Distributed Shared Memory: A Survey of Issues and Algorithms

Bill Nitzberg and Virginia Lo, University of Oregon

Distributed shared-memory systems implement the shared-memory abstraction on multicomputer architectures, combining the scalability of network-based architectures with the convenience of shared-memory programming.

As we slowly approach the physical limits of processor and memory speed, it is becoming more attractive to use multiprocessors to increase computing power. Two kinds of parallel processors have become popular: tightly coupled shared-memory multiprocessors and distributed-memory multiprocessors. A tightly coupled multiprocessor system — consisting of multiple CPUs and a single global physical memory — is more straightforward to program because it is a natural extension of a single-CPU system. However, this type of multiprocessor has a serious bottleneck: Main memory is accessed via a common bus — a serialization point — that limits system size to 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. If designers choose the network topology carefully, the system can contain many orders of magnitude more processors than a tightly coupled system. Because all communication between concurrently executing processes must be performed over the network in such a system, until recently the programming model was limited to a message-passing paradigm. However, recent systems have implemented a shared-memory abstraction on top of message-passing distributed-memory systems. The shared-memory abstraction gives these systems the illusion of physically shared memory and allows programmers to use the shared-memory paradigm.

As Figure 1 shows, distributed shared memory provides a virtual address space shared among processes on loosely coupled processors. The advantages offered by DSM include ease of programming and portability achieved through the shared-memory programming paradigm, the low cost of distributed-memory machines, and scalability resulting from the absence of hardware bottlenecks.

DSM has been an active area of research since the early 1980s, although its foundations in cache coherence and memory management have been extensively studied for many years. DSM research goals and issues are similar to those of research in multiprocessor caches or networked file systems, memories for nonuniform memory access multiprocessors, and management systems for distributed or replicated databases.[1] Because of this similarity, many algorithms and lessons learned in these domains can be transferred to DSM systems and vice versa.

0018-9162/91/0800-0052$01.00 © 1991 IEEE

However, each of the above systems has unique features (such as communication latency), so each must be considered separately.

The advantages of DSM can be realized with reasonably low runtime overhead. DSM systems have been implemented using three approaches (some systems use more than one approach):

(1) hardware implementations that extend traditional caching techniques to scalable architectures,
(2) operating system and library implementations that achieve sharing and coherence through virtual memory-management mechanisms, and
(3) compiler implementations where shared accesses are automatically converted into synchronization and coherence primitives.

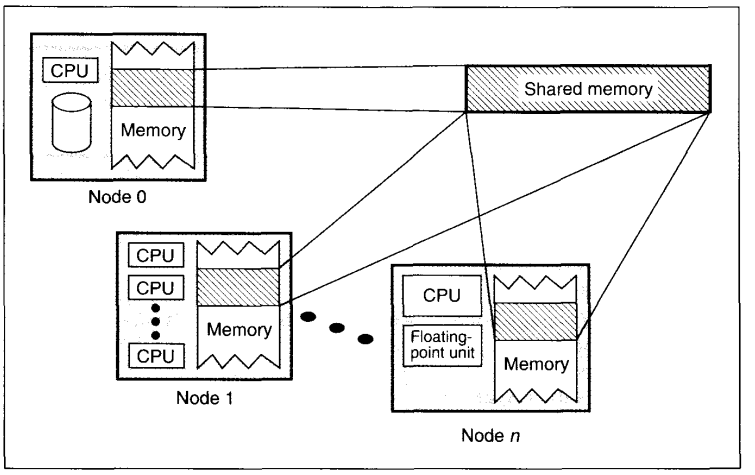

Figure 1. Distributed shared memory.

These systems have been designed on common networks of workstations or minicomputers, special-purpose message-passing machines (such as the Intel iPSC/2), custom hardware, and even heterogeneous systems.

This article gives an integrated overview of important DSM issues: memory coherence, design choices, and implementation methods. In our presentation, we use examples from the DSM systems listed and briefly described in the sidebar on page 55. Table 1 compares how design issues are handled in a selected subset of the systems.

