization variables in the same order. Effectively, when a synchronization variable is accessed, this fact is broadcast to the world, and no other synchronization variable can be accessed in any other process until this one is finished everywhere.

Point 2 says that accessing a synchronization variable "flushes the pipeline." It forces all writes that are in progress or partially completed or completed

at some memories but not others to complete everywhere. When the synchronization access is done, all previous writes are guaranteed to be done as well. By doing a synchronization after updating shared data, a process can force the new values out to all other memories.

Point 3 says that when ordinary (i.e., not synchronization) variables are accessed, either for reading or writing, all previous synchronizations have been performed. By doing a synchronization before reading shared data, a process can be sure of getting the most recent values.

It is worth mentioning that quite a bit of complexity lurks behind the word "performed" here and elsewhere in the context of DSM. A read is said to have been performed when no subsequent write can affect the value returned. A write is said to have performed at the instant when all subsequent reads return the value written by the write. A synchronization is said to have performed when all shared variables have been updated. One can also distinguish between operations that have performed locally and globally. Dubois et al. (1988) go into this point in detail.

From an implementation standpoint, when the contract between the software and the memory says that memory only has to be brought up to date when a synchronization variable is accessed, a new write can be started before the previous ones have been completed, and in some cases writes can be avoided altogether. Of course, this contract puts a greater burden on the programmer, but the potential gain is better performance. Unlike the previous memory models, it enforces consistency on a group of operations, not on individual reads and writes. This model is most useful when isolated accesses to shared variables are rare, with most coming in clusters (many accesses in a short period, then none for a long time).

```
int a, b, c, d, e, x, y;                /* variables */
int *p, *q;                             /* pointers */
int f(int *p, int *q);                  /* function prototype */

a = x * x;                              /* a is stored in a register */
b = y * y;                              /* b too */
c = a * a * a + b * b + a * b;          /* used later */
d = a * a * c;                          /* used later */
p = &a;                                 /* p gets the address of a */
q = &b;                                 /* q gets the address of b */
e = f(p, q);                            /* function call */
```

Fig. 6-21. A program fragment in which some variables may be kept in registers.

The idea of having memory be wrong is nothing new. Many compilers cheat too. For example, consider the program fragment of Fig. 6-21, with all the variables initialized to appropriate values. An optimizing compiler may decide

to compute a and b in registers and keep the values there for a while, not updating their memory locations. Only when the function f is called does the compiler have to put the current values of a and b back in memory, because f might try to access them.

Having memory be wrong is acceptable here because the compiler knows what it is doing (i.e., because the software does not insist that memory be up-to-date). Clearly, if a second process existed that could read memory in an unconstrained way, this scheme would not work. For example, if during the assignment to d, the second process read a, b, and c, it would get inconsistent values (the old values of a and b, but the new value of c). One could imagine a special way to prevent chaos by having the compiler first write a special flag bit saying that memory was out-of-date. If another process wanted to access a, it would busy wait on the flag bit. In this way one can live with less than perfect consistency, provided that synchronization is done in software and all parties obey the rules.

Now let us consider a somewhat less far-fetched situation. In Fig. 6-22(a) we see that process P_1 does two writes to an ordinary variable, and then synchronizes (indicated by the letter S). If P_2 and P_3 have not yet been synchronized, no guarantees are given about what they see, so this sequence of events is valid.

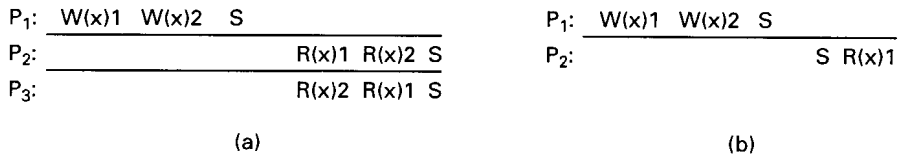


Fig. 6-22. (a) A valid sequence of events for weak consistency. (b) An invalid sequence for weak consistency.

Figure 6-22(b) is different. Here P_2 has been synchronized, which means that its memory is brought up to date. When it reads x, it must get the value 2. Getting 1, as shown in the figure, is not permitted with weak consistency.

6.3.6. Release Consistency

Weak consistency has the problem that when a synchronization variable is accessed, the memory does not know whether this is being done because the process is finished writing the shared variables or about to start reading them. Consequently, it must take the actions required in both cases, namely making sure that all locally initiated writes have been completed (i.e., propagated to all other machines), as well as gathering in all writes from other machines. If the memory could tell the difference between entering a critical region and leaving

one, a more efficient implementation might be possible. To provide this information, two kinds of synchronization variables or operations are needed instead of one.

Release consistency (Gharachorloo et al., 1990) provides these two kinds. **Acquire** accesses are used to tell the memory system that a critical region is about to be entered. **Release** accesses say that a critical region has just been exited. These accesses can be implemented either as ordinary operations on special variables or as special operations. In either case, the programmer is responsible for putting explicit code in the program telling when to do them, for example, by calling library procedures such as *acquire* and *release* or procedures such as *enter_critical_region* and *leave_critical_region*.

It is also possible to use barriers instead of critical regions with release consistency. A **barrier** is a synchronization mechanism that prevents any process from starting phase $n + 1$ of a program until all processes have finished phase n. When a process arrives at a barrier, it must wait until all other processes get there too. When the last one arrives, all shared variables are synchronized and then all processes are resumed. Departure from the barrier is acquire and arrival is release.

In addition to these synchronizing accesses, reading and writing shared variables is also possible. Acquire and release do not have to apply to all of memory. Instead, they may only guard specific shared variables, in which case only those variables are kept consistent. The shared variables that are kept consistent are said to be **protected**.

The contract between the memory and the software says that when the software does an acquire, the memory will make sure that all the local copies of the protected variables are brought up to date to be consistent with the remote ones if need be. When a release is done, protected variables that have been changed are propagated out to other machines. Doing an acquire does not guarantee that locally made changes will be sent to other machines immediately. Similarly, doing a release does not necessarily import changes from other machines.

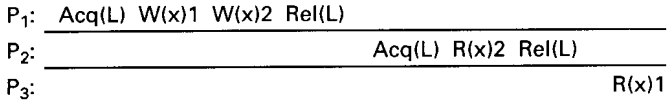


Fig. 6-23. A valid event sequence for release consistency.

Fig. 6-23 depicts a valid sequence of events for release consistency. Process P_1 does an acquire, changes a shared variable twice, and then does a release. Process P_2 does an acquire, and reads x. It is guaranteed to get the value x had at the time of the release, namely 2 (unless P_2's acquire performs before P_1's acquire). If the acquire had been done before P_1 did the release, the acquire

would have been be delayed until the release had occurred. Since P_3 does not do an acquire before reading a shared variable, the memory has no obligation to give it the current value of x, so returning 1 is allowed.

To make release consistency clearer, let us briefly describe a possible simple-minded implementation in the context of distributed shared memory (release consistency was actually invented for the Dash multiprocessor, but the idea is the same, even though the implementation is not). To do an acquire, a process sends a message to a synchronization manager requesting an acquire on a particular lock. In the absence of any competition, the request is granted and the acquire completes. Then an arbitrary sequence of reads and writes to the shared data can take place locally. None of these are propagated to other machines. When the release is done, the modified data are sent to the other machines that use them. After each machine has acknowledged receipt of the data, the synchronization manager is informed of the release. In this way, an arbitrary number of reads and writes on shared variables can be done with a fixed amount of overhead. Acquires and releases on different locks occur independently of one another.

While the centralized algorithm described above will do the job, it is by no means the only approach. In general, a distributed shared memory is release consistent if it obeys the following rules:

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.*
2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.*
3. *The acquire and release accesses must be processor consistent (sequential consistency is not required).*

If all these conditions are met and processes use acquire and release properly (i.e., in acquire-release pairs), the results of any execution will be no different than they would have been on a sequentially consistent memory. In effect, blocks of accesses to shared variables are made atomic by the acquire and release primitives to prevent interleaving.

A different implementation of release consistency is **lazy release consistency** (Keleher et al., 1992). In normal release consistency, which we will henceforth call **eager release consistency**, to distinguish it from the lazy variant, when a release is done, the processor doing the release pushes out all the modified data to all other processors that already have a cached copy and thus might potentially need it. There is no way to tell if they actually will need it, so to be safe, all of them get everything that has changed.

Although pushing all the data out this way is straightforward, it is generally inefficient. In lazy release consistency, at the time of a release, nothing is sent anywhere. Instead, when an acquire is done, the processor trying to do the acquire has to get the most recent values of the variables from the machine or machines holding them. A timestamp protocol can be used to determine which variables have to be transmitted.

In many programs, a critical region is located inside a loop. With eager release consistency, on every pass through the loop a release is done, and all the modified data have to be pushed out to all the processors maintaining copies of them. This algorithm wastes bandwidth and introduces needless delay. With lazy release consistency, at the release nothing is done. At the next acquire, the processor determines that it already has all the data it needs, so no messages are generated here either. The net result is that with lazy release consistency no network traffic is generated at all until another processor does an acquire. Repeated acquire-release pairs done by the same processor in the absence of competition from the outside are free.

6.3.7. Entry Consistency

Another consistency model that has been designed to be used with critical sections is **entry consistency** (Bershad et al., 1993). Like both variants of release consistency, it requires the programmer (or compiler) to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent. Entry consistency differs from lazy release consistency in that the latter does not associate shared variables with locks or barriers and at acquire time has to determine empirically which variables it needs.

Associating with each synchronization variable a list of shared variables reduces the overhead associated with acquiring and releasing a synchronization variable, since only a few shared variables have to be synchronized. It also allows multiple critical sections involving disjoint shared variables to execute simultaneously, increasing the amount of parallelism. The price paid is the extra overhead and complexity of associating every shared data variable with some synchronization variable. Programming this way is also more complicated and error prone.

Synchronization variables are used as follows. Each synchronization variable has a current owner, namely, the process that last acquired it. The owner

may enter and exit critical regions repeatedly without having to send any messages on the network. A process not currently owning a synchronization variable but wanting to acquire it has to send a message to the current owner asking for ownership and the current values of the associated variables. It is also possible for several processes simultaneously to own a synchronization variable in nonexclusive mode, meaning that they can read, but not write, the associated data variables.

Formally, a memory exhibits entry consistency if it meets all the following conditions (Bershad and Zekauskas, 1991):

1. *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*
2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.*
3. *After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.*

The first condition says that when a process does an acquire, the acquire may not complete (i.e., return control to the next statement) until all the guarded shared variables have been brought up to date. In other words, at an acquire, all remote changes to the guarded data must be made visible.

The second condition says that before updating a shared variable, a process must enter a critical region in exclusive mode to make sure that no other process is trying to update it at the same time.

The third condition says that if a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable guarding the critical region to fetch the most recent copies of the guarded shared variables.

6.3.8. Summary of Consistency Models

Although other consistency models have been proposed, the main ones are discussed above. They differ in how restrictive they are, how complex their implementations are, their ease of programming, and their performance. Strict consistency is the most restrictive, but because its implementation in a DSM system is essentially impossible, it is never used.

Sequential consistency is feasible, popular with programmers, and widely used. It has the problem of poor performance, however. The way to get around this result is to relax the consistency model. Some of the possibilities are shown in Fig. 6-24(a), roughly in order of decreasing restrictiveness.

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
Sequential	All processes see all shared accesses in the same order
Causal	All processes see all casually-related shared accesses in the same order
Processor	PRAM consistency + memory coherence
PRAM	All processes see writes from each processor in the order they were issued. Writes from different processors may not always be seen in the same order

(a)

Weak	Shared data can only be counted on to be consistent after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered

(b)

Fig. 6-24. (a) Consistency models not using synchronization operations. (b) Models with synchronization operations.

Causal consistency, processor consistency, and PRAM consistency all represent weakenings in which there is no longer a globally agreed upon view of which operations appeared in which order. Different processes may see different sequences of operations. These three differ in terms of which sequences are allowed and which are not, but in all cases, it is up to the programmer to avoid doing things that work only if the memory is sequentially consistent.

A different approach is to introduce explicit synchronization variables, as weak consistency, release consistency, and entry consistency do. These three are summarized in Fig. 6-24(b). When a process performs an operation on an ordinary shared data variable, no guarantees are given about when they will be visible to other processes. Only when a synchronization variable is accessed are changes propagated. The three models differ in how synchronization works, but in all cases a process can perform multiple reads and writes in a critical section without invoking any data transport. When the critical section has been

completed, the final result is either propagated to the other processes or made ready for propagation should anyone else express interest.

In short, weak consistency, release consistency, and entry consistency require additional programming constructs that, when used as directed, allow programmers to pretend that memory is sequentially consistent, when, in fact, it is not. In principle, these three models using explicit synchronization should be able to offer the best performance, but it is likely that different applications will give quite different results. More research is needed before we can draw any firm conclusions here.

