View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by VU Research Portal

VU Research Portal

Using Broadcasting to Implement Distributed Shared Memory Efficiently

Tanenbaum, A.S.; Kaashoek, M.F.; Bal, H.E.

published in
Reading in Distributed Computing Systems Chapter 8
1994

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
Tanenbaum, A. S., Kaashoek, M. F., & Bal, H. E. (1994). Using Broadcasting to Implement Distributed Shared Memory Efficiently. In *Reading in Distributed Computing Systems Chapter 8* (pp. 387-408). IEEE Press.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 27. May. 2021

Using Broadcasting to Implement Distributed Shared Memory Efficiently

Andrew S. Tanenbaum,

M. Frans Kaashoek, and

Henri E. Bal

As computers have continued to get cheaper, interest has increased in harnessing together multiple CPUs to build large, powerful parallel systems that are high in performance and low in cost. The two major design approaches taken so far are multiprocessors and multicomputers. NUMA (nonuniform memory access) architectures are an attempt at combining the two types of parallel machines, each of which has its strengths and weaknesses. This hybrid form uses an unusual software organization on conventional hardware to achieve a system that is easy to build and easy to program. We have built a prototype system that is similar to NUMA in some important respects. The central concept in our method is to efficiently do reliable broadcasting on unreliable hardware. Here, we describe some applications we have written for our system, give measurements of its performance, look at a paradigm for programming using this method, and discuss a parallel programming language that we have designed and implemented to allow programmers to use our system conveniently.

Much confusion exists in the literature about the terms *distributed system* and *parallel system*. For our purposes, a distributed system is one in which independent jobs execute on independent CPUs—on behalf of independent users—but share some resources, such as a common file server. A typical example of a distributed system is a collection of (possibly heterogeneous) engineering workstations connected by a fast local area network (LAN) and sharing a file server. In contrast, a parallel system is one in which a user attempts to utilize multiple (usually identical) CPUs in order to speed up the execution of a single program. The CPUs may be in a single rack and connected by a high-speed backplane bus, they may be distributed around a building and connected by a fast LAN, or they may be used in some other topology. What is important is that a large number of CPUs cooperate to solve a single problem. Thus, the difference between the two types of systems is really a question of how the software—not the hardware—is organized. Here, we concentrate on parallel systems, using as a metric how much speedup can be achieved on a single problem by using *n* identical processors rather than only one processor.

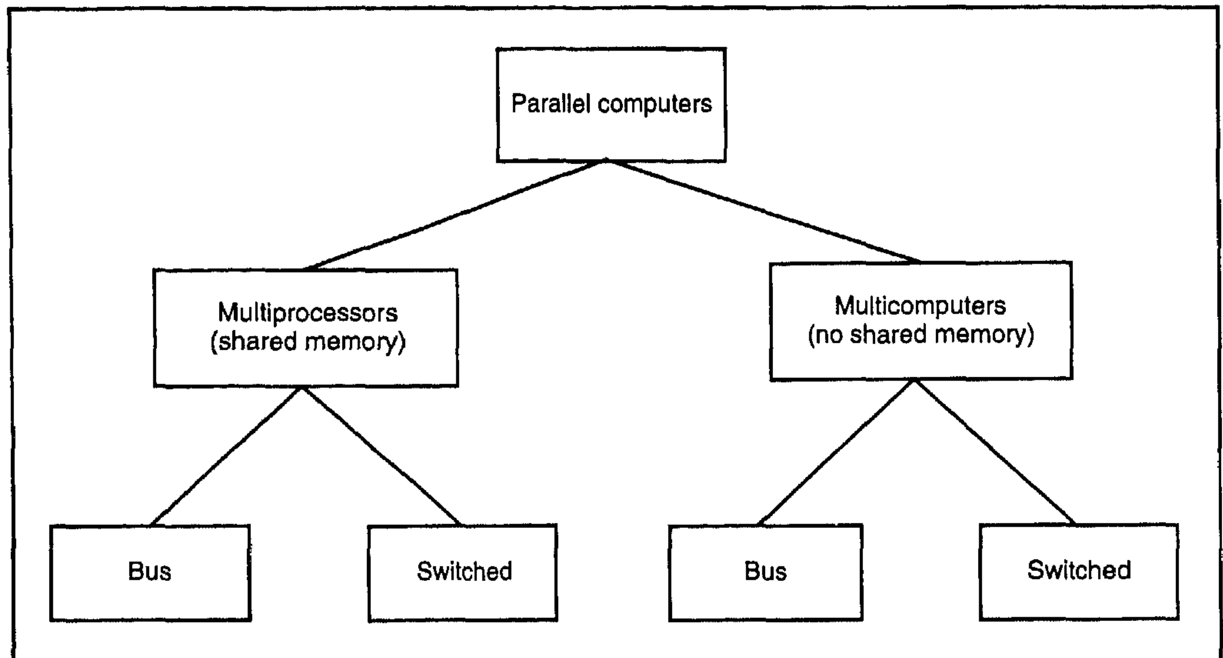

Figure 1. A taxonomy of parallel computers.

Multiprocessors versus multicomputers

Parallel computers can be divided into two categories: *multiprocessors*, which contain physical shared memory, and *multicomputers*, which do not. A simple taxonomy is given in Figure 1. In multiprocessors, there is a single, global, shared address space visible to all processors. Any processor can read or write any word in the address space by simply moving data from or to a memory address. In multicomputers, the processors need some other way to communicate—for example, by message passing.

Multiprocessors

The key property required of any multiprocessor is *memory coherence*. When any processor writes a value v to memory address m, any other processor that subsequently reads the word at memory address m, no matter how quickly after the write, will get the value v just written.

Multiprocessor hardware. There are two basic methods of building a multiprocessor, both of them expensive. In the first method, all the processors are put on a single bus, along with a memory module. To read or write a word of data, a processor makes a normal memory request over the bus. Because there is only one memory module and there are no copies of memory words anywhere else, the memory is always *coherent*. The problem with this method is that the bus will be completely saturated with as few as four to eight processors. To get around this problem, each processor is normally given a cache memory, as shown in Figure 2. However, the caches introduce a new problem: If Processors 1 and 2 both read the contents of memory address m into their caches, and one of them modifies m, then when the other processor next reads that address, it will get a *stale* value. Memory is then not coherent. Because *incoherent* memory is hard for programmers to deal with, it is unacceptable.

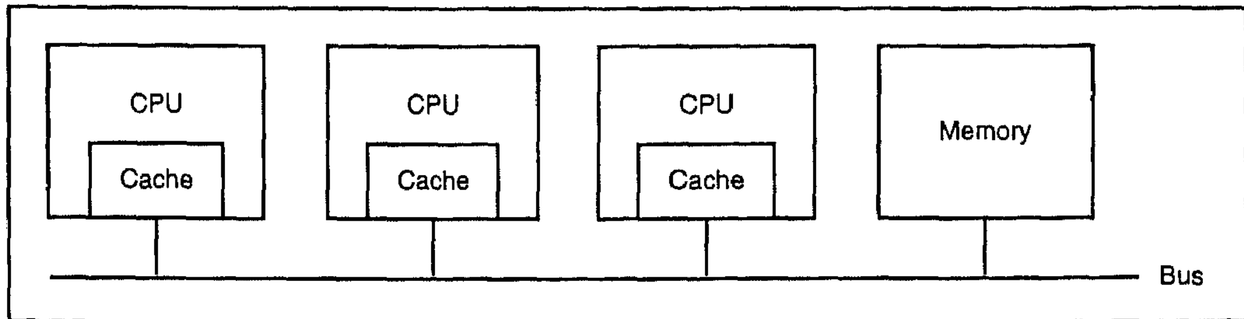

Figure 2. A single-bus multiprocessor.

The stale-data problem can be solved in many ways. One way is to have all writes go through the cache to update the external memory. Another is to have each cache constantly *snoop* on the bus (that is, monitor it) and take some appropriate action — such as invalidating or updating its cache entry — whenever another processor tries to write a word for which it has a local copy. Nevertheless, caching only delays the problem of bus saturation. Instead of saturating at four to eight processors, a well-designed single-bus system might saturate at 32 to 64 processors. Building a single-bus system with appreciably more than 64 processors is not feasible with current bus technology.

In the second method, a multiprocessor is built using some kind of switching network, such as the *crossbar switch*, which is shown in Figure 3(a). Each of the *n* processors can potentially be connected to any one of the *n* memory banks via a matrix of little electronic switches. When switch *ij* is closed (by hardware), processor *i* is connected to memory bank *j* and can read or write data there. Since several switches may be closed simultaneously, multiple reads and writes can occur at the same time between disjoint processor-memory combinations. The problem with the crossbar switch is that connecting *n* processors to *n* memory banks requires n^2 switches. As *n* becomes large — say 1024 processors and 1024 memories — the switch becomes prohibitively expensive and unmanageable.

An alternative switching scheme for multiprocessors is the omega network shown in Figure 3(b). In this figure, the CPUs (on the left) and the memories (on the right) are connected by the omega network, a sophisticated packet-switching network. To read a word of memory, a CPU sends a request packet to the appropriate memory via the switching network, which sends the reply back the other way. Many variations of this basic design have been proposed, but they all have the problem that for a system with *n* processors and *n* memories, the number of switching stages needed is on the order of $\log_2 n$ and the number of switching elements is $n \log_2 n$. For example, consider a system of 1024 reduced instruction-set computer (RISC) processors running at 50 megahertz (MHz). With $n = 1024$, 10 switches must be traversed from the CPU to the memory and 10 more on the way back. If this is to occur in one cycle (20 nanoseconds [ns]), each switching step must take no longer than one ns, and 10,240 such switches are required. A machine with such a large number of very high speed packet switches is clearly going to be expensive and difficult to build and maintain, even if the designers are able to make many optimizations.

