Distributed Shared Memory on Loosely Coupled Systems

Vicente Cholvi-Juan
Department of Computer Science
University Jaume I
Campus Penyeta Roja, Castelló, Spain
E-mail: vcholvi@inf.uji.es
AND
Roy Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Av, Urbana, IL 61801
E-mail: roy@cs.uiuc.edu

Keywords: distributed systems, distributed shared memory, concurrency, operating systems

Edited by: Rudi Murn
Received: April 4, 1996 **Revised:** November 5, 1996 **Accepted:** November 29, 1996

The distributed shared memory model (DSMM) is considered a feasible alternative to the traditional communication model (CM), especially in loosely coupled distributed systems. While the CM is usually considered a low-level model, the DSMM provides a shared address space that can be used in the same way as local memory.
This paper provides a taxonomy of distributed shared memory systems, focusing on different implementations and the factors which affect the behavior of those implementations.

1 Introduction

Many computational problems benefit from the availability of *parallel-processing* power: the computational problem is split into subproblems and each one is solved concurrently. There are many multiprocessor computers, ranging from only a few to thousands of processors. Typically, such a multicomputer is much more expensive than a collection of loosely coupled computers, having each only a few number of processors. The main advantage of the large multicomputer systems is the speed of the interconnection network joining its processors. However, trends in network technology will make possible to have high performance networks joining loosely coupled systems. In fact, the number of loosely coupled distributed systems being used as parallel computers is quickly increasing [4, 12, 32]. Thus, such systems constitute a low-cost approach entry into the parallel computing domain without necessarily requiring special (and often expensive) hardware. They can be easily upgraded and customized, and even though the performance gap between them and supercomputers is still relatively big, it is expected a notable reduction as high-speed networks become more popular (e.g., ATM or HiPPI networks). We will focus our work in this type of systems.

A typical (loosely coupled) distributed system is composed of a collection of independent computers interconnected through some type of network. In order to cooperate, applications written to span several computers on such a system need to have some mechanism to allow each one of their parts to exchange information.

Within the *communication model* (CM) [17, 18, 28], this information exchange is accomplished by means of explicit transfer of messages: a given node sends a message to another node using the following primitives:

- **send**(data,address)

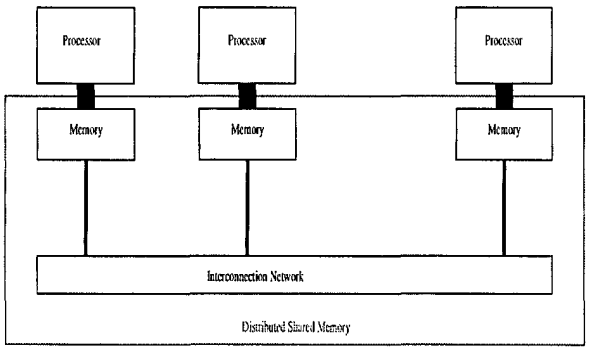

Figure 1: Distributed Shared Memory (DSM).

– **receive**(data)

The CM model provides explicit control over the communication to the programmers, being relatively easy to overlap communication with computation. Nevertheless, that explicit control constitutes the main disadvantage of the CM [17, 18], as it increases its complexity. Thus, it is necessary that the source process of a message knows the target processes. In addition, target processes must exist when data is sent, and must eventually be able to receive that data. Finally, each process must dynamically extract its state when receiving random messages.

On the other hand, the *shared memory model* (SMM) [51] provides a shared address space which can be used by processes in the same way as local memory, even if they are executed concurrently in different processors. Thus, every process can access any address by means of two basic operations:

– data = **read**(address)

– **write**(address,data)

read returns the *data* in *address*, and **write** associates *data* with *address*.

Using the SMM model has several important benefits. In the first place, it hides the particular communication mechanisms employed, thus application developers do not need to be involved in the management of messages, or know whether the application runs on a multiprocessor or on a distributed system (they should know, however, the cost of exchanging information, so they can decide on a performant partition). Besides, it allows complex shared structures to be passed by reference, providing a simple and well known paradigm.

When a SMM is built on top of a distributed system, we get what is known as a DSMM. Even though a DSMM is built on top of a CM (suggesting a decrease in the performance), it has been shown that DSMM can perform well [15]. Factors, such as high locality of references [23], allow communication costs to be compensated against multiple accesses. Multiple replicas can also reduce transfers between nodes, while distributing the communication over a larger interval of time (transfers of data are made on demand), increasing concurrence.