6.4. PAGE-BASED DISTRIBUTED SHARED MEMORY

Having studied the principles behind distributed shared memory systems, let us now turn to these systems themselves. In this section we will study "classical" distributed shared memory, the first of which was IVY (Li 1986; and Li and Hudak 1989). These systems are built on top of multicomputers, that is, processors connected by a specialized message-passing network, workstations on a LAN, or similar designs. The essential element here is that no processor can directly access any other processor's memory. Such systems are sometimes called **NORMA** (**NO Remote Memory Access**) systems to contrast them with NUMA systems.

The big difference between NUMA and NORMA is that in the former, every processor can directly reference every word in the global address space just by reading or writing it. Pages can be randomly distributed among memories without affecting the results that programs give. When a processor references a remote page, the system has the option of fetching it or using it remotely. The decision affects the performance, but not the correctness. NUMA machines are true multiprocessors—the hardware allows every processor to reference every word in the address space without software intervention.

Workstations on a LAN are fundamentally different from a multiprocessor. Processors can only reference their own local memory. There is no concept of a global shared memory, as there is with a NUMA or UMA multiprocessor. The goal of the DSM work, however, is to add software to the system to allow a multicomputer to run multiprocessor programs. Consequently, when a processor references a remote page, that page *must* be fetched. There is no choice as there is in the NUMA case.

Much of the early research on DSM systems was devoted to the question of how to run existing multiprocessor programs on multicomputers. Sometimes this is referred to as the "dusty deck" problem. The idea is to breathe new life into old programs just by running them on new (DSM) systems. The concept is

especially attractive for applications that need all the CPU cycles they can get and whose authors are thus interested in using large-scale multicomputers rather than small-scale multiprocessors.

Since programs written for multiprocessors normally assume that memory is sequentially consistent, the initial work on DSM was carefully done to provide sequentially consistent memory, so that old multiprocessor programs could work without modification. Subsequent experience has shown that major performance gains can be had by relaxing the memory model, at the cost of reprogramming existing applications and writing new ones in a different style. We will come back to this point later, but first we will look at the major design issues in classical DSM systems of the IVY type.

6.4.1. Basic Design

The idea behind DSM is simple: try to emulate the cache of a multiprocessor using the MMU and operating system software. In a DSM system, the address space is divided up into chunks, with the chunks being spread over all the processors in the system. When a processor references an address that is not local, a trap occurs, and the DSM software fetches the chunk containing the address and restarts the faulting instruction, which now completes successfully. This concept is illustrated in Fig. 6-25(a) for an address space with 16 chunks and four processors, each capable of holding four chunks.

In this example, if processor 1 references instructions or data in chunks 0, 2, 5, or 9, the references are done locally. References to other chunks cause traps. For example, a reference to an address in chunk 10 will cause a trap to the DSM software, which then moves chunk 10 from machine 2 to machine 1, as shown in Fig. 6-25(b).

6.4.2. Replication

One improvement to the basic system that can improve performance considerably is to replicate chunks that are read only, for example, program text, read-only constants, or other read-only data structures. For example, if chunk 10 in Fig. 6-25 is a section of program text, its use by processor 1 can result in a copy being sent to processor 1, without the original in processor 2's memory being disturbed, as shown in Fig. 6-25(c). In this way, processors 1 and 2 can both reference chunk 10 as often as needed without causing traps to fetch missing memory.

Another possibility is to replicate not only read-only chunks, but all chunks. As long as reads are being done, there is effectively no difference between

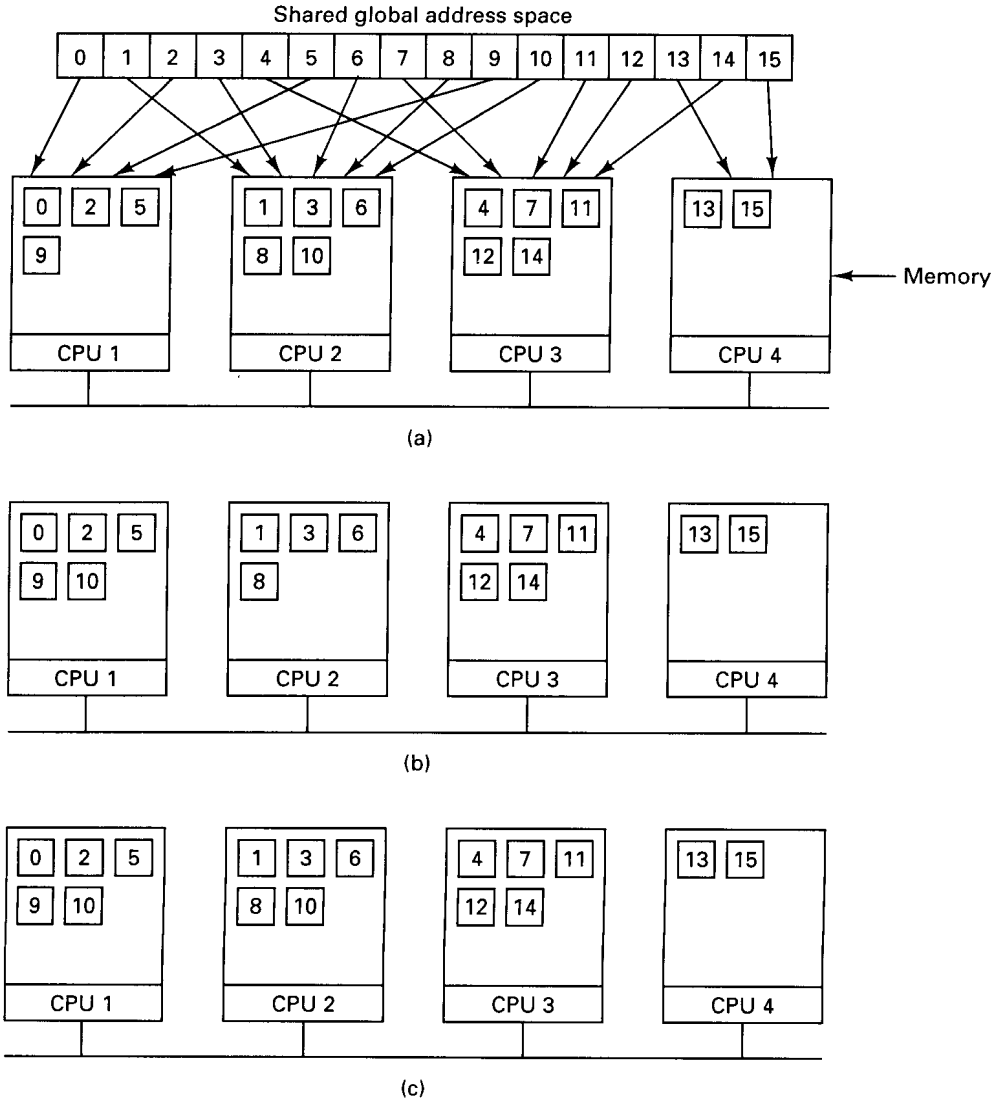


Fig. 6-25. (a) Chunks of address space distributed among four machines. (b) Situation after CPU 1 references chunk 10. (c) Situation if chunk 10 is read only and replication is used.

replicating a read-only chunk and replicating a read-write chunk. However, if a replicated chunk is suddenly modified, special action has to be taken to prevent having multiple, inconsistent copies in existence. How inconsistency is prevented will be discussed in the following sections.

6.4.3. Granularity

DSM systems are similar to multiprocessors in certain key ways. In both systems, when a nonlocal memory word is referenced, a chunk of memory containing the word is fetched from its current location and put on the machine making the reference (main memory or cache, respectively). An important design issue is how big should the chunk be? Possibilities are the word, block (a few words), page, or segment (multiple pages).

With a multiprocessor, fetching a single word or a few dozen bytes is feasible because the MMU knows exactly which address was referenced and the time to set up a bus transfer is measured in nanoseconds. Memnet, although not strictly a multiprocessor, also uses a small chunk size (32 bytes). With DSM systems, such fine granularity is difficult or impossible, due to the way the MMU works.

When a process references a word that is absent, it causes a page fault. An obvious choice is to bring in the entire page that is needed. Furthermore, integrating DSM with virtual memory makes the total design simpler, since the same unit, the page, is used for both. On a page fault, the missing page is just brought in from another machine instead of from the disk, so much of the page fault handling code is the same as in the traditional case.

However, another possible choice is to bring in a larger unit, say a region of 2, 4, or 8 pages, including the needed page. In effect, doing this simulates a larger page size. There are advantages and disadvantages to a larger chunk size for DSM. The biggest advantage is that because the startup time for a network transfer is substantial, it does not take much longer to transfer 1024 bytes than it does to transfer 512 bytes. By transferring data in large units, when a large piece of address space has to be moved, the number of transfers may often be reduced. This property is especially important because many programs exhibit locality of reference, meaning that if a program has referenced one word on a page, it is likely to reference other words on the same page in the immediate future.

On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processes. Also, too large an effective page size introduces a new problem, called **false sharing**, illustrated in Fig. 6-26. Here we have a page containing two unrelated shared variables, *A* and *B*. Processor 1 makes heavy use of *A*, reading and writing it. Similarly, process 2 uses *B*. Under these circumstances, the page containing both variables will constantly be traveling back and forth between the two machines.

The problem here is that although the variables are unrelated, since they appear by accident on the same page, when a process uses one of them, it also gets the other. The larger the effective page size, the more often false sharing will occur, and conversely, the smaller the effective page size, the less often it

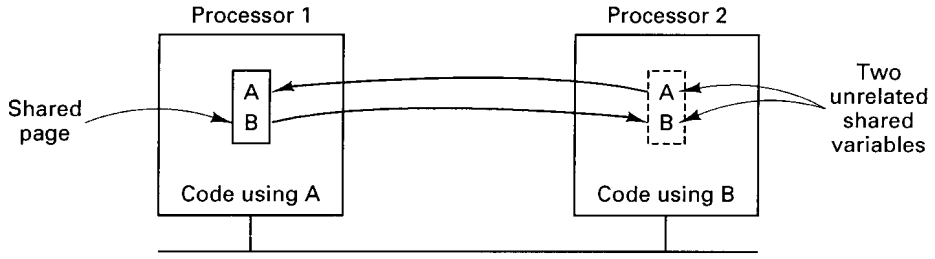


Fig. 6-26. False sharing of a page containing two unrelated variables.

will occur. Nothing analogous to this phenomenon is present in ordinary virtual memory systems.

Clever compilers that understand the problem and place variables in the address space accordingly can help reduce false sharing and improve performance. However, saying this is easier than doing it. Furthermore, if the false sharing consists of processor 1 using one element of an array and processor 2 using a different element of the same array, there is little that even a clever compiler can do to eliminate the problem.

6.4.4. Achieving Sequential Consistency

If pages are not replicated, achieving consistency is not an issue. There is exactly one copy of each page, and it is moved back and forth dynamically as needed. With only one copy of each page, there is no danger that different copies will have different values.

If read-only pages are replicated, there is also no problem. The read-only pages are never changed, so all the copies are always identical. Only a single copy is kept of each read-write page, so inconsistencies are impossible here, too.

The interesting case is that of replicated read-write pages. In many DSM systems, when a process tries to read a remote page, a local copy is made because the system does not know what is on the page or whether it is writable. Both the local copy (in fact, all copies) and the original page are set up in their respective MMUs as read only. As long as all references are reads, everything is fine.

However, if any process attempts to write on a replicated page, a potential consistency problem arises because changing one copy and leaving the others alone is unacceptable. This situation is analogous to what happens in a multiprocessor when one processor attempts to modify a word that is present in multiple caches, so let us review what multiprocessors do under these circumstances.

In general, multiprocessors take one of two approaches: update or invalidation. With update, the write is allowed to take place locally, but the address of

the modified word and its new value are broadcast on the bus simultaneously to all the other caches. Each of the caches holding the word being updated sees that an address it is caching is being modified, so it copies the new value from the bus to its cache, overwriting the old value. The final result is that all caches that held the word before the update also hold it afterward, and all acquire the new value.

The other approach multiprocessors can take is invalidation. When this strategy is used, the address of the word being updated is broadcast on the bus, but the new value is not. When a cache sees that one of its words is being updated, it invalidates the cache block containing the word, effectively removing it from the cache. The final result with invalidation is that only one cache now holds the modified word, so consistency problems are avoided. If one of the processors that now holds an invalid copy of the cache block tries to use it, it will get a cache miss and fetch the block from the one processor holding a valid copy.

