Distributed Shared Memory: A Survey of Issues and Algorithms

Bill Nitzberg and Virginia Lo

CIS-TR-90-26
January, 1991

Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

Distributed Shared Memory: A Survey of Issues and Algorithms*

Bill Nitzberg† Virginia Lo*

Dept. of Computer Science
University of Oregon
Eugene, OR 97403

Tel (503) 346-4408
FAX (503) 346-3660

Email: lastname@cs.uoregon.edu

January 7, 1991

Abstract

A distributed shared memory (DSM) is an implementation of the shared memory abstraction on a multicomputer architecture which has no physically shared memory. Shared memory is important (as a programming model) not only because of the vast number of existing applications which use it, but also because it is a more appropriate paradigm for certain algorithms. The DSM concept was demonstrated to be viable by Li, in IVY. Recently, there has been a surge of new projects which implement DSM in a variety of software and hardware environments.

This paper gives an integrated overview of distributed shared memory. We discuss theoretical lower bounds on the performance of DSM systems, design choices such as structure and granularity, access, coherence semantics, scalability, and heterogeneity, and open problems in DSM. In addition, we describe algorithms used to implement and improve efficiency: reducing thrashing, eliminating false sharing, matching the coherence protocol to the type of sharing, and relaxing the semantics of the memory coherence provided. A spectrum of current DSM systems are used as illustrative examples.

Keywords: Distributed shared memory, cache coherence, synchronization, memory consistency, distributed systems.

*partially supported by NSF Grant CCR-8808532
†partially supported by a Tektronix Research Fellowship

Contents

1 Introduction

As we slowly come up against the physical limits of processor and memory speed, it is becoming more attractive to use multiprocessors to increase computing power. In particular, two kinds of parallel processors have become popular: the tightly-coupled shared-memory multiprocessor and the distributed memory multiprocessor. The tightly-coupled multiprocessor system consists of multiple CPU's and a single global physical memory. This type of system is more straightforward to program because it is a natural extension of a single CPU system. However, a tightly-coupled multiprocessor suffers from one of the same drawbacks that brought us towards parallel computing in the first place. This type of multiprocessor has a bottleneck: main memory is accessed via a common bus—a serialization point—that limits the size to a few tens of processors.

Distributed memory multiprocessors, however, do not suffer from this drawback. These systems consist of a collection of independent computers connected by a high speed interconnection network. These systems range from single user back-end machines, to workstations connected by an Ethernet, to a wide area network of heterogeneous computers. If the network topology is chosen carefully, then the system can contain many orders of magnitude more processors than a tightly coupled system. As all communication between concurrently executing processes must be performed over the network in such a system, the programming model (until recently) was limited to a message passing paradigm. However, recent systems

have implemented a shared memory abstraction on top of distributed memory systems—giving these systems the illusion of physically shared memory and allowing the programmer to program using the shared memory paradigm.

Distributed Shared Memory (DSM) is a virtual address space shared among processes on loosely coupled processors (see figure 1). It has advantages over tightly-coupled shared memory multiprocessors as well as over message passing systems:

- It is often more natural to design a parallel algorithm using the shared memory paradigm rather than explicit message passing. For example, passing structures containing pointers is either impossible or expensive (in programming and communication cost) on message passing systems. Consider passing a circular list.
- Many programs have been written for shared memory multiprocessors. A DSM system allows these programs to be executed on distributed memory multiprocessors, improving their portability. It also allows the program to take advantage of the increased parallelism available on such a machine.
- The architectures of distributed memory multiprocessors are more easily scaled upwards. Shared memory multiprocessors are limited to tens of processors as they share a single pathway to the memory, whereas a distributed memory multiprocessor can have 1000's of processors.
- Distributed memory machines are cheaper to build than tightly coupled shared memory multiprocessors—they can consist of stock hardware coupled with stock networking. Shared memory multiprocessors must have complex interfaces to a shared, high-speed bus.
- Programs which require large amounts of memory can be run on DSM systems because a distributed system with thousands of nodes will have an enormous physical memory available. A DSM system will make this physical memory appear as a single shared memory, but avoid the disk latency associated with traditional virtual memory paging systems.
- Exchanging data in page sized units is more efficient than sending many smaller messages, one at a time. In this way, DSM systems amortize the cost of communication over message passing systems.

DSM has been an active area of research since the early 1980's, although the foundations for this work in cache coherence and memory management have been around much longer. The goals and issues of DSM are similar to those of the caches of multiprocessors or networked file systems, the memories of a non-uniform memory access multiprocessor, and the management system of a distributed/replicated database. Because of this similarity, many algorithms and lessons learned in these domains can be transferred to DSM systems and vice versa. However, all of these systems have unique features (such as communication latency) which makes it necessary to consider each separately.

Research has shown that the advantages of DSM can be realized with reasonably low runtime overhead and reasonably small modifications to operating system algorithms. The

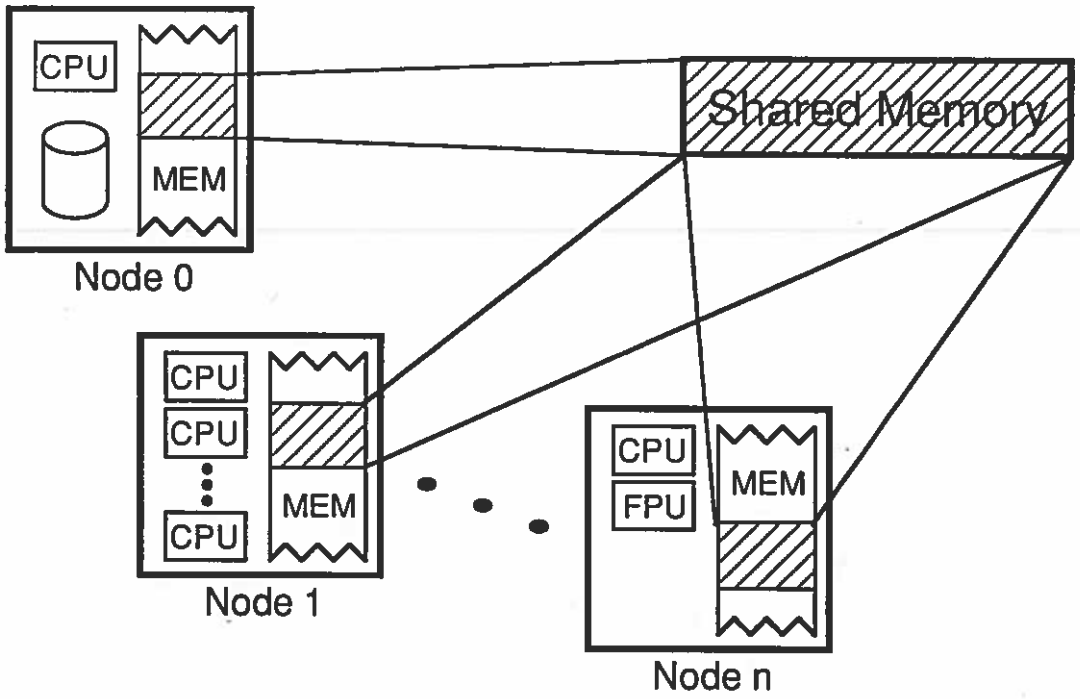


Figure 1: Distributed Shared Memory

existence proof for DSM was given by Li in IVY [LH86][Li86][Li88]. He showed not only that DSM is viable, but also that super-linear speedups can be realized in certain instances, most likely due to the increased overall physical memory and cache sizes. There are three distinct approaches to implementing DSM: software implementations in which sharing and coherence are achieved through the paged memory management mechanisms, hardware implementations which extend traditional caching techniques to scalable architectures, and object based systems in which the unit of sharing is the variable sized application dependent object. These systems have been designed on common networks of workstations or mini-computers, special purpose message passing machines (such as the Intel iPSC/2), custom hardware, and even heterogeneous systems.