Of course, those paradigms do not have to be necessarily exclusive. Indeed, systems such as SAM [49], Locust [19] and CarlOS [38] support the DSMM, providing at the same time mechanisms for communication and synchronization.

The rest of the paper is organized as follows: Section 2.1 contains an overview of different approaches to implement the DSMM. Section 2.2 addresses implementation mechanisms. Section 2.3 focuses on the problem of consistency between shared units, while Section 2.4 analyzes the importance of the shared units structure. Finally, in Section 3 we give some concluding remarks and suggest future research directions.

2 Characterization of the DSMM

As we have pointed previously, the DSMM has to be built on the CM in such a manner that it transforms the memory access requests into messages between processes. There are a lot of factors that affect the way such transformations take place. In the next sections we identify principal issues that characterize the behavior of DSM systems, presenting some of the proposed implementations.

2.1 Implementation Approaches

The field of research in DSM systems was open up in 1985 by D.R. Cheriton [17]. Since then, a huge amount of work has been done in that area.

The earliest DSM systems provided implementations of the DSMM principally by using *operating system* resources, through virtual memory management mechanisms. IVY [43, 44] constitutes a classical example of a system that implements the DSMM by adding coherence mech-

anisms [1] to a distributed demand paging policy. More recently, Choices [48] incorporates custom designed distributed virtual memory protocols for different applications, which can be altered to trade off characteristics such as resiliee ncto packet loss, network loading, etc. In the same way, the virtual memory management system of Mach [47, 54], a well known operating system kernel that runs on a wide variety of architectures, is designed to be architecture and operating system independent, allowing programmers to handle directly memory as a system resource. Thus, individual memory manager systems that implement the DSMM can be customized for specific applications (e.g., Agora [11] or Midway [10]).

Another approach consists of making use of *hardware* components. For instance, MemNet [22, 52] is an entirely hardware implementation of the DSMM. Every node has a *MemNet-device* that includes both the host's system bus and the network interface, and a *MemNet-cache* (structured in blocks of 32 bytes) divided into a large cache and a reserved area. The cache is used to store the blocks whose reserved area is another node, while the reserved area is used to store the blocks which have to be flushed when a cache area become full. On every memory access, the local *MemNet-device* decides if it can alone handle that request. If it needs the cooperation of other devices, it will send a message and will block the node until receiving a reply. That message will circulate through the net (a token ring), being inspected by every *MemNet-device* (thus, the maximum reply time is limited). If there is a read access, the first *MemNet-device* with a copy will send it to the requester node, while if there is a write access, in addition it will be necessary to invalidate all the replicas in order to maintain some type of consistency between them.

Compilers can also provide support for transforming shared accesses into primitives to manage both coherency and synchronizations. Among the languages for implementing the DSMM we can mention EDS Lisp [30], an extension of an existing sequential language, and Orca [6], a new language designed from scratch in such a way that data shared structures can be accessed through higher level operations.

[1] Basically they are very similar to those used in the Berkeley multiprocessor system [5]

However, currently most of the efforts are addressed in order to implement DSM *environments*. They consist of user-level libraries providing operations that programmers can use directly [21]. For instance, TreadMarks [35] constitutes a DSM environment that implements the DSMM using standard Unix systems such as SunOS and Ultrix without requiring any modification of them (the implementation is done at user level), avoiding the performance problems by focusing on reducing the communication between nodes. Also SAM [49], a shared object system for distributed memory machines, has been implemented as a C library on a variety of platforms: on the CM-5, Intel iPSC/860, Intel Paragon, IBM SP1 and on heterogeneous networks of workstations using PVM. Other DSM environments are Quarks [16] and CarlOS [38].

2.2 Implementation Issues

Placement. The DSMM provides a shared address space which can be used by processes in the same way as local memory.

However, the implementation of such a shared address space requires placing physically shared units (*blocks*) at the local address spaces composing the global one.

That placement can be done *statically* in su ach way that the same block is always placed at the same node. A simple way to implement static placement consists of employing a central server which will store all the blocks. Thus it will manage every access to them [17, 18, 51]. Unfortunately, this implementation needs twice as much messages as the CM. Besides, the central server constitutes a potential bottleneck and although this problem can be solved by using several servers, troubles will still remain if load is not properly distributed.

