

A SCALABLE OBJECT-BASED ARCHITECTURE

Susan Flynn Hummel, M. Frans Kaashoek, Andrew S. Tanenbaum

Vrije Universiteit

ABSTRACT

Although large-scale shared-memory multiprocessors are believed to be easier to program than disjoint-memory multicomputers with similar numbers of processors, they have proven harder to build. To date, the efficiency of software implementations of virtual shared-memory (VSM) on multicomputers with even a modest number of processors have not approached that of a physical shared-memory. Often VSMs are implemented by using the local memories of processors as caches for shared data. The overhead of maintaining the consistency of these caches, both in terms of processing time and bandwidth consumed, is a major contributor to the inefficiency of the implementations.

In this paper, we describe an object-based scheme for implementing a VSM on hierarchical multicomputers that is both efficient and scalable. By implementing an object-based style of programming at a low level, we are able to make effective use of bandwidth, while giving support for modern programming languages. We call our VSM scheme *multiple instruction single data* (MISD) since processors controlled by separate instruction streams operate, conceptually at least, on a single broadcast stream of shared data. MISD relies on an efficient software coherency scheme and on dedicated hardware to achieve its performance. The hardware required by MISD need not be special purpose. Indeed, MISD can be incorporated into existing multiple computer systems, or an MISD machine can be constructed from off-the-shelf components.

1. INTRODUCTION

Although the speed of uniprocessors continues to improve, it is ultimately limited by the speed of light. In order to build faster machines, we are forced to consider systems composed of multiple processor and memory chips. Unfortunately, inter-chip communication delays can seriously degrade the performance of such parallel machines. Moreover, the structures used to connect the chips have limited bandwidth and hence do not easily scale to large numbers (thousands) of processors and memories. To obtain the maximum (or even acceptable) performance from a large-scale parallel machine, it has been assumed that one must program at a very low level that is close to the hardware. Thus the benefits of high-level programming methodologies (increased productivity, reliability, etc.) have not been available in the highly parallel machine arena.

In short, the utility of today's parallel machines is hampered by both hardware and software induced limitations. The ideal parallel machine would possess the following (interrelated) properties:

- **Efficient:** Inter-chip communication overhead should be negligible, that is, it should be balanced with computations.
- **Scalable:** The machine should be easily extensible and should not require excessive amounts of special purpose hardware.
- **Support High-Level Programming Languages:** Users should be able to use modern programming methodologies (such as procedure-level abstractions) without sacrificing performance.

Another scalability issue (that is often overlooked by the designers of parallel machines) is that increasing the number of processors will increase the probability that some component in the system will fail. Early multiprocessors such as the C.mmp were crippled by component failures [37]. While today's hardware is considerably more robust, a large-scale parallel machine should at least take precautionary measures to allow execution to continue after partial failure of the system.

In this paper, we propose an architecture that we believe comes closer than existing proposed architectures to meeting the above criteria. By supporting an object-based style of programming at a low level (part of the

operating system kernel and optionally the hardware), we are able to mask communications delays and to make effective use of bandwidth, while giving support for modern programming languages. The low-level physical nature of the support allows us to efficiently implement application programs, run-time supervisors and even the operating system itself using object-based programming. Our design achieves its efficiency by using an integrated software/hardware approach that calls for simple (off-the-shelf) hardware and sophisticated software. Software intensive implementations are attractive for highly parallel machines as software is more flexible than hardware and hence better able to deal with the unpredictable nature of parallel programs.

The hardware and software that we envision for a scalable, efficient, object-based parallel machine is the subject of the rest of this paper. In the next section, we lay some technical groundwork by classifying and identifying the weaknesses of current memory models for parallel machines. We then introduce a new memory paradigm, *broadcast operations*, and describe a machine, MISD, whose memory is based on this paradigm. After giving the (hardware and software) architectural details of MISD, we discuss and evaluate a prototype implementation in §5. Finally, we discuss the direction of our future work.

2. MEMORY MODELS

The configuration of the processor and memory chips determines how processors communicate. The two main choices are shared-variables and message passing. At the two extremes, all the processors are connected to all the memories (as in the Ultracomputer [21]) or processor/memory pairs are connected to other processor/memory pairs (as in the iPSC [2]). The former scheme, where processors share memories, is called *multiprocessor*, while the later scheme where processors have disjoint (i.e., local) memories is called a *multicomputer*. Intermediate configurations, where some memories are shared and others disjoint, are common. Another trend, which we elaborate upon below, is to use software to implement a *virtual shared-memory* (VSM) on a machine with physically disjoint-memories.

The two most salient properties of the structure used to connect processor and memory chips are its bandwidth and its latency. Ideally, the processing power and memory capacity on the one hand, and the bandwidth and latency of the interconnection structure on the other, should be balanced. Special purpose hardware (such as communication co-processors or caches) or software optimizations (such as pipelining, data replication and processes partitioning) can reduce bandwidth requirements and mask latencies. Unfortunately, many software optimizations are intractable and hence may not be practical when there are large numbers of processors (and hence processes).

The simplest interconnection structure is the bus, where processors take turns exchanging messages. The finite bandwidth of a bus limits the number of processors that can be connected. The current rule-of-thumb is that a bus will become saturated by 64 processors [1]. To connect larger numbers of chips, either multiple buses (hierarchies) or *switching networks* must be used. On multicomputers, processors may double as switches, while on multiprocessors a separate network made up of multiple stages of switches is used to connect processors and memories. A multiple stage network composed of $N \log_k N$, $k \times k$ switches is needed to connect N processors to N memories. While hierarchies and switching networks may have higher bandwidths than buses, they suffer from similarly high latencies. How to satisfy the bandwidth requirements of highly parallel machines without incurring intolerable latencies is a difficult and open question.

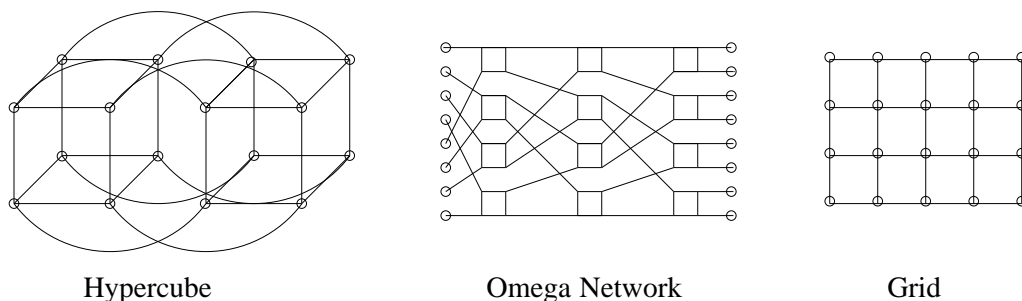


Figure 1. Switching Networks.

When processors share a communication channel (e.g. a wire), they form a *broadcast subnet*, and broadcast communication is possible. On a broadcast subnet, a broadcast message will take (almost) the same amount of time and consume the same amount of bandwidth as a point-to-point message. Examples of broadcast subnets are buses and rings. When its central node acts as a repeater, a star configuration can also be considered a broadcast subnet. The number of processors that can be effectively connected in a broadcast subnet is limited by their bandwidth

(buses), latency (rings) or fan-in constraints (stars). Although many machines contain broadcast subnets, most software does not take advantage of their inherent broadcast capabilities.