Current research has shown that DSM algorithms must be sensitive to the memory access patterns of applications. Stumm and Zhou [SZ90] pairwise compare four algorithms for supporting DSM, and show that the efficiency of these algorithms is directly affected by the memory faulting rate of the application. Bennett, Carter, and Zwaenepoel show in Munin [BCZ90a][BCZ90b] that there are at least nine different types of memory access patterns exhibited by typical parallel applications, and give algorithms tailored to support each separately. Finally, the importance of locality of reference is demonstrated in the context of DSM by Fleisch and Popek with Mirage [FP89][Fle89]. The focus of current research emphasizes fine tuning the efficiency of DSM systems.

This paper gives an integrated overview of distributed shared memory covering memory coherence, design choices, implementation methods, and open research areas. We discuss these issues using illustrative examples from a spectrum of DSM systems. Figure 2 and the Appendix summarize these systems.

2 Theoretical Lower Bounds

To be efficient, an implementation of DSM should allow shared memory programs to execute with only a small constant multiple (very close to 1) of the number of operations necessary to perform memory accesses. In general, this is impossible. Upfal and Wigerson have shown [UW84] that it takes at least $\Omega(T\frac{\log n}{\log\log n})$ steps on a distributed memory multiprocessor to perform T shared memory operations of a PRAM, given that there are n processors. They derive this lower bound by simulating a PRAM on a complete network of processors, so even with the most general interconnection network, the amortized time to access shared memory is more than a constant.

However, they show that on a bounded degree network distributed memory multiprocessor, a more realistic interconnection network than fully connected, a shared memory access can be performed in $O((\log n \log\log n)^2)$ steps. Karlin [KU86][Kar87] obtains a better bound by looking at probabilistic complexity. She shows how to simulate shared memory in only $\Theta(T\log n)$ steps for T operations. Both of these algorithms simulate shared memory by using hashing functions to distribute the shared variables to reduce contention. Karlin's algorithm uses random routing to avoid communication bottlenecks.

Of course, these are theoretical bounds. Actual performance of real systems is much better. In fact, most of the systems perform at least as well as their shared memory counterparts on many applications.

3 Memory Coherence

In order for programmers to write correct programs on any shared memory machine, they must understand how parallel memory updates are propagated throughout the system. The most intuitive semantics for memory coherence is *strict consistency*[1] which means that a read operation returns the most recently written value. This is most often the naive users' model of shared memory. However, "most recently" is an ambiguous concept in a distributed system. For this reason, and in order to improve performance, some DSM systems only provide a reduced form of memory coherence. Relaxed semantics require less synchronization and less data movement—this makes them more efficient. However, existing programs which depend on a stronger form of consistency may not perform correctly if executed in a system which only supports a weaker form. There are several forms of memory coherence:

Strict Consistency [CF78] A read returns the most recently written value. This form of consistency is what most programmers intuitively expect. However, it implies a total ordering on all memory operations in the system so the *most recent* write can be determined. This forced total ordering leads to inefficiency.

Sequential Consistency [Lam79] A system is sequentially consistent if the result of any execution is the same as if the operations of all the nodes were executed in some sequential order, and the operations of each individual node appear in this sequence in the order specified by its program. A sequentially consistent multiprocessor is indistinguishable from a multi-threaded uniprocessor.

Concurrent Consistency [Sch89] A system is concurrently consistent if it provides sequential consistency except for programs which explicitly test for sequential consistency or take access timings into account. Note that it is wasteful for a program which expects sequential consistency to explicitly test for it's existence.

Processor Consistency [Goo89] Writes issued by a processor are never seen out of order. This allows updates to propagate through the system in a pipelined fashion, although two simultaneous reads of the same location from different processors may yield different results.

Weak Consistency [DSB88] Synchronization accesses are sequentially consistent, and before a synchronization access can be performed all previous regular accesses must be completed and before a regular access can be performed all previous synchronization accesses must be done. This basically leaves consistency up to the programmer. The memory will only be consistent immediately after a synchronization operation.

Release Consistency [GLL+90] Basically weak consistency, but synchronization accesses must only be processor consistent with respect to each other. Synchronization operations are broken down into *acquire* and *release*. All pending acquires must be done

[1] Strict consistency is defined in [CF78] as *memory coherence* and appears in [Sch89] as *strict coherence*. The terms "coherence" and "consistency" are used somewhat interchangeably in the literature. We use "coherence" as the general term for the semantics of memory operations, and "consistency" to refer to a specific kind of memory coherence.

before you can do a regular access, and all regular accesses must be done before doing a release. This is just a further relaxation of *weak* consistency without very much loss of coherence.

Since time is not mentioned in any of the consistency semantics above, nothing can be said regarding the propagation of updates throughout the system. This makes it impossible (in general) to use shared memory to perform any kind of mutually recursive synchronization operations without the possibility of deadlock. In practice, however, coherence protocols propagate updates in a timely manner.

4 DSM Design Choices

A DSM system must address certain design issues: structure, granularity, access, coherence semantics, scalability, and heterogeneity. By looking at these design issues for several real implementations of DSM, we can better understand the intricacies of such a system. Figure 2 summarizes these design choices for a collection of DSM systems, as well as the implementation issues discussed in section 5.

4.1 Structure & Granularity

The structure and granularity of a DSM system are closely related. Structure refers to the layout of the shared data in memory. Most DSM systems do not structure memory (it is a linear array of words), but some structure the data as objects, language types, or even an associative memory. Granularity refers to the size of the unit of sharing: byte, word, page, complex data structure.

IVY [LH86][Li86][Li88], one of the first transparent DSM systems, implemented shared memory as virtual memory. This memory was unstructured, and was shared in 1k byte pages. It is convenient in systems implemented using the virtual memory hardware of the underlying architecture (such as IVY) to choose a multiple of the hardware page size as the unit of sharing. Hardware implementations of DSM typically support smaller grain sizes. For example, DASH [LLG+90][GLL+90] and MemNet [DF86][Del88][TSF90] also support unstructured sharing, but the unit of sharing is 16 and 32 bytes respectively—typical cache line sizes. PLUS [BR90] is somewhat of a hybrid of the two: the unit of replication is a page, while the unit of coherence is one 32 bit word. As shared memory programs exhibit locality of reference, a process will most likely access a large region of its shared address space in a small amount of time. This makes larger "page" sizes better, because there is less overhead associated with paging activity. However, sharing may also cause contention, and the larger the page size, the more chance there is that more than one process will require access to that page. In this case, a smaller page size is better, as it reduces the possibility of *false sharing*. This occurs when two unrelated variables (each used by different processes) are placed in the same page—the page appears shared, even though the original variables were not. Another factor affecting the choice of page size is the necessity of keeping directory information about the pages in the system. The smaller the page size, the larger the directory must be to keep track of the pages.