Another possibility consists of using *dynamic* placement. In this case, blocks are transferred to the requester node before to be accessed. That approach avoids any communication between nodes if data is locally available, although it may force superfluous data transfers.

Location. While finding blocks can be done in a straightforward way when using static placement, if the placement is dynamic it is necessary to follow circulating blocks. In the same way as in the placement of blocks, the simplest way

of controlling circulation consists of using a single node. But analogously to that case, if the node becomes heavily loaded, the entire system will also become overloaded. That problem can be also solved by using several controller nodes, but the effectiveness of that solution still will depend on the proper distribution of load. Also, it requires maintaining a mechanism to find the proper controller node, thus loading the system with a new task.

Replication. To increase concurrency, most of the DSM systems support *replication* of data. That allows different processes to use the same data at the same time. However, and in order to guarantee consistency of shared data, systems using replication must carry out control of replicas.

That control can be done by *invalidating* outdated replicas, as for instance systems as IVY [43] or Clouds [36] or by *propagating* data to outdated replicas. Stumm et al. [1, 51] have proposed several algorithms intended to propagate values. Basically they use a single node, varying only the moment when the propagation takes place.

Whereas propagation is more expensive than invalidation due that, in addition to the invalidating messages, data have to be sent, by using invalidation each block-fault (a block-fault happens when a request can not be locally served) leads to starting a process that will create a new replica, thus increasing latency.

Application Customization. Application-specific protocols constitute a well known approach to improve performance [17, 18]. However, although it has been shown to be an efficient means to reduce extra communication against general purpose protocols [26], it requires writing protocols from scratch, which has been also shown to be difficult and error-prone.

System-provided protocols, even though with reduced performance, seems to be a compromising solution to that problem. Indeed, experimental studies of several shared memory parallel programs [7, 15] support the hypothesis that a system employing a type-specific memory coherency scheme may outperform systems using only a single mechanism.

Nevertheless, that technique requires a relatively small number of identifiable patterns that characterize the behavior of the majority of blocks (so that customized mechanisms can be developed).

Fault tolerance. Fault tolerance and error recovery constitute topics also addressed by using the DSMM. Let's introduce the approach taken by Wu & Kent [53]. They have designed a recoverable distributed virtual memory system which stands up to fail-stop processors [50] without any global re-starting. To do that they use *security copies* that store the necessary data to restart the execution [8]. Given that every process shares the global memory, a backward propagation might be needed if each process simply creates an independent security copy [37]. That happens if a process, after creating a security copy, modifies the value of a page and sends it to another process. Then, if the first process fails, the second one will have to get a security copy created previously to that failure.

To solve this problem, every node creates a security copy before sending any modified page since the last checkpoint (also the operating system or even the program can create additional copies). That is done by using *twin* disk pages. One of them is a security copy. The other is either a work copy or a wrong copy (due to a failure or because it is an old security copy). Thus, every restart, the "right" page is chosen, which will avoid a backward propagation because data do not have to be invalidated in any node.

However, to develop truly reliable systems, both processors and memory failures must be considered. In this way, Hoepman et al. [33] have addressed the construction of self-stabilizing wait-free shared memory objects (these objects occur naturally in systems in which both processors and memory may be faulty).

2.3 Coherency Models

As it as been previously pointed out, the use of replication may increase concurrency. In turn, it is necessary to maintain some kind of *coherency* between replicas.

This problem is similar to the cache coherency problem in multiprocessor systems [5, 24], where several processors share the same data in local caches. In this case, the size of the caches is relatively small, the connections fast and the coherency protocols are implemented by hardware. On the contrary, in distributed systems the communication cost is bigger, and the coherency pro-

tocols are usually implemented by software.

A memory coherency model is characterized by its constrains on initiation and completion of memory accesses [20]. Depending on the properties guaranteed by the coherency model, algorithms will vary in complexity. Programmers must ensure that accesses to data conform to the rules of the model.

Basically coherency models can be split into *non-synchronized* and *synchronized*. Non-synchronized models use only read and write operations while synchronized ones have, in addition, another operations (synchronizations) intended to enforce dependencies at specific points.