Multiprocessor software. In contrast to building multiprocessor hardware — which, for large systems, is complicated, difficult, and expensive—building multiprocessor software is straightforward. Since all processes run within a single, shared address space, they can easily share data structures and variables. When one process updates a variable and another one reads it immediately afterward, the reader always gets the value just stored; that is, memory is coherent.

To avoid chaos, cooperating processes must synchronize their activities. For example, while one process is updating a linked list, it is essential that no other process even attempt to read the list, let alone modify it. Many well-known techniques—including spin locks, semaphores, and monitors—

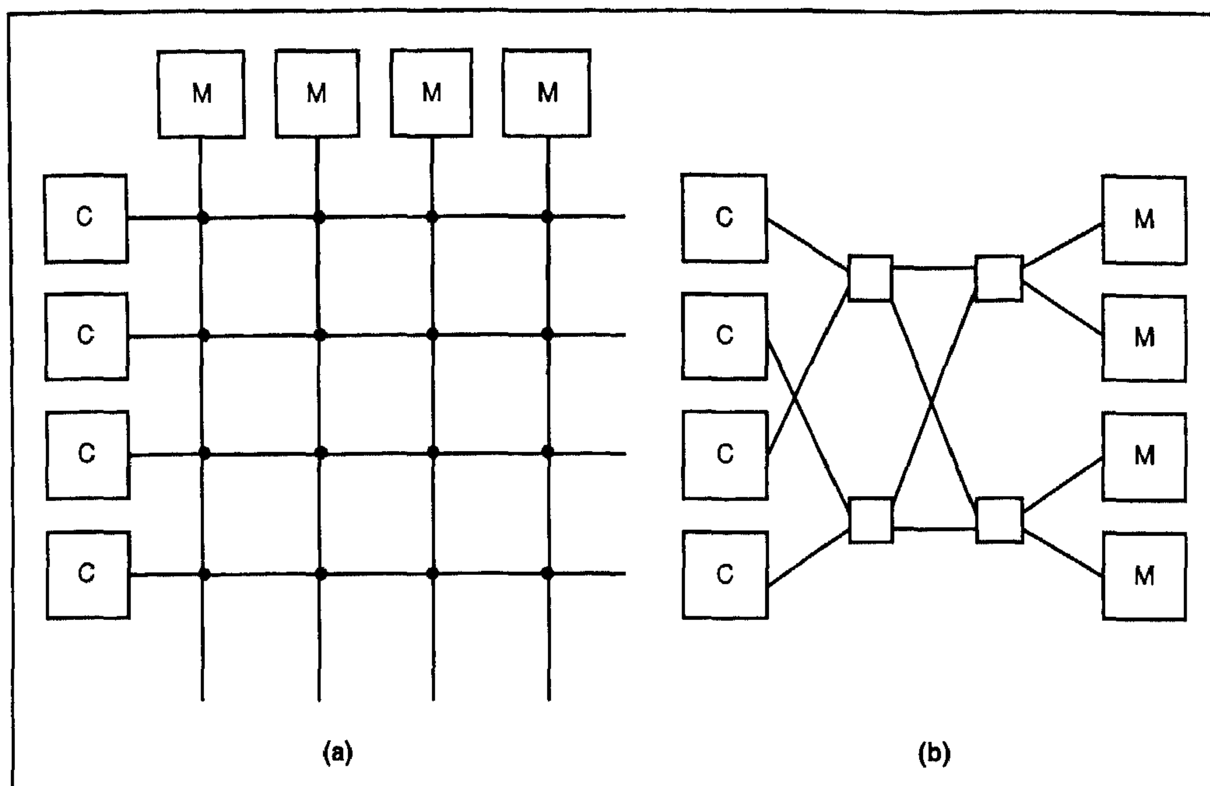

Figure 3. (a) Crossbar switch. (b) Omega network. (C = CPU and M = memory.)

can provide the necessary synchronization. (These techniques are discussed in any standard textbook on operating systems.) The advantages of multiprocessor software are that sharing is easy and cheap and uses a well-understood methodology that has been around for years.

Multicomputers

In contrast to multiprocessors, which—by definition—share primary memory, *multicomputers* do not. Each CPU in a multicomputer has its own, private memory, which it alone can read and write. This difference leads to a significantly different architecture, both in hardware and in software.

Multicomputer hardware. Just as there are bus and switched multiprocessors, there are bus and switched multicomputers. Figure 4 shows a simple bus-based multicomputer. Note that each CPU has its own local memory, which is not accessible by remote CPUs. Since there is no shared memory in this system, communication occurs via message passing between CPUs. The "bus" in this example can either be a LAN or a high-speed backplane; conceptually, these two are the same, differing only in their performance. Since each CPU-memory pair is essentially independent of all the others, building very large multicomputer systems—certainly much larger than multiprocessor systems—is straightforward.

Switched multicomputers do not have a single bus over which all traffic goes. Instead, they have a collection of point-to-point connections. Figure 5 shows a grid and a hypercube, two examples of the many designs that have been proposed and built. A grid is easy to understand and easy to lay out on a printed circuit board or chip. This architecture is best suited to problems that are two-dimensional in nature, such as graph theory and vision. A hypercube is an n-dimensional cube. One can imagine a four-dimensional hypercube as a pair of ordinary cubes with the corresponding vertices connected, as shown in Figure 5(b). (In the figure, the lower left nodes at the rear have been omitted for clarity.) Similarly, a five-dimensional hypercube can be represented as two copies of Figure 5(b) with the

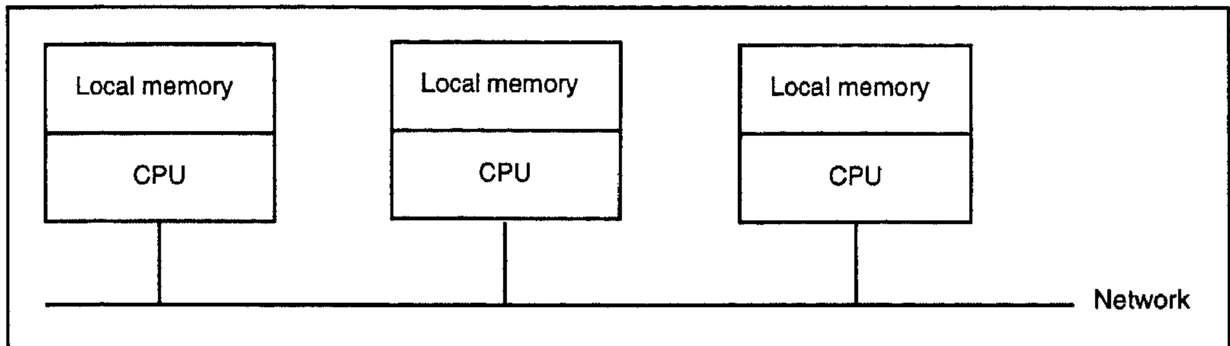

Figure 4. A single-bus multicomputer.

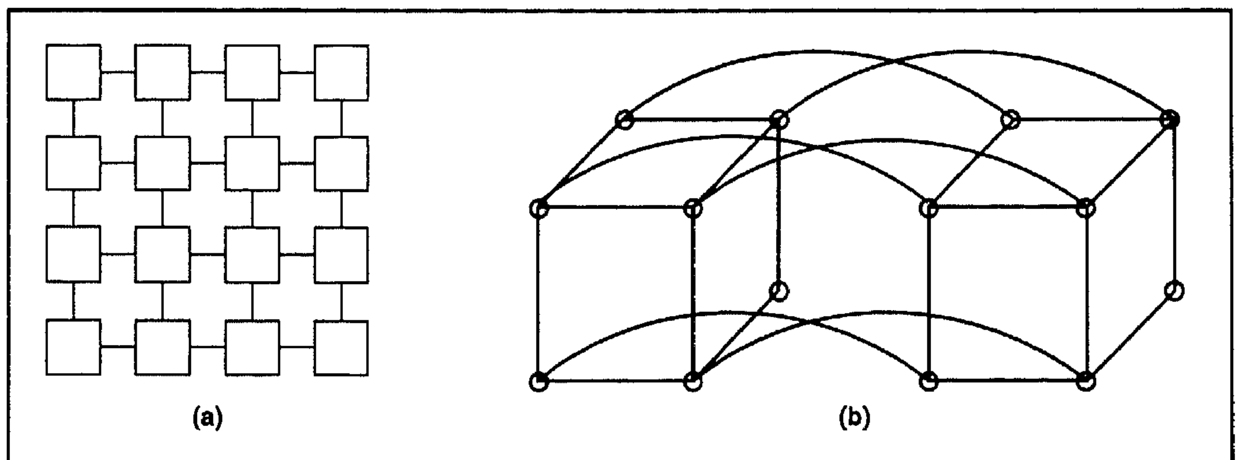

Figure 5. Multicomputers. (a) Grid. (b) Hypercube.

corresponding vertices connected, and so on. In general, an n-dimensional hypercube has 2^n vertices, each holding one CPU. Each CPU has a fan-out of n, so the interconnection complexity grows logarithmically with the number of CPUs.