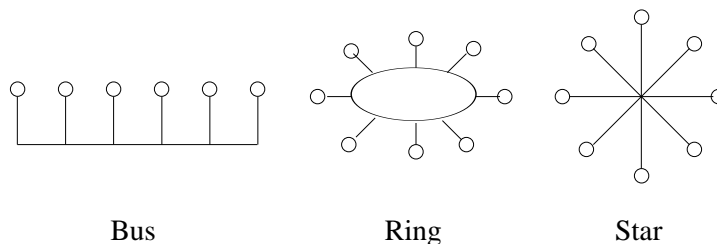


Figure 2. Broadcast Subnets.

All of the memory models of current (proposed or built) highly parallel machines suffer to some extent from high latencies, low bandwidth or both. We review these memory models below. In §3, we present a new memory model, broadcast memory, that makes effective use of bandwidth by balancing computations with access latencies.

2.1. SHARED MEMORY

On multiprocessors, processors communicate by writing and reading shared variables. From the programmer's point-of-view, the shared variable paradigm is generally regarded as preferable to a message passing one as it provides a convenient means for maintaining global state information. Read-modify-write instructions are typically included in shared-memory machines as they greatly facilitate inter-processor synchronization. The atomicity of *read-modify-write* instructions can be guaranteed by having them executed by DMA controllers that are attached to the memories.

A shared memory is equally accessible by all processors. Unfortunately, for N processors connected by a multiple stage switching network, this means that all memory accesses take $O(\log N)$ time. Moreover, the cost of building such a shared-memory machine is dominated by the cost of its switching network, which contains $O(N \log N)$ switches.

2.2. DISJOINT MEMORY

When processors do not share a memory, they must communicate via messages. Separate switching elements can be added to the processor/memory pair of a multicomputer, so that the processor does not have to participate in the routing of messages. Such special purpose hardware can have a dramatic impact on performance. For instance, by adding to each processor/memory pair of the iPSC multicomputer a processor that is microprogrammed to handle message routing, latency was reduced by three orders of magnitude [2]. More sophisticated DMA controllers can be instructed to perform complex operations (i.e., read-modify-write instructions).

For N processors connected by a switching network, each message must traverse on average $O(\log N)$ nodes. The bandwidth provided by a switching network or a multistage network depends on their number of wires, which may be comparable. However, the cost a switching network is lower, $O(N)$ (when separate switching elements are used). Large-scale multicomputers have also proven easier to build and to extend than multicomputers.

2.3. HIERARCHICAL MEMORY

A disjoint-memory machine can be viewed as two-level memory machine, where each processor can either access its local memory or remote memories. The remotely accessed data in essence forms a shared-memory. In general, a machine may have a hierarchy of memories, where some are disjoint and others shared. Often each level of memory functions as a cache for the next higher layer. By the exploiting the principle of locality, the average access latency can approach that of the fastest (i.e., closest) memory. Caching can also reduce bandwidth requirements as more data references may be satisfied locally.

Unfortunately, caching introduces the problem of coherence: if more than processor caches the same datum, their copies may become inconsistent. Various hardware and software schemes have been proposed for maintaining the consistency of caches. For instance, a write to a cached datum can be broadcast to all other copies, or alternatively, an invalidation message can be broadcast to the copies. These two approaches are especially attractive on broadcast subnets.

For a modest number of processors connected by a bus (or buses), cache coherency can be maintained by hardware. *Snoopy caches* monitor the bus for changes made to the data that they contain and then take the appropriate action (e.g., update or invalidate) [18]. Software coherency schemes are becoming increasingly

common on highly parallel machines (see for example [8, 21, 28, 30, 31]) because it is difficult to scale hardware cache coherency schemes to large numbers of processors [21]. Due to the potential optimizations that they introduce, software control over caching has even been proposed for uniprocessors [9].

2.4. VIRTUAL SHARED-MEMORY

Schemes for maintaining cache coherency and implementing VSMs on multicomputers have a similar flavor, which is not surprising as in both cases shared data must be stored (cached) in disjoint memories. One of the earliest implementations of a VSM is Kai Li's IVY [27], which allows processors with disjoint memories to share a paged, virtual address space. At any given time, a page is owned by a single processor. While a processor owns a page, reads and writes to the page can be satisfied locally. If another processor needs to access the page, the processor must request (and be granted) ownership from the owner and the page must be fetched over the network.

Although its unit of sharing is a page, IVY attempts to implement a VSM at the level of simple reads and writes. There may be a severe performance penalty due to the mismatch of computation and access overhead: an entire page must be fetched just to read or write one byte. Moreover, care must be taken lest two processor simultaneously access (perhaps distinct) data in the same page, causing the page to be "ping-ponged" back and forth between the two.

A higher-level approach to a VSM is supported by the Linda programming language [7]. Their VSM, called a tuple space (TS), is associatively accessed (by content) using three predefined operations, *out*, *in* and *read*. *Out*() and *in*() are used to add tuples (records) to or extract tuples from the TS (respectively). *Read*() returns the value of a tuple without removing it from the TS. (There is also an *eval* operation that causes a tuple-producing processing to be spawned.)

While Linda implementations may be able to take advantage of the relatively high-level nature of its predefined operations, Linda does not allow user-defined TS operations. Allowing users to define their own operations on shared variables is not only user-friendly, but prudent as the optimal read-modify-write instruction is still very much a topic of research (see for example, [15, 20]).

Like Linda, the Orca programming language supports a VSM [3]. Shared data-objects encapsulate shared variables and their operations. The operations are executed indivisibly and can be safely and efficiently invoked via messages on disjoint-memory machines. Unlike Linda, operations are user-defined. Hence, Orca provides a high-level flexible communication mechanism.

It is straightforward to implement procedure-level communication mechanisms, such as shared-data objects, when there is a shared memory. Unfortunately, as mentioned above, large-scale multiprocessors have proven hard to build and moreover are expensive. Multicomputers, on the other hand, are easy to build; however, to date, implementations of VSMs on multicomputers have not approached that of physical shared-memory.

Rather than considering what kind of highly parallel machines it is possible to build and then considering how the machine can be programmed, we will take a system-wide view and consider what type of highly parallel machine provides the most effective platform for implementing object-based communication mechanisms. Thus our machine architecture will be both software and hardware driven.

3. BROADCAST MEMORY

Other than the degree of coupling of their processors and memories, parallel machines differ in the degree of autonomy that they grant their processors. Processors can be controlled by either (independent) *multiple instruction* streams or a *single instruction* stream, and can operate on either *multiple data* streams or a *single data* stream [16]. This gives rise to the four classifications: SISD, MISD, SIMD, MIMD. Uniprocessors are SISD, while all existing parallel machines are either SIMD or MIMD.