Whereas most of the systems support only one coherency model, there are systems which support multiple coherency models within a single parallel program. For instance, Midway [10], which has been implemented using Mach 3.0 with CMU's Unix server on MPIS R3000-based DECstations and 5000/120s, supports release consistency, entry consistency and processor consistency (described below).

2.3.1 Non-Synchronized Models

One of the most widely known non-synchronized models is the *atomic*. It was formalized by Lamport [41] in the case of one writer, and by Misra [46] in the case of several writers. Also the *linearizability condition* for objects introduced by Herlihy and Wing [31] is equivalent to the atomic model when restricted to objects that support read and write operations. This model requires each read operation to obtain the "most recently written" value. It also preserves "real-time" ordering of operations without blocking every process while an operation is taking place. An interesting property of this model is that to guarantee that a system is atomic, it is enough to guarantee that each variable in isolation is atomic, i.e. the atomic model is compositional.

The *sequential* model [40] resembles the atomic, although this one does not preserve any kind of global order between operations (only operations from the same process are forced to preserve real-time ordering). Sequential memory, on the contrary to what happens to atomic memory, does not satisfy the compositional property. Thus, in contrast with the atomic model, it is not possible in general to obtain a sequential system out of the composition of independent sequential components.

On the other hand and in order to improve the performance, other coherency models do not preserve the "most recently written" property.

For instance, the *cache* model (it was introduced by Goodman as *cache consistency* [29]) forces only operations affecting the same variable to "appear" as executed under the sequential model.

That condition is also fulfilled by the *PRAM* (Pipelined RAM) model [45]. Only now, operations appearing as sequential are those in the same process and write ones. That allows pipelining of the write operations, which, even though may potentially delay the effect of write operations to different processes, permits programs take advantage of the better performance of a PRAM implementation as compared to a sequential implementation.

The *causal* model [2], besides to the conditions of the PRAM model, forces read operations to return the value written by the last causally ordered operation [42]. Similarly to PRAM implementations, implementations of the causal model result in far less communications than on sequential ones, providing also a good scalability.

Also, the *processor* model [29] imposes additional conditions on the PRAM one. Now, restrictions are imposed on the write operations to the same variable.

Finally, the *safe* and the *regular* models (they were introduced by Lamport [41] in order to provide a way for implementing stronger models in terms of weaker ones) force the restriction of their executions to the write and non-overlapping operations be atomic. Moreover and in the case of the regular model, read operations are forced to return the value of any previous or overlapping write operation to the same variable.

2.3.2 Synchronized Models

The approach of synchronized models consists of obtaining algorithms that behave sequentially by forcing explicit dependencies between events (by using synchronizations) when necessary. However, that requires identifying dependencies in a proper way, which may induce additional complexity in the design of programs.

by one process be allocated on shared units with no data for other processes. However, the analysis of data dependencies uses to be a difficult task.

3 Conclusions

While many studies have shown the usefulness of the DSMM and a big amount of work has been done to improve the performance of DSM systems, some areas still seem to require paying more attention [16, 19].

Performance of the DSMM is greatly affected by memory access patterns. As a matter of fact, the consistency mistmach between the DSM systems and the application programs constitutes one of the most important factors that favors low performance. Therefore, an important approach in order to avoid performance problems consists of exploiting data dependencies. However, that requires knowing access patterns, which may not be always available.

Real-time implementations and auto-configuring systems are other areas which also need deeper study.

Contrary to available message passing systems such as MPI or PVM, the DSMM has not yet had a significant impact on non-researcher users. The earliest systems provided experimental environments useful to be used as benchmarks. Now, new generation DSM systems are overcoming former problems, which allow us to envisage a wider acceptance of the DSMM.

References

[1] A. Krishnamurthy and K. Yelick. Optimizing parallel programming with explicit synchronization. In *Programming Language Design and Implementation*, June 1995.

[2] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, August 1995.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[4] T.E. Anderson, D.E. Culler, and D.A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[5] J. Archibals and J.L. Baer. Cache coherence protocols: Evaluation using a multiprocessor model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[6] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[7] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 168–176. ACM, 1990.

[8] P.A. Bernstein, N. Goodman, and V. Hadzilacos. Recovery algorithms for database systems. In *IFIP*, pages 799–807, 1983.