Whereas these two strategies are approximately equally easy to implement in a multiprocessor, they differ radically in a DSM system. Unlike in a multiprocessor, where the MMU knows which word is to be written and what the new value is, in a DSM system the software does not know which word is to be written or what the new value will be. To find out, it could make a secret copy of the page about to be changed (the page number is known), make the page writable, set the hardware trap bit, which forces a trap after every instruction, and restart the faulting process. One instruction later, it catches the trap and compares the current page with the secret copy it just made, to see which word has been changed. It could then broadcast a short packet giving the address and new value on the network. The processors receiving this packet could then check to see if they have the page in question, and if so, update it.

The amount of work here is enormous, but worse yet, the scheme is not foolproof. If several updates, originating on different processors, take place simultaneously, different processors may see them in a different order, so the memory will not be sequentially consistent. In a multiprocessor this problem does not occur because broadcasts on the bus are totally reliable (no lost messages), and the order is unambiguous.

Another issue is that a process may make thousands of consecutive writes to the same page because many programs exhibit locality of reference. Having to catch all these updates and pass them to remote machines is horrendously expensive in the absence of multiprocessor-type snooping.

For these reasons, page-based DSM systems typically use an invalidation protocol instead of an update protocol. Various protocols are possible. Below we will describe a typical example, in which all pages are potentially writable (i.e., the DSM software does not know what is on which page).

In this protocol, at any instant of time, each page is either in *R* (readable) or *W* (readable and writable) state. The state a page is in may change as execution

progresses. Each page has an owner, namely the process that most recently wrote on the page. When a page is in *W* state, only one copy exists, mapped into the owner's address space in read-write mode. When a page is in *R* state, the owner has a copy (mapped read only), but other processes may have copies, too.

Six cases can be distinguished, as shown in Fig. 6-27. In all the examples in the figure, process *P* on processor 1 wants to read or write a page. The cases differ in terms of whether *P* is the owner, whether *P* has a copy, whether other processes have copies, and what the state of the page is, as shown.

Let us now consider the actions taken in each of the cases. In the first four cases of Fig. 6-27(a), *P* just does the read. In all four cases the page is mapped into its address space, so the read is done in hardware. No trap occurs. In the fifth and sixth cases, the page is not mapped in, so a page fault occurs and the DSM software gets control. It sends a message to the owner asking for a copy. When the copy comes back, the page is mapped in and the faulting instruction is restarted. If the owner had the page in *W* state, it must degrade to *R* state, but may keep the page. In this protocol, the other process keeps ownership, but in a slightly different protocol that could be transferred as well.

Writes are handled differently, as depicted in Fig. 6-27(b). In the first case, the write just happens, without a trap, since the page is mapped in read-write mode. In the second case (no other copies), the page is changed to *W* state and written. In the third case there are other copies, so they must first be invalidated before the write can take place.

In the next three cases, some other process is the owner at the time *P* does the write. In all three cases, *P* must ask the current owner to invalidate any existing copies, pass ownership to *P*, and send a copy of the page unless *P* already has a copy. Only then may the write take place. In all three cases, *P* ends up with the only copy of the page, which is in *W* state.

In all six cases, before a write is performed the protocol guarantees that only one copy of the page exists, namely in the address space of the process about to do the write. In this way, consistency is maintained.

6.4.5. Finding the Owner

We glossed over a few points in the description above. One of them is how to find the owner of the page. The simplest solution is by doing a broadcast, asking for the owner of the specified page to respond. Once the owner has been located this way, the protocol can proceed as above.

An obvious optimization is not just to ask who the owner is, but also to tell whether the sender wants to read or write and say whether it needs a copy of the page. The owner can then send a single message transferring ownership and the page as well, if needed.

Broadcasting has the disadvantage of interrupting each processor, forcing it

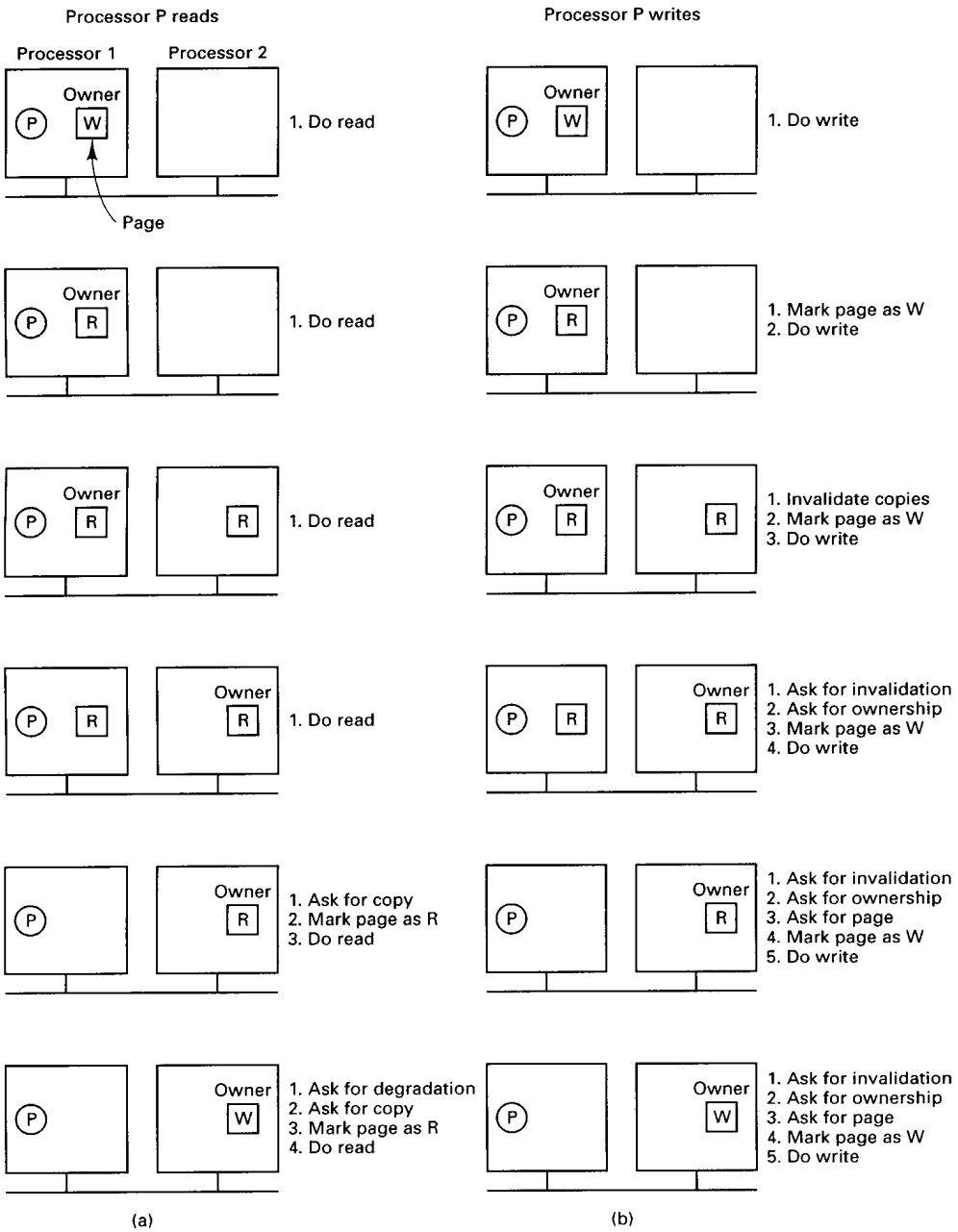


Fig. 6-27. (a) Process *P* wants to read a page. (b) Process *P* wants to write a page.

to inspect the request packet. For all the processors except the owner's, handling the interrupt is essentially wasted time. Broadcasting may use up considerable network bandwidth, depending on the hardware.

Li and Hudak (1989) describe several other possibilities as well. In the first of these, one process is designated as the page manager. It is the job of the manager to keep track of who owns each page. When a process, *P*, wants to read a page it does not have or wants to write a page it does not own, it sends a message to the page manager telling which operation it wants to perform and on which page. The manager then sends back a message telling who the owner is. *P* now contacts the owner to get the page and/or the ownership, as required. Four messages are needed for this protocol, as illustrated in Fig. 6-28(a).

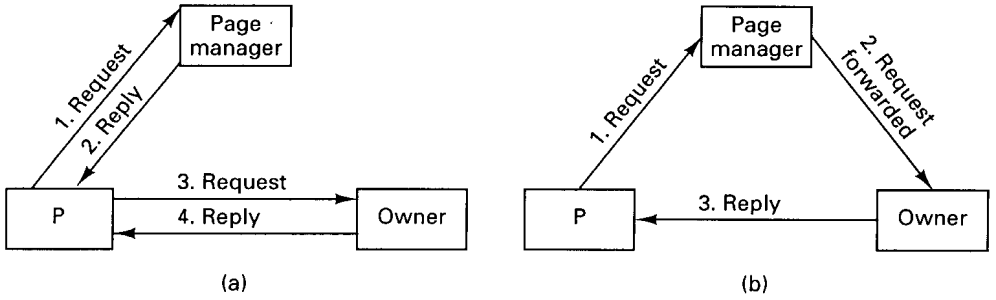


Fig. 6-28. Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol.

An optimization of this ownership location protocol is shown in Fig. 6-28(b). Here the page manager forwards the request directly to the owner, which then replies directly back to *P*, saving one message.

A problem with this protocol is the potentially heavy load on the page manager, handling all the incoming requests. This problem can be reduced by having multiple page managers instead of just one. Splitting the work over multiple managers introduces a new problem, however—finding the right manager. A simple solution is to use the low-order bits of the page number as an index into a table of managers. Thus with eight page managers, all pages that end with 000 are handled by manager 0, all pages that end with 001 are handled by manager 1, and so on. A different mapping, for example by using a hash function, is also possible. The page manager uses the incoming requests not only to provide replies but also to keep track of changes in ownership. When a process says that it wants to write on a page, the manager records that process as the new owner.

Still another possible algorithm is having each process (or more likely, each processor) keep track of the probable owner of each page. Requests for ownership are sent to the probable owner, which forwards them if ownership has

changed. If ownership has changed several times, the request message will also have to be forwarded several times. At the start of execution and every n times ownership changes, the location of the new owner should be broadcast, to allow all processors to update their tables of probable owners.

The problem of locating the manager also is present in multiprocessors, such as Dash, and also in Memnet. In both of these systems it is solved by dividing the address space into regions and assigning each region to a fixed manager, essentially the same technique as the multiple-manager solution discussed above, but using the high-order bits of the page number as the manager number.

6.4.6. Finding the Copies

Another important detail is how all the copies are found when they must be invalidated. Again, two possibilities present themselves. The first is to broadcast a message giving the page number and ask all processors holding the page to invalidate it. This approach works only if broadcast messages are totally reliable and can never be lost.

The second possibility is to have the owner or page manager maintain a list or **copyset** telling which processors hold which pages, as depicted in Fig. 6-29. Here page 4, for example, is owned by a process on CPU 1, as indicated by the double box around the 4. The copyset consists of 2 and 4, because copies of page 4 can be found on those machines.

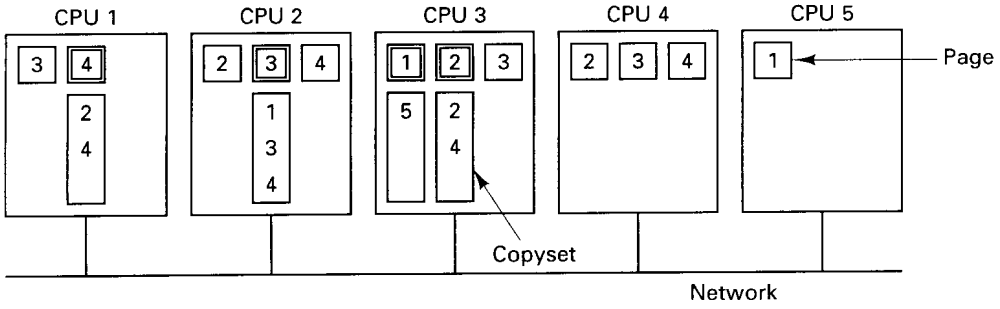


Fig. 6-29. The owner of each page maintains a copyset telling which other CPUs are sharing that page. Page ownership is indicated by the double boxes.

When a page must be invalidated, the old owner, new owner, or page manager sends a message to each processor holding the page and waits for an acknowledgement. When each message has been acknowledged, the invalidation is complete.

Dash and Memnet also need to invalidate pages when a new writer suddenly appears, but they do it differently. Dash uses directories. The writing process

sends a packet to the directory (the page manager in our terminology), which then finds all the copies from its bit map, sends each one an invalidation packet, and collects all the acknowledgements. Memnet fetches the needed page and invalidates all copies by broadcasting an invalidation packet on the ring. The first processor having a copy puts it in the packet and sets a header bit saying it is there. Subsequent processors just invalidate their copies. When the packet comes around the ring and arrives back at the sender, the needed data are present and all other copies are gone. In effect, Memnet implements DSM in hardware.