To design a cost-effective large-scale parallel machine, we will take a cue from the less often considered class of machines, MISD. Our strategy will be to use the single data stream to maintain the consistency of cached shared data, i.e., to implement a VSM. To emphasize the way in which we implement our VSM, we will call our machine MISD; we will, however, relax MISD execution enough to allow for an efficient implementation.

Specifically, we will partially decouple the processors from the single data stream to gain flexibility and performance, and hence arrive at a hybrid MISD/MIMD machine. To do so, we distinguish between instructions that operate on shared data and those that operate on private data. There will be a single broadcast stream of the former, while each processor will have its own stream of the latter. As the broadcast stream will keep shared data synchronized, processors are able to read local copies of the data without having to traverse the interconnection structure. To allow finer control over shared data placement and cache coherency, the MISD machine will be implemented in software (Thereby also enabling us to use easily realized multicomputer hardware.).

Instead of broadcasting data, we will broadcast data-generating operations. The motivation for broadcasting operations (instructions and data) as opposed to either instructions (SIMD) or data (MISD) is two-fold. First, as discussed above, our primary goal is to support an object-based style of programming. Operations on shared data are trivially implemented with our scheme. Second, fortuitously, operations allow us to balance processor speed with the (as of yet unspecified) interconnection structure latency and bandwidth.¹

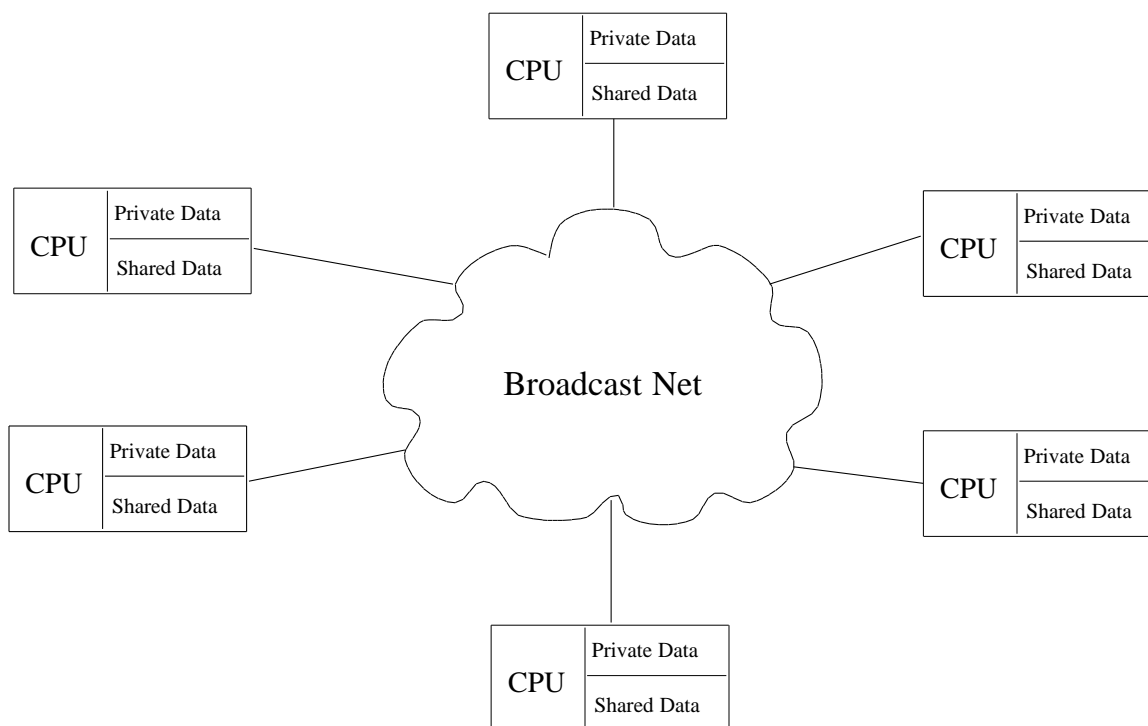


Figure 3. Processors Connected by a Broadcast Network.

In our proposed architecture, processors are connected by some kind of broadcast net, for example an Ethernet or a high-speed backplane. Although the broadcast net is not assumed to be completely reliable, as is true of modern hardware, failures are assumed to be the exception. Occasionally, messages may be lost and processors may fail. System software must therefore detect lost messages and recover from processor failures whenever possible. Nevertheless, the design is optimized for the case of relatively few lost messages.

The net is used to broadcast the operations to processors, which perform them on cached data. It is important that all processors execute the operations in the same order, so that cached data remains consistent. Achieving such total ordering over an unreliable broadcast network has been assumed to be prohibitively expensive [5]. However, the broadcast protocol we use is extremely efficient.

Very briefly, the basis of our reliable broadcasting scheme is that the collection of processors elect one of themselves as the *sequencer*. A simple algorithm is to choose the processor with the highest network address. To perform a reliable broadcast, any processor sends a point-to-point message to the sequencer which attaches a *sequence number*, broadcasts it and stores it in case it is needed later. If a process ever receives a broadcast message with a nonconsecutive sequence number, it knows it has missed one, so it asks the sequencer for a (point-to-point) retransmission. The protocol will be described in detail in §4.1.

Sequential Consistency

Although all processors perform operations in the same order, because of non-uniform network delays, not all processors will perform the operations at the same time. Moreover, since we permit processors to access cached shared data, it is possible that concurrent reads of the same variable will return different values. A similar situation arises on multiprocessors whose shared-memory has non-uniform access latencies (so-called NUMA machines). An important question is therefore, What does program order mean in our MISD context?

¹ It is interesting to note that the CM [23] takes the opposite approach to balancing computations and communication: rather than increasing the weight (information content) of messages they use very weak processors.

As has been suggested for shared-memory accesses on multiprocessors [26], what is desired is that local instructions and broadcast operations be performed in some permissible serial order. That is, program order is respected when the system behaves as if the processors' local execution streams (including any broadcast operations) were executed in some interleaved fashion by a single processor. Lamport refers to this requirement as the *sequential consistency principle*. A simple way of ensuring sequential consistency on multiprocessors is to not allow a processor (or a process when processors are multitasked) to have more than one outstanding shared-memory access; a simple way of ensuring sequential consistency on MISD is to not allow a processor (or process) to have more than one outstanding operation (i.e., the maximum depth of pipelined operations is one).

To enhance the scalability of MISD, cache coherency, that is, the decision to broadcast an operation, will be under software control. It will therefore ultimately be the user's responsibility to ensure that program order is respected. In the next section, we give the architectural details of MISD and justify our design choices.

4. MISD SYSTEM ARCHITECTURE

An emerging trend for microprocessors is that hardware and software are being designed together (e.g. MIPS [22] and TRACE [12]). A processor with unusual, albeit powerful, instructions will not sell unless it is accompanied by software that can make ample use of these instructions. It has been argued that the best cost/performance ratio currently seems to be obtained by packaging simplified hardware with highly-sophisticated software (e.g. RISC microprocessor packaged with optimizing compilers).