Design choices

A DSM system designer must make choices regarding structure, granularity, access, coherence semantics, scalability, and heterogeneity. Examination of how designers handled these issues in several real implementations of DSM shows the intricacies of such a system.

Structure and 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, or complex data structure.

Ivy,[2] one of the first transparent DSM systems, implemented shared memory as virtual memory. This memory was unstructured and was shared in 1-Kbyte pages. In systems implemented using the virtual memory hardware of the underlying architecture, it is convenient to choose a multiple of the hardware page size as the unit of sharing. Mirage[3] extended Ivy's single shared-memory space to support a paged segmentation scheme. Users share arbitrary-size regions of memory (segments) while the system maintains the shared space in pages.

Hardware implementations of DSM typically support smaller grain sizes. For example, Dash[4] and Memnet[5] also support unstructured sharing, but the unit of sharing is 16 and 32 bytes respectively — typical cache line sizes. Plus[6] is somewhat of a hybrid: The unit of replication is a page, while the unit of coherence is a 32-bit word.

Because shared-memory programs provide locality of reference, a process is likely to access a large region of its shared address space in a small amount of time. Therefore, larger "page" sizes reduce paging overhead. However, sharing may also cause contention, and the larger the page size, the greater the likelihood that more than one process will require access to a page. A smaller page reduces the possibility of *false sharing*, which 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 need to keep directory information about the pages in the system: the smaller the page size, the larger the directory.

A method of structuring the shared memory is by data type. With this method, shared memory is structured as objects in distributed object-oriented systems, as in the Emerald, Choices, and Clouds[7] systems; or it is structured as variables in the source language, as in the Shared Data-Object Model and Munin systems. Because with these systems the sizes of objects and data types vary greatly, the grain size varies to match the application. However, these systems can still suffer from false sharing when different parts of an object (for example, the top and bottom halves of an array) are accessed by distinct processes.

Another method is to structure the shared memory like a database. Linda,[8] a system that has such a model, orders its shared memory as an associative memory called a *tuple space*. This structure allows the location of data to be separated from its value, but it also requires programmers to use special access functions to interact with the shared-memory space. In most other systems, access to shared data is transparent.

Coherence semantics. For programmers to write correct programs on a shared-memory machine, they must understand how parallel memory updates are propagated throughout the

Table 1. DSM design issues.

System Name	Current Implementation	Structure and Granularity	Coherence Semantics	Coherence Protocol	Sources of Improved Performance	Support for Synchronization	Heterogeneous Support
Dash	Hardware, modified Silicon Graphics Iris 4D/340 workstations, mesh	16 bytes	Release	Write-invalidate	Relaxed coherence, prefetching	Queued locks, atomic incrementation and decrementation	No
Ivy	Software, Apollo workstations, Apollo ring, modified Aegis	1-Kbyte pages	Strict	Write-invalidate	Pointer chain collapse, selective broadcast	Synchronized pages, semaphores, event counts	No
Linda	Software, variety of environments	Tuples	No mutable data	Varied	Hashing		?
Memnet	Hardware, token ring	32 bytes	Strict	Write-invalidate	Vectored interrupt support of control flow		No
Mermaid	Software, Sun workstations DEC Firefly multiprocessors, Mermaid/native operating system	8 Kbytes (Sun), 1 Kbyte (Firefly)	Strict	Write-invalidate		Messages for semaphores and signal/wait	Yes
Mirage	Software, VAX 11/750, Ethernet, Locus distributed operating system, Unix System V interface	512-byte pages	Strict	Write-invalidate	Kernel-level implementation, time window coherence protocol	Unix System V semaphores	No
Munin	Software, Sun workstations, Ethernet, Unix System V kernel and Presto parallel programming environment	Objects	Weak	Type-specific (delayed write update for read-mostly protocol)	Delayed update queue	Synchronized objects	No
Plus	Hardware and software, Motorola 88000, Caltech mesh, Plus kernel	Page for sharing, word for coherence	Processor	Nondemand write-update	Delayed operations	Complex synchronization instructions	No
Shiva	Software, Intel iPSC/2, hypercube, Shiva/native operating system	4-Kbyte pages	Strict	Write-invalidate	Data structure compaction, memory as backing store	Messages for semaphores and signal/wait	No

system. The most intuitive semantics for memory coherence is *strict consistency*. (Although "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.) In a system with strict consistency, a read operation returns the most recently written value. However, "most recently" is an ambiguous concept in a distributed system. For this reason, and to improve performance, some DSM systems provide only a reduced form of memory coherence. For example, Plus provides processor consistency, and Dash provides only

release consistency. In accordance with the RISC philosophy, both of these systems have mechanisms for forcing coherence, but their use must be explicitly specified by higher level software (a compiler) or perhaps even the programmer.