Multicomputer software. Since by definition multicomputers do not contain shared memory, they must communicate by message passing. Various software paradigms have been devised to express message passing. The simplest one is to have two operating system primitives: *Send* and *Receive*. The *Send* primitive typically has three parameters: the destination address, a pointer to the data buffer, and the number of bytes to be sent. In its simplest form, the *Receive* primitive might just provide a buffer, accepting messages from any sender.

Many variations on this theme exist. For one, the primitives can be *blocking* (synchronous) or *nonblocking* (asynchronous). With a blocking *Send*, the sending process is suspended after the *Send* until the message has been actually accepted and acknowledged. With a nonblocking *Send*, the sender may continue immediately. The problem with allowing the sender to continue is that the sender may be in a loop, with messages being sent much faster than the underlying communication hardware can deliver them. The result may be lost messages. This problem may be alleviated somewhat by senders addressing messages not to processes but to *mailboxes*. A mailbox is a special kind of buffer that can hold multiple messages. However, overflow is still a possibility.

A fundamental problem with message passing is that conceptually it is really input/output. Many people believe that input/output should not be the central abstraction of a modern programming language. Birrell and Nelson [6] proposed a scheme called *remote procedure call (RPC)*, shown in Figure 6, to hide the bare input/output. The idea is to hide the message passing and make the

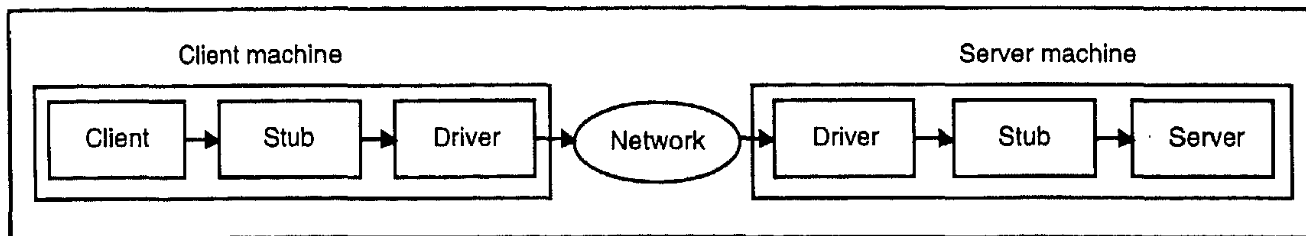

Figure 6. Remote procedure call (RPC) from a client to a server. The reply follows the same path in the reverse direction.

communication look like an ordinary procedure call. The sender, called the *client*, calls a *stub routine* on its own machine that builds a message containing the name of the procedure to be called and all the parameters. The stub then passes this message to the driver for transmission over the network. When the message arrives, the remote driver gives it to a stub, which unpacks the message and makes an ordinary procedure call to the *server*. The reply from server to client follows the reverse path. Because the client and server each think of the other as an ordinary local procedure, this scheme hides the message passing to some extent; however, making this scheme entirely transparent would be difficult. For example, passing pointers as parameters is difficult and passing arrays is costly. Thus, the programmer usually must be aware that the semantics of local and remote procedure calls are different.

In conclusion, the characteristics of multiprocessors and multicomputers differ significantly. Multiprocessors are hard to build but easy to program. In contrast, multicomputers are easy to build but hard to program. People want a system that is easy to build (that is, with no shared memory) and easy to program (that is, with shared memory). Reconciling these two contradictory demands is the subject of the rest of this paper.

NUMA machines

In developing intermediate designs, various researchers have tried to capture the desirable properties of both multiprocessor and multicomputer architectures. In most of these designs, an attempt was made to simulate shared memory on multicomputers. In all of the designs, a process executing on any machine can access data from its own memory without any delay. However, access to data located on another machine entails considerable delay and overhead, because a request message must be sent there and a reply must be received. In a system with *distributed shared memory*, a single address space is shared among otherwise disjoint machines.

Computers in which references to some memory addresses are cheap (that is, local) and others are expensive (that is, remote) have become known as *NUMA machines*. Three of the more interesting types of NUMA machines—*word-oriented*, *page-oriented*, and *object-oriented* machines —differ primarily in the granularity of access and the mechanism by which remote references are handled.

(1) Word-oriented NUMA machines. One of the earliest word-oriented NUMA machines was Cm*, built at Carnegie Mellon University, Pittsburgh, Pennsylvania. Cm* consisted of a collection of LSI-11 minicomputers [16]. Each LSI-11 had a microprogrammed memory management unit (MMU) and a local memory. The MMU microcode could be downloaded when execution began, allowing part of the operating system to be run there. The LSI-11s were grouped together into clusters, the machines in each cluster were connected by an intracluster bus, and the clusters were connected by intercluster buses.

When an LSI-11 referenced its own local memory, the MMU simply read or wrote the required word directly. However, when an LSI-11 referenced a word in a remote memory, the microcode

in the MMU built a request packet specifying which memory was being addressed, which word was needed, the opcode (*Read* or *Write*), and the value to be written for *Writes*. The packet was then sent out over the buses to the destination MMU via a store-and-forward network. At the destination, it was accepted and processed, and a reply containing the requested word (for *Read*) or an acknowledgment (for *Write*) was generated. The more remote the memory, the longer the operation was, with the worst case taking about 10 times as long as the best case. It was possible for a program to run out of remote memory entirely, with a performance penalty of a factor of about 10. There was no caching and no automatic data movement. It was up to the programmer to place code and data appropriately for optimal performance. Nevertheless, this system appeared — to the programmer—to have a single, shared address space accessible to all processors, even though — in actuality — it was implemented by an underlying packet-switching network.

(2) Page-oriented NUMA machines. Cm* represents one extreme — sending requests for individual words from MMU to MMU in "hardware" (actually MMU microcode) over a set of tightly coupled backplane buses. At the other extreme are systems that implement virtual shared memory on a collection of workstations on a LAN. As in Cm*, users are presented with a single, shared virtual address space, but the implementation is quite different.

In its simplest form, the virtual memory in page-oriented NUMA machines is divided up into fixed-size pages, with each page residing on exactly one processor. When a processor references a local page, the reference is done by the hardware in the usual way. However, when a remote page is referenced, a page fault occurs, and a trap to the operating system occurs. The operating system fetches the page, just as in a traditional virtual-memory system, only now the page is fetched from another processor (which loses the page) instead of from the disk.

As in Cm*, pages are fetched by request and reply messages, only here these messages are generated and processed by the operating system instead of by the MMU microcode. Since the overhead is so much higher, an entire page is transferred each time, in the hope that subsequent references will be to the same page. If two processors are actively using the same page at the same time, the page will thrash back and forth wildly, degrading performance.

Li and Hudak [14] proposed, implemented, and analyzed a significant improvement to the basic algorithm. In their design, thrashing is reduced by permitting read-only pages to be replicated on all the machines that need them. When a read-only page is referenced, a copy is made instead of the page being sent, so the original owner may continue using it. Other pages may also be shared, but when a page is written, other copies of it are invalidated.

Li and Hudak [14] presented several algorithms for locating pages. In the simplest, a centralized manager keeps track of the location of all pages. All page requests are sent to the manager, which then forwards them to the processor holding the page. A variation of this scheme is to have multiple managers, with the leftmost *n* bits of the page number telling which manager is responsible for the page. This approach spreads the load over multiple managers. Even more decentralized page-location schemes are possible, including the use of hashing or broadcasting.

It is worth pointing out that the basic page-oriented strategy could (almost) have been used with Cm*. For example, if the MMUs noticed that a particular page were being heavily referenced by a single remote processor and not at all referenced by its own processor, the MMUs could have decided to ship the whole page to the place it was being used instead of constantly making expensive remote references. This approach was not taken because the message-routing algorithm used the page number to locate the page.

To reduce thrashing, NUMA machines have been designed so that a page may be transported a maximum of only *k* times. After that, the page is wired down, and all remote references to it are done as in Cm*. Alternatively, one can have a rule saying that once moved, a page may not be moved again for a certain time interval.

(3) Object-oriented NUMA machines. An inherent problem with page-oriented NUMA

systems is that the amount of data transferred on each fault is fixed at exactly one page. Usually this is too much, sometimes it is too little, but hardly ever is it exactly right. The next step in the evolution of NUMA machines was an object-based system, in which the shared memory contains well-defined objects, each one with certain operations defined on it. When a process invokes an operation on a local object, it is executed directly, but when it invokes an operation on a remote object, the object is shipped to the invoker. Alternatively, the operation name and parameters can be sent to the object, with the operation being carried out on the remote processor and the result being sent back. In both of these schemes, no unnecessary data are moved (unlike in the page-oriented NUMA machines, which move one K or four K or eight K to fetch even a single byte).

It should be noted that our definition of a NUMA machine is somewhat broader than that some other writers have used. We regard as a NUMA machine any machine that presents the programmer with the illusion of shared memory but implements it by sending messages across a network to fetch chunks of data. Whether the code that implements this is in the MMU — as in Cm* — or in the operating system — as in the work of Li and Hudak — is simply an implementation decision. Similarly, whether the unit fetched is a word, a page, or an object is also an implementation decision. The essential commonality of NUMA machines is that when a remote read is made, the system sees an exception, sends a message to the remote memory, gets back a reply containing the data needed, and restarts the instruction. In this view, Cm* and the model of Li and Hudak simply differ in where the exception handler runs and the block size returned on a fault.