The need for a system-wide outlook seems to be even more imperative for parallel machines, where the complexity of both hardware and software is greater. Simple hardware and sophisticated software is also an attractive mix for highly parallel machines as hardware implementations (of cache coherency schemes, for example) do not scale as readily as software ones. Bearing this in mind, we have designed the hardware and system software that we envision for an efficient, scalable machine that supports object-based programming.

In order to make the writing and maintaining of system software manageable, the software is often structured in layers, where each layer can be considered as the virtual machine upon which higher layers are constructed [36]. Higher layers of software need not be concerned with the implementation details of lower layers. Moreover, by separating *policies* enforced in one layer from *mechanisms* implemented in a lower layer, the flexibility of software is enhanced [6].

Application Programs
Other System Software (e.g. run-time supervisors)
Operating System Functions (e.g. process, memory and operation management)
Operating System Kernel (e.g. broadcast protocol)
Hardware (e.g broadcast net)

Figure 4. Layers of MISD

As depicted in figure 4, our MISD design is divided into five distinct layers. The lowest layer of system software, the kernel of the operating system, implements a reliable broadcast protocol atop the (not-necessarily reliable) hardware broadcast net. The kernel takes care of actually broadcasting operations and of ensuring their sequential consistency, that is, it implements the communication mechanism. (The kernel also supports point-to-point messages.) Higher layers of software use the reliable broadcast protocol to implement a VSM. It is the responsibility of application programs or their run-time supervisors to ensure the coherency of shared data (i.e., to initiate the appropriate broadcast operations), which is a policy decision. Operating system functions handle operation invocation and map processes and data onto processors and memories.

In the next subsection, we describe the kernel of MISD. Its hardware and VSM implementation are discussed in §4.2 and §4.3 (respectively).

4.1. BROADCAST PROTOCOL

The lowest layer of MISD system software implements the protocol used to ensure the sequential consistency of broadcast operations. Processes are assumed to be divided into groups based on their mutual use of shared data; using the protocol, the operations to be performed by a process group on its shared data are broadcast to the group. The protocol uses broadcast as well as point-to-point messages. In the protocol, we assume that each processor runs the same communications kernel and executes user processes and that one distinguished processor doubles as the sequencer. However, a dedicated processor (that does not execute user programs) may act as the sequencer, affecting the optimal division of labor between the sequencer and kernels (see §4.2).

There are three salient properties of a broadcast protocol: synchrony, reliability and fault tolerance. Synchrony is concerned with the ordering of messages; to achieve synchrony, we must ensure that all processors execute broadcast operations in the same order. If, in addition, we only allow a processor to have one outstanding operation, then sequential consistency is achieved. Reliability is the ability to recover from communication failures (i.e. lost messages), and fault tolerance from processor failures (i.e. site crashes).

With current technology, communication failures happen rarely and processor failures even more rarely. Although the protocol we present can handle communication and processor failures, it is optimistic. That is, it is optimized for the usual case when communication and processor failures do not occur.

The protocol used in MISD is a derivative of the one presented in [25]. The protocol described therein is extremely efficient but does not address the problem of fault tolerance and hence was extended for use in MISD. Another extension to the protocol is the handling of overlapping groups, that is, groups that have data in common. (It may be desirable to have overlapping groups so that processors have similar numbers of processes and to localize inter-processor communication—see §4.2. and §4.3) Groups that overlap are problematic as operations generated by distinct groups must be performed synchronously. (When a group *A* overlaps with a group *B*, then all the processes in their intersection must perform the operations generated by both groups in the same overall order.) Our protocol is easily extended to overlapping groups by viewing them as an additional compound group with its own sequencer. We first give the broadcast protocol for non-overlapping groups with emphasis on its novel fault tolerance mechanism, and then the protocol for overlapping groups.

Synchrony

To guarantee that all operations are performed in the same order, we require that operations are logged with a special site, the sequencer, which actually broadcasts the operations. Each broadcast operation is sent to the sequencer via a point-to-point message. The sequencer stamps operations with consecutive sequence numbers before broadcasting them.

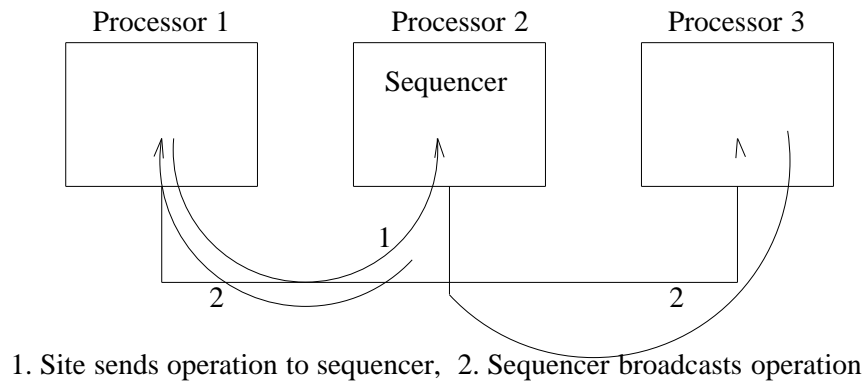


Figure 6. Broadcasting an Operation

The original sender interprets this broadcast as an acknowledgement for its operation; the operation cannot be discarded until such an acknowledgement has been received. All sites process operations according to the sequence numbers added by the sequencer. Thus, all sites process the operations in the same order and synchrony is achieved.

Reliability

A site also maintains a local sequence number, where it stores the highest consecutive sequence number (stamped onto the operations) it has received from the sequencer. This local sequence number is piggybacked onto each operation that the site sends to the sequencer, and in essence, acts as an acknowledgement for all operations with lower sequence numbers.

A gap in the sequence numbers of operations received at a site indicates that a broadcast operation has been missed. When such a situation occurs, the site requests a retransmission from the sequencer. In order to be able to fulfill such requests, the sequencer stores operations in a buffer, called its *history*. (For pragmatic reasons, the size of the history must of course be bounded.)

The acknowledgements that are piggybacked onto operations sent to the sequencer are used to determine when operations can be discarded from the history. The sequencer records the highest piggy-backed acknowledgement it has received from each processor. When the history is in danger of becoming full, the sequencer calculates the minimum of these acknowledgements and removes all operations with lower sequence numbers from the history. Thus, the overhead of removing operations is proportional to the number of processors, not the history size. As the history size HS is expected to be at least as large as the number of processors P (so that each processor can have one outstanding operation), the cost is $O(1)$ per broadcast operation.

It is possible that a site will not need to broadcast an operation for an extended period of time, and hence, will not have the opportunity to piggy-back its acknowledgement onto an operation. To ensure that the history can still be emptied, each site maintains a counter; if the site receives C operations since it has last sent a message to the sequencer, then the site piggy-backs its local sequence number onto a dummy message which it transmits to the sequencer. For example, if we set $C = P$ (where HS is some multiple of P), then in the worse case, one processor continuously broadcasting operations (i.e., other processors are unable to piggy-back acknowledgements onto operations), $P - 1$ dummy messages will be sent for every P operations broadcast. Obviously, it is desirable to have the history as large as possible in order to minimize the number of dummy messages (required to ensure that there is very little possibility of history becoming full).