One method of structuring the shared memory is by data type. In this case, it is either structured as objects in distributed object oriented systems (e.g. Emerald [JLHB88]

	Current Implementation	Structure & Granularity	Access	Coherence Semantics	Coherence Protocol	Sources of Improved Performance	Heterogeneous Support
Clouds	SW Sun-3 DSMC/Ra Kernel	object segments	trans.	strict	write-invalidate		no
DASH	HW modified SGI 4D/240 workstations; mesh	16 bytes	trans.	release	write-invalidate	relaxed coherence	no
IVY	SW Apollo workstations; Apollo ring; Apollo Domain	1K pages	trans.	strict	write-invalidate	pointer chain collapse, selective broadcast	no
Linda	SW/HW variety of environments	tuples	access funcs.	strict/no mutable data	broadcast (no replication)		?
Memnet	HW token ring	32 bytes	trans.	strict	write-invalidate		no
Mermaid	SW Sun workstations + DEC Firefly multiprocessor Mermaid/native OS	8K pages (single data type)	trans.	strict	write-invalidate	See IVY	yes
Mirage	SW Vax 11/750; ether-net; Locus Distributed OS	512 byte pages	trans.	strict	write-invalidate	time window coherence protocol	no
Munin	SW Sun workstations; ether-net; V kernel + Presto parallel programming env.	objects	language level trans.	strict	type-specific (delayed write update for read-mostly)	delayed update queue	no
PLUS	HW/SW Motorola 88000; CalTech mesh; PLUS kernel	page for sharing; word for coherence	trans.	processor	non-demand write-update	specialized synchronization primitives to complement DSM	no
Shiva	SW iPSC/2; hypercube; Shiva/native OS	4K pages	trans.	strict	write-invalidate	See IVY	no

Figure 2: DSM Design Issues

and Clouds [RK89][RK88] [RAK88][RK89]), or it is structured as variables in the source language (e.g. Shared Data Object Model [BKT89] and Munin [BCZ90a][BCZ90b]). In these systems, the grain size is variable as the sizes of objects and data types vary greatly, and as such, may be better matched to the application. In addition, these systems do not suffer from false sharing.

Finally, the shared memory can be structured like a database. One system that has such a model, Linda [BCGL87][WL88], orders its shared memory as an associative memory called a *tuple space*. This allows the location of data to be separated from its value, but it also requires that the programmer use special access functions to interact with the shared memory space.

4.2 Access

The shared memory can be accessed either transparently or by calling special shared memory access functions. Early DSM systems used access functions, and Linda may be considered to have shared memory access functions, as there is no way to access the shared memory except via the Linda primitives. Shared memory is considered transparent if access to it is represented by normal language level variable access.

Accessing shared memory through explicit primitives defeats some of the purposes of having shared memory in the first place. Fine grained sharing is no longer trivial. As in a message passing system, all communication must be explicitly marked and packaged into pieces. In addition, having transparent DSM makes it easier to port applications written for shared memory multiprocessors to distributed memory machines. Newer systems and hardware implementations support transparent access to DSM, allowing shared memory to be accessed and modified through normal machine instructions.

4.3 Coherence Semantics

DSM systems try to balance memory coherence with performance. In order to ensure that existing parallel applications will run on a DSM system, the system must provide strict consistency. Most systems provide strict consistency. However, in an effort to improve efficiency, some of the newer systems (especially hardware implementations) have chosen to provide a looser form of coherence. PLUS, for example, provides processor consistency, and DASH only provides release consistency. In the RISC philosophy, both of these systems have mechanisms for forcing coherence, but their use must be explicitly specified by higher level software (compiler or perhaps even the programmer). This relaxation of the coherence semantics allows shared accesses to be performed more efficiently.

4.4 Scalability

One of the benefits of DSM is that it theoretically scales better than a tightly coupled shared memory multiprocessor. The limits of scalability are greatly reduced by two factors: central bottlenecks (such as the bus of a tightly-coupled shared memory multiprocessor), and global common knowledge operations and storage (such as broadcast messages or full directories, size proportional to the number of nodes). Li [Li86] went through several iterations of refining a cache coherence protocol before arriving at his dynamic distributed

manager algorithm which avoids centralized bottlenecks. IVY, however, and most other DSM systems are currently implemented on top of Ethernet (itself a centralized bottleneck) which can only support about 100 nodes at a time. This is most likely an artifact of these systems being research tools rather than any real design flaw. Shiva [LS89] is an implementation of DSM on the Intel iPSC/2 hypercube, and should scale nicely. Nodes in the DASH system are connected on two meshes which would imply that the machine should be expandable, however, the DASH prototype currently uses a full bit vector (one bit per node) to keep track of page replication.

4.5 Heterogeneity

At first glance, sharing memory between two machines with different architectures seems almost impossible. The machines may not even use the same representation for basic data types (integers, floating point numbers, etc.). It is a bit easier if the DSM system is structured as variables or objects in the source language. In this case, a DSM compiler can add conversion routines to all accesses to shared memory. In Agora [BF87][FBC87], memory is structured as objects which are shared among heterogeneous machines. Another novel approach is being explored in Mermaid [LSWZ89]. Mermaid shares memory in pages where a page can only contain one type of data. Whenever a page is moved between two architecturally different systems, a conversion routine is called to convert the data in the page to the appropriate format. Although heterogeneous DSM might allow more machines to participate in a computation, the overhead of conversion seems like it would outway the benefits.

5 Implementation of DSM

A DSM system must automatically transform shared memory access into interprocess communication. Implementing this requires algorithms to handle locating and accessing shared data, maintaining coherence, and replacing data. The DSM system may also have additional schemes to improve performance. These algorithms represent direct support for DSM. In addition, operating system algorithms to support process synchronization as well as memory management must be specifically tailored for DSM. We focus on the algorithms used in IVY, DASH, Munin, PLUS, Mirage, and MemNet since these systems illustrate the concepts we wish to cover. Figure 2 summarizes these implementation alternatives. Other systems are mentioned in the Appendix. Both Li in [Li86] and Stumm and Zhou in [SZ90] give a good evolutionary overview of algorithms to support static, migratory, and replicated data.

5.1 Data Location and Access

In order to share data in a DSM system, a program must be able to find and retrieve the data it needs. If the data does not move around in the system—it only resides in a single static location, then locating it is easy. All processes can simply "know" where to obtain any piece of data. For example, some Linda implementations use hashing on the tuples to distribute the data statically. This has the advantages of being simple and fast, but may

cause a bottleneck if the data is not distributed properly (e.g. all the shared data ends up on a single node).