Weakened semantics

A completely different approach to implementing distributed shared memory is to weaken the semantics, giving up the demand for absolute coherence. For example, suppose three processes — Processes *A*, *B*, and *C* — all simultaneously update a word, followed by Process *D* reading it. If total coherence is required, the word will probably have to be sent first to *A*, then to *B*, then to *C*, and finally to *D* to carry out the three writes and the read. However, from *D*'s point of view, it has no way of knowing which of the three writers went last (and thus whose value did not get overwritten). In the Munin system, Bennett, Carter, and Zwaenepoel [4] proposed that returning any one of the values written should be legal. The next step is to delay remote writes until a read is done that might observe the values written. It may even be possible to perform an optimization and avoid some of the writes altogether. This strategy was implemented in the Munin system, which uses a form of strong typing on shared objects to make it easier to perform these and other optimizations.

Other systems go even farther in this direction. For example, Mether [15] maintains a primary copy of each page and possibly one or more secondary copies. The primary copy is writable; the secondary copies are read-only. If a write is done to the primary copy, the secondary copies become obsolete and stale data are returned by reads to them. Applications simply have to accept this. In Mether, three ways are provided for getting resynchronized: The owner of a secondary copy can ask for an up-to-date copy; the owner of the primary copy can issue an up-to-date copy to the readers; or the owner of a secondary copy can just discard it, getting a fresh copy on the next read.

A new model

Although weakening the semantics of the shared memory may improve the performance, it has the disadvantage of ruining the ease of programming that the shared-memory model was designed to provide. In effect, it offers only the syntax of shared variables, while forcing the programmer to deal with semantics even less convenient than message passing.

We have devised an alternative model that preserves the coherency of (object-based) shared

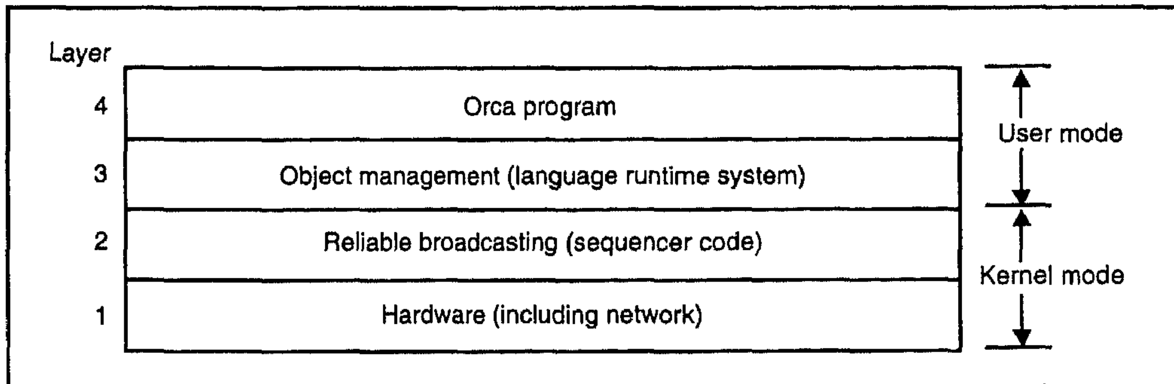

Figure 7. The layer structure of reliable broadcasting.

memory yet has been implemented efficiently. Our model, which we call *reliable broadcasting*, consists of four layers, as shown in Figure 7.

Layer 1 is the bare CPU and networking hardware. Our scheme is most efficient on networks that support *broadcasting* (sending a message to all machines) or *multicasting* (sending a message to a selected group of machines) in hardware, but supporting broadcasting or multicasting is not required. Ethernet, earth satellites, and cellular radio are examples of networks having broadcasting or multicasting. (For simplicity, we henceforth use the term *broadcasting* to mean either one.) Broadcasting is assumed to be *unreliable*; that is, it is possible for messages to be lost.

Layer 2 is the software necessary to turn the unreliable broadcasting offered by Layer 1 into reliable broadcasting. It is normally part of the operating system kernel. As a simple example of a possible (but highly inefficient) protocol, reliably broadcasting a message to n machines can be done by having the kernel send each machine, in turn, a point-to-point message and then wait for an acknowledgment. This protocol takes $2n$ messages per reliable broadcast. Below, we describe a different protocol that takes (on the average) a fraction more than two messages per reliable broadcast, instead of $2n$. The main issue to understand is that when Layer 3 hands a message to Layer 2 and asks for it to be reliably broadcast, Layer 3 does not have to worry about how this is implemented or what happens if the hardware loses a message. All of this is taken care of by Layer 2.

In addition to its inefficiency, the protocol that sends a point-to-point message to every machine has a more serious problem. When two machines—Machines A and B—simultaneously do a broadcast, the results can be interleaved. That is, depending on network topology, lost messages, and so on, some machines may get the broadcast from Machine A before that from Machine B and other machines may receive the broadcasts in the reverse order. This property makes programming difficult. In the protocol that we describe below, this cannot happen. Either all the machines get Machine A's broadcast and then B's broadcast or all the machines get B's broadcast and then A's broadcast. Broadcasts are globally ordered, and it is guaranteed that all user processes get them in the same order.

Layer 3 is the language runtime system, which is usually a set of library procedures compiled into the application program. Conceptually, programmers can have variables and objects can be *Private* or *Shared*. *Private* objects are not visible on other machines, so they can be accessed by direct memory reads and writes. *Shared* objects are replicated on all machines. Reads to them are local, the same as reads to *Private* objects. Writes are done by reliable broadcasting. Objects are entirely passive; they contain only data, not processes.

Layer 4 provides language support. Although it is possible for programmers to use the distributed shared memory by making calls directly on Layer 3, having language support is much more convenient. We designed a language, called *Orca* [1], for parallel programming using distributed shared objects and implemented a compiler for it. In Orca, programmers can declare shared objects,

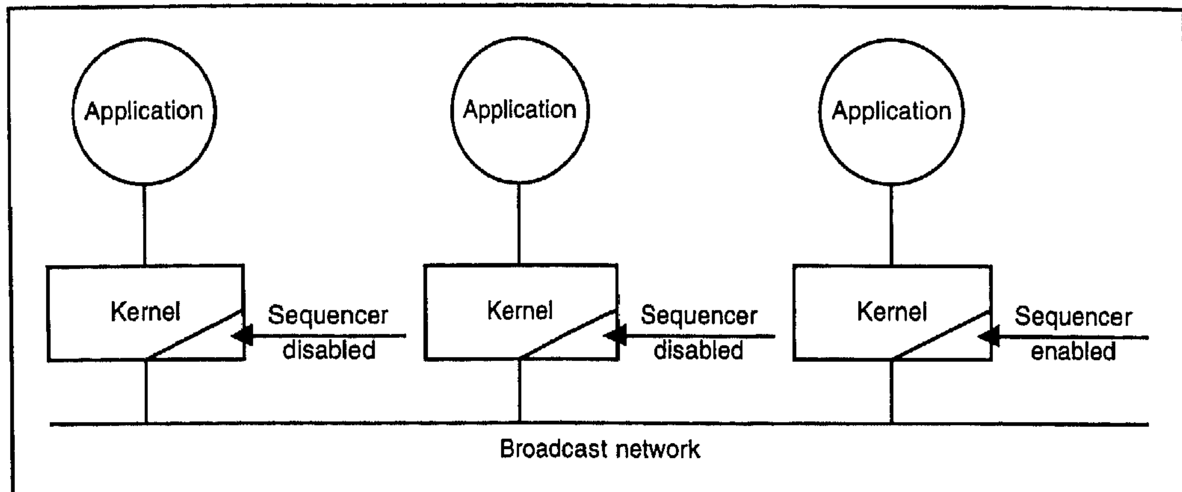

Figure 8. System structure. Each kernel is capable of becoming the sequencer; however, at any instant, only one of them functions as sequencer.

with each shared object containing a data structure for the object and a set of procedures that operate on it. Operations on shared objects are indivisible and serializable. In other words, when multiple processes update the same object simultaneously, the final result is as if the updates had been performed sequentially, in some unspecified order, with each update having been completed before the next one began.

Reliable broadcasting

The heart of our proposal is the efficient implementation of indivisible, reliable broadcasting (Layer 2). Once that has been achieved, the rest is relatively straightforward. In this section, we describe the mechanism used to achieve reliable broadcasting (in software) over an unreliable network. (Kaashoek et al. [12] described this mechanism more comprehensively.)

The hardware/software configuration required for reliable broadcasting is shown in Figure 8. The circles represent the application programs and their runtime systems (Layers 3 and 4). The kernel represents Layer 2. The network is Layer 1. The hardware of all the machines is identical, and they all run exactly the same kernel and application software. However, when the application starts up, one of the machines is elected as sequencer, like a committee electing a chairman [13]. If the sequencer machine subsequently crashes, the remaining members elect a new one. (Many election algorithms are known. For example, the one used in the IEEE 802.4 Token Bus standard is to pick the machine with the highest network address. We do not discuss this point further here.)

The actual sequence of events involved in achieving reliable broadcasting can be summarized as follows:

- The user traps to the kernel, passing it the message.
- The kernel accepts the message and blocks the user.
- The kernel sends the message to the sequencer using an ordinary point-to-point message.
- When the sequencer gets the message, it allocates the next available sequence number, puts the sequencer number in a header field reserved for it, and broadcasts the message (and sequence number).
- When the sending kernel sees the broadcast message, it unblocks the calling process to let it continue execution.