Relaxed coherence semantics allows more efficient shared access because it requires less synchronization and less data movement. However, programs that depend on a stronger form of coherence may not perform correctly if executed in a system that supports only a weaker form. Figure 2 gives brief definitions of strict, sequential, processor, weak, and release consistency, and illustrates the hierarchical relationship among these types of coherence. Table 1 indicates the coherence semantics supported by some current DSM systems.

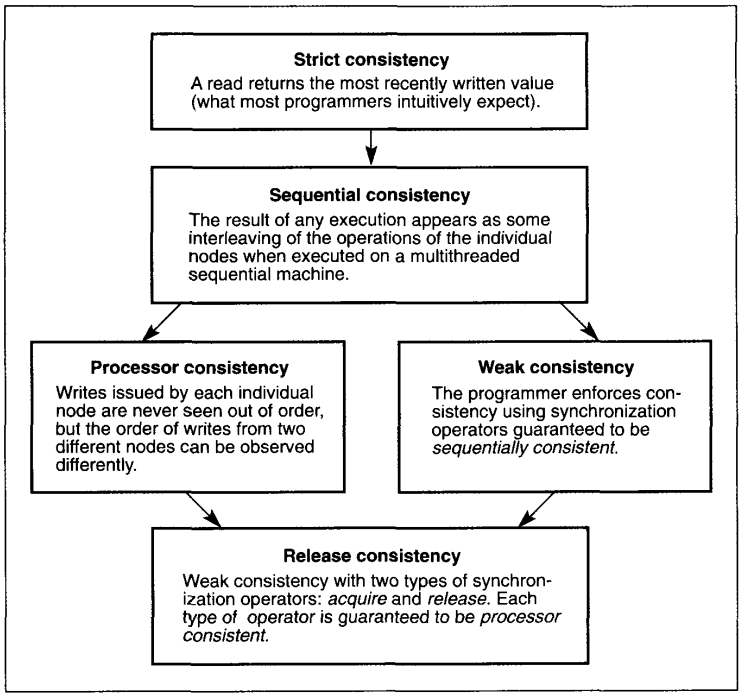

Figure 2. Intuitive definitions of memory coherence. The arrows point from stricter to weaker consistencies.

DSM systems

This partial listing gives the name of the DSM system, the principal developers of the system, the site and duration of their research, and a brief description of the system. Table 1 gives more information about the systems followed with an asterisk.

Agora (Bisiani and Forin, Carnegie Mellon University, 1987-): A heterogeneous DSM system that allows data structures to be shared across machines. Agora was the first system to support weak consistency.

Amber (Chase, Feeley, and Levy, University of Washington, 1988-): An object-based DSM system in which sharing is performed by migrating processes to data as well as data to processes.

Capnet (Tam and Farber, University of Delaware, 1990-): An extension of DSM to a wide area network.

Choices (Johnston and Campbell, University of Illinois, 1988-): DSM incorporated into a hierarchical object-oriented distributed operating system.

Clouds (Ramachandran and Khalidi, Georgia Institute of Technology, 1987-): An object-oriented distributed operating system where objects can migrate.

Dash* (Lenoski, Laudon, Gharachorloo, Gupta, and Hennessy, Stanford University, 1988-): A hardware implementation of DSM with a directory-based coherence protocol. Dash provides release consistency.

Emerald (Jul, Levy, Hutchinson, and Black, University of Washington, 1986-1988): An object-oriented language and system that indirectly supports DSM through object mobility.

Ivy* (Li, Yale University, 1984-1986): An early page-oriented DSM on a network of Apollo workstations.