6.4.7. Page Replacement

In a DSM system, as in any system using virtual memory, it can happen that a page is needed but that there is no free page frame in memory to hold it. When this situation occurs, a page must be evicted from memory to make room for the needed page. Two subproblems immediately arise: which page to evict and where to put it.

To a large extent, the choice of which page to evict can be made using traditional virtual memory algorithms, such as some approximation to the least recently used (LRU) algorithm. One complication that occurs with DSM is that pages can be invalidated spontaneously (due to the activities of other processes), which affects the possible choices. However, by maintaining the estimated LRU order of only those pages that are currently valid, any of the traditional algorithms can be used.

As with conventional algorithms, it is worth keeping track of which pages are "clean" and which are "dirty." In the context of DSM, a replicated page that another process owns is always a prime candidate to evict because it is known that another copy exists. Consequently, the page does not have to be saved anywhere. If a directory scheme is being used to keep track of copies, the owner or page manager must be informed of this decision, however. If pages are located by broadcasting, the page can just be discarded.

The second best choice is a replicated page that the evicting process owns. It is sufficient to pass ownership to one of the other copies by informing that process, the page manager, or both, depending on the implementation. The page itself need not be transferred, which results in a smaller message.

If no replicated pages are suitable candidates, a nonreplicated page must be chosen, for example, the least recently used valid page. There are two possibilities for getting rid of it. The first is to write it to a disk, if present. The other is to hand it off to another processor.

Choosing a processor to hand a page off to can be done in several ways. For example, each page could be assigned a home machine, which must accept it, although this probably implies reserving a large amount of normally wasted space to hold pages that might be sent home some day. Alternatively, the

number of free page frames could be piggybacked on each message sent, with each processor building up an idea of how free memory was distributed around the network. An occasional broadcast message giving the exact count of free page frames could help keep these numbers up to date.

As an aside, note that a conflict may exist between choosing a replicated page (which may just be discarded) and choosing a page that has not been referenced in a long time (which may be the only copy). The same problem exists in traditional virtual memory systems, however, so the same compromises and heuristics apply.

One problem that is unique to DSM systems is the network traffic generated when processes on different machines are actively sharing a writable page, either through false sharing or true sharing. An ad hoc way to reduce this traffic is to enforce a rule that once a page has arrived at any processor, it must remain there for some time ΔT. If requests for it come in from other machines, these requests are simply queued until the timer expires, thus allowing the local process to make many memory references without interference.

As usual, it is instructive to see how page replacement is handled in multiprocessors. In Dash, when a cache fills up, the option always exists of writing the block back to main memory. In DSM systems, that possibility does not exist, although using a disk as the ultimate repository for pages nobody wants is often feasible. In Memnet, every cache block has a home machine, which is required to reserve storage for it. This design is also possible in a DSM system, although it is wasteful in both Memnet and DSM.

6.4.8. Synchronization

In a DSM system, as in a multiprocessor, processes often need to synchronize their actions. A common example is mutual exclusion, in which only one process at a time may execute a certain part of the code. In a multiprocessor, the TEST-AND-SET-LOCK (TSL) instruction is often used to implement mutual exclusion. In normal use, a variable is set to 0 when no process is in the critical section and to 1 when one process is. The TSL instruction reads out the variable and sets it to 1 in a single, atomic operation. If the value read is 1, the process just keeps repeating the TSL instruction until the process in the critical region has exited and set the variable to 0.

In a DSM system, this code is still correct, but is a potential performance disaster. If one process, A, is inside the critical region and another process, B, (on a different machine) wants to enter it, B will sit in a tight loop testing the variable, waiting for it to go to zero. The page containing the variable will remain on B's machine. When A exits the critical region and tries to write 0 to the variable, it will get a page fault and pull in the page containing the variable.

Immediately thereafter, *B* will also get a page fault, pulling the page back. This performance is acceptable.

The problem occurs when several other processes are also trying to enter the critical region. Remember that the TSL instruction modifies memory (by writing a 1 to the synchronization variable) every time it is executed. Thus every time one process executes a TSL instruction, it must fetch the entire page containing the synchronization variable from whoever has it. With multiple processes each issuing a TSL instruction every few hundred nanoseconds, the network traffic can become intolerable.

For this reason, an additional mechanism is often needed for synchronization. One possibility is a synchronization manager (or managers) that accept messages asking to enter and leave critical regions, lock and unlock variables, and so on, sending back replies when the work is done. When a region cannot be entered or a variable cannot be locked, no reply is sent back immediately, causing the sender to block. When the region becomes available or the variable can be locked, a message is sent back. In this way, synchronization can be done with a minimum of network traffic, but at the expense of centralizing control per lock.

6.5. SHARED-VARIABLE DISTRIBUTED SHARED MEMORY

Page-based DSM takes a normal linear address space and allows the pages to migrate dynamically over the network on demand. Processes can access all of memory using normal read and write instructions and are not aware of when page faults or network transfers occur. Accesses to remote data are detected and protected by the MMU.

A more structured approach is to share only certain variables and data structures that are needed by more than one process. In this way, the problem changes from how to do paging over the network to how to maintain a potentially replicated, distributed data base consisting of the shared variables. Different techniques are applicable here, and these often lead to major performance improvements.

The first question that must be addressed is whether or not shared variables will be replicated, and if so, whether fully or partially. If they are replicated, there is more potential for using an update algorithm rather than on a page-based DSM system, provided that writes to individual shared variables can be isolated.

Using shared variables that are individually managed also provides considerable opportunity to eliminate false sharing. If it is possible to update one variable without affecting other variables, then the physical layout of the variables on the pages is less important. Two of the most interesting examples of such systems are Munin and Midway, which are described below.

6.5.1. Munin

Munin is a DSM system that is fundamentally based on software objects, but which can place each object on a separate page so the hardware MMU can be used for detecting accesses to shared objects (Bennett et al., 1990; and Carter et al., 1991, 1993). The basic model used by Munin is that of multiple processors, each with a paged linear address space in which one or more threads are running a slightly modified multiprocessor program. The goal of the Munin project is to take existing multiprocessor programs, make minor changes to them, and have them run efficiently on multicomputer systems using a form of DSM. Good performance is achieved by a variety of techniques to be described below, including the use of release consistency instead of sequential consistency.

The changes consist of annotating the declarations of the shared variables with the keyword *shared*, so that the compiler can recognize them. Information about the expected usage pattern can also be supplied, to permit certain important special cases to be recognized and optimized. By default, the compiler puts each shared variable on a separate page, although large shared variables, such as arrays, may occupy multiple pages. It is also possible for the programmer to specify that multiple shared variables of the same Munin type be put in the same page. Mixing types does not work since the consistency protocol used for a page depends on the type of variables on it.

To run the compiled program, a root process is started up on one of the processors. This process may generate new processes on other processors, which then run in parallel with the main one and communicate with it and with each other by using the shared variables, as normal multiprocessor programs do. Once started on a particular processor, a process does not move.

Accesses to shared variables are done using the CPU's normal read and write instructions. No special protected methods are used. If an attempt is made to use a shared variable that is not present, a page fault occurs, and the Munin system gets control.

Synchronization for mutual exclusion is handled in a special way and is closely related to the memory consistency model. Lock variables may be declared, and library procedures are provided for locking and unlocking them. Barriers, condition variables, and other synchronization variables are also supported.

Release Consistency

Munin is based on a software implementation of (eager) release consistency. For the theoretical baggage, see the paper by Gharachorloo et al. (1990). What Munin does is to provide the tools for users to structure their programs around critical regions, defined dynamically by acquire (entry) and release (exit) calls.

Writes to shared variables must occur inside critical regions; reads can occur inside or outside. While a process is active inside a critical region, the system gives no guarantees about the consistency of shared variables, but when a critical region is exited, the shared variables modified since the last release are brought up to date on all machines. For programs that obey this programming model, the distributed shared memory acts like it is sequentially consistent.

Munin distinguishes three classes of variables:

1. Ordinary variables.
2. Shared data variables.
3. Synchronization variables.

Ordinary variables are not shared and can be read and written only by the process that created them. Shared data variables are visible to multiple processes and appear sequentially consistent, provided that all processes use them only in critical regions. They must be declared as such, but are accessed using normal read and write instructions. Synchronization variables, such as locks and barriers, are special, and are only accessible via system-supplied access procedures, such as *lock* and *unlock* for locks and *increment* and *wait* for barriers. It is these procedures that make the distributed shared memory work.

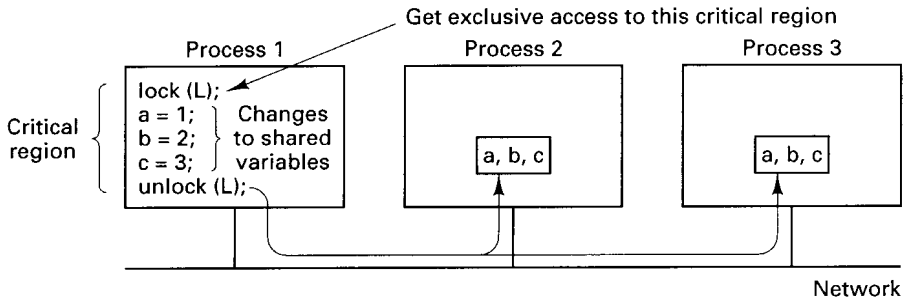


Fig. 6-30. Release consistency in Munin.

The basic operation of Munin's release consistency is illustrated in Fig. 6-30 for three cooperating processes, each running on a different machine. At a certain moment, process 1 wants to enter a critical region of code protected by the lock L (all critical regions must be protected by some synchronization variable). The *lock* statement makes sure that no other well-behaved process is currently executing this critical region. Then the three shared variables, a, b, and c, are accessed using normal machine instructions. Finally, *unlock* is called and the results are propagated to all other machines which maintain copies of a, b, or c. These changes are packed into a minimal number of messages. Accesses to

these variables on other machines while process 1 is still inside its critical region produce undefined results.

Multiple Protocols

In addition to using release consistency, Munin also uses other techniques for improving performance. Chief among these is allowing the programmer to annotate shared variable declarations by classifying each one into one of four categories, as follows:

1. Read-only.
2. Migratory.
3. Write-shared
4. Conventional.

Originally, Munin supported some other categories as well, but experience showed them to be of only marginal value, so they were dropped. Each machine maintains a directory listing each variable, telling, among other things, which category it belongs to. For each category, a different protocol is used.

Read-only variables are easiest. When a reference to a read-only variable causes a page fault, Munin looks up the variable in the variable directory, finds out who owns it, and asks the owner for a copy of the required page. Since pages containing read-only variables do not change (after they have been initialized), consistency problems do not arise. Read-only variables are protected by the MMU hardware. An attempt to write to one causes a fatal error.

Migratory shared variables use the acquire/release protocol illustrated with locks in Fig. 6-30. They are used inside critical regions and must be protected by synchronization variables. The idea is that these variables migrate from machine to machine as critical regions are entered and exited. They are not replicated.

To use a migratory shared variable, its lock must first be acquired. When the variable is read, a copy of its page is made on the machine referencing it and the original copy is deleted. As an optimization, a migratory shared variable can be associated with a lock, so when the lock is sent, the data are sent along with it, eliminating extra messages.

A write-shared variable is used when the programmer has indicated that it is safe for two or more processes to write on it at the same time, for example, an array in which different processes can concurrently access different subarrays. Initially, pages holding write-shared variables are marked as being read only, potentially on several machines at the same time. When a write occurs, the fault handler makes a copy of the page, called the **twin**, marks the page as dirty, and

sets the MMU to allow subsequent writes. These steps are illustrated in Fig. 6-31 for a word that is initially 6 and then changed to 8.

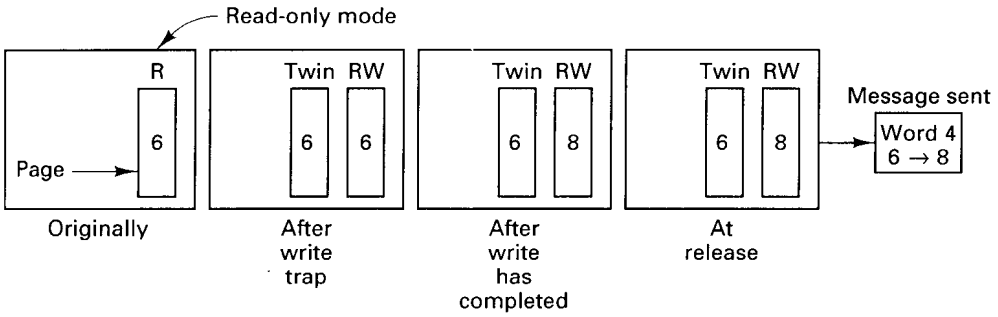


Fig. 6-31. Use of twin pages in Munin.

When the release is done, Munin runs a word-by-word comparison of each dirty write-shared page with its twin, and sends the differences (along with all the migratory pages) to all processes needing them. It then resets the page protection to read only.