[9] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[10] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *COMPCON*, 1993.

[11] R. Bisiani and A. Forin. Architectural support for multilanguage parallel programming on heterogeneus systems. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 21–30, October 1987.

[12] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21th International Symposium on Computer Architectures*, April 1994.

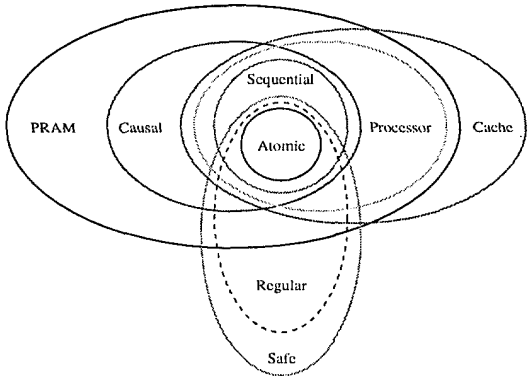

Figure 2: Relations between non-synchronized models: The sets represent the executions they allow.

We begin the description of synchronized models with the *weak* model [25]. It only uses a single synchronization type (*weak*). Roughly speaking, it forces dependencies between synchronizations and the preceding and following operations. However, slightly different versions of this model have been proposed varying the set of operations forced to be related with synchronizations.

Contrary to the weak model, both the *lazy-release* (LR) [34] and the *eager-release* (ER) models [27] use two types of synchronizations (*acq* and *rel*). That permits addressing typical problems (e.g., implementing critical sections) in an easier way.

Whereas the ER model sets up dependencies from the *rel* synchronizations to the whole set of operations, the LR model sets up dependencies from the *rel* synchronizations to the *acq* synchronizations.

Moreover, and independently from the set up dependencies, they require the first synchronization operation for each process to be an *acq* synchronization and impose an alternating use of the *acq* and *rel* synchronizations. Besides, after an *acq* synchronization completes, the next completing synchronization has to be executed by the same process.

The last synchronized model we introduce is the *entry* [9]. It is very similar to the LR model. Only now synchronizations are associated with "synchronization variables". As well as the release models, it requires the first synchronization for each process has to be an *acq* synchronization and it imposes an alternating use of the *acq* and *rel* synchronizations. Also, the *rel* synchronizations must be executed on the same variable that the previous *acq* synchronization, and after an *acq* synchronization completes, the next completing synchronization to the same variable has be executed by the same process.

2.4 Shared Data Characteristics

DSM systems are intended to provide an address space where data can be shared among several nodes. Therefore it is not surprising that the characteristics of those data may affect the behavior of such systems.

Heterogeneous size and structure greatly affect the system performance. That is due to the data conversion when interchanging information between modules (e.g., MMUs having to manage pages with different sizes [55]).

On the other hand, in loosely coupled distributed systems, sending a big packet of data is not, relatively speaking, much more expensive than sending a small packet. Therefore, if programs have a high locality and we use dynamic placement, using a big size of the shared units may reduce the number of block-faults. But the more we increase the size the more *false sharing* arises. False sharing occurs when two non-related variables, each one referred from a different node, are located in the same shared unit, thereby inducing unnecessary coherence operations. It is believed to be a serious problem for parallel program performance. This belief is also supported by experimental evidence [13].

Multi-writer protocols address that problem by allowing multiple nodes to write one block at the same time and merging changes in a consistent way at specified points. Examples of systems using multi-writer protocols are Munin [14] and TreadMarks [35].

Delayed protocols attack false sharing by communicating updates at the latest possible moment. For instance, synchronized models, because they only suffer delays at synchronization points, are used to reduce false sharing.

Systems supporting structured data provide the user with control of the shared units, which can be used to avoid false sharing. Orca [6], Indigo [39], Linda [3] or Agora [11] are examples of systems that allow data structures to be shared between nodes. In this case, a careful analysis must be done in such a way that data manipulated mostly

[13] W.J. Bolosky and M.L. Scott. False sharing and its effects on shared memory performance. Technical report, Computer Science Department, University of Rochester, 1994.

[14] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. *Operating System Review*, 25(5):152–164, October 1991.

[15] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *Transactions on Computer Systems*, 13(3):205–244, August 1995.

[16] J.B. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proceedings of the 5th Workshop on Hot Topics in Operating Sytems*, May 1995.