If, despite all precautions, the history does become full, then a synchronization phase is entered, where the sequencer explicitly requests that silent sites send their local sequence numbers.

Fault Tolerance

The protocol given in [25] supports synchrony and reliability, but does not address fault tolerance. By viewing our system as one that broadcasts *operations* rather than general *messages*, we are able to achieve fault tolerance in a very efficient manner. Since processor failures occur very infrequently, we will use a fault tolerance mechanism that is optimized for normal operation. While the cost of actually recovering from a sequencer failure may be high, the mechanism overhead (in both space and time) per message is minimal.

The basic idea behind our fault tolerance mechanism is that when a sequencer fails, the most up-to-date processor broadcasts its data and a new processor takes over as sequencer. Thus, instead of replicating the operation history to increase its availability (so that missed operations can be reapplied after the sequencer fails), we take advantage of the fact that the data upon which the operations have been performed is replicated. Each processor must keep a copy of the last operation broadcast by the sequencer of its behalf, until the sequence number assigned to this operation is equal to its local sequence number. Since this is precisely the time at which the operation can be executed by the processor, there is no time or storage penalty incurred by the mechanism.

When the site executing the sequencer fails, the shared data must be synchronized and a new sequencer must be selected. Since every site maintains a local sequence number, this is easily accomplished: Each site broadcasts a message containing a tuple made up of its local sequence and identification numbers; the processor whose tuple has the highest cardinality is chosen as the new sequencer, and broadcasts its shared data to the other sites. Sites with an outstanding operation or whose last operation has a sequence number that is greater than that of the chosen processor must resend the operation to the new sequencer.

For the above scheme to function correctly, the identification numbers of all of the members of a group must be known to the group. This is easily accomplished by including a group membership list in the shared data of a group: When the sequencer detects a site failure (because the site does not respond to requests for its local sequence number), the sequencer broadcasts an operation to delete the site from the membership list. Similarly, when a site wishes to be added to a membership list, it sends the sequencer an enrollment message that the sequencer subsequently broadcasts. (Note that we assume the existence of a name server from which the name of the current sequencer of a group can be obtained.)

Compound Groups

A scheme for handling broadcast messages to overlapping groups is presented in [19]. Like our scheme there is a sequencer per group. The scheme uses a forest of sequencers, called a *propagation graph*, to merge broadcast messages from overlapping groups. Although the broadcast protocol used within process groups is different than the one described here, their protocol is (in some sense) orthogonal to the propagation graph and could easily be substituted with ours. The basic idea behind the propagation graph is that messages destined for multiple groups are ordered as they are propagated up the tree of sequencers before being broadcast down the tree.

The key to the propagation graph algorithm in [19] is to assign sequencers to processes in the intersection of groups. For MISD, we will use a different approach and create a new additional *compound* group for groups that overlap. (The motivation for taking this approach is architectural—see §4.2.) The shared data of a compound group will be the data that its subgroups have in common (i.e., the union of the intersections of their data). Thus each subgroup will have a compound group sequencer as well as a local sequencer.

The specific algorithm used by MISD to broadcast operations to a compound group is: A forest of sequencers is built, where the group of each sequencer is a superset of the groups of its children. An operation from a site destined to a compound group G is first sent to the local sequencer of the site. The operation is propagated up the tree of sequencers until it reaches the sequencer for G . This sequencer stamps the operation with a sequence number and then broadcasts the operation to the sequencers in its subtree (via point-to-point messages). Each sequencer assigns its next sequence number to the operation and then broadcasts the operation to its subgroup. Thus, operations are delivered to the subgroups in the same relative order and within each subgroup in the same overall order. Reliability and fault tolerance are handled as in the single group case (i.e., by having subgroup sequencers maintain local sequence numbers and counters and the compound group sequencer a history).

4.2. HARDWARE

Broadcast subnets are ideal for running MISD because of their inherent broadcast capabilities. Unfortunately, as discussed in §2, the number of processors that can be effectively connected by a broadcast subnet is limited. While we expect that sophisticated software-based cache management schemes can decrease bandwidth requirements and pipelining can mask latencies, and perhaps raise this limit, nevertheless a limit will be reached.

In this subsection, we consider how to:

- 1) Reduce the overhead of broadcast operations.
- 2) Reduce the latency of broadcast operations.
- 3) Increase the bandwidth of MISD.

To achieve 1) and 2), we use dedicated hardware: A processor is dedicated to executing the sequencer code, and DMA controllers are added to each processor/memory pair. To achieve 3), we use a hierarchy of broadcast subnets.

The optimal division of labor between the (processor acting as) sequencer and the other processors will be dependent on the degree of dedicated hardware. For instance, when a dedicated processor acts as sequencer, it may be desirable to have the sequencer execute the operations and to then broadcast the results rather than the operations themselves. For comparison purposes, we will first assess the cost of the implementation without any dedicated hardware, that is, when the processor that acts as sequencer also executes user processes. We will (initially) assume that processors are connected by a broadcast subnet and only consider non-compound groups (both of these restrictions are removed below). By identifying deficiencies, we can determine where it is best to apply dedicated hardware.

Protocol Performance

For each operation broadcast to a process group, the protocol under normal conditions uses two messages (one point-to-point and one broadcast) per operation. Occasionally (when a processor has been silent for an extended period), an extra (point-to-point) dummy messages will be sent. In addition, the sequencer must maintain an operation history. As discussed above, there is a space/time tradeoff between the size of the history and the number of dummy messages sent.

The protocol itself is very scalable. Increasing the number of processors does not increase the number of messages needed to broadcast an operation. The cost in terms time for the sequencer to maintain the history is $O(1)$ per message. Assuming that the history size is set to a multiple of the number of processors P , the cost in terms of space is will be $O(P)$ (including the acknowledgement per processor that must be recorded). Thus the scalability of the protocol is only limited by the available bandwidth.

One cost of broadcasting operations is that each processor must execute every operation. As these operations are executed in parallel this cost may not be very significant; however, the cost of processing the messages from the sequencer (i.e., an interrupt and two context switches) may be considerable. These two concerns are addressed in the next two subsections (respectively). In the third subsection, we consider how to increase the bandwidth of MISD.

Broadcast Server

By dedicating a processor to act as sequencer, that is by having a *broadcast server*, we can offload the execution of the operations from the other processors. Instead of broadcasting the operation logged with it, the sequencer can broadcast the results of executing the operations, i.e. the new values of any data that has been modified. In

some sense, a broadcast server can be considered as a rudimentary file server, the shared data viewed as shared files and the operations can be thought of as transactions.

When a dedicated processor acts as sequencer, then almost the entire memory of the broadcast server is available for the history, thereby greatly reducing the number of dummies necessary to ensure that there is only a small probability that the history will become full. The disadvantage of having a broadcast server is that when a new sequencer is elected, its user processes must be migrated to new processors. However, since the shared data accessed by these processes already resides on the new processors, and since sequencer failures are expected to be rare, this probably is not very serious.