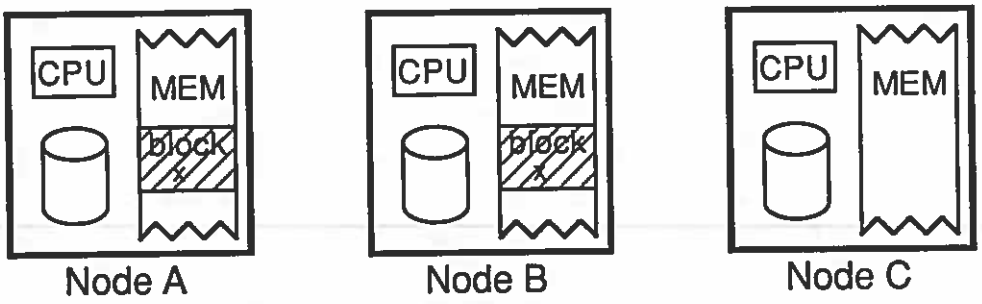
An alternative is to allow the data to freely migrate throughout the system. This allows the data to be dynamically redistributed so that the data can be moved to where it is being used. However, locating the data then becomes more difficult. In this case, the simplest method of data location is to have a centralized server. The server can keep track of the locations of all shared data. The centralized method suffers from two drawbacks: the server serializes location queries which reduces parallelism, and the server may become heavily loaded which will slow the entire system down.

Another method (used by MemNet) is to broadcast requests for data. Unfortunately, broadcasting is expensive as all nodes must process a broadcast request, not just the nodes containing the data. The network latency of a broadcast may also be significant, requiring accesses to take extended periods of time to complete.

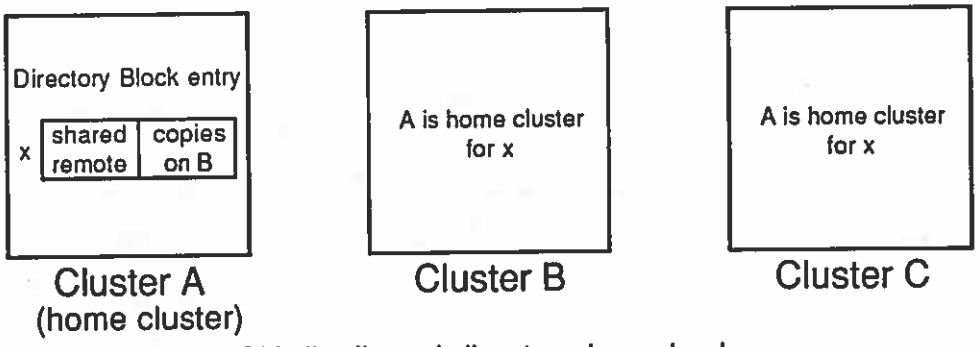
To avoid broadcasts and distribute the load more evenly, several systems use an owner based distributed scheme.[2] In this scheme, each piece of data has an associated owner—a node which has the primary copy of the data. The owner changes as the data migrates throughout the system. When another node wishes to get a copy of the data, a request is sent to the owner. If the owner still has the data, it is returned. The owner, however, may have given the data to some other node (the new owner). In this case, the old owner forwards the request to the new owner. The drawback with this scheme is that a request may be forwarded many times before reaching the current owner. This can be time consuming, and in some cases, more wasteful than broadcasting. In Li's scheme [Li86], all of the nodes involved in forwarding a request (including the requester) are given the identity of the current owner. This "collapsing of pointer chains" helps reduce the forwarding overhead and delay.

When data is replicated, the DSM system must provide a means for keeping track of the replicated copies. Figure 3 illustrates the data location schemes used in DASH, IVY, and PLUS. DASH uses a distributed directory-based scheme, implemented in hardware. Briefly, the directory for a given cluster (node) keeps track of the physical blocks in that cluster. Each block is represented by a directory entry which specifies whether the block is *unshared remote* (local copy only), *shared remote*, or *shared dirty*. If the block is shared remote, the directory entry also indicates the location of replicated copies of the block; if the block is shared dirty, the directory entry indicates the location of the single dirty copy. Note that only the home cluster possesses the directory block entry. Non-local data is accessed for reading by sending a message to the home cluster. Ivy's dynamic distributed scheme also supports replicated data. A *ptable* on each node contains an entry for each page which indicates the probable location for the referenced page. As described above, data is located by following the chain of probable owners. Finally, the copy-list scheme implemented by PLUS uses a distributed linked list to keep track of replicated data. Memory references are mapped to the physically closest copy by the page map table. A discussion of how writes are handled for replicated data appears in the next section.

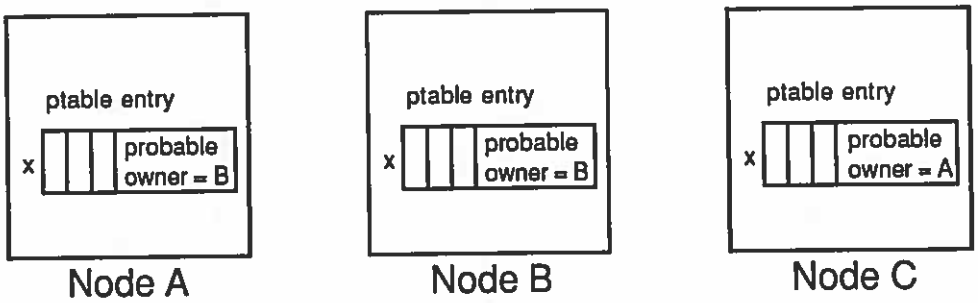
[2]This scheme is independent of data replication, but is seen mostly in systems which support both data migration and replication.



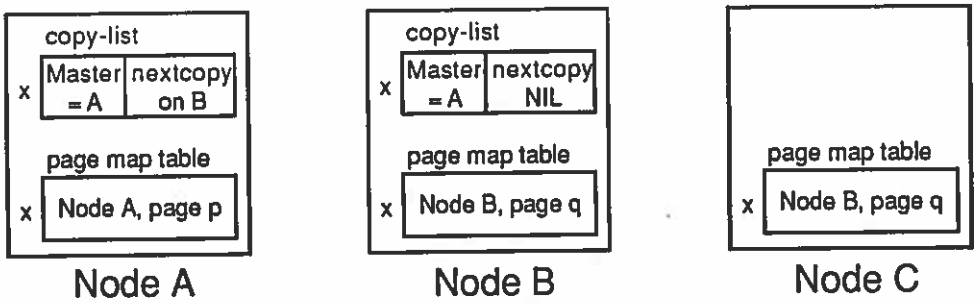
(a) Block x is replicated on Nodes A and B, referenced by Node C



(b) DASH distributed directory-based scheme



(c) IVY - dynamic distributed scheme



(d) PLUS copy-list scheme

Figure 3: DSM Data Location Schemes

5.2 Coherence Protocol

All DSM systems provide some form of memory coherence. If the shared data is not replicated, then enforcing memory coherence is trivial. Requests are automatically serialized (in the order they occur) by the underlying network. A node which is handling shared data can merely perform each request as it is received, and this will ensure strict memory consistency—the strongest form of coherence. Unfortunately, this serializing of data access causes a bottleneck, and defeats one of the major advantages of DSM: parallelism.