Linda* (Carriero and Gelernter, Yale University, 1982-): A shared associative object memory with access functions. Linda can be implemented for many languages and machines.

Memnet* (Delp and Farber, University of Delaware, 1986-1988): A hardware implementation of DSM implemented on a 200-Mbps token ring used to broadcast invalidates and read requests.

Mermaid* (Stumm, Zhou, Li, and Wortman, University of Toronto and Princeton University, 1988-1991): A heterogeneous DSM system where the compiler forces shared pages to contain a single data type. Type conversion is performed on reference.

Mether (Minnich and Farber, Supercomputing Research Center, Bowie, Md., 1990-): A transparent DSM built on SunOS 4.0. Mether allows applications to access an inconsistent state for efficiency.

Mirage* (Fleisch and Popek, University of California at Los Angeles, 1987-1989): A kernel-level implementation of DSM. Mirage reduces thrashing by prohibiting a page from being stolen before a minimum amount of time (Δ) has elapsed.

Munin* (Bennett, Carter, and Zwaenepoel, Rice University, 1989-): An object-based DSM system that investigates type-specific coherence protocols.

Plus* (Bisiani and Ravishankar, Carnegie Mellon University, 1988-): A hardware implementation of DSM. Plus uses a write-update coherence protocol and performs replication only by program request.

Shared Data-Object Model (Bal, Kaashoek, and Tannenbaum, Vrije University, Amsterdam, The Netherlands, 1988-): A DSM implementation on top of the Amoeba distributed operating system.

Shiva* (Li and Schaefer, Princeton University, 1988-): An Ivy-like DSM system for the Intel iPSC/2 hypercube.

Scalability. A theoretical benefit of DSM systems is that they scale better than tightly coupled shared-memory multiprocessors. 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, whose sizes are proportional to the number of nodes).

Li and Hudak[2] went through several iterations to refine a coherence protocol for Ivy before arriving at their dynamic distributed-manager algorithm, which avoids centralized bottlenecks. However, Ivy and most other DSM systems are currently implemented on top of Ethernet (itself a centralized bottleneck), which can support only about 100 nodes at a time. This limitation is most likely a result of these systems being research tools rather than an indication of any real design flaw. Shiva[9] is an implementation of DSM on an Intel iPSC/2 hypercube, and it should scale nicely. Nodes in the Dash system are connected on two meshes. This implies that the machine should be expandable, but the Dash prototype is currently limited by its use of a full bit vector (one bit per node) to keep track of page replication.

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, and so on). It is a bit easier if the DSM system is structured as variables or objects in the source language. Then a DSM compiler can add conversion routines to all accesses to shared memory. In Agora, memory is structured as objects shared among heterogeneous machines.

Mermaid[10] explores another novel approach: Memory is shared in pages, and a page can contain only one type of data. Whenever a page is moved between two architecturally different systems, a conversion routine converts 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 to outweigh the benefits.

Implementation

A DSM system must automatically transform shared-memory access into interprocess communication. This requires algorithms to locate and access shared data, maintain coherence, and replace data. A DSM system may also have additional schemes to improve performance. Such algorithms directly support DSM. In addition, DSM implementers must tailor operating system algorithms to support process synchronization and memory management. We focus on the algorithms used in Ivy, Dash, Munin, Plus, Mirage, and Memnet because these systems illustrate most of the important implementation issues. Stumm and Zhou[1] give a good evolutionary overview of algorithms that support static, migratory, and replicated data.

Data location and access. To share data in a DSM system, a program must be able to find and retrieve the data it needs. If data does not move around in the system — it resides only in a single static location — then locating it is easy. All processes simply "know" where to obtain any piece of data. Some Linda implementations use hashing on the tuples to distribute data statically. This has the advantages of being simple and fast, but may cause a bottleneck if data is not distributed properly (for example, all shared data ends up on a single node).

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

Instead of using a centralized server, a system can broadcast requests for data. Unfortunately, broadcasting does not scale well. All nodes — not just the nodes containing the data — must process a broadcast request. The network latency of a broadcast may also require accesses to take a long time to complete.