Let us now consider these events in more detail. When an application process executes a *Broadcast* primitive, a trap to its kernel occurs. The kernel then blocks the caller and builds a message containing a kernel-supplied header and the application-supplied data. The header contains the message type (*Request for Broadcast*, in this case), a unique message identifier (used to detect duplicates), the number of the last broadcast received by the kernel (a kind of piggybacked acknowledgment), and some other information. The kernel sends the message to the sequencer using a normal point-to-point message and simultaneously starts a timer. If the broadcast comes back before the timer runs out (as happens in practice well over 99 percent of the time, because LANs are highly reliable), the sending kernel stops the timer and returns control to the caller.

On the other hand, if the broadcast has not come back before the timer expires, the kernel assumes that either the message or the broadcast has been lost. Either way, it retransmits the message. If the original message was lost, no harm has been done, and the second (or subsequent) attempt will trigger the broadcast in the usual way. If the message got to the sequencer and was broadcast, but the sender missed the broadcast, the sequencer will detect the retransmission as a duplicate (from the message identifier) and simply tell the sender that everything is all right. The message is not broadcast a second time.

A third possibility is that a broadcast comes back before the timer expires, but it is the wrong broadcast. This situation arises when two processes attempt to broadcast simultaneously. One of them, Process *A*, gets to the sequencer first, and its message is broadcast. Process *A* sees the broadcast and unblocks its application program. However, its competitor, Process *B*, sees Process *A*'s broadcast and realizes that it has failed. Nevertheless, Process *B* knows that its message probably got to the sequencer (since lost messages are rare), where it will be queued and broadcast next. Thus, Process *B* accepts Process *A*'s broadcast and continues to wait for its own broadcast to come back or its timer to expire.

Let us consider what happens at the sequencer when a *Request for Broadcast* arrives there. First, a check is made to see if the message is a retransmission; if so, the sender is informed that the broadcast has already been done, as mentioned above. If the message is new (as is the normal case), the next sequence number is assigned to it, and the sequencer counter is incremented by one for next time. The message and its identifier are then stored in a *history buffer*, and the message is then broadcast. (The management of the history buffer is described shortly.) The message is also passed up to the application running on the sequencer's machine (because the broadcast does not cause an interrupt on the machine that issued the broadcast).

Now, let us consider what happens when a kernel receives a broadcast. First, the sequence number is compared to the sequence number of the most recently received broadcast. If the new one is one higher (as is the normal case), no broadcasts have been missed, so the message is passed up to the application program. If the application program is multithreaded, usually one thread will be waiting for the message; if it is single threaded, the kernel waits for the program to try to receive a message.

Suppose that the newly received broadcast has sequence number 25, while the previous one had number 23. The kernel is immediately alerted to the fact that it has missed number 24, so it sends a point-to-point message to the sequencer asking for a private retransmission of the missing message. The sequencer fetches the missing message from its history buffer and sends it. When it arrives, the receiving kernel processes 24 and 25, passing them up to the application program in numerical order. Thus, the only effect of a lost message is a minor time delay. All application programs see all broadcasts in the same order, even if some messages are lost.

The reliable-broadcast protocol is illustrated in Figure 9. Here, the application program running on Machine *A* passes a message *M* to its kernel for broadcasting. The kernel sends the message to the sequencer, where it is assigned sequence number 25. The message (containing the sequence number 25) is now broadcast to all machines and also passed up to the application program running on the sequencer itself. This broadcast message is denoted by *M*25 in the figure.

Message *M*25 arrives at Machines *B* and *C*. At Machine *B*, the kernel sees that it has already

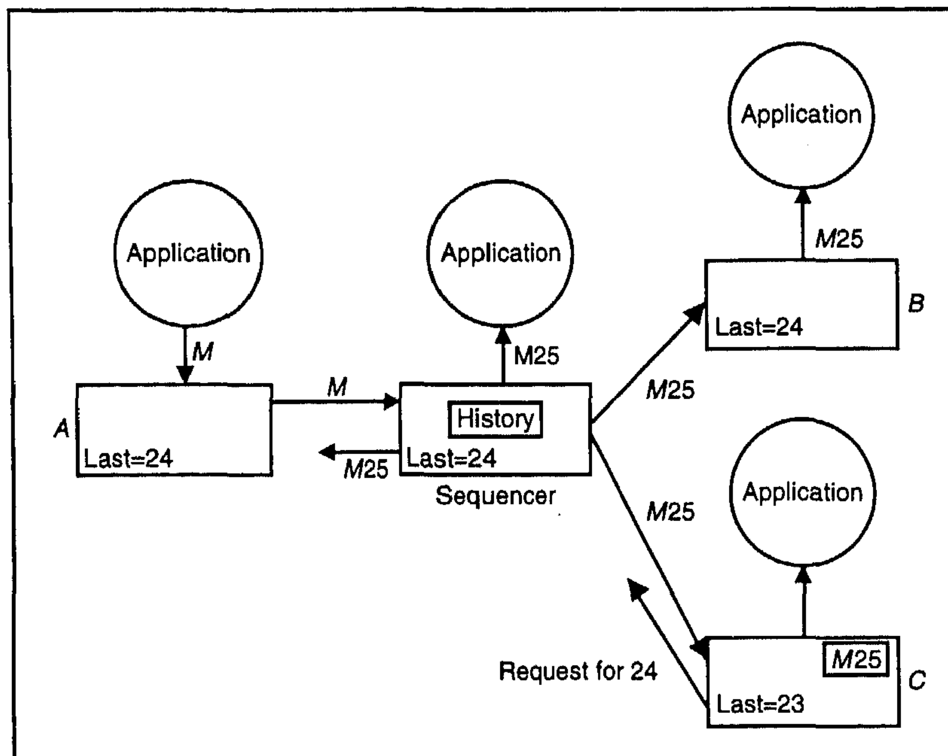

Figure 9. The application of Machine *A* sends a message *M* to the sequencer, which then adds a sequence number (25) and broadcasts it. At *B* it is accepted, but at *C* it is buffered until 24, which was missed, can be retrieved from the sequencer.

processed all broadcasts up to and including 24, so it immediately passes *M*25 up to the application program. However, at *C*, the last message to arrive was 23 (24 must have been lost), so *M*25 is buffered in the kernel, and a point-to-point message requesting 24 is sent to the sequencer. Only after the reply has come back and been given to the application program will *M*25 be passed upward as well.

Now, let us look at the management of the history buffer. Unless something is done to prevent it, the history buffer will quickly fill up. However, if the sequencer knows that all machines have correctly received broadcasts — say zero through 23 — it can delete these from its history buffer. Several mechanisms are provided to allow the sequencer to discover this information. The basic one is that each *Request for Broadcast* message sent to the sequencer carries a piggybacked acknowledgment *k*, meaning that all broadcasts up to and including *k* have been correctly received. This way, the sequencer can maintain a piggyback table, indexed by machine number, telling for each machine which broadcast is the last one received. Whenever the history buffer begins to fill up, the sequencer can make a pass through this table to find the smallest value. It can then safely discard all messages up to and including this value.

To inform the sequencer, a machine is required to send a short acknowledgment message when it has sent no broadcast messages for a certain period of time. Otherwise, if one machine happens to be silent for an unusually long period of time, the sequencer will not know what its status is. Furthermore, the sequencer can broadcast a *Request for Status* message, which directs all other machines to send it a message giving the number of the highest broadcast received in sequence. In this way, the sequencer can update its piggyback table and then truncate its history buffer. In practice, *Request for Status* messages are rare; however, when they do occur, the mean number of messages required for a reliable broadcast is raised to slightly above two, even when there are no lost messages. This effect increases slightly as the number of machines grows.

We would like to clarify a subtle design point concerning the reliable-broadcast protocol. There are two methods for doing the broadcast. In Method 1 (described above), the user sends a point-to-point message to the sequencer, which then broadcasts it. In Method 2, the user broadcasts the message, including a unique identifier. When the sequencer sees this, it broadcasts a special *Accept* message containing the unique identifier and its newly assigned sequence number. A broadcast is only "official" when the *Accept* message has been sent. Although these protocols are logically equivalent, they have different performance characteristics. In Method 1, each message appears in full on the network twice: once to the sequencer and once from the sequencer. Thus, a message of length m bytes consumes $2m$ bytes worth of network bandwidth. However, only the second of these is broadcast, so each user machine is interrupted only once (for the second message). In Method 2, the full message appears only once on the network, in addition to a very short *Accept* message from the sequencer; therefore, only half the bandwidth is consumed. On the other hand, every machine is interrupted twice: once for the message and once for the *Accept*. Thus, Method 1 wastes bandwidth to reduce interrupts compared to Method 2. We have implemented both methods and are now running experiments comparing them. Depending on the results of these experiments, we may go to a hybrid scheme that uses Method 1 for short messages and Method 2 for long ones.

In summary, our reliable-broadcast protocol allows reliable broadcasting to be done on an unreliable network in just over two messages per reliable broadcast. Each broadcast is indivisible, and all applications receive all messages in the same order, no matter how many are lost. The worst that can happen is that a short delay is introduced when a message is lost, which rarely occurs. If two processes attempt to broadcast at the same time, one of them will get to the sequencer first and win. The other will see a broadcast from its competitor coming back from the sequencer and will realize that its request has been queued and will appear shortly, so it simply waits.

Comparison

Kaashoek et al. [12] gave a detailed comparison of our scheme with other published protocols for doing reliable broadcasting. Here, we compare our scheme with some of the major other ones. Three other kinds of work to which we can compare ours are

- Distributed shared memory,
- Broadcasting to achieve fault tolerance, and
- Broadcasting as an operating system service.