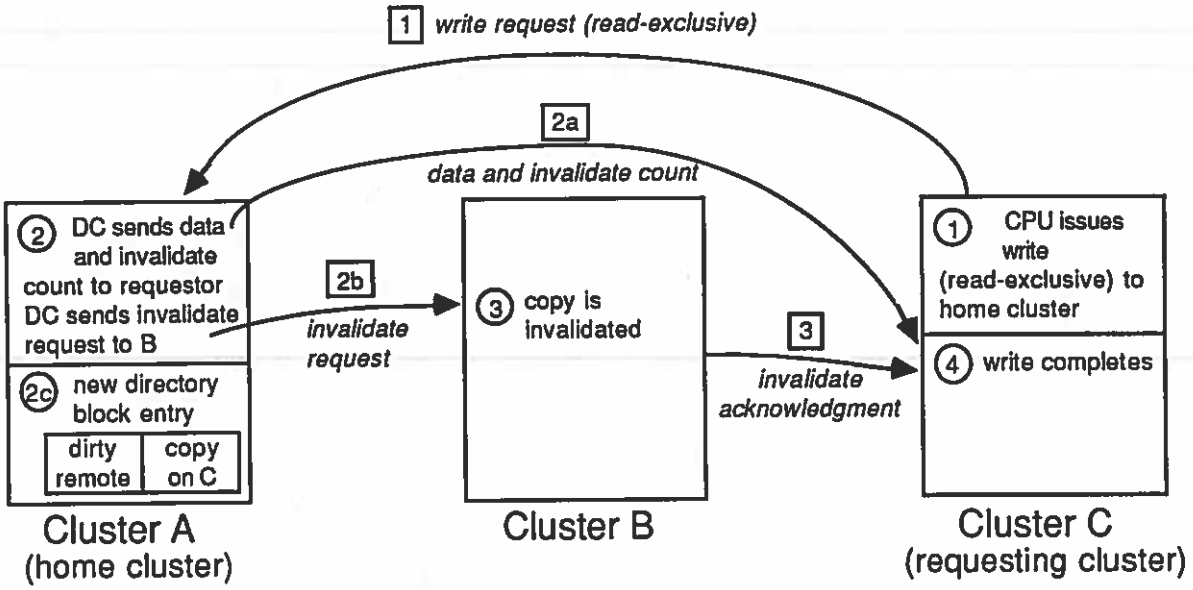
To increase parallelism, virtually all DSM systems allow data to be replicated. So, for example, multiple reads may be performed in parallel. However, replication complicates the coherence protocol. There are basically two types of protocols that handle replication: write-invalidate and write-update. In a write-invalidate protocol, there can be many copies of a read-only piece of data, but only one copy of a writable piece of data. It is called write-*invalidate* because all copies of a piece of data except one are *invalidated* before a write can proceed. In a write-update scheme, however, a write causes all copies of a piece of data to be updated.

The majority of DSM systems have write-invalidate coherence protocols. All the protocols for these systems are similar. Each piece of data is tagged with a status which indicates whether the data is valid, whether it is read-only, whether it is shared, and whether it is writable. On a read, if the data is valid, it is returned immediately. If the data is not valid, a read request is sent (the location of a valid copy is determined via one of the mechanisms described above), and a copy of the data is returned. If the data was writable on another node, this request will cause it to become read-only. The copy remains valid until an invalidate request is received. On a write, if the data is valid and writable, the request is satisfied immediately. Otherwise, an invalidate request is sent out along with a request for a copy of the data (in case only part of it is to be written). When the invalidate completes, the data is valid locally and writable, and the original write request may complete.

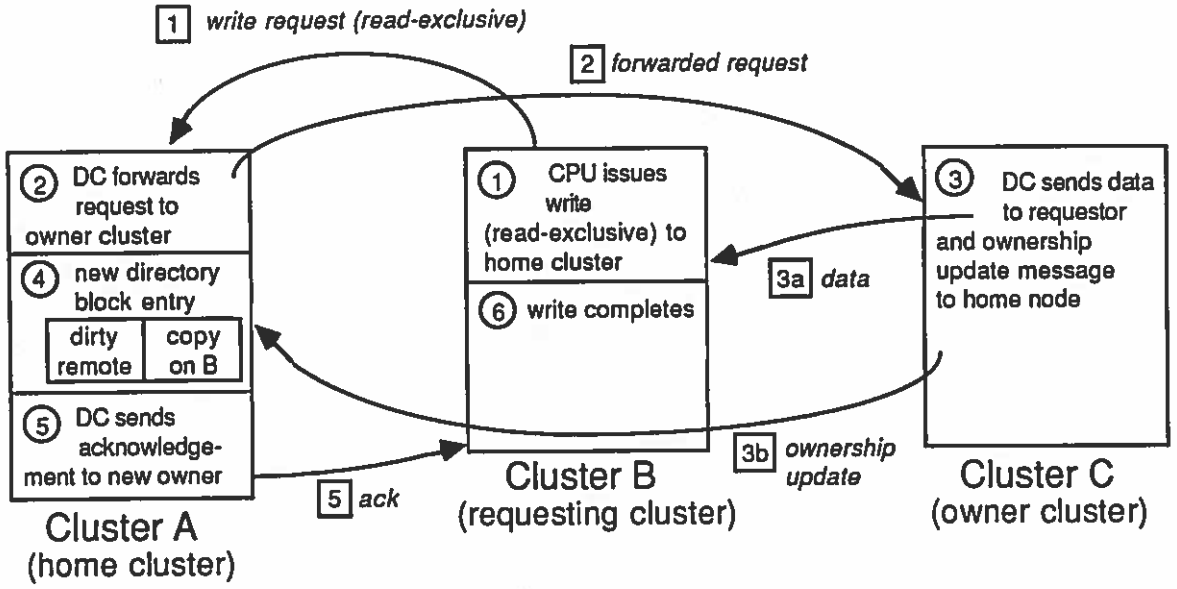
Figure 4 illustrates the DASH directory based coherence protocol. The sequence of events and messages shown in Figure 4(a) occur when the block to be written is in *shared-remote* state (multiple read-only copies) just before the write. Figure 4(b) shows the events and messages that would occur when the block is to be written is in *shared-dirty* state (single dirty copy) just before the write. DASH's coherence protocol supports release consistency. Further details of the DASH coherence protocol and the methods used to fine-tune the protocol for high performance are given in [LLG+90].

Li shows that the write-invalidate protocol performs well for a variety of applications. In fact, he shows super-linear speedups for a linear equation solver and a three dimensional partial differential equation solver. However, he dismissed using a write-update protocol at the onset with the reasoning that it would be inefficient due to network latency. It turns out that write-update can be implemented efficiently, and PLUS is one system which uses a write-update protocol.

Figure 5 traces the PLUS write-update protocol which begins all updates with the block's master node, then proceeds down the copy-list chain. The write operation is completed when the last node in the chain sends an acknowledgment message to the originator of the write request. The PLUS protocol provides the weaker form of consistency known as processor consistency.