To avoid broadcasts and distribute the load more evenly, several systems use an owner-based distributed scheme. This scheme is independent of data replication, but is seen mostly in systems that support both data migration and replication. Each piece of data has an associated owner — a node with the primary copy of the data. The owners change as the data migrates through the system. When another node needs a copy of the data, it sends a request to the owner. If the owner still has the data, it returns the data. If the owner has given the data to some other node, it 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. In some cases, this is more wasteful than broadcasting. In Ivy, all 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 it replicates data, a DSM system must keep track of the replicated copies. Dash uses a distributed directory-based scheme, implemented in hardware. The Dash directory for a given cluster (node) keeps track of the physical blocks in that cluster. Each block is represented by a directory entry that 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. Only the special node known as the *home cluster* possesses the directory block entry. A node accesses nonlocal data 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 for each page an entry that indicates the probable location for the referenced page. As described above, a node locates data by following the chain of probable owners. 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.

Coherence protocol. All DSM systems provide some form of memory coherence. If the shared data is not replicated, then enforcing memory coherence is trivial. The underlying network automatically serializes requests in the order they

Charles F. Goldfarb, IBM Almaden Research Center

HyTime is being developed as an ment, plus the large capital and orga- and SPDL. Some industry standards

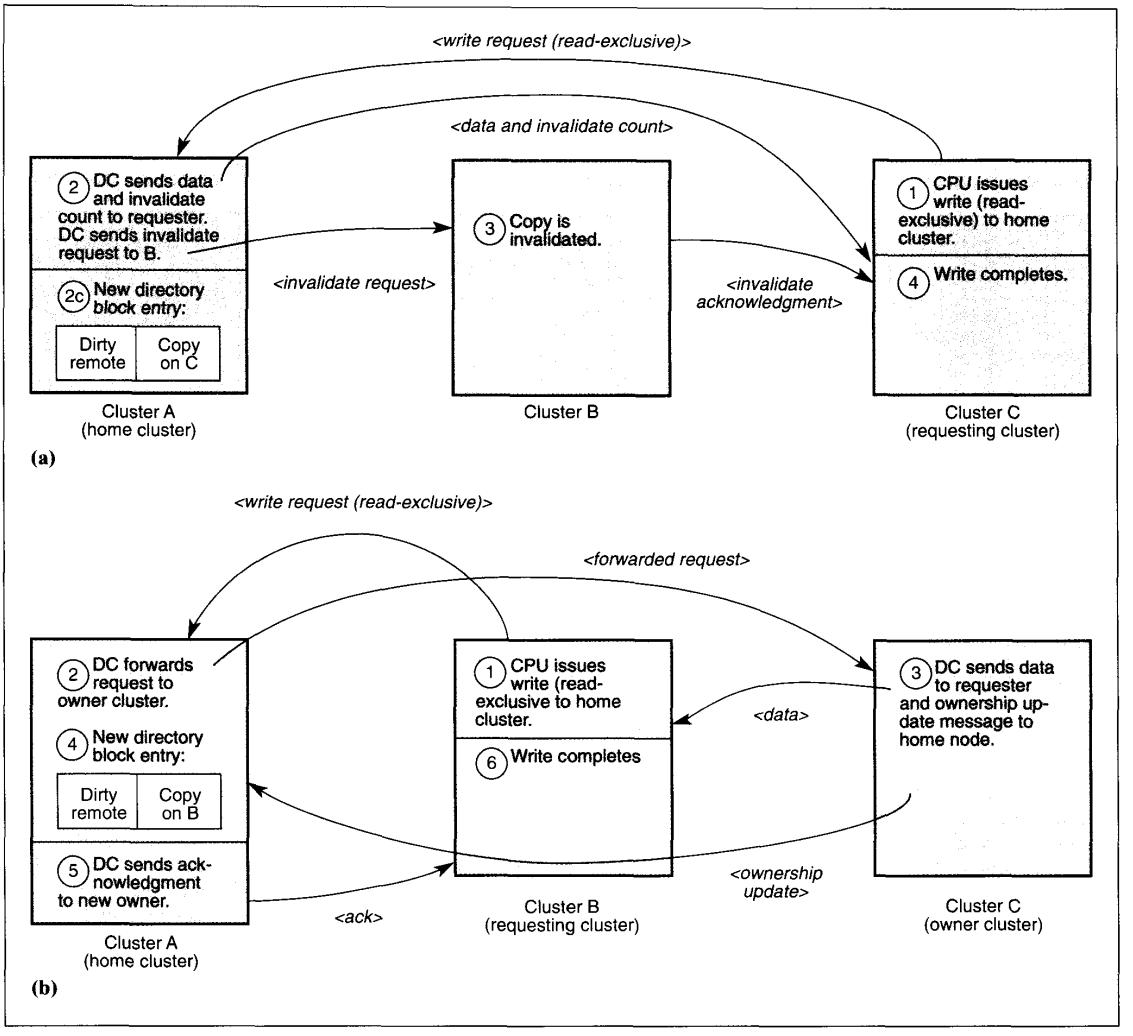