When a list of differences comes into a process, the receiver checks each page to see if it has modified the page, too. If a page has not been modified, the incoming changes are accepted. If, however, a page has been modified locally, the local copy, its twin, and the corresponding incoming page are compared word by word. If the local word has been modified but the incoming one has not been, the incoming word overwrites the local one. If both the local and incoming words have been modified, a runtime error is signaled. If no such conflicts exist, the merged page replaces the local one and execution continues.

Shared variables that are not annotated as belonging to one of the above categories are treated as in conventional page-based DSM systems: only one copy of each writable page is permitted, and it is moved from process to process on demand. Read-only pages are replicated as needed.

Let us now look at an example of how the multiwriter protocol is used. Consider the programs of Fig. 6-32(a) and (b). Here, two processes are each incrementing the elements of the same array. Process 1 increments the even elements using function f and process 2 increments the odd elements using function g. Before starting this phase, each process blocks at a barrier until the other one gets there, too. After finishing this phase, they block at another barrier until both are done. Then they both continue with the rest of the program. Parallel programs for quicksort and fast Fourier transforms exhibit this kind of behavior.

With pure sequentially consistent memory both processes pause at the barrier as shown in Fig. 6-32(c). The barrier can be implemented by having each

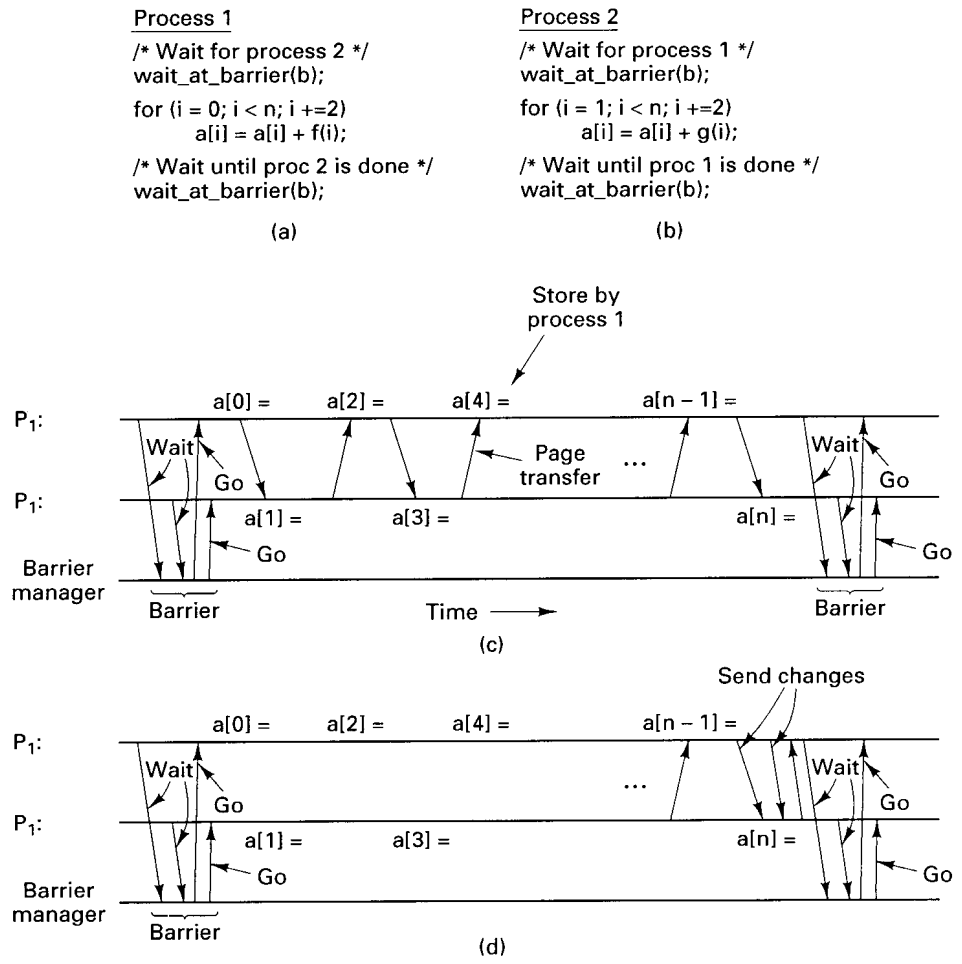


Fig. 6-32. (a) A program using a. (b) Another program using a. (c) Messages sent for sequentially consistent memory. (d) Messages sent for release consistent memory.

process send a message to a barrier manager and block until the reply arrived. The barrier manager does not send any replies until all processes have arrived at the barrier.

After passing the barrier, process 1 might start off, storing into $a[0]$. Then process 2 might try to store into $a[1]$, causing a page fault to fetch the page containing the array. After that, process 1 might try to store into $a[2]$, causing another fault, and so on. With a little bad luck, each of the stores might require a full page to be transferred, generating a great deal of traffic.

With release consistency, the situation is illustrated in Fig. 6-32(d). Again, both processes first pass the barrier. The first store into $a[0]$ forces a twin page to be created for process 1. Similarly, the first store into $a[1]$ forces a twin page to be created for process 2. No page transfers between machines are required at this point. Thereafter, each process can store into its private copy of a at will, without causing any page faults.

When each process arrives at the second barrier statement, the differences between its current values of a and the original values (stored on the twin pages) are computed. These are sent to all the other processes known to be interested in the pages affected. These processes, in turn, may pass them on to other interested processes unknown to the source of the changes. Each receiving process merges the changes with its own version. Conflicts result in a runtime error.

After a process has reported the changes in this way, it sends a message to the barrier manager and waits for a reply. When all processes have sent out their updates and arrived at the barrier, the barrier manager sends out the replies, and everyone can continue. In this manner, page traffic is needed only when arriving at a barrier.

Directories

Munin uses directories to locate pages containing shared variables. When a fault occurs on a reference to a shared variable, Munin hashes the virtual address that caused the fault to find the variable's entry in the shared variable directory. From the entry, it sees which category the variable is, whether a local copy is present, and who the probable owner is. Write-shared pages do not necessarily have a single owner. For a conventional shared variable, it is the last process to acquire write access. For a migratory shared variable, the owner is the process currently holding it.

The location of the probable owner is kept track of using the following algorithm. When a Munin process starts up, the root process owns all the shared variables. When process P_1 later references a shared variable it gets a fault, which generates a message to the root asking for it. The root gives it the page it wants, and notes that P_1 is now the owner. If P_2 asks for the page, the root tells it that P_1 is probably the owner. When P_2 asks P_1 for the variable, it gets it. If P_2 wants to write or the page is migratory, P_2 becomes the new owner and P_1 records this fact. The state of the probable owners at this point in time is shown in Fig. 6-33(a) for a writable or migratory variable.

Now suppose that P_3 and P_4 successively ask for the page. They, too, each follow the chain, resulting in Fig. 6-33(b). Now P_1 needs the variable again. Since it thinks P_2 has it, it sends P_2 a message, only to learn that P_2 thinks that

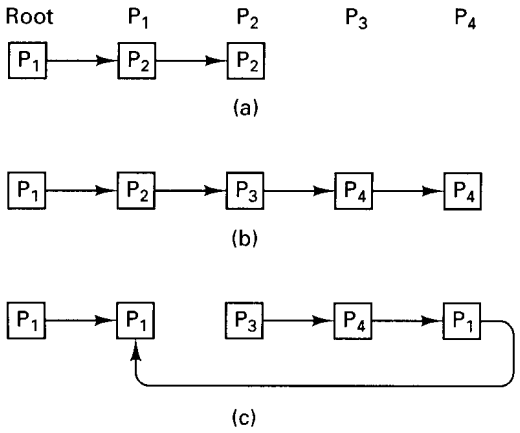


Fig. 6-33. At each point in time, a process can think another process is the probable owner of some page.

P_3 is the owner. After following the chain, P_1 gets the page and the chain looks like Fig. 6-33(c). In this way, every process eventually has an idea of who the probable owner might be, and can follow the chain all the way to find the actual owner.

The directories are also used to keep track of the copysets. However, the copysets need not be perfectly consistent. For example, suppose that P_1 and P_2 are each holding some write-shared variable and each of them knows about the other one. Then P_3 asks the owner, P_1, for a copy and gets it. P_3 records P_1 as having a copy, but does not tell P_2. Later, P_4, which thinks P_2 is the owner, acquires a copy, which updates P_2's copyset to include P_4. At this point no one process has a complete list of who has the page.

Nevertheless, it is possible to maintain consistency. Imagine that P_4 now releases a lock, so it sends the updates to P_2. The acknowledgement message from P_2 to P_4 contains a note saying that P_1 also has a copy. When P_4 contacts P_1 it hears about P_3. In this way, it eventually discovers the entire copyset, so all copies can be updated and it can update its own copyset.

To reduce the overhead of having to send updates to processes that are no longer interested in particular write-shared pages, a timer-based algorithm is used. If a process holds a page, does not reference it within a certain time interval and receives an update, it drops the page. The next time it receives an update for the dropped page, the process tells the updating process that it no longer has a copy, so the updater can reduce the size of its copyset. The probable owner chain is used to denote the copy of last resort, which cannot be dropped without finding a new owner or writing it to disk. This mechanism ensures that a page cannot be dropped by all processes and thus lost.

Synchronization

Munin maintains a second directory for synchronization variables. These are located in a way analogous to the way ordinary shared variables are located. Conceptually, locks act like they are centralized, but in fact a distributed implementation is used to avoid sending too much traffic to any one machine.

When a process wants to acquire a lock, it first checks to see if it owns the lock itself. If it does and the lock is free, the request is granted. If the lock is not local, it is located using the synchronization directory, which keeps track of the probable owner. If the lock is free, it is granted. If it is not free, the requester is added to the tail of the queue. In this way, each process knows the identity of the process following it in the queue. When a lock is released, the owner passes it to the next process on the list.

Barriers are implemented by a central server. When a barrier is created, it is given a count of the number of processes that must be waiting on it before they can all be released. When a process has finished a certain phase in its computation it can send a message to the barrier server asking to wait. When the requisite number of processes are waiting, all of them are sent a message freeing them.

6.5.2. Midway

Midway is a distributed shared memory system that is based on sharing individual data structures. It is similar to Munin in some ways, but has some interesting new features of its own. Its goal was to allow existing and new multiprocessor programs to run efficiently on multicomputers with only small changes to the code. For more information about Midway, see (Bershad and Zekauskas, 1991; and Bershad et al., 1993).

Programs in Midway are basically conventional programs written in C, C++, or ML, with certain additional information provided by the programmer. Midway programs use the Mach C-threads package for expressing parallelism. A thread may fork off one or more other threads. The children run in parallel with the parent thread and with each other, potentially with each thread on a different machine (i.e., each thread as a separate process). All threads share the same linear address space, which contains both private data and shared data. The job of Midway is to keep the shared variables consistent in an efficient way.

Entry Consistency

Consistency is maintained by requiring all accesses to shared variables and data structures to be done inside a specific kind of critical section known to the Midway runtime system. Each of these critical sections is guarded by a special

synchronization variable, generally a lock, but possibly also a barrier. Each shared variable accessed in a critical section must be explicitly associated with that critical section's lock (or barrier) by a procedure call. In this way, when a critical section is entered or exited, Midway knows precisely which shared variables potentially will be accessed or have been accessed.

Midway supports entry consistency, which works as follows. To access shared data, a process normally enters a critical region by calling a library procedure, *lock*, with a lock variable as parameter. The call also specifies whether an exclusive lock or a nonexclusive lock is required. An exclusive lock is needed when one or more shared variables are to be updated. If shared variables are only to be read, but not modified, a nonexclusive lock is sufficient, which allows multiple processes to enter the same critical region at the same time. No harm can arise because none of the shared variables can be changed.

When *lock* is called, the Midway runtime system acquires the lock, and at the same time, brings all the shared variables associated with that lock up to date. Doing so may require sending messages to other processes to get the most recent values. When all the replies have been received, the lock is granted (assuming that there are no conflicts) and the process starts executing the critical region. When the process has completed the critical section, it releases the lock. Unlike release consistency, no communication takes place at release time, that is, modified shared variables are *not* pushed out to the other machines that use the shared variables. Only when one of their processes subsequently acquires a lock and asks for the current values are data transferred.

To make the entry consistency work, Midway requires that programs have three characteristics that multiprocessor programs do not have:

1. Shared variables must be declared using the new keyword *shared.*
2. Each shared variable must be associated with a lock or barrier.
3. Shared variables may only be accessed inside critical sections.

Doing these things requires extra effort from the programmer. If these rules are not completely adhered to, no error message is generated and the program may yield incorrect results. Because programming in this way is somewhat error prone, especially when running old multiprocessor programs that no one really understands any more, Midway also supports sequential consistency and release consistency. These models require less detailed information for correct operation.