[17] D.R. Cheriton. Preliminary thoughts on problem-oriented shared memory: A decentralized approach to distributed systems. *ACM Operating System Review*, pages 26–33, October 1985.

[18] D.R. Cheriton. Problem-oriented shared memory: A decentralizad approach to distributed system design. In *Proceedings of the Sixth International Conference on Distributed Computer Systems*, pages 190–197. IEEE, May 1986.

[19] T. Chiueh and M. Verma. A compiler-directed distributed shared memory system. In *International Conference on Supercomputing*, 1995.

[20] V. Cholvi-Juan. *Formalizing Memory Models.* PhD thesis, Department of Computer Science, Polytechnic University of Valencia, December 1994.

[21] V. Cholvi-Juan and J.M. Bernabéu-Aubán. Implementing a distributed compiler library that provides a $\mathcal{N}$-mixed memory model. In *IEEE/USP International Workshop on High Performance Computing*, pages 229–244, March 1994.

[22] G. Delp. *The Architecture and Implementation of MemNet: A High Speed-Shared Memory Computer Communication Network.* PhD thesis, Computer Science Department, University of Delaware, 1988.

[23] P.J. Denning. On modeling program behavior. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 937–944, 1972.

[24] M. Dubois and C. Scheurich. Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, pages 9–21, February 1988.

[25] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 434–442, June 1986.

[26] B. Falsafi, A.R. Lebeck, S.K. Reinhardt, I. Schoinas, M. Hill, J.R. Larus, A. Rogers, and D.A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing*, November 1994.

[27] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26. ACM, May 1990.

[28] D.K. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3):258–283, August 1988.

[29] J.R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[30] C. Hammer and T. Henties. Using a weak coherency model for a parallel lisp. In A. Bode, editor, *Distributed Memory Computing*, volume 487 of *Lecture Notes in Computer Science*, pages 42–51. 1991.

[31] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming*

Languages and Systems, 12(3):463–492, July 1990.

[32] M.D. Hill, J.R. Larus, and D.A. Wood. Tempest: A substrate for portable parallel programs. In *COMPCON Spring'95*, March 1995.

[33] J.-H. Hoepman, M. Papatriantafilou, and P. Tsigas. Toward self-stabilizing wait-free shared memory objects. Technical Report CS-R9514, Centrum voor Wiskunde en Informatica, 1995.

[34] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[35] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Winter USENIX*, 1994.

[36] Y.A. Khalidi. *Hardware Support for Distributed Object-Based Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1989.

[37] K.H. Kim. Programmer-transparent coordination of recovery concurrent processes: Philosophy and rules for efficient implementation. *IEEE Transactions on Software Engenieering*, 14(8):810–821, June 1988.

[38] P.T. Koch, R.J. Fowler, and E. Jul. Message-driven relaxed consistency in a software distributed shared memory. In *First Symposium on Operating System Design and Implementation*, pages 75–85. USENIX Association, November 1994.

[39] P. Kohli, M. Ahamad, and K. Schwan. Indigo: User-level support for building distributed shared abstractions. Technical Report GIT-ICS-94/53, School of Information and Computer Science, Georgia Institute of Technology, March 1995.

[40] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[41] L. Lamport. On interprocess communication: Parts I and II. *Distributed Commputing*, 1(2):77–1101, 1986.

[42] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1991.

[43] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[44] K. Li and P. Hudak. Memory coherence in shared memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[45] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[46] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[47] R.F. Rashid et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[48] A. Sane, K. MacGregor, and R. Campbell. Distributed virtual memory consistency protocols: Design and performance. In *Second IEEE Workshop on Experimental Distributed Systems*, 1990.

[49] D.J. Scales and M.S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *First Symposium on Operating Systems Design and Implementation.* USENIX.

[50] F.B. Schneider. Fail-stop processors. In *Proceedings IEEE*, pages 66–70. IEEE, 1983.

[51] S. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1988.

[52] S. Sureshchandran and T.A. Gonsalves. The performance of the MemNet distributed shared memory architectures. Technical Report TR-CSE-90-02, Department of Computer Science, Indian Institute of Technology, January 1990.

[53] K.-L. Wu and W.K. Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[54] M. Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76. ACM, November 1987.

[55] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environments. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 30–37. IEEE, 1990.