Figure 3. Simplified Dash write-invalidate protocol: (a) Data is shared remote; (b) data is dirty remote (after events depicted in Figure 3a). (DC stands for directory controller.)

occur. A node handling shared data can merely perform each request as it is received. This method will ensure strict memory consistency — the strongest form of coherence. Unfortunately, serializing data access creates a bottleneck and makes impossible a major advantage of DSM: parallelism.

To increase parallelism, virtually all DSM systems replicate data. Thus, for example, multiple reads can be performed in parallel. However, replication complicates the coherence protocol. Two types of protocols — write-invalidate and write-update protocols — handle replication. 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. The protocol is called write-invalidate because it invalidates all copies of a piece of data except one before a write can proceed. In a write-update scheme, however, a write updates all copies of a piece of data.

Most DSM systems have write-invalidate coherence protocols. All the protocols for these systems are similar. Each piece of data has a status tag that indicates whether the data is valid, whether it is shared, and whether it is read-only or writable. For a read, if the data is valid, it is returned immediately. If the data is not valid, a read request is sent to the location of a valid copy, and a copy of the data is returned. If the data was writable on another node, this read request will cause it to become read-only. The copy remains valid until an invalidate request is received.

For a write, if the data is valid and writable, the request is satisfied immediately. If the data is not writable, the directory controller sends out an invalidate request, along with a request for a copy of the data if the local copy is not valid. When the invalidate completes, the data is valid locally and writable, and the original write request may complete.