(a) Data is shared remote



(a) Data is dirty remote (after events depicted in (a))

Figure 4: Simplified DASH write-invalidate protocol

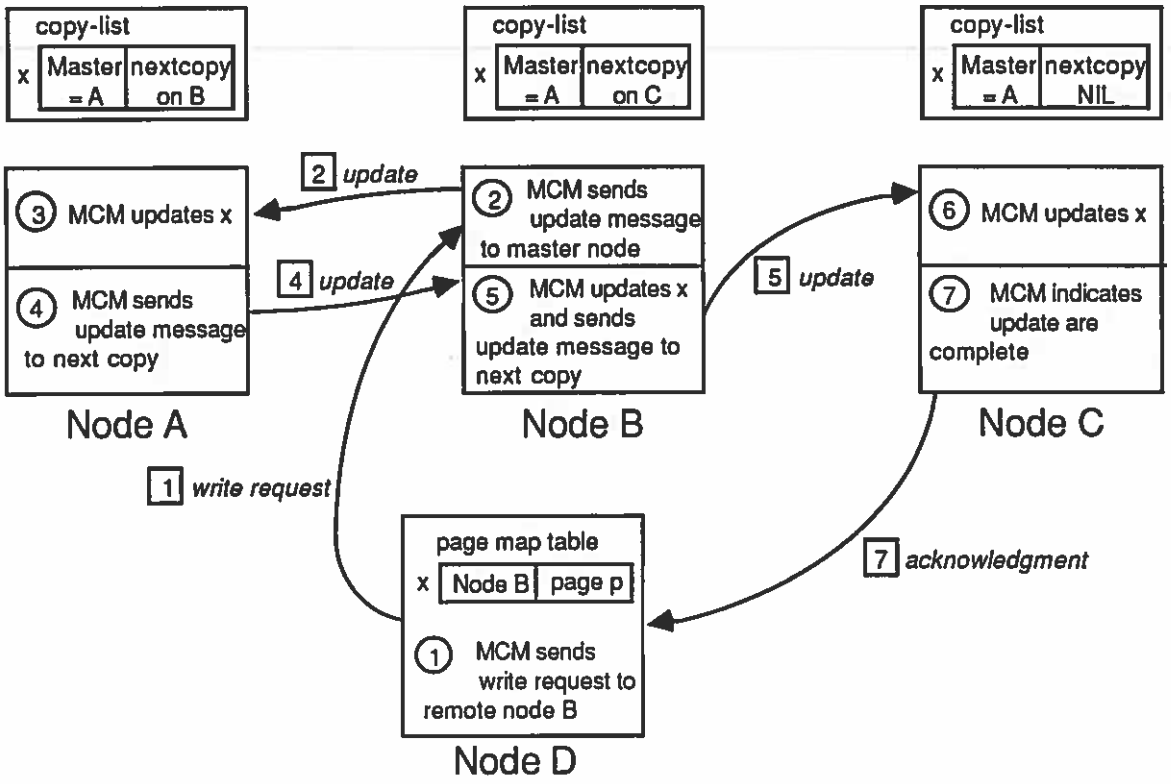


Figure 5: PLUS write-update protocol

Munin suggests the notion of *type specific memory coherence,* coherence protocols tailored for different types of data. For example, data that is read much more frequently than it is written (read-mostly data) is kept coherent using a write-update protocol. Since an invalidation message is about the same size as an update message, an update costs no more than an invalidate. However, the overhead of making multiple read-only copies of the data item is avoided. As another example, the Munin producer-consumer memory type is supported by an eager paging strategy. Data, once written by the producer process, is transferred to the consumer process, where it remains available until the consumer process is ready to use it. Notice that this saves message events, since the consumer does not need to send request messages to the producer when data is available in the "buffer". Munin, however, does not deduce the memory access patterns of applications, but relies on the programmer to explicitly specify the type of each piece of shared data.

A coherence protocol which combines both write-update and write-invalidate is given by Karlin et. al. [KMRS88] in the domain of shared-bus multiprocessors with local caches. Their protocol acts as a write-update protocol until a threshold is reached at which time it switches to write-invalidate. Basically, it performs updates until the time spent on updating is equal to the time it would take to perform an invalidate and transfer the entire piece of data to the updating node. When this threshold is reached, the algorithm performs the invalidate and transfer. They prove that this algorithm takes at most twice as much time as an optimal strategy. Unfortunately, they only consider shared-bus systems, and it is unclear how much of this theory will transfer over to a generalized network of processors.

5.3 Replacement Strategy

In systems which allow data to migrate around the system, two problems arise when the available space for "caching" shared data fills up: what data should be replaced to free space and where it should go. In terms of choosing the data item to be replaced, a DSM system is very similar to the caching system of a shared memory multiprocessor. Unlike most caching systems which use a simple least recently used (LRU) or random replacement strategy, however, most DSM systems differentiate the status of data items and prioritize them. For example, priority is given to shared items over exclusively owned items because the latter will have to be transferred over the network. Simply deleting a read-only shared copy of a data item is possible because no data will be lost. Shiva prioritizes pages based on a linear combination of type (*read-only, owned read-only, writable*) and LRU statistics.

Once a piece of data is to be replaced, there is still the problem of making sure it is not lost. In the caching system of a multiprocessor, the item would simply be placed in main memory. An equivalent scheme is used in some DSM systems, such as MemNet, which transfer the data item to a "home node" which has a statically allocated space (perhaps on disk) to store a copy of an item when it is not needed elsewhere in the system. This method is simple to implement, but it wastes a lot of memory. One improvement to this approach is to have the node which wishes to delete the item simply page it out onto disk. Although this does not waste any (memory) space, it is time consuming. An alternative approach would be to keep track of free memory in the system and to simply page the item out to a node with space available to it. Algorithms to keep track of free space and perform the page out have not been investigated.

5.4 Thrashing

DSM systems are particularly prone to thrashing. For example, if two nodes compete for write access to a single data item, it may be transferred back and forth at such a high rate that no "real" work can get done (a ping-pong effect). Two systems attack this directly: Munin and Mirage.

Munin allows the programmer to associate types with shared data: *write-once, write-many, producer-consumer, private, migratory, result, read-mostly, synchronization* and *general read/write*. Shared data of different types get different coherence protocols. In the case above with two competing writers, the user could specify the type as *write-many* and the system would use a delayed write policy. (Note that Munin does not guarantee strict consistency of memory in this case). The ability to tailor the coherence algorithm to the shared data usage patterns can greatly reduce thrashing. However, Munin requires the programmer to specify the type of shared data, and once specified, the type cannot change. As programmers are notoriously bad at estimating the behavior of their programs, this may not be any better (in general) than choosing any particular protocol. In addition, since the type remains static once specified, Munin cannot dynamically adjust to the changing behavior of an application.

Another method designed to reduce thrashing is explored in Mirage. Mirage specifically looks at the case when many nodes are competing for access to the same piece of data. To stop the ping-pong effect, Mirage adds a dynamically tunable parameter to their coherence protocol. This parameter determines the minimum amount of time (Δ) a data item will be available at a node. For example, if a node performed a write on a piece of shared data, the data would be writable on that node for Δ time. This solves the problem of having a data item stolen away after only a single request on the node can be satisfied. Δ is tuned dynamically based on access patterns so that a process can complete a write-run (or read-run) before losing access to the data. Δ is akin to a time-slice in a multitasking operating system, except in Mirage, it is dynamically adjusted to meet an application's specific needs.