Most research on distributed shared memory is based on the work of Li and Hudak [14]. In their method, fixed-size pages are moved around the network in point-to-point messages. If broadcasting is used at all, it is only for invalidating pages. In contrast, broadcasting is the basis of the entire system in our method. Furthermore, messages are variable sized, containing only the data that are needed. Most of the time, the amount of data needed is much less than a page; therefore, being able to send short messages is a significant performance improvement.

Some recent work in distributed shared memories emphasized weakening the semantics of what shared memory means [15]. We consider this unacceptable, since the goal is to give the programmer an easy-to-understand model to work with, and weak-semantics systems put all the burden of coherency back on the programmer. Other work concentrated on using broadcasting to achieve fault tolerance. Chang and Maxemchuk [8] described a family of protocols for reliable broadcasting, of which nonfault tolerance is a special case. Like our protocol, their protocol for the nonfault-tolerant case uses a sequencer, but they have the nodes ordered in a logical ring, with the sequencer advancing along the ring with every message sent. This motion of the sequencer is almost free when the traffic is heavy, but costs an extra message per broadcast when it is not. On the average, their protocol

requires two to three messages per reliable broadcast, whereas ours requires just a fraction over two. In addition, their protocol uses more storage: They have to store the history buffer on all machines, because of the moving sequencer. With a one-megabyte (Mbyte) history buffer and 100 machines, we need one Mbyte of memory for the history buffer and they need 100 Mbytes.

Finally, in our Method 1, each reliable broadcast uses one point-to-point message and one broadcast message. With $n >> 1$ machine, we generate about n interrupts per reliable broadcast. In their protocol (similar to what we have called Method 2), all messages are broadcast, so they need between $2n$ and $3n$ interrupts per reliable broadcast. When there are hundreds of broadcasts per second, their scheme uses much more CPU time than ours.

Another family of fault-tolerant protocols is the Isis system of Birman and Joseph [5]. They used a distributed two-phase commit protocol to achieve global ordering, something we achieve in two messages per broadcast. Since they realized that their protocol is inefficient, they proposed alternative protocols with weaker semantics. Our method shows that it is not necessary to weaken and complicate the semantics to achieve efficiency. However, in Isis's favor, it does provide a high degree of fault tolerance, which our basic protocol does not. Our method can also provide fault tolerance, if desired [11]. Furthermore, Isis supports simultaneous use of overlapping process groups, whereas — in our scheme — groups are disjoint. Each application forms its own group, and no attempt is made to provide global ordering between different applications.

A few researchers proposed providing broadcasting as an operating system service. For example, the system of Cheriton and Zwaenepoel [9] supports broadcasting, but it does not guarantee reliability. A problem with their system is that the programmer's job is made more difficult than that in a system in which broadcasts are guaranteed to be reliable.

Performance of reliable broadcasting

We modified the Amoeba kernel to support our reliable-broadcast protocol. Amoeba runs on a collection of 16-MHz Motorola 68030 processors connected by a 10-megabit- (Mbit)-per-second Ethernet. We measured the performance of this system by running various experiments on it. First, to measure the maximum broadcast rate by a single process, we had one process continuously broadcast null messages as fast as it could. In this experiment, we achieved 370 reliable broadcasts per second with up to 16 processors. Note that this experiment tested the worst possible case. Since all machines but one were silent, they were not sending back piggybacked acknowledgments to the sequencer. Without these acknowledgments, the sequencer had to send out a *Request for Status* every 64 messages. It is instructive to point out that even though we were measuring broadcasting, the performance was better than that for most (point-to-point) RPC systems, partly because our protocol requires only two messages per broadcast, whereas RPCs often require three, with the last being an acknowledgment. (As an aside, these results differed from our earlier results, because we had designed and implemented a new kernel that can handle broadcasting over an arbitrary internetwork consisting of LANs and buses. Also, this scheme supports transparent process migration and automatic network reconfiguration and management. The differences between the results here and the previously published ones are entirely accounted for by these changes.)

Second, to see the effect of contention, we had not one but multiple processes broadcasting at the same time. In this experiment, we varied the number of senders from two to 14, with the number in the receiving group equal to the number of senders. Table 1 gives the results of our second experiment. Multiple senders get a higher throughput than just one, because if two machines send messages to the sequencer almost simultaneously, the first one to arrive will be broadcast first, but the second one will be buffered and broadcast immediately afterward. This simple form of pipelining increases the parallelism of the system and thus increases the broadcast rate. As the number of senders increases, performance drops slightly because of contention for the Ethernet. (We were unable to make consistent

measurements for 15 and 16 processors because of technical limitations of our equipment.)

A word about scaling is appropriate here. It has been pointed out to us that the sequencer will become a bottleneck in very large systems. While this is true, few current systems or applications suffer from this limit. With current technology (two-million-instructions-per-second [MIPS] CPUs and 10-Mbit-per-second Ethernet), we can support on the order of 800 reliable broadcasts per second, as shown above. Because broadcast messages are usually short, there is bandwidth to spare. For many problems, the read-to-write ratio is quite high. Suppose (conservatively) that 90 percent of the operations are reads and 10 percent are writes. Then, we can support 8000 operations per second on shared objects (7200 reads, done locally, and 800 writes, done by broadcasting). We know of very few applications that would be hindered by being able to perform only 8000 operations per second on shared objects. Furthermore, with 20-MIPS RISC processors and 100-Mbit-per-second fiber-optic networks, this limit will be closer to 80,000 operations per second on shared objects within a few years.

Table 1. Performance with multiple processes broadcasting at the same time.

Number of senders	Broadcasts per second
2	714
3	769
4	769
5	793
6	789
7	795
8	800
9	782
10	781
11	780
12	780
13	718
14	710

Object management

On top of the broadcast layer is the object-management layer, implemented by a package of library procedures (in user space). This layer manages shared objects. Our design is based on the explicit assumption that shared objects are read much more often than they are written. Ratios of 10 to one or even 100 to one are not at all unusual. Therefore, we have chosen to replicate each object on all machines that use the object. (Note that multiple, independent applications may be running at the same time, so not every machine needs every object.) All replicas have equal status: There is no concept of a primary object and secondary copies of it.

Two operations are defined on objects: *Read* and *Write*. *Reads* are done on the local copy, without any network traffic. A *Read* on a shared object is only slightly more expensive than a *Read* on a *Private* object (because of some locking). A *Write* to a shared object can be done by sending a reliable broadcast with the new value of the object, in whole or in part, or it can be done by sending an operation code and some parameters, letting each machine recompute the new value. The former strategy is attractive for small objects and the latter is attractive for large objects; it is up to the runtime system to pick one. Since all machines process all broadcasts in the same order, all objects will settle down to the same value when equilibrium is reached.

This scheme does not provide complete memory coherence, because if Machine *A* initiates a reliable broadcast to update a shared object and Machine *B* reads the (local copy of the) object one ns later, *B* will get the old value. On the other hand, it does provide for indivisible update and serializability, which is almost as good, as can easily be seen in the example that follows. Consider a multiprocessor with a true shared memory. At a certain moment, Process *A* wants to write a word and Process *B* wants to read it. Since the two operations may take place a microsecond apart, the value read by *B* depends on who went first. Despite the memory being coherent, the value read by *B* is

determined by the detailed timing. Our shared-object model has a similar property. In both cases, programs whose correct functioning depends on who wins the race to memory are living dangerously, at best. Thus, although our memory model does not exhibit true coherence, serializability plus global message ordering are in reality sufficient properties, and our model does have these properties.

Orca

While it is possible to program directly with shared objects, it is much more convenient to have language support for them. For this reason, we have designed the Orca parallel programming language and written a compiler for it. The following are the four guiding principles behind the Orca design:

(1) *Transparency.* With transparency, programs (and programmers) should not be aware of where objects reside. Location management should be fully automatic. Furthermore, the programmer should not be aware of whether the program is running on a machine with physical shared memory or one with disjoint memories. The same program should run on both, unlike nearly all other languages for parallel programming, which are aimed at either one or the other, but not both. (Of course, one can always simulate message passing on a multiprocessor, but this is far from optimal.)
(2) *Semantic simplicity.* With semantic simplicity, programmers should be able to form a simple mental model of how the shared memory works. This principle rules out incoherent memory, in which reads to shared data sometimes return good values and sometimes return stale (incorrect) ones.
(3) *Serializability.* In a parallel system, many events happen simultaneously. By making operations serializable, we guarantee that operations on objects are indivisible and that the observed behavior is the same as some sequential execution would have been. Operations on objects are guaranteed not to be interleaved, which contributes to semantic simplicity, as does the fact that all machines see the same sequence of serial events. Of course, the compiler and runtime system do not actually serialize events any more than they have to guarantee the semantics.
(4) *Efficiency.* Efficiency is also important, since we are proposing a system that can actually be used for solving real problems.

Now let us look at the principal aspects of Orca that relate to parallelism and shared objects. Orca is a procedural language whose sequential constructs are roughly similar to those of languages like C or Modula 2. Parallelism is based on two orthogonal concepts: *processes* and *objects.* Processes are active entities that execute programs. They can be created and destroyed dynamically. It is possible to read in an integer *n* and then execute a loop *n* times, creating a new process on each iteration. Thus, the number of processes is not fixed at compile time, but is determined during execution.

The Orca construct for creating a new process is the

fork func(param, ...)

statement, which creates a new process running the procedure *func* with the specified parameters. The user may specify which processor to use or let the runtime system use its own load-balancing heuristics. Objects may be passed as parameters (call by reference). A process may fork many times, passing the same objects to each of the children. This is how objects come to be shared among a collection of processes. There are no global objects in Orca.