Figure 3 illustrates the Dash directory-

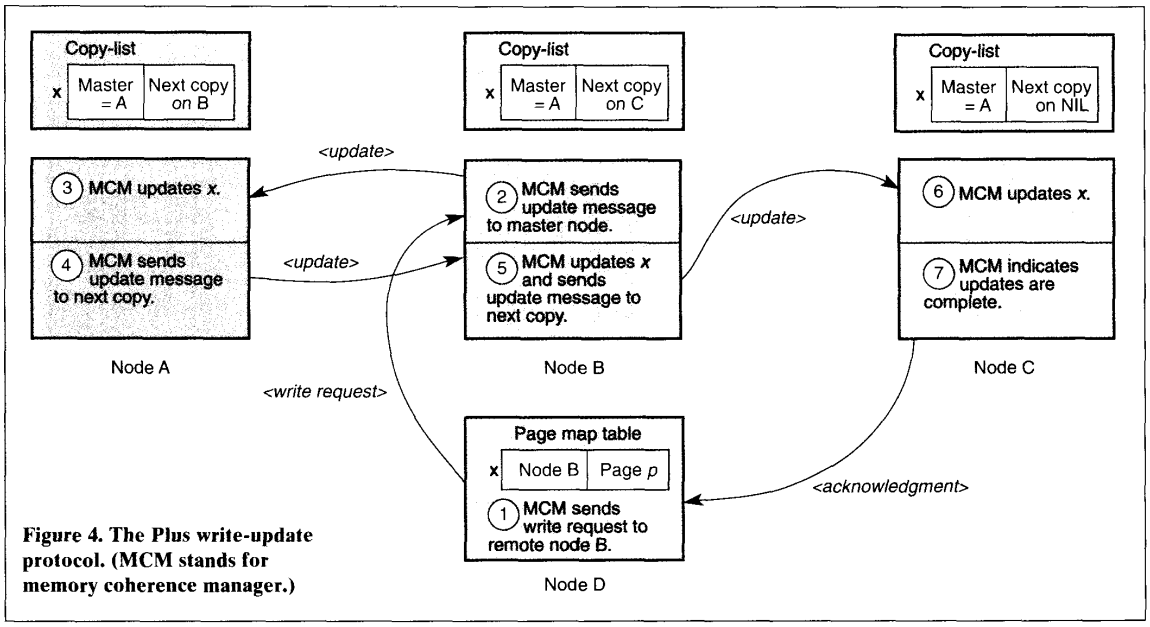

Figure 4. The Plus write-update protocol. (MCM stands for memory coherence manager.)

based coherence protocol. The sequence of events and messages shown in Figure 3a occurs when the block to be written is in shared-remote state (multiple read-only copies on nodes A and B) just before the write. Figure 3b shows the events and messages that occur when the block to be written is in shared-dirty state (single dirty copy on node C) just before the write. In both cases, the initiator of the write sends a request to the home cluster, which uses the information in the directory to locate and transfer the data and to invalidate copies. Lenoski et al.[4] give further details about the Dash coherence protocol and the methods they used to fine-tune the protocol for high performance.

Li and Hudak[2] show that the write-invalidate protocol performs well for a variety of applications. In fact, they show superlinear speedups for a linear equation solver and a three-dimensional partial differential equation solver, resulting from the increased overall physical memory and cache sizes. Li and Hudak rejected use of a write-update protocol at the onset with the reasoning that network latency would make it inefficient.

Subsequent research indicates that in the appropriate hardware environment write-update protocols can be implemented efficiently. For example, Plus is a hardware implementation of DSM that uses a write-update protocol. Figure 4 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.

Munin[11] uses *type-specific memory coherence*, coherence protocols tailored for different types of data. For example, Munin uses a write-update protocol to keep coherent data that is read much more frequently than it is written (read-mostly data). Because 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 after each invalidate is avoided. An eager paging strategy supports the Munin producer-consumer memory type. Data, once written by the producer process, is transferred to the consumer proce where it remains available until the consumer process is ready to use it. This reduces overhead, since the consumer does not request data already available in the buffer.

Replacement strategy. In systems that allow data to migrate around the system, two problems arise when the available space for "caching" shared data fills up: Which data should be replaced to free space and where should it go? In choosing the data item to be replaced, a DSM system works almost like the caching system of a shared-memory multiprocessor. However, unlike most caching systems, which use a simple least recently used or random replacement strategy, 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 have to be transferred over the network. Simply deleting a read-only shared copy of a data item is possible because no data is lost. Shiva prioritizes pages on the basis of a linear combination of type (read-only, owned read-only, and writable) and least recently used statistics.

Once a piece of data is to be replaced, the system must make sure it is not lost. In the caching system of a multiprocessor, the item would simply be placed in main memory. Some DSM systems, such as Memnet, use an equivalent scheme. The system transfers the data item to a "home node" that 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. An improvement is to have the node that wants to delete the item simply page it out onto disk. Although this does not waste any memory space, it is time consuming. Because it may be faster to transfer something over the network than to transfer it to disk, a better solution (used in Shiva) is to keep track of free memory in the system and to simply page the item out to a node with space available to it.

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, Munin and Mirage, attack this problem directly.

Munin allows programmers 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. To avoid thrashing with two competing writers, a programmer could specify the type as write-many and the system would use a delayed write policy. (Munin does not guarantee strict consistency of memory in this case.)

Tailoring the coherence algorithm to the shared-data usage patterns can greatly reduce thrashing. However, Munin requires programmers to specify the type of shared data. Programmers are notoriously bad at predicting the behavior of their programs, so this method may not be any better than choosing a particular protocol. In addition, because the type remains static once specified, Munin cannot dynamically adjust to an application's changing behavior.

Mirage[3] uses another method to reduce thrashing. It specifically examines the case when many nodes compete for access to the same page. To stop the