The extra information required by Midway should be thought of as part of the contract between the software and the memory that we studied earlier under consistency. In effect, if the program agrees to abide by certain rules known in advance, the memory promises to work. Otherwise, all bets are off.

Implementation

When a critical section is entered, the Midway runtime system must first acquire the corresponding lock. To get an exclusive lock, it is necessary to locate the lock's owner, which is the last process to acquire it exclusively. Each process keeps track of the probable owner, the same way that IVY and Munin do, and follows the distributed chain of successive owners until the current one is found. If this process is not currently using the lock, ownership is transferred. If the lock is in use, the requesting process is made to wait until the lock is free. To acquire a lock in nonexclusive mode, it is sufficient to contact any process currently holding it. Barriers are handled by a centralized barrier manager.

At the same time the lock is acquired, the acquiring process brings its copy of all the shared variables up to date. In the simplest protocol, the old owner would just send them all. However, Midway uses an optimization to reduce the amount of data that must be transferred. Suppose that this acquire is being done at time T_1 and the previous acquire done by the same process was done at T_0. Only those variables that have been modified since T_0 are transferred, since the acquirer already has the rest.

This strategy brings up the issue of how the system keeps track of what has been modified and when. To keep track of which shared variables have been changed, a special compiler can be used that generates code to maintain a runtime table with an entry in it for each shared variable in the program. Whenever a shared variable is updated, the change is noted in the table. If this special compiler is not available, the MMU hardware is used to detect writes to shared data, as in Munin.

The time of each change is kept track of using a timestamp protocol based on Lamport's (1978) "happens before" relation. Each machine maintains a logical clock, which is incremented whenever a message is sent and included in the message. When a message arrives, the receiver sets its logical clock to the larger of the sender's clock and its own current value. Using these clocks, time is effectively partitioned into intervals defined by message transmissions. When an acquire is done, the acquiring processor specifies the time of its previous acquire and asks for all the relevant shared variables that have changed since then.

The use of entry consistency implemented in this way potentially has excellent performance because communication occurs only when a process does an acquire. Furthermore, only those shared variables that are out of date need to be transferred. In particular, if a process enters a critical region, leaves it, and enters it again, no communication is needed. This pattern is common in parallel programming, so the potential gain here is substantial. The price paid for this performance is a programmer interface that is more complex and error prone than that used by the other consistency models.

6.6. OBJECT-BASED DISTRIBUTED SHARED MEMORY

The page-based DSM systems that we studied use the MMU hardware to trap accesses to missing pages. While this approach has some advantages, it also has some disadvantages. In particular, in many programming languages, data are organized into objects, packages, modules, or other data structures, each of which has an existence independent of the others. If a process references part of an object, in many cases the entire object will be needed, so it makes sense to transport data over the network in units of objects, not in units of pages.

The shared-variable approach, as taken by Munin and Midway, is a step in the direction of organizing the shared memory in a more structured way, but it is only a first step. In both systems, the programmer must supply information about which variables are shared and which are not, and must also provide protocol information in Munin and association information in Midway. Errors in these annotations can have serious consequences.

By going further in the direction of a high-level programming model, DSM systems can be made easier and less error prone to program. Access to shared variables and synchronization using them can also be integrated more cleanly. In some cases, certain optimizations can also be introduced that are more difficult to perform in a less abstract programming model.

6.6.1. Objects

An **object** is a programmer-defined encapsulated data structure, as depicted in Fig. 6-34. It consists of internal data, the **object state**, and procedures, called **methods** or **operations**, that operate on the object state. To access or operate on the internal state, the program must invoke one of the methods. The method can change the internal state, return (part of) the state, or something else. Direct access to the internal state is not allowed. This property, called **information hiding** (Parnas, 1972). Forcing all references to an object's data to go through the methods helps structure the program in a modular way.

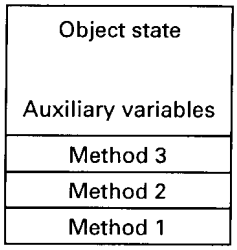


Fig. 6-34. An object.

In an object-based distributed shared memory, processes on multiple

machines share an abstract space filled with shared objects, as shown in Fig. 6-35. The location and management of the objects is handled automatically by the runtime system. This model is in contrast to page-based DSM systems such as IVY, which just provide a raw linear memory of bytes from 0 to some maximum.

Any process can invoke any object's methods, regardless of where the process and object are located. It is the job of the operating system and runtime system to make the act of invoking a method work no matter where the process and object are located. Because processes cannot directly access the internal state of any of the shared objects, various optimizations are possible here that are not possible (or at least are more difficult) with page-based DSM. For example, since access to the internal state is strictly controlled, it may be possible to relax the memory consistency protocol without the programmer even knowing it.

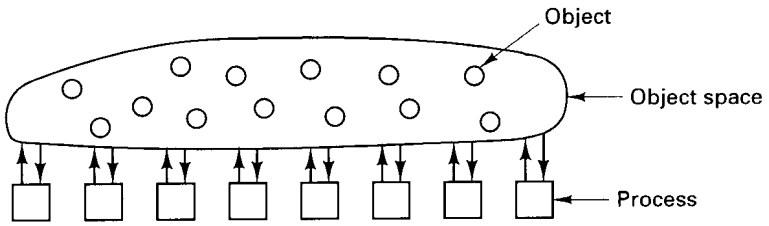


Fig. 6-35. In an object-based distributed shared memory, processes communicate by invoking methods on shared objects.

Once a decision has been made to structure a shared memory as a collection of separate objects instead of as a linear address space, there are many other choices to be made. Probably the most important issue is whether objects should be replicated or not. If replication is not used, all accesses to an object go through the one and only copy, which is simple, but may lead to poor performance. By allowing objects to migrate from machine to machine, as needed, it may be possible to reduce the performance loss by moving objects to where they are needed.

On the other hand, if objects are replicated, what should be done when one copy is updated? One approach is to invalidate all the other copies, so that only the up-to-date copy remains. Additional copies can be created later, on demand, as needed. An alternative choice is not to invalidate the copies, but to update them. Shared-variable DSM also has this choice, but for page-based DSM, invalidation is the only feasible choice. Similarly, object-based DSM, like shared-variable DSM, eliminates most false sharing.

To summarize, object-based distributed shared memory offers three advantages over the other methods:

1. It is more modular than the other techniques.
2. The implementation is more flexible because accesses are controlled.
3. Synchronization and access can be integrated together cleanly.

Object-based DSM also has disadvantages. For one thing, it cannot be used to run old "dusty deck" multiprocessor programs that assume the existence of a shared linear address space that every process can read and write at random. However, since multiprocessors are relatively new, the existing stock of multiprocessor programs that anyone cares about is small.

A second potential disadvantage is that since all accesses to shared objects must be done by invoking the objects' methods, extra overhead is incurred that is not present with shared pages that can be accessed directly. On the other hand, many experts in software engineering recommend objects as a structuring tool, even on single machines, and accept the overhead as well worth the price paid.

Below we will study two quite different examples of object-based DSM: Linda and Orca. Other distributed object-based systems also exist, including Amber (Chase et al., 1989), Emerald (Jul et al., 1988), and COOL (Lea et al., 1993).

6.6.2. Linda

Linda provides processes on multiple machines with a highly structured distributed shared memory. This memory is accessed through a small set of primitive operations that can be added to existing languages, such as C and FORTRAN to form parallel languages, in this case, C-Linda and FORTRAN-Linda. In the description below, we will focus on C-Linda, but conceptually the differences between the variants are small. More information about Linda can be found in (Carriero and Gelernter, 1986, 1989; and Gelernter, 1985).

This approach has several advantages over a new language. A major advantage is that users do not have to learn a new language. This advantage should not be underestimated. A second one is simplicity: turning a language, X, into X-Linda can be done by adding a few primitives to the library and adapting the Linda preprocessor that feeds Linda programs to the compiler. Finally, the Linda system is highly portable across operating systems and machine architectures and has been implemented on many distributed and parallel systems.

Tuple Space

The unifying concept behind Linda is that of an abstract **tuple space**. The tuple space is global to the entire system, and processes on any machine can insert tuples into the tuple space or remove tuples from the tuple space without regard to how or where they are stored. To the user, the tuple space looks like a big, global shared memory, as we saw in Fig. 6-35. The actual implementation may involve multiple servers on multiple machines, and will be described later.

A **tuple** is like a structure in C or a record in Pascal. It consists of one or more fields, each of which is a value of some type supported by the base language. For C-Linda, field types include integers, long integers, and floating-point numbers, as well as composite types such as arrays (including strings) and structures, (but not other tuples). Figure 6-36 shows three tuples as examples.

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Carolyn", "Elinor")
```

Fig. 6-36. Three Linda tuples.

Operations on Tuples

Linda is not a fully general object-based system since it provides only a fixed number of built-in operations and no way to define new ones. Four operations are provided on tuples. The first one, *out*, puts a tuple into the tuple space. For example,

```
out("abc", 2, 5);
```

puts the tuple ("abc", 2, 5) into the tuple space. The fields of *out* are normally constants, variables, or expressions, as in

```
out("matrix-1", i, j, 3.14);
```

which outputs a tuple with four fields, the second and third of which are determined by the current values of the variables i and j.

Tuples are retrieved from the tuple space by the *in* primitive. They are addressed by content rather than by name or address. The fields of *in* can be expressions or formal parameters. Consider, for example,

```
in("abc", 2, ? i);
```

This operation "searches" the tuple space for a tuple consisting of the string "abc", the integer, 2, and a third field containing any integer (assuming that i is an integer). If found, the tuple is removed from the tuple space and the variable

i is assigned the value of the third field. The matching and removal are atomic, so if two processes execute the same *in* operation simultaneously, only one of them will succeed, unless two or more matching tuples are present. The tuple space may even contain multiple copies of the same tuple.

The matching algorithm used by *in* is straightforward. The fields of the *in* primitive, called the **template**, are (conceptually) compared to the corresponding fields of every tuple in the tuple space. A match occurs if the following three conditions are all met:

1. The template and the tuple have the same number of fields.
2. The types of the corresponding fields are equal.
3. Each constant or variable in the template matches its tuple field.

Formal parameters, indicated by a question mark followed by a variable name or type, do not participate in the matching (except for type checking), although those containing a variable name are assigned after a successful match.

If no matching tuple is present, the calling process is suspended until another process inserts the needed tuple, at which time the caller is automatically revived and given the new tuple. The fact that processes block and unblock automatically means that if one process is about to output a tuple and another is about to input it, it does not matter which goes first. The only difference is that if the *in* is done before the *out*, there will be a slight delay until the tuple is available for removal.

The fact that processes block when a needed tuple is not present can be put to many uses. For example, it can be used to implement semaphores. To create or do an *UP* (*V*) on semaphore *S*, a process can execute

```
out("semaphore S");
```

To do a *DOWN* (*P*), it does

```
in("semaphore S");
```

The state of semaphore *S* is determined by the number of ("semaphore S") tuples in the tuple space. If none exist, any attempt to get one will block until some other process supplies one.

In addition to *out* and *in*, Linda also has a primitive *read*, which is the same as *in* except that it does not remove the tuple from the tuple space. There is also a primitive *eval*, which causes its parameters to be evaluated in parallel and the resulting tuple to be deposited in the tuple space. This mechanism can be used to perform an arbitrary computation. This is how parallel processes are created in Linda.

A common programming paradigm in Linda is the **replicated worker**

model. This model is based on the idea of a **task bag** full of jobs to be done. The main process starts out by executing a loop containing

```
out("task-bag", job);
```

in which a different job description is output to the tuple space on each iteration. Each worker starts out by getting a job description tuple using