DMA Controller

We can further reduce the load on the processors by adding DMA controllers to each processor/memory pair. The DMA controllers can handle all the communication with the sequencer. Since many memory chips are dual ported, the controllers could be implemented with off-the-shelf processor chips. Alternatively, tailored-made DMA controller chips could be used. Such special purpose DMA controllers could be used to speedup message delivery: each processor stores a list of its processes' group identifiers in an associative memory; the group identifier field of passing messages would be matched against this memory and non-matching messages ignored. (Network controllers that match multicast group names already exist—for example in some Ethernet implementations [24].) The broadcast protocol (described in §5.3) could also be implemented in microcode on the DMA controller.

Note that the controllers we propose are similar to, but fundamentally different than, snoopy caches [18]. Snoopy caches attempt to match the cache-line address contained in passing messages with the contents of its cache, whereas our controllers attempt to match a process group. Assuming that each shared segment is at least as large as a cache line, then the number of distinct cache lines will be larger (and probably much larger) than the number of process groups. Matching against process groups, which are symbolic names, rather than addresses, is another instance of our taking (and benefiting from) a high-level software-assisted approach to managing cache coherence.

With the addition of DMA controllers, it becomes less attractive to have the sequencer perform operations (i.e., to broadcast results rather than the operations themselves), as the controllers can perform the operations in parallel with the processors. Moreover, reducing the workload of the sequencer decreases the likelihood that the sequencer itself becomes a bottleneck.

Adding a DMA controller may significantly reduce the cost of broadcasting operations, but it does not make MISD truly scalable. Indeed, by speeding up inter-processor communication we increase the rate at which operations can be generated and hence decrease the number of processors which will saturate the bandwidth of the subnet! In the next section, we address the problem of scalability.

Hierarchy of Broadcast Subnets

Since the number of processors that can be connected by a single broadcast net is limited, we propose to build an MISD machine by connecting subnets hierarchically. One processor per subnet will act as sequencer and will propagate operations to its parent if necessary (i.e., if a process group is assigned to more than one subnet). Thus, we will have a tree of sequencers, each broadcasting to its subnet. (The ability to achieve fault tolerance will depend critically on the connectivity of this tree.)

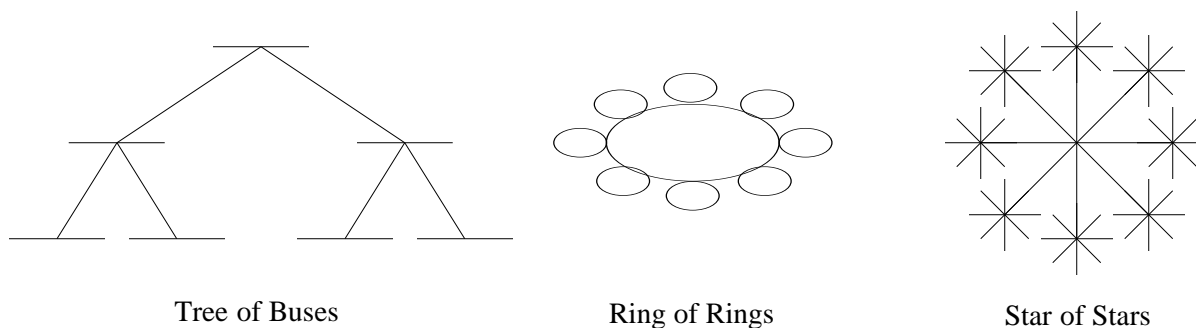


Figure 5. Possible Topologies for an MISD Machine.

Various hierarchical topologies are possible, for example a tree, ring or star. In principle, the hierarchy could be extended to wide area networks (WANs) (satellites and cellular radios, for instance, have broadcast capabilities); however, the latencies would increase. For concreteness, we will restrict our discussion to a tree topology, although

a ring or star structure may be preferable because of their inherent broadcast capabilities. For generality, we will not assume that the backbone or subnets are reliable.

Because a broadcast subnet can contain up to 64 processors (e.g. when they are bus-connected), the height h of a tree consisting on broadcast subnet nodes does not have to be large to include thousands of processors. For example, a 4-ary tree of height 3 could contain 1344 processors.

Ideally each process group would run on an single subnet obviating the need for the inter-subnet propagation of broadcast operations. For load-balancing reasons, however, a process group may be assigned to more than one subnet. In such cases, the backbone of the machine must be used to propagate messages from one subnet to the other. Obviously, it is desirable to have the subnets assigned to a group be as close as possible in the tree.

When a group is assigned to more than one broadcast subnet, then the sequencer will be assigned to their nearest common ancestor in the hierarchy. The sequencer of a compound group will similarly be assigned to the nearest common ancestor of its subgroups. Indeed, the protocol for multi-subnet groups is similar to that of compound groups: operations must be propagated up the tree to the sequencer and then broadcast down the tree. The protocol is slightly simpler than the compound group one since the local sequencers do not need to re-stamp the operations before broadcasting.

Both protocols use two messages, one point-to-point and one broadcast, per broadcast subnet. In the worst case, i.e., when a group is assigned to every subnet of a k -ary tree of height h , h point-to-point messages are required to send an operation to the sequencer and $(k^{h+1} - 1)/(k - 1)$ point-to-point messages and broadcast messages are required to broadcast an operation to all

$$P = n \frac{(k^{h+1} - 1)}{(k - 1)}$$

processors (each subnet consists of n processors). Fortunately, many of the messages can be passed concurrently as they traverse distinct edges of the tree, and the elapsed time is $O(h)$ (assuming a constant delay per message). Thus, our protocol remains scalable when executed on a hierarchy of subnets: only when a group is assigned to an additional subnet, does increasing the number of processors increase the number of messages needed to broadcast an operation. As described below, it is the responsibility of software to assign process groups to subnets and to propagate broadcast messages (when necessary).

4.3. VIRTUAL SHARED-MEMORY ARCHITECTURE

In this subsection we describe how MISD system software uses the above reliable broadcast protocol to implement a VSM. The software must map processes and data onto processors and memories (at compile-time), and implement object invocation (at run-time).

Process Management

We define a *job* to be the collection of parallel processes created from a single application program. Some of the processes within a job may share data and operations on that data. Recall that, processes are partitioned into proper subsets, called groups, based on their mutual use of shared data and operations.

While most languages require that shared data be explicitly identified (Ada being the exception), it is not beyond the state-of-the-art of compiler technology to detect shared data (see for example [17]). Thus, it should be no problem for a compiler to determine which processes share data.

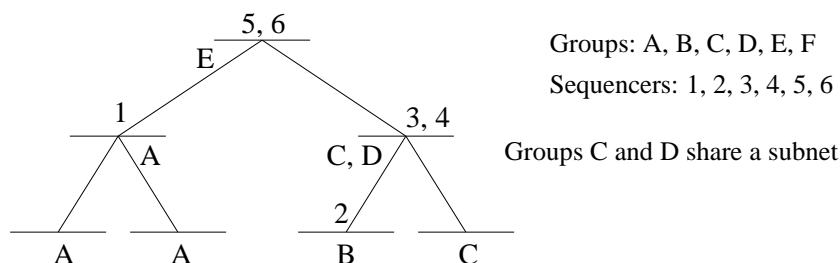


Figure 7. Possible Assignment of Groups and Sequencers to Subnets

Groups A, B, C, D and E are disjoint. Group $F = A \cup B \cup C \cup D \cup E$, hence its shared data is global.