Objects are passive; they do not contain processes or other active elements. Each object contains some data structures, along with definitions of one or more operations that use the data structures. The operations are defined by Orca procedures written by the programmer. An object

has a specification part and an implementation part; in this respect, Orca is similar to Ada packages or Modula 2 modules.

A common way of programming in Orca is the *replicated worker paradigm* [7]. In this model, the main program starts out by creating a large number of identical worker processes, each getting the same objects as parameters, so they are shared among all the workers. Once the initialization phase has been completed, the system consists of the main process, along with some number of identical worker processes, all of which share some objects. Processes can perform operations on any of their objects whenever they want to, without having to worry about all the mechanics of how many copies are stored and where, how updates take place, which synchronization technique is used, and so on. As far as the programmer is concerned, all the objects are effectively located in one big shared memory somewhere, but they are protected by a kind of monitor that prevents multiple updates to an object at the same time.

As a minimal example of an object specification, consider a simple object consisting of an integer variable with two operations on it: *read* and *write*. If this object is subsequently shared among multiple processes, any of them can read or write the value of the integer. The Orca specification part looks like the following:

```
object specification SharedInt;
        operation read(): integer;
        operation write(val: integer): integer;
end;
```

The implementation part looks like the following:

```
object implementation SharedInt;
        n: integer; # the value

        operation read(): integer;
        begin
                return n;
        end;

        operation write(val: integer);
        begin
                n := val;
        end;
end;
```

To declare and use an object of type *SharedInt*, the programmer might write the following:

```
s: SharedInt;
i: integer; # ordinary integer

s$write(100); # set object to 100
i := s$read(); # set i to 100
```

Although the programming style suggested by this trivial example is sufficient for some programs, for many others some kind of synchronization method is required. A common example is *barrier*

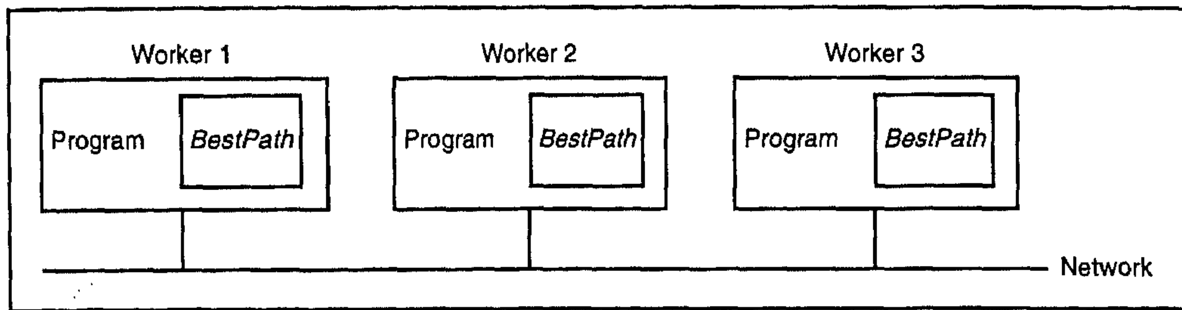

Figure 10. Model of the traveling salesman problem (TSP) program using the replicated worker paradigm.

synchronization, in which *n* workers are busy computing something, and only when all of them have finished may the next step begin. Synchronization in Orca is handled by *guarded commands*, in which an operation consists of a number of (guard, statement) pairs. Each guard is a side-effect-free Boolean expression and each statement is an arbitrary piece of sequential Orca code. When the operation is invoked, the language runtime system evaluates the guards one at a time (in an unspecified order); as soon as it finds one that is true, the corresponding statement is executed. For example, to implement barrier synchronization, the main process could create a shared object with operations to initialize it, increment it, and synchronize on it (that is, wait until it reached *n*). The main process would *initialize* it to zero and then fork off all the workers. When each worker had finished its work, it would invoke the *increment* operation and then the *synchronize* operation. The latter would block the calling process until the value reached *n*, at which point all the processes would be released to start the next phase.

As a slightly more elaborate example, consider an implementation of the *traveling salesman problem (TSP)* in Orca. In this problem, the computer is a given a starting city and a list of cities to visit; it has to find the shortest path that visits each city exactly once and returns to the starting city. Usually, the TSP is solved using the branch-and-bound algorithm. Suppose the starting city is New York and the cities to be visited are London, Sydney, Tokyo, Nairobi, and Rio de Janeiro. The main program begins by computing some possible path (for example, using the closest-city-next algorithm) and determining its length. Then, it initializes a soon-to-be-shared object, *BestPath*, containing this path and its length. As the program executes, this object will always contain the best path found so far and its length. Next, it forks off some number of workers, each getting the shared object as parameter, as shown in Figure 10.

Each worker is given a different partial path to investigate. The first worker tries paths beginning New York-London, the second one tries paths beginning New York-Sydney, the third one tries paths beginning New York-Tokyo, and so on. Very roughly, the algorithm used by a worker that is given a partial path plus *k* cities to visit is to first check if the length of the partial path is longer than the best complete path found so far. If so, the process terminates itself. If the partial path is still a potential candidate, the process generates *k* new partial paths to be investigated — one per city in the list — and forks off these to *k* new workers. To avoid forking off too many processes, when the list of remaining cities to be visited is less than *n*, the process tries all the possible combinations itself. Other optimizations are also used. Whenever a new best path is found, an update operation on the shared object *BestPath* is executed to replace the previous best path by the new one. When the Orca compiler detects an assignment to a variable in a shared object, it generates code to cause the runtime system to issue a reliable broadcast of the new value. In this manner, all the details of managing shared objects are hidden from the programmer.

A moment's thought reveals that reads of *BestPath* will occur very often, while writes will hardly occur at all, certainly not after the program has been running for a while and has found a path close

to the optimal one. Remember that reads are done entirely locally on each machine, whereas writes require a reliable broadcast. The net result is that the vast majority of operations on the shared object do not require network traffic, and the few that do take only two messages. Consequently, the solution is highly efficient. We have achieved almost linear speedup with 10 processes. In fact, actual measurements showed the distributed version to outperform one run on a shared-memory machine (without caching), because the latter suffers from serious memory contention as large numbers of processors constantly try to read the memory word containing the length of the best path so far.

Comparison

It is instructive to briefly compare Orca with some alternative approaches to parallel programming. A large number of languages are based on message passing, which is of a conceptually lower level than shared objects. The approach usually used in languages that support shared memory is the use of semaphores or monitors to protect critical regions. Both work adequately on shared-memory machines with a small number of processors but poorly on large distributed systems, because they use a locking scheme that is inherently centralized.

A system that is somewhat similar to Orca is Emerald [10]. Like Orca, Emerald supports shared objects. However, unlike Orca, it does not replicate objects. This means that when a caller on Machine 1 invokes an operation on an object located on Machine 2 using a parameter on Machine 3, messages must be sent to collect all the necessary information in the same place. There is no automatic migration, so if the programmer does not arrange for things that go together to be colocated, execution will be inefficient. Alternatively, for efficiency, the programmer can specify on each call that the parameter objects are to be sent to the machine where the object resides and to remain there after the invocation. The programmer can also specify whether or not the result is to remain there. In a truly transparent system, these issues would not arise.

In Orca, in contrast, the runtime system decides whether or not it wants to replicate objects and, if so, where. When the broadcast system described above is used, all objects are replicated on machines that need them (but other Orca implementations do it differently). In any event, object management is not the programmer's responsibility; it is handled automatically, and the decision on whether to perform operations locally or remotely is made by the system. The Orca approach comes much closer than does Emerald to providing the semantics of a shared-memory multiprocessor.

Yet another shared-object scheme is Linda [7]. Linda supports the concept of a shared tuple space that is equally accessible to processes on all machines. Linda is fully location-transparent, but the primitives are low-level: inserting and deleting tuples. In Orca, in contrast, the operations on shared objects can be simple or arbitrarily complex, as the programmer wishes.

In summary, Orca supports the model of having dynamically created parallel processes perform user-defined operations on shared objects. These objects are implemented by maintaining identical copies on all machines. An operation that reads an object—but does not change it—is done entirely locally, with no network traffic. The efficiency of this operation is only slightly worse than that of an operation on an unshared object.

An operation that modifies an object is handled differently. The operation name and parameters are put in a message that is passed to the runtime system (object-management layer) for transmission to all other machines in one indivisible reliable broadcast. The method by which the kernel provides this abstraction is discussed in detail above. When the message arrives, the operation is carried out locally. This method is preferable to carrying out the operation on the sending machine and broadcasting the new object to reduce the size of the message. This division of labor between the layers yields a great conceptual simplicity: The programmer defines and invokes operations on shared objects, the runtime system handles reads and writes on these objects, and the reliable-broadcast layer implements indivisible updates to objects using the sequencer protocol.