```
in("task-bag", ?job);
```

which it then carries out. When it is done, it gets another. New work may also be put into the task bag during execution. In this simple way, work is dynamically divided among the workers, and each worker is kept busy all the time, all with little overhead.

In certain ways, Linda is similar to Prolog, on which it is loosely based. Both support an abstract space that functions as a kind of data base. In Prolog, the space holds facts and rules; in Linda it holds tuples. In both cases, processes can provide templates to be matched against the contents of the data base.

Despite these similarities, the two systems also differ in significant ways. Prolog was intended for programming artificial intelligence applications on a single processor, whereas Linda was intended for general programming on multicomputers. Prolog has a complex pattern-matching scheme involving unification and backtracking; Linda's matching algorithm is much simpler. In Linda, a successful match removes the matching tuple from the tuple space; in Prolog it does not. Finally, a Linda process unable to locate a needed tuple blocks, which forms the basis for interprocess synchronization. In Prolog, there are no processes and programs never block.

Implementation of Linda

Efficient implementations of Linda are possible on various kinds of hardware. Below we will discuss some of the more interesting ones. For all implementations, a preprocessor scans the Linda program, extracting useful information and converting it to the base language where need be (e.g., the string "? int" is not allowed as a parameter in C or FORTRAN). The actual work of inserting and removing tuples from the tuple space is done during execution by the Linda runtime system.

An efficient Linda implementation has to solve two problems:

1. How to simulate associative addressing without massive searching.
2. How to distribute tuples among machines and locate them later.

The key to both problems is to observe that each tuple has a **type signature**, consisting of the (ordered) list of the types of its fields. Furthermore, by

convention, the first field of each tuple is normally a string that effectively partitions the tuple space into disjoint subspaces named by the string. Splitting the tuple space into subspaces, each of whose tuples has the same type signature and same first field, simplifies programming and makes certain optimizations possible.

For example, if the first parameter to an *in* or *out* call is a literal string, it is possible to determine at compile time which subspace the call operates on. If the first parameter is a variable, the determination is made at run time. In both cases, this partitioning means that only a fraction of the tuple space has to be searched. Figure 6-37 shows four tuples and four templates. Together they form four subspaces. For each *out* or *in*, it is possible to determine at compile time which subspace and tuple server are needed. If the initial string was a variable, the determination of the correct subspace would have to be delayed until run time.

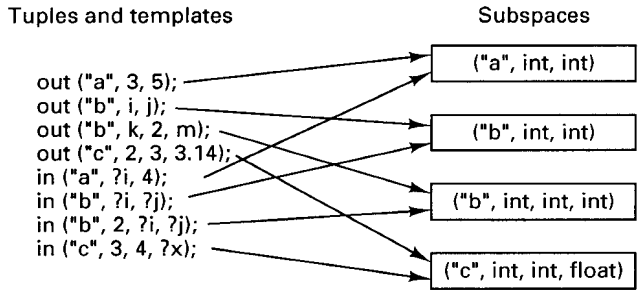


Fig. 6-37. Tuples and templates can be associated with subspaces.

In addition, each subspace can be organized as a hash table using its *i*th tuple field as the hash key. If field *i* is a constant or variable (but not a formal parameter), an *in* or *out* can be executed by computing the hash function of the *i*th field to find the position in the table where the tuple belongs. Knowing the subspace and table position eliminates all searching. If the *i*th field of a certain *in* is a formal parameter, hashing is not possible, so a complete search of the subspace is needed except in some special cases. By carefully choosing the field to hash on, however, the preprocessor can usually avoid searching most of the time. Other subspace organizations beside hashing are also possible for special cases (e.g., a queue when there is one writer and one reader).

Additional optimizations are also used. For example, the hashing scheme described above distributes the tuples of a given subspace into bins, to restrict searching to a single bin. It is possible to place different bins on different machines, both to spread the load more widely and to take advantage of locality. If the hashing function is the key modulo the number of machines, the number of bins scales linearly with the system size.

Now let us examine various implementation techniques for different kinds of hardware. On a multiprocessor, the tuple subspaces can be implemented as hash tables in global memory, one for each subspace. When an *in* or an *out* is performed, the corresponding subspace is locked, the tuple entered or removed, and the subspace unlocked.

On a multicomputer, the best choice depends on the communication architecture. If reliable broadcasting is available, a serious candidate is to replicate all the subspaces in full on all machines, as shown in Fig. 6-38. When an *out* is done, the new tuple is broadcast and entered into the appropriate subspace on each machine. To do an *in*, the local subspace is searched. However, since successful completion of an *in* requires removing the tuple from the tuple space, a delete protocol is required to remove it from all machines. To prevent races and deadlocks, a two-phase commit protocol can be used.

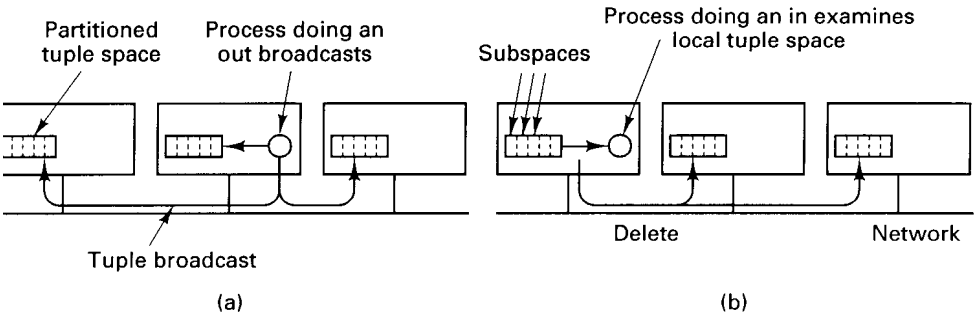


Fig. 6-38. Tuple space can be replicated on all machines. The dotted lines show the partitioning of the tuple space into subspaces. (a) Tuples are broadcast on *out*. (b). *In*s are local, but the deletes must be broadcast.

This design is straightforward, but may not scale well as the system grows in size, since every tuple must be stored on every machine. On the other hand, the total size of the tuple space is often quite modest, so problems may not arise except in huge systems. The S/Net Linda system uses this approach because S/Net has a fast, reliable, word-parallel bus broadcast (Carriero and Gelernter, 1986).

The inverse design is to do *out*s locally, storing the tuple only on the machine that generated it, as shown in Fig. 6-39. To do an *in*, a process must broadcast the template. Each recipient then checks to see if it has a match, sending back a reply if it does.

If the tuple is not present, or if the broadcast is not received at the machine holding the tuple, the requesting machine retransmits the broadcast request ad infinitem, increasing the interval between broadcasts until a suitable tuple materializes and the request can be satisfied. If two or more tuples are sent, they

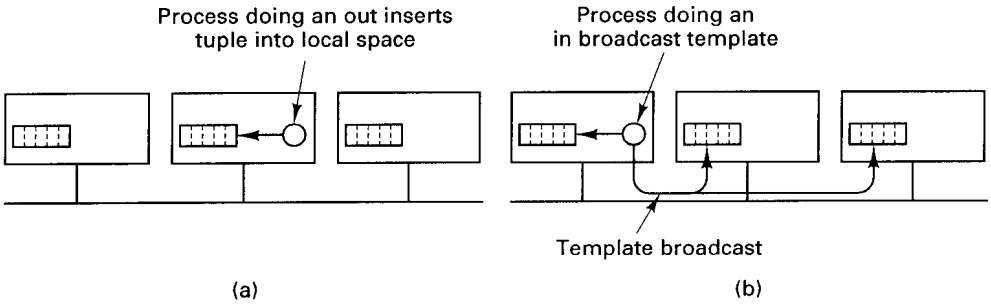


Fig. 6-39. Unreplicated tuple space. (a) An *out* is done locally. (b) An *in* requires the template to be broadcast in order to find a tuple.

are treated like local *out*s and the tuples are effectively moved from the machines that had them to the one doing the request. In fact, the runtime system can even move tuples around on its own to balance the load. Carriero et al. (1986) used this method for implementing Linda on a LAN.

These two methods can be combined to produce a system with partial replication. A simple example is to imagine all the machines logically forming a rectangle, as shown in Fig. 6-40. When a process on a machine, *A*, wants to do an *out*, it broadcasts (or sends by point-to-point message) the tuple to all machines in its row of the matrix. When a process on a machine, *B*, wants to do an *in* it broadcasts the template to all machines in its column. Due to the geometry, there will always be exactly one machine that sees both the tuple and the template (*C* in this example), and that machine makes the match and sends the tuple to the process asking for it. Krishnaswamy (1991) used this method for a hardware Linda coprocessor.

Finally, let us consider the implementation of Linda on systems that have no broadcast capability at all (Bjornson, 1993). The basic idea is to partition the tuple space into disjoint subspaces, first by creating a partition for each type signature, then by dividing each of these partitions again based on the first field. Potentially, each of the resulting partitions can go on a different machine, handled by its own tuple server, to spread the load around. When either an *out* or an *in* is done, the required partition is determined, and a single message is sent to that machine either to deposit a tuple there or to retrieve one.

Experience with Linda shows that distributed shared memory can be handled in a radically different way than moving whole pages around, as in the page-based systems we studied above. It is also quite different from sharing variables with release or entry consistency. As future systems become larger and more powerful, novel approaches such as this may lead to new insights into how to program these systems in an easier way.

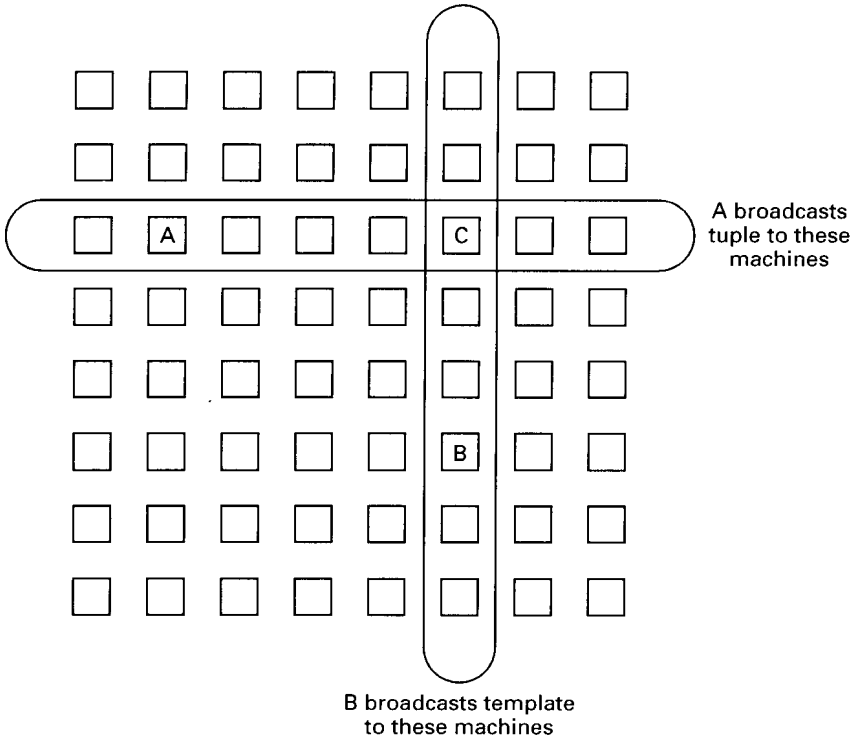


Fig. 6-40. Partial broadcasting of tuples and templates.

6.6.3. Orca

Orca is a parallel programming system that allows processes on different machines to have controlled access to a distributed shared memory consisting of protected objects (Bal, 1991; and Bal et al., 1990, 1992). These objects can be thought of as a more powerful (and more complicated) form of the Linda tuples, supporting arbitrary operations instead of just *in* and *out*. Another difference is that Linda tuples are created on-the-fly during execution in large volume, whereas Orca objects are not. The Linda tuples are used primarily for communication, whereas the Orca objects are also used for computation and are generally more heavyweight.

The Orca system consists of the language, compiler, and runtime system, which actually manages the shared objects during execution. Although language, compiler, and runtime system were designed to work together, the runtime system is independent of the compiler and could be used for other languages as well. After an introduction to the Orca language, we will describe how the runtime system implements an object-based distributed shared memory.

The Orca Language

In some respects, Orca is a traditional language whose sequential statements are based roughly on Modula-2. However, it is a type secure language with no pointers and no aliasing. Array bounds are checked at runtime (except when the checking can be done at compile time). These and similar features eliminate or detect many common programming errors such as wild stores, into memory. The language features have been chosen carefully to make a variety of optimizations easier.

Two features of Orca important for distributed programming are shared data-objects (or just **objects**) and the **fork** statement. An object is an abstract data type, somewhat analogous to a package in Ada®. It encapsulates internal data structures and user-written procedures, called **operations** (or **methods**) for operating on the internal data structures. Objects are passive, that is, they do not contain threads to which messages can be sent. Instead, processes access an object's internal data by invoking its operations. Objects do not inherit properties from other objects, so Orca is considered an object-based language rather than an object-oriented language.

Each operation consists of a list of (guard, block-of-statements) pairs. A guard is a Boolean expression that does not contain any side effects, or the empty guard, which is the same as the value *true*. When an operation is invoked, all of its guards are evaluated in an unspecified order. If all of them are *false*, the invoking process is delayed until one becomes *true*. When a guard is found that evaluates to *true*, the block of statements following it is executed. Figure 6-41 depicts a *stack* object with two operations, *push* and *pop*.