5.5 Related Algorithms

To support a DSM system, synchronization operations and memory management must be specially tuned. Semaphores, for example, are typically implemented on shared memory systems by using spin-locks. In a DSM system, a spin-lock can easily cause thrashing because multiple nodes may be heavily accessing shared data. For performance reasons, some systems provide synchronization primitives along with DSM. Munin supports the synchronization memory type with distributed locks. PLUS supplies a variety of synchronization instructions, and supports delayed execution where the synchronization can be initiated, then later tested for successful completion. The relationship between coherence and synchronization are discussed in [DSB88].

Memory management can also benefit from being rewritten for DSM. A typical memory allocation scheme (as in the C library `malloc()`) allocates memory out of a common pool which is searched each time a request is made. A linear search of all shared memory can be expensive. A better approach is to partition available memory into private buffers on each node and only allocate from the global buffer space when the private buffer is empty.

6 Open Problems in DSM

During the early and mid 80's, research in DSM focused on the design of the basic algorithms needed to achieve coherent sharing across the physically disjoint memories. The challenge today is performance. The leading research projects in DSM are investigating ways to realize the full advantages of DSM through means such as hardware innovations, fine tuning of algorithms, and dynamic protocols. Below is a brief description of some of the exciting problems in this area. We suggest some interesting directions in these areas.

- **RISC and DSM** The load/store architecture of RISC processors can be exploited to easily allow single words of DSM to be read/written instead of entire pages. This facilitates easy implementation of write-update, write-through, and read-through protocols. In addition, the overhead is greatly reduced compared to typical DSM systems which require an entire page to be sent over the network.

  Some RISC's already perform delayed loads and stores. It might be possible to modify the compiler to schedule a load far enough in advance to allow the DSM system time to retrieve the needed value. This seems most plausible in a hardware implementation of DSM such as PLUS or DASH.

- **Page Replacement Strategies** There has been little research directed towards page replacement algorithms in DSM systems. Most systems (especially simulations) either consider the DSM infinite, provide a *home-node* to allow for a place for memory to overflow, or do random page replacement. Good distributed algorithms for keeping track of free space could double the available memory as compared with a home-node based system. In addition, there has been no work to determine if the memory behaviors of DSM applications are similar to demand paged uniprocessor applications or if they require entirely new policies for page replacement.

- **Very Large Systems** Most work in the area of DSM has focused on showing that loosely coupled implementations can perform as well as (sometimes better) than tightly coupled shared memory systems. The limited size of tightly coupled shared memory machines and the fact that virtually all DSM systems to date have been prototypes has led to little research on systems with more than 100 nodes. Systems with thousands, or tens of thousands of nodes have not been addressed. The advantage of such a system is its extremely large "physical" memory. However, clever (probably hierarchical) algorithms would need to be developed to provide even the most rudimentary form of memory coherence. Even then, such a system may only provide good performance for certain applications.

- **Dynamic Protocols** The large latency times associated with remote access in DSM warrant a more complicated protocol if it can improve efficiency. Work has been done on matching the coherence protocol to the type of data sharing in Munin, but this system requires the programmer specify the data type to the system. A more automated and dynamic approach would have the coherence algorithm change as memory access patterns change. This both eliminates the need for programmer hints and allows the system to adjust to changing memory needs over time. Examples

of this are dynamically choosing the coherence protocol, dynamically choosing the location algorithm, and dynamically adjusting the grain (page) size.

- **Fault Tolerance** In systems where failures would be disastrous (critical applications) or in systems where failures would be unavoidable (very large systems), some form of fault tolerance is necessary. In theory, replicating all data (at least on two nodes) will allow the system to survive at least a single node crash. However, total replication may be inefficient. Further, fault tolerant DSM must be integrated with the rest of the operating system to provide a fully fault tolerant environment. Some work in this area has been done by Wu and Fuchs [WF89] (they suggest a checkpointing scheme with a twin-page disk storage system), but more work in the area is warranted.

- **Compiler Support** Many of the DSM systems discussed in this paper require the user to supply some information about the behavior of the application so that the DSM system can be effective. Munin requires the user to specify the type of data which is to be kept coherent. Mirage requires the user to choose an initial time window (Δ). DASH is working on compiler generation of "fence" operations which must be inserted at appropriate places in the executable code to enforce coherence. At the University of Oregon, we are looking at the design of a "smart" compiler which could efficiently partition shared data in memory (reducing false sharing), generate usage hints to the operating system (reducing thrashing), merge arbitrary sequences of concurrent writes (eliminating unnecessary code), and, in some cases, even automatically convert shared memory operations into explicit message passing primitives (reducing overhead).

7 Summary

Research has shown DSM systems to be viable and to yield clear performance gains over traditional multiprocessor architectures. The systems described in this paper have demonstrated that distributed shared memory can be implemented in a variety of hardware and software environments ranging from commercial workstations with native operating systems software, to innovative customized hardware, and even to heterogeneous systems. Many of the design choices and algorithms needed to implement DSM are well-understood and integrated with existing related areas of computer science. In addition, the research has demonstrated that the performance of DSM is greatly affected by memory access patterns and replication of shared data. Hardware implementations have yielded enormous reductions in communication latency as well as the advantages of a smaller unit of sharing. It must be noted, however, that the performance results to date are preliminary in nature: most systems are experimental or prototypes consisting of only a few nodes. In addition, the dearth of test programs has meant that most studies are based on a small group of applications or a synthetic workload. Nevertheless, DSM has been proven as an important architecture to support parallel processing and promises to be a fruitful and exciting area of research for the coming decade.

Appendix A: Summary of DSM systems

Agora (Bisiani & Forin & Correrini, CMU, 1987–?)
A heterogeneous DSM system [BF87][FBC87] which allows objects or data structures to be shared across machines. All writes to an object are funneled through the node where it was created, then all relevant nodes are notified of the change.

Amber (Chase & Amador & Lazowska & Levy & Littlefield, 1989–?)
An object model based DSM system [CAL+89] where sharing is performed by migrating processes to data instead of data to processes.

CapNet (Tam & Farber, U. Penn, 1990–?) This research [TF90] looks at extending DSM to a wide area network—across the country.

Clouds (Ramachandran & Khalidi, Georgia Institute of Tech, 1987–?)
This is an object-oriented distributed operating system [RK89][RK88] [RAK88][RK89] where objects can migrate. DSM is used to simplify the implementation of the operating system.

DASH (Lenoski & Laudon & Gharachorloo & Gupta & Hennessy, Stanford, 1988–?)
A hardware implementation of DSM [LLG+90][GLL+90] providing release consistency for coherence. It supports two explicit fence mechanisms—a full-fence and a write-fence. The coherence protocol is an invalidation based ownership protocol where the order operations reach the owning node (cluster) is their "global ordering". The current prototype consists of four SGI POWER 4D/240's connected in two meshes (one for requests and one for replies). Pages are located by having the owner of a page keep a full bit vector of copy locations—this is not very scalable.

Emerald (Jul & Levy & Hutchinson & Black, U. Washington, 1986–1988)
An object-oriented distributed operating system [JLHB88] which supports object mobility. DSM is not supported per-se, but is indirect since objects can migrate.

IVY (Li, Yale, 1986–1988)
Li implements transparent page-oriented DSM on a network of Apollo workstations [LH86][Li86][Li88] and shows that DSM is not only feasible, but can yield very good performance. He also investigates several coherence mechanisms for implementing strict consistency.