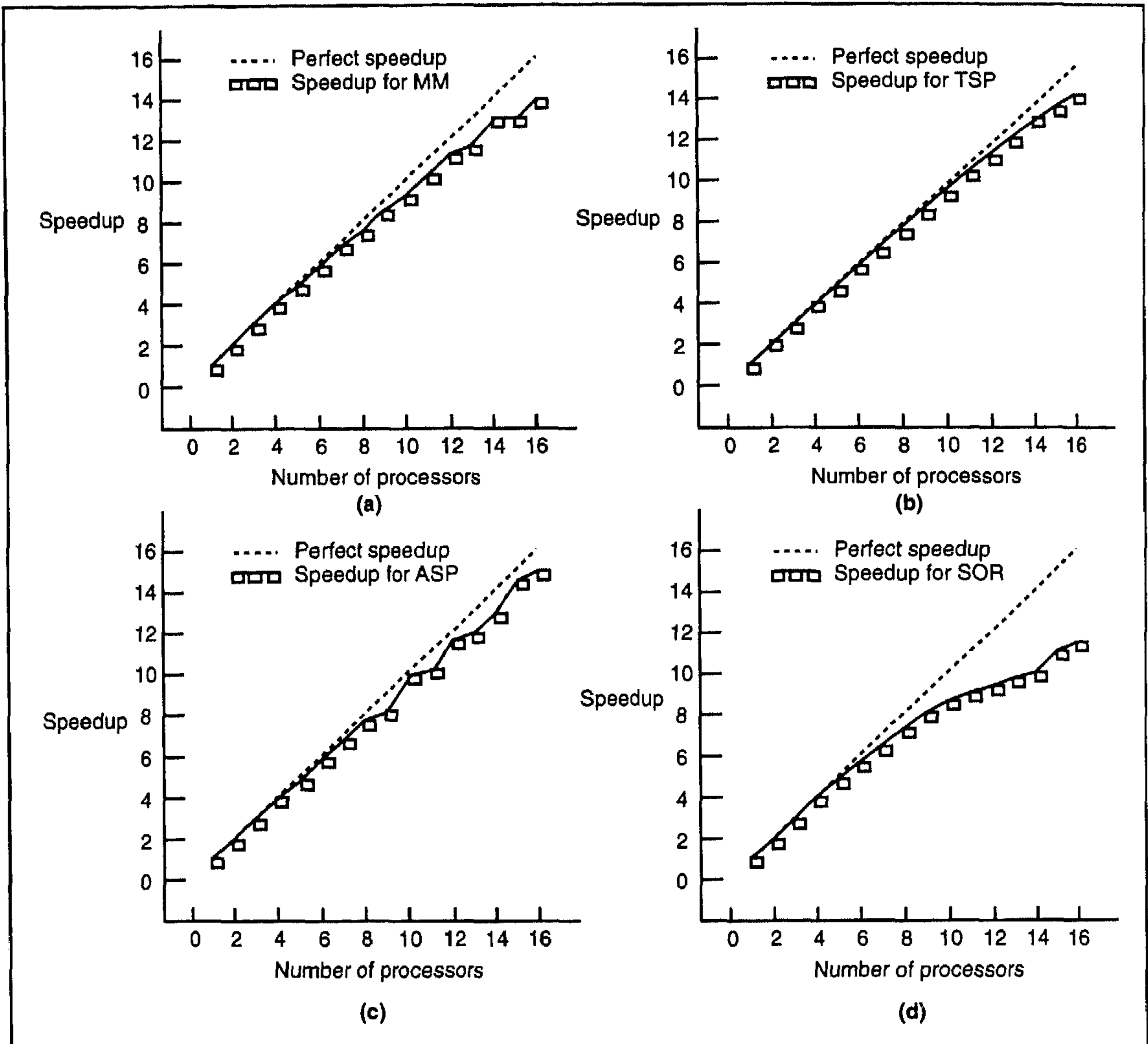

Figure 11. Performance (measured speedup) for four Orca programs. Each graph shows the speedup of the parallel Orca program on *N* CPUs over the same program running on one CPU. (a) Matrix multiplication (MM), using input matrices of size 250 × 250. (b) The traveling salesman problem (TSP), averaged over three randomly generated graphs with 12 cities each. (c) The all-pairs shortest paths (ASP) problem, using an input graph with 300 nodes. (d) Successive overrelaxation (SOR), using a grid with 80 columns and 242 rows.

Parallel applications in Orca

Orca is a procedural, type-secure language intended for implementing parallel applications on distributed systems [1], [2], [3]. Below, we briefly discuss four examples of problems for which Orca has been used. Figure 11 presents the measured speedup of the Orca program for these four problems.

(1) *Matrix multiplication (MM).* MM is an example employing "trivial parallelism." Each processor is assigned a fixed portion of the result matrix. Once the work-to-do has been

distributed, all processors can proceed independently from each other. The speedup is not perfect, because it takes some time to initialize the source matrices (of size 250×250) and the portions are fixed but not necessarily equal if the matrix size is not divisible by the number of processors.

(2) *The traveling salesman problem (TSP).* The main characteristic of the TSP is the shared variable containing the current best solution. This variable is stored in a shared object that is read very frequently. If a new better route is found, all copies of the object are updated immediately, using the efficient broadcast protocol. It is important that the updating take place immediately, lest some processors continue to use an inferior bound, reducing the effectiveness of the pruning. The TSP achieves a speedup close to linear. (The TSP is discussed in detail in this paper.)

(3) *The all-pairs shortest paths (ASP) problem.* In the ASP problem, communication overhead is much higher than in the TSP. The ASP problem uses an iterative algorithm. Before each iteration, some process selects one row of the distances matrix as pivot row and sends it to all other processes. If implemented with point-to-point messages, the communication overhead would be linear with the number of processors. However, with our multicast protocol, the overhead is reduced to a few messages, resulting in high speedups.

(4) *Successive overrelaxation (SOR).* Of course, not all parallel applications benefit from broadcasting. In SOR, each processor communicates mainly with its neighbors. SOR is a worst-case example for our system, because point-to-point messages between neighboring nodes are implemented as broadcast messages received by all nodes. Still, the program achieves a reasonable speedup.

Summary and conclusions

Multiprocessors (with shared memory) are easy to program but hard to build, while multicomputers (no shared memory) are easy to build but hard to program. Here, we introduce a new model that is easy to program, easy to build, and has an acceptable performance on problems with a moderate grain size in which reads are much more common than writes. The essence of our approach is to implement reliable broadcasting as a distinct semantic layer and then use this layer to implement shared objects. Reliable broadcasting is achieved by having users send their messages to a sequencer, which then numbers them sequentially and broadcasts them. Machines that miss a message can get it by asking the sequencer, which maintains a history buffer. This scheme requires slightly over two messages per reliable broadcast. We built a kernel based on these principles and measured its performance at up to 1500 reliable broadcasts per second.

Also, we designed and implemented a shared-object layer and a language, Orca, that allows programmers to declare objects shared by multiple processes. Objects are replicated on all the machines that need to access them. When the language runtime system needs to read a shared object, it simply uses the local copy. When it needs to update a shared object, it reliably broadcasts the new object (or operation code and parameters). This scheme is simple and has a semantic model that is easy for programmers to understand. It is also efficient, because the amount of data broadcast is exactly what is needed. It is not constrained — as in the work of Li and Hudak [14] — to be one K or eight K because the page size happens to be that. Most of our broadcasts are much smaller (typically a few hundred bytes, at most).

In conclusion, the use of reliable broadcasting to support replicated, shared objects is a good way to approach parallel programming. Reliable broadcasting is simple to understand and efficient to implement. The use of a language like Orca makes it easier to use shared objects; however, even without Orca, this paradigm offers a new and effective way to exploit parallelism in future computing systems.

Acknowledgments

We would like to thank Erik Baalbergen, Dick Grune, and Wiebren de Jonge for carefully reading the paper and making many helpful suggestions.

References

[1] H.E. Bal, *Programming Distributed Systems*, Silicon Press, Summit, N.J., 1990.

[2] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, "A Distributed Implementation of the Shared Data-Object Model," *Proc. USENIX/SERC Workshop Experiences with Building Distributed and Multiprocessor Systems*, Usenix, Berkeley, Calif., 1989, pp. 1-9.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, "Experience with Distributed Programming in Orca," *Proc. 1990 Int'l Conf. Computer Languages*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 79-89.

[4] J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Second ACM Symp. Principles and Practice Parallel Programming*, ACM Press, New York, N.Y., 1990, pp. 168-177.

[5] K.P. Birman, and T.A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Computer Systems*, Vol. 5, No. 1, Feb. 1987, pp. 47-76.

[6] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Vol. 2, No. 1., Feb. 1984, pp. 39-59.

[7] N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, Vol. 32, No. 4, Apr. 1989, pp. 444-458.

[8] J. Chang and N.F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Trans. Computer Systems*, Vol. 2, No. 3, Aug. 1984, pp. 251-273.

[9] D.R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel," *ACM Trans. Computer Systems*, Vol. 3, No. 2, May 1985, pp. 77-107.

[10] E. Jul, H. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Trans. Computer Systems*, Vol. 6, No. 1, Feb. 1988, pp. 109-133.

[11] M.F. Kaashoek, *Group Communication in Distributed Computer Systems*, doctoral thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1992.

[12] M.F. Kaashoek et al., "An Efficient Reliable Broadcast Protocol," *Operating Systems Rev.*, Vol. 23, No. 4, Oct. 1989, pp. 5-19.

[13] M.F. Kaashoek and A.S. Tanenbaum, "Group Communication in the Amoeba Distributed Operating System," *Proc. 11th Int'l Conf. Distributed Computing Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1991, pp. 222-230.

[14] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems*, Vol. 7, No. 4, Nov. 1989, pp. 321-359.

[15] R.G. Minnich and D.J. Farber, "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," *Proc. 10th Int'l Conf. Distributed Computing Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 468-475.

[16] R.J. Swan, S.H. Fuller, and D.P. Siewiorek, "Cm* — A Modular, MultiMicroprocessor System," *Proc. Nat'l Computer Conf.*, AFIPS, Reston, Va., 1977, pp. 645-655.