```
Object implementation stack;                    # variable indicating the top
    top: integer;                               # storage for the stack
    stack: array [integer 0..N–1] of integer;
    operation push (item: integer);             # function returning nothing
    begin
        stack [top] := item;                    # push item onto the stack
        top := top + 1;                         # increment the stack pointer
    end;
    operation pop(): integer;                   # function returning an integer
    begin
        guard top > 0 do                        # suspend if the stack is empty
            top := top – 1;                     # decrement the stack pointer
            return stack [top];                 # return the top item
        od;
    end;
begin
    top := 0;                                   # initialization
end;
```

Fig. 6-41. A simplified stack object, with internal data and two operations.

Once a *stack* has been defined, variables of this type can be declared, as in

```
s, t: stack;
```

which creates two stack objects and initializes the *top* variable in each to 0. The integer variable *k* can be pushed onto the stack *s* by the statement

```
s$push(k);
```

and so forth. The *pop* operation has a guard, so an attempt to pop a variable from an empty stack will suspend the caller until another process has pushed something on the stack.

Orca has a **fork** statement to create a new process on a user-specified processor. The new process runs the procedure named in the **fork** statement. Parameters, including objects, may be passed to the new process, which is how objects become distributed among machines. For example, the statement

```
for i in 1 .. n do fork foobar(s) on i; od;
```

generates one new process on each of machines 1 through *n*, running the process **foobar** in each of them. As these *n* new processes (and the parent) execute in parallel, they can all push and pop items onto the shared stack *s* as though they were all running on a shared-memory multiprocessor. It is the job of the runtime system to sustain the illusion of shared memory where it really does not exist.

Operations on shared objects are atomic and sequentially consistent. The system guarantees that if multiple processes perform operations on the same shared object nearly simultaneously, the net effect is that it looks like the operations took place strictly sequentially, with no operation beginning until the previous one finished.

Furthermore, the operations appear in the same order to all processes. For example, suppose that we were to augment the *stack* object with a new operation, *peek*, to inspect the top item on the stack. Then if two independent processes push 3 and 4 simultaneously, and all processes later use *peek* to examine the top of the stack, the system guarantees that either every process will see 3 or every process will see 4. A situation in which some processes see 3 and other processes see 4 cannot occur in a multiprocessor or a paged-based distributed shared memory, and it cannot occur in Orca either. If only one copy of the stack is maintained, this effect is trivial to achieve, but if the stack is replicated on all machines, more effort is required, as described below.

Although we have not emphasized it, Orca integrates shared data and synchronization in a way not present in page-based DSM systems. Two kinds of synchronization are needed in parallel programs. The first kind is mutual exclusion synchronization, to keep two processes from executing the same critical region at the same time. In Orca, each operation on a shared object is

effectively like a critical region because the system guarantees that the final result is the same as if all the critical regions were executed one at a time (i.e., sequentially). In this respect, an Orca object is like a distributed form of a monitor (Hoare, 1975).

The other kind of synchronization is condition synchronization, in which a process blocks waiting for some condition to hold. In Orca, condition synchronization is done with guards. In the example of Fig. 6-41, a process trying to pop an item from an empty stack will be suspended until the stack is no longer empty.

Management of Shared Objects in Orca

Object management in Orca is handled by the runtime system. It works on both broadcast (or multicast) networks and point-to-point networks. The runtime system handles object replication, migration, consistency, and operation invocation.

Each object can be in one of two states: single copy or replicated. An object in single-copy state exists on only one machine. A replicated object is present on all machines containing a process using it. It is not required that all objects be in the same state, so some of the objects used by a process may be replicated while others are single copy. Objects can change from single-copy state to replicated state and back during execution.

The big advantage of replicating an object on every machine is that reads can be done locally, without any network traffic or delay. When an object is not replicated, all operations must be sent to the object, and the caller must block waiting for the reply. A second advantage of replication is increased parallelism: multiple read operations can take place at the same time. With a single copy, only one operation at a time can take place, slowing down execution. The principal disadvantage of replication is the overhead of keeping all the copies consistent.

When a program performs an operation on an object, the compiler calls a runtime system procedure, *invoke_op*, specifying the object, the operation, the parameters, and a flag telling whether the object will be modified (called a write) or not modified (called a read). The action taken by *invoke_op* depends on whether the object is replicated, whether a copy is available locally, whether it is being read or written, and whether the underlying system supports reliable, totally-ordered broadcasting. Four cases have to be distinguished, as illustrated in Fig. 6-42.

In Fig. 6-42(a), a process wants to perform an operation on a nonreplicated object that happens to be on its own machine. It just locks the object, invokes the operation, and unlocks the object. The point of locking it is to inhibit any remote invocations temporarily while the local operation is in progress.

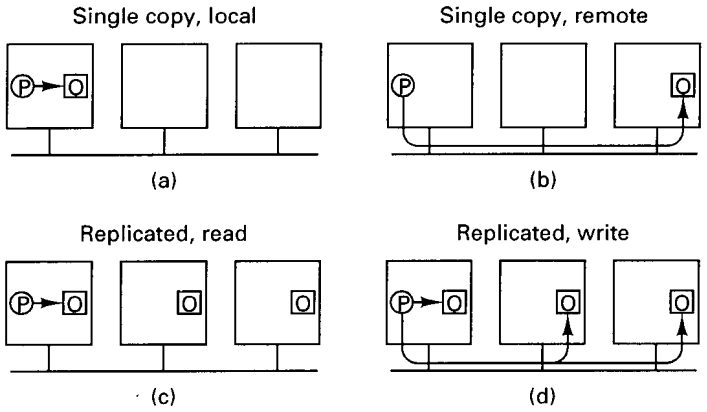


Fig. 6-42. Four cases of performing an operation on an object, *O*.

In Fig. 6-42(b) we still have a single-copy object, but now it is somewhere else. The runtime system does an RPC with the remote machine asking it to perform the operation, which it does, possibly with a slight delay if the object was locked when the request came in. In neither of these two cases is a distinction made between reads and writes (except that writes can awaken blocked processes).

If the object is replicated, as in Fig. 6-42(c) and (d), a copy will always be available locally, but now it matters if the operation is a read or a write. Reads are just done locally, with no network traffic and thus no overhead.

Writes on replicated objects are trickier. If the underlying system provides *reliable*, totally-ordered broadcasting, the runtime system broadcasts the name of the object, the operation, and the parameters and blocks until the broadcast has completed. All the machines, including itself, then compute the new value.

Note that the broadcasting primitive must be reliable, meaning that lower layers automatically detect and recover from lost messages. The Amoeba system, on which Orca was developed, has such a feature. Although the algorithm will be described in detail in Chap. 7, we will summarize it here very briefly. Each message to be broadcast is sent to a special process called the **sequencer**, which assigns it a sequence number and then broadcasts it using the unreliable hardware broadcast. Whenever a process notices a gap in the sequence numbers, it knows that it has missed a message and takes action to recover.

If the system does not have reliable broadcasting (or does not have any broadcasting at all), the update is done using a two-phase, primary copy algorithm. The process doing the update first sends a message to the primary copy of the object, locking and updating it. The primary copy then sends individual messages to all other machines holding the object, asking them to lock their

copies. When all of them have acknowledged setting the lock, the originating process enters the second phase and sends another message telling them to perform the update and unlock the object.

Deadlocks are impossible because even if two processes try to update the same object at the same time, one of them will get to the primary copy first to lock it, and the other request will be queued until the object is free again. Also note that during the update process, all copies of the object are locked, so no other processes can read the old value. This locking guarantees that all operations are executed in a globally unique sequential order.

Notice that this runtime system uses an update algorithm rather than an invalidation algorithm as most page-based DSM systems do. Most objects are relatively small, so sending an update message (the new value or the parameters) is often no more expensive than an invalidate message. Updating has the great advantage of allowing subsequent remote reads to occur without having to refetch the object or perform the operation remotely.

Now let us briefly look at the algorithm for deciding whether an object should be in single-copy state or replicated. Initially, an Orca program consists of one process, which has all the objects. When it forks, all other machines are told of this event and given current copies of all the child's shared parameters. Each runtime system then calculates the expected cost of having each object replicated versus having it not replicated.

To make this calculation, it needs to know the expected ratio of reads to writes. The compiler estimates this information by examining the program, taking into account that accesses inside loops count more and accesses inside if-statements count less than other accesses. Communication costs are also factored into the equation. For example, an object with a read/write ratio of 10 on a broadcast network will be replicated, whereas an object with a read/write ratio of 1 on a point-to-point network will be put in single-copy state, with the single copy going on the machine doing the most writes. For a more detailed description, see (Bal and Kaashoek, 1993).

Since all runtime systems make the same calculation, they come to the same conclusion. If an object currently is present on only one machine and needs to be on all, it is disseminated. If it is currently replicated and that is no longer the best choice, all machines but one discard their copy. Objects can migrate via this mechanism.

Let us see how sequential consistency is achieved. For objects in single-copy state, all operations genuinely are serialized, so sequential consistency is achieved for free. For replicated objects, writes are totally ordered either by the reliable, totally-ordered broadcast or by the primary copy algorithm. Either way, there is global agreement on the order of the writes. The reads are local and can be interleaved with the writes in an arbitrary way without affecting sequential consistency.

Assuming that a reliable, totally-ordered broadcast can be done in two (plus epsilon) messages, as in Amoeba, the Orca scheme is highly efficient. Only after an operation is finished are the results sent out, no matter how many local variables were changed by the operation. If one regards each operation as a critical region, the efficiency is the same as for release consistency—one broadcast at the end of each critical region.

Various optimizations are possible. For example, instead of synchronizing *after* an operation, it could be done when an operation is started, as in entry consistency or lazy release consistency. The advantage here is that if a process executes an operation on a shared object repeatedly (e.g., in a loop), no broadcasts at all are sent until some other process exhibits interest in the object.

Another optimization is not to suspend the caller while doing the broadcast after a write that does not return a value (e.g., *push* in our stack example). Of course, this optimization must be done in such a transparent way. Information supplied by the compiler makes other optimizations possible.

In summary, the Orca model of distributed shared memory integrates good software engineering practice (encapsulated objects), shared data, simple semantics, and synchronization in a natural way. Also, in many cases an implementation as efficient as release consistency is possible. It works best when the underlying hardware and operating system must provide efficient, reliable, totally-ordered broadcasting, and the application must has an inherently high ratio of reads to writes for accesses to shared objects.

6.7. COMPARISON

Let us now briefly compare the various systems we have examined. IVY just tries to mimic a multiprocessor by doing paging over the network instead of to a disk. It offers a familiar memory model—sequential consistency, and can run existing multiprocessor programs without modification. The only problem is the performance.

Munin and Midway try to improve the performance by requiring the programmer to mark those variables that are shared and by using weaker consistency models. Munin is based on release consistency, and on every release transmits all modified pages (as deltas) to other processes sharing those pages. Midway, in contrast, does communication only when a lock changes ownership.

Midway supports only one kind of shared data variable, whereas Munin has four kinds (read only, migratory, write-shared, and conventional). On the other hand, Midway supports three different consistency protocols (entry, release, and processor), whereas Munin only supports release consistency. Whether it is better to have multiple types of shared data or multiple protocols is open to discussion. More research will be needed before we understand this subject fully.

Finally, the way writes to shared variables are detected is different. Munin uses the MMU hardware to trap writes, whereas Midway offers a choice between a special compiler that records writes and doing it the way Munin does, with the MMU. Not having to take a stream of page faults, especially inside critical regions, is definitely an advantage for Midway.

Now let us compare Munin and Midway to object-based shared memory of the Linda-Orca variety. Synchronization and data access in Munin and Midway are up to the programmer, whereas they are tightly integrated in Linda and Orca. In Linda there is less danger that a programmer will make a synchronization error, since *in* and *out* handle their own synchronization internally. Similarly, when an operation on a shared object is invoked in Orca, the locking is handled completely by the runtime system, with the programmer not even being aware of it. Condition synchronization (as opposed to mutual exclusion synchronization) is not part of the Munin or Midway model, so it is up to the programmer to do all the work explicitly. In contrast, it is an integral part of the Linda model (blocking when a tuple is not present) and the Orca model (blocking on a guard).

In short, the Munin and Midway programmers must do more work in the area of synchronization and consistency, with little support, and must get it all right. There is no encapsulation and there are no methods to protect shared data, as Linda and Orca have. On the other hand, Munin and Midway allow programming in only slightly modified C or C++, whereas Linda's communication is unusual and Orca is a whole new language. In terms of efficiency, Midway is best in terms of the number and size of messages transmitted, although the use of fundamentally different programming models (open C code, objects, and tuples) may lead to substantially different algorithms in the three cases, which also can affect efficiency.

6.8. SUMMARY

Multiple CPU computer systems fall into two categories: those that have shared memory and those that do not. The shared memory machines (multiprocessors) are easier to program but harder to build, whereas the machines without shared memory (multicomputers) are harder to program but easier to build. Distributed shared memory is a technique for making multicomputers easier to program by simulating shared memory on them.

Small multiprocessors are often bus based, but large ones are switched. The protocols the large ones use require complex data structures and algorithms to keep the caches consistent. NUMA multiprocessors avoid this complexity by forcing the software to make all the decisions about which pages to place on which machine.

A straightforward implementation of DSM, as done in IVY, is possible, but