Linda (Ahuja & Carriero & Gelernter, Yale, 1985–?)
This [BCGL87][WL88] is a non-transparent associative form of DSM. All operations are on tuples. The authors say that Linda provides *generative communication*, as tuples are created and destroyed, unlike shared memory which has values that always exist, but are mutated. Linda is an architecture, not a system. The implementations of Linda typically have the Linda compiler generate the coherence primitives. For example, in the Linda Machine [ACGK88] (a mesh of connected processors), the tuple space is partitioned among the columns of the system. An access to the tuple space is broadcast along a row and the node with that tuple responds. An update

is sent to all nodes in a column and each adds the tuple to its local cache. This is basically a replicated, write-update coherence scheme.

MemNet (Delp & Farber, U. Pennnsylvania, 1986–1988)
A hardware implementation of DSM [DF86][Del88][TSF90] with a small grain size (32 bytes). The system is implemented on a 200Mbps token ring which is used to broadcast invalidates and read requests.

Mermaid (Li & Stumm & Wortman & Zhou, Princeton, 1989)
This [LSWZ89] is a heterogeneous DSM system. The compiler separates shared data by type and forces shared memory pages to contain only a single type. Conversion routines are called when a page moves to a system with different data-type representations. *Mermaid* runs on Sun-3 and DEC Firefly computers, and only handles integer data-type conversions.

Mirage (Fleisch & Popek, UCLA, 1989)
This [FP89] is an IVY like implementation of DSM used to explore a method of reducing thrashing by prohibiting a page from being stolen before a minimum amount of time (Δ) has elapsed.

Munin (Bennett & Carter & Zwaenepoel, Rice University, 1989–?)
A DSM system [BCZ90a][BCZ90b] designed to investigate the benefits of using multiple cache coherence protocols at the same time. They empirically determine types of memory sharing: *write-once, write-many, producer-consumer, private, migratory, result, read-mostly, synchronization* and *general read/write*, propose different coherence protocols for each type, and show a performance improvement by using multiple protocols for a single application. In addition, they determined that the unit of sharing (grain size) of an application can effect how well the protocol performs—in most cases, sharing element-wise is better than sharing object-wise.

PLUS (Bisiani & Ravishankar, CMU, 1988–?)
This hardware implementation of DSM [BR90] uses a non-demand, write-update coherence protocol. Pages (the unit of sharing) are only replicated at remote nodes at the request of software. The unit of coherence and access is, however, the 32-bit word. Interprocessor coherence is maintained through use of *write-fences*, which must be explicitly specified by the program. A *write-fence* blocks subsequent writes by a processor until previous writes have completed. This gives processor consistency. They have a one node prototype as of Nov '89.

Shared Data-Object Model (Bal & Kaashoek & Tannenbaum, Vrije Univ., 1988–?)
A DSM implementation [BKT89] on top of the Amoeba distributed operating system [MT86]. It is a basis for *Orca*[BST89]—a language that supports parallelism and data-objects. This implementation uses replication at every site for access and reliable broadcasting for updates. The unit of data sharing is the entire object.

Shiva (Li, Princeton, 1989–?)
An implementation of IVY on the Intel iPSC/2 hypercube [LS89].

References

[ACGK88] Sudhir Ahuja, Nicholas J. Carriero, David H. Gelernter, and Venkatesh Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988.

[BCGL87] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. Linda, the portable parallel. Technical Report Research Report YALE/DCS/RR-520, Yale University, February 1987.

[BCZ90a] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 125–134, May 1990.

[BCZ90b] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[BF87] R. Bisiani and A. Forin. Architectural support for multilanguage parallel programming on heterogeneous systems. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems Proceedings*, pages 21–30, October 1987.

[BKT89] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tannenbaum. A distributed implementation of the shared data-object model. In *First Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, October 1989.

[BR90] Roberto Bisiani and Mosur Ravishankar. PLUS: a distributed shared-memory system. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 115–124, May 1990.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tannenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), September 1989.

[CAL+89] Jeffery S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceeding of the 14th ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[Del88] G. Delp. *The Architecture and Implementation of MemNet: A High Speed Shared Memory Computer Communication Network.* PhD thesis, University of Delaware, 1988.

[DF86] G. Delp and D. Farber. MemNet: an experiment on high-speed memory mapped network interface. Technical Report Technical Report 85-11-IR Delaware, University of Delaware, 1986.

[DSB88] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[FBC87] A. Forin, R. Bisiani, and F. Correrini. Parallel processing with Agora. Technical Report Technical Report CMU-CS-87-183, Carnegie-Mellon University, Computer Science Dept., December 1987.

[Fle89] Brett Dwayne Fleisch. *Distributed Shared Memory in a Loosely Coupled Environment.* PhD thesis, UCLA, 1989.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: a coherent distributed shared memory design. In *Proceeding of the 14th ACM Symposium on Operating System Principles*, pages 211–223, December 1989.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report Technical Report no. 61, IEEE Computer Society, SCI Committee, March 1989.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Kar87] Anna Rochelle Karlin. *Sharing Memory in Distributed Systems—Methods and Applications.* PhD thesis, Stanford University, January 1987.

[KMRS88] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[KU86] Anna R. Karlin and Eli Upfal. Parallel hashing - an efficient implementation of shared memory. In *Symposium on the Theory of Computing*, pages 160–168, 1986.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceeding of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[Li88] Kai Li. IVY: a shared virtual memory system for parallel computing. In *International Conference on Parallel Processing 1988*, pages 94–101, August 1988.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[LS89] Kai Li and Richard Schaefer. Shiva: an operating system transforming a hypercube into a shared-memory machine. March 1989.

[LSWZ89] Kai Li, Michael Stumm, David Wortman, and Songnian Zhou. Shared virtual memory accomodating heterogeneity. Technical Report CS-TR-210-89, Princeton University, February 1989.

[MT86] S. J. Mullender and A. S. Tannenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–300, March 1986.

[RAK88] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef Amin Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report Technical Report GIT-ICS-88/23, Georgia Institute of Technology, June 1988.

[RK88] Umakishore Ramachandran and M. Yousef Amin Khalidi. Programming with distributed shared memory. Technical Report Technical Report GIT-ICS-88/38, Georgia Institute of Technology, October 1988.

[RK89] Umakishore Ramachandran and M. Yousef A. Khalidi. An implementation of distributed shared memory. In *First Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 21–38, October 1989.

[Sch89] Christopher E. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors.* PhD thesis, University of Southern California, May 1989. Tech Report No. CENG 89-19.

[SZ90] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990.

[TF90] Ming-Chit Tam and David J. Farber. CapNet - an alternative approach to ultra high speed networks. In *Proceedings of the International Communication Conference*, 1990.

[TSF90] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber. A survey of distributed shared memory systems. March 1990.

[UW84] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. In *Proceedings of the 25th IEEE Conference on the Foundations of Computer Science*, pages 171–180, 1984.

[WF89] K. L. Wu and W. K. Fuchs. Recoverable distributed shared virtual memory: Memory coherence and storage structures. In *International Symposium on Fault-Tolerant Computing*, 1989.

[WL88] Robert A. Whiteside and Jerrold S. Leichter. Using Linda for supercomputing on a local area network. March 1988.