To cluster processes into groups and to assign these groups to broadcast subnets, variants of well-know partitioning and scheduling algorithms (for example [34]) that (attempt to) minimize inter-subnet communication can be

used. As mentioned above, the sequencer of a group is assigned to the nearest common ancestor of the subnets assigned to the group. Figure 7 depicts an example of a possible assignment of groups and their sequencers to subnets. In the example, group F is a compound group composed of all the other groups; thus, the sequencer for F (6) ensures the synchrony of operations performed on shared global data (which are factored out of the subgroups' shared data). The tree structure of MISD, where each subnet (containing up to 64 processors) can be considered a single entity, may greatly reduce the cost of assignment algorithms. (Within the subnets conventional run-time optimizations, e.g., dynamic load balancing, can be used as there is an illusion of shared-memory.)

Operations

On MISD, users can construct their own read-modify-write operations. Like procedures, operations consist of sequences of a subset of the native machine instruction set. To simplify the implementation, we will exempt any instruction that either can block or is non-interruptible (in particular, system calls cannot be included in operations). A minimal instructions set would include: reads, writes, increments, decrements, multiplies, divides and branches. To reduce the number of instructions that must actually be broadcast, users (e.g. compilers) can install operations and to refer to them by symbolic names.

After we get more experience with what instructions and operations are truly valuable, we will consider providing predefined, microcoded operations. Operations that we suspect may be useful include: block reads, block writes and queue operations. Hardware support for these operations have been included in various other designs, e.g. [13, 32].

By replicating shared data in all of the local memories, only operations that modify shared data need be broadcast, the others can be executed locally. While it is an applications' responsibility to initiate broadcast operations, again, it is not beyond the state-of-the-art of compiler technology to detect which operations modify shared data and to insert the broadcast operations into user code (see [3]).

Since broadcasting is handled by the kernel, to broadcast an operation user programs must execute a system call, i.e., a protected procedure call. (Such system calls can be inserted into user programs by a compiler.) Alternatively, for security reasons, when a kernel receives a broadcasted operation, the operations must be executed in user space via an "up-call" [10].

When complex data structures need to be accessed indivisibly, some sort of synchronization mechanism must be used to coordinate the accesses made by user processes and the broadcast operations (executed on the same processor); however, since only local accesses need to be coordinated, the synchronization mechanism can be optimistic and need not entail a system call. (Assuming that user processes can only be interrupted by systems calls, a simple implementation of the required mechanisms is: Before initiating a read in user space, a flag is reset; broadcast operations that modify the data set the flag; when the read completes, if the flag is set, i.e., there were intervening writes, then the read is restarted.)

Memory Management

The local memory of each processor of MISD is divided into variable-sized segments. Segments come in three flavors: private, shared and system. The private segments contain the local data and code of the user processes that run on the processor. The shared segments contain copies of the data and code (i.e., operations) that are accessed by more than one user process. There is a shared segment for each process group that contains the shared data and operations belonging to the group. System segments contain the operating system kernel data.

We distinguish between private and system segments because they belong to different protection domains: As is true of most operating systems, MISD software supports two protection domains, supervisor and user. The kernel processes run in supervised mode and have access rights to all segments; whereas user processes run in user mode and only have access to private and shared segments. (Note that if user mode processes were only given read access to shared segments, i.e., no write or execute permission, then a trap to the kernel would be automatically generated whenever an attempt was made to write a shared datum or execute a shared operation obviating the need for explicit system calls to broadcast operations.)

On most architectures the protected procedures calls and up-calls needed to broadcast operations are usually somewhat expensive as they require two context switches that cross protection domains. The cost of context switches can be reduced with the proper hardware support, such as separate address spaces and segmentation. For instance, Ramachandran and Khalidi propose a three-level virtual address space that includes a separate *object* address space [33]. Unfortunately, although object-based architectures are appearing, and indeed, the idea of segmentation is not new (it is used in MULTICS for example [29]), such hardware support is not commonplace.

In the absence of hardware support, Ramachandran and Khalidi suggest using code in-lining (for local calls) and *slave* processes (for calls to remote processors) to reduce the overhead of context switches. Upon arrival of an

operation, a pre-spawned slave process is activated to execute the operation. If the slave processes can be made light-weight, then the cost of context switching may be less than an up-call. Such a scheme has also been used to implement distributed up-calls (for a window server) [11].

5. PROTOTYPE IMPLEMENTATION

We have implemented a prototype kernel and integrated it with an existing distributed operating system, Amoeba [35]. The kernel runs on a collection of 10 16 Mhz Motorola 68020 CPUs connected by an 10 Mbit/s Ethernet. As reported in [4], an Orca run-time supervisor has been implemented atop the kernel.

To quantify the overhead of the broadcast protocol, we have measured the delay for a message when a single site was continuously broadcasting to all 10 CPUs. With a history size of 20, the average delay for a broadcast message is less than most RPCs implementation, 1.5 msec. With a history size of 1000, the delay was reduced to 1.3 msec. 1.3 msec is almost the minimum time required to send two messages given our current hardware. Thus, as predicted above, with a large enough history, the cost of a broadcast message is indeed roughly equal to that of 2 messages.

To investigate the scalability of the protocol, we measured the average message delay as the number of continuous broadcasters was increased. The protocol did not scale perfectly because of limited bandwidth. Due to collisions, each additional broadcaster added approximately 0.6 msec to the delay per broadcast operation. The average delay for ten continuous broadcasters was around 7 msec. We, nevertheless, find these results encouraging, as we believe that the actual rate that processors will broadcast operations will be an order of magnitude lower than in our experiments. (If nothing else the processors must spend time executing the operations.) Although more experience with application programs is necessary, preliminary studies have show that only a small percentage of data accesses are writes to shared data ($\approx 3\%$) [14], and the programs that we have run typically use less than 2% of the available bandwidth.

We therefore believe that, with a dedicated broadcast server (i.e., nearly unlimited memory to hold the history), it is not unrealistic for 64 processors running application programs to be able to broadcast operations with an average delay that is significantly less than 7 msec using our protocol and hardware. Nor do we think it overly optimistic for the delay to drop by an order of magnitude with the addition of DMA controllers. An implementation using more advanced technologies (such as fiber optics) would be expected to reduce the delay similarly.

6. CONCLUSIONS AND FUTURE WORK

How well does our design meet our goals of scalability, efficiency and support for high-level programming languages? MISD's hierarchy of broadcast subnets is easy to build and to extend and does not require special purpose hardware. The cost of a P processor MISD machine is $O(P)$, even when dedicated hardware is used. We have outlined the system software for efficiently supporting broadcast operations. This software can be run a wide spectrum of machines, ranging from off-the-shelf processors connected by a local-area network (LAN) to a specialized MISD machine. Finally, because of MISD's low-level support for operations, modern programming languages with procedure-level communication mechanisms can be readily implemented. Thus we feel as though MISD realizes our goals.

However, several unresolved issues and areas for improvement remain. For instance, on the hardware side, What is the optimal hierarchical topology for a given class of processors and buses? We are also considering what types of special purpose hardware could be used to advantage by MISD. DMA controllers could reduce communications costs as well as provide support for operations, while object-based processors (separate address spaces, tagged data etc.) could speed up object invocation.

On the software side, we plan to investigate various optimizations. MISD enhances the opportunities for optimizations by exposing cache coherency to software control and by using a hierarchical architecture. Software controlled caching will allow us to explore intermediate replication strategies, where operations on some shared data are broadcast, and operations on other shared data are performed locally or remotely. Pipelining (i.e., allowing more than one outstanding operation) and optimistic execution (i.e., performing operations immediately and then recovering if necessary) are other potential enhancements whose implementations may benefit from a software-based scheme. The advantage of a hierarchical architecture is that by considering each subnet in the hierarchy as a single entity, the cost of intractable optimizations (or their approximations) may become less prohibitive.

1. Almasi, G. S. and Gottlieb, A., *Highly Parallel Computing*, Benjamin/Cummings Publishing Company, Inc., Redwood City (1989).
2. Athas, W. C. and Seitz, C. L., "Multicomputers: Message Passing Concurrent Computers," *Computer*, pp. 9-24 (August 1988).

3. Bal, H. E., *Programming Distributed Systems*, Silicon Press, Summit NJ (1989).
4. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., "A Distributed Implementation of the Shared Data-Object Model," *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19 (October 1989).
5. Birman, K. P. and Joseph, T. A., "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems* **5**(1), pp. 39-59 (February 1987).
6. Brinch Hansen, P., "The Nucleus of a Multiprogramming System," *Communications of the ACM* **13**(4) (April 1970).
7. Carriero, N. and Gelernter, D., "Linda in Context," *Communications of the ACM* **32**(4), pp. 444-458 (April 1989).
8. Cheriton, D. R., Goosen, H. A., and Boyle, P. D., "Multi-Level Sharing Caching Techniques for Scalability in VMP-MC," *16th International Symposium on Computer Architecture* (June 1989).
9. Chi, C.-H. and Dietz, H., "Unified Management of Registers and Cache using Liveness and Cache Bypass," *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 344-355 (1989).
10. Clark, D., "The Structuring of Systems using Upcalls," *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 171-180 (October 1985).
11. Cohrs, D. L., Miller, B. P., and Call, L. A., "Distributed Upcalls: a Mechanism for Layering Asynchronous Abstractions," *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 55-62 (1988).
12. Cowell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B., and Rodman, P. K., "A VLIW Architecture for a Trace Scheduling Compiler," *Proceeding Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180-192 (1987).
13. Dally, W. J., Chao, L., Chien, A., Hassoun, S., Horwat, W., Kaplan, J., Song, P., Totty, B., and Wills, S., "Architecture of a Message-Driven Processor," *14th Annual International Symposium on Computer Architecture*, pp. 189-196 (1987).
14. Eggers, S. J. and Katz, R. H., "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," *Proceedings Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257-270 (1989).
15. Feitelson, D. G., "Implementation of a Wait-Free Synchronization Primitive that solves the n-process consensus," *Information Processing Letters* **32**, pp. 81-83 (July 1989).
16. Flynn, M. J., "Some Computer Organizations and their Effectiveness," *IEEE Transactions on Computers* **C-21**, pp. 948-960 (1972).
17. Flynn Hummel, S., Dewar, R. B. K., and Schonberg, E., "A Storage Model for Ada on Hierarchical-Memory Multiprocessors," *Proceedings of the Ada-Europe International Conference*, pp. 205-214, Cambridge University Press (June 1989).
18. Frank, S. J., "Tightly Coupled Multiprocessor System Speeds Memory-Access Times," *Electronics*, pp. 164-169 (January 12 1984).
19. Garcia-Molina, H. and Spauster, A., "Message Ordering in a Multicast Environment," *International Conference on Distributed Computing Systems*, pp. 354-361 (1989).
20. Goodman, J. R., Vernon, M. K., and Woest, P. J., "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proceedings Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64-75 (1989).
21. Gottlieb, A., "An Overview of the NYU Ultracomputer Project," pp. 25-95 in *Experimental Computing Architectures*, ed. J. J. Dongarra, North-Holland (1987).
22. Hennessy, J. L., Jouppi, N., Baskett, F., and Gill, J., "MIPS: a VLSI Processor Architecture," *Proceedings of the CMU Conference on VLSI Systems and Computations*, pp. 337-346, Computer Science Press (October 1981).
23. Hillis, W. D., *The Connection Machine*, M.I.T. Press, Cambridge, Mass. (1985).
24. Hughes, L., "Multicast response handling taxonomy," *Computer Communications* **12**(1), pp. 39-46 (February 1989).
25. Kaashoek, M. F., Tanenbaum, A. S., Hummel, S. Flynn, and Bal, H. E., "An Efficient Reliable Broadcast

- Protocol,” *Operating Systems Review* (October 1989).
26. Lamport, L., “How to Make a Multiprocessor that Correctly Executes Multiprocess Programs,” *IEEE Transactions on Computers* **C-28, 9**, pp. 690-691 (September 1979).
 27. Li, K., “Shared Virtual Memory on Loosely Coupled Multiprocessors,” *Ph.D. Thesis*, Yale University (1986).
 28. McGrath, R. E. and Emrath, P. A., “Using Memory in the Cedar System,” *Lecture Notes in Computer Science* **297**, pp. 43-67, Springer-Verlag (1987).
 29. Organic, E. I., *The Multics System*, M.I.T. Press, Cambridge, Mass. (1972).
 30. Owicki, S. and Agarwal, A., “Evaluating the Performance of Software Cache Coherence,” *Proceedings Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 230-242 (1989).
 31. Pfister, G. F., “An Introduction to the RP3,” in *Experimental Computing Architectures*, ed. J. J. Dongarra, North-Holland (1987).
 32. Ramachandran, U., Solomon, M., and Vernon, M., “Hardware Support for Interprocessor Communication,” *14th Annual International Symposium on Computer Architecture*, pp. 178-188 (1987).
 33. Ramachandran, U. and Khalidi, M. Y. A., “A Measurement-based Study of Hardware Support for Object Invocation,” *Software-Practice and Experience* **19(9)**, 809-828 (September 1989).
 34. Sakar, V. and Hennessy, J., “Compile-Time Partitioning and Scheduling of Parallel Programs,” *Proceedings of SIGPLAN Symposium on Compiler Construction* (June 1986).
 35. Tanenbaum, A. S., Renesse, R. van, Staveren, H. van, Sharp, G. J., Mullender, S. J., Jansen, J., and Rossum, G. van, “Experiences with the Amoeba Distributed Operating System,” *Technical Report IR194*, Vrije Universiteit (July 1989).
 36. Tanenbaum, A. S., *Structured Computer Organization 3rd Edition*, Prentice-Hall (1990).
 37. Wulf, W. A., Levin, R., and Harbison, S. P., *Hydra/C.mmp: an Experimental Computer System*, McGraw-Hill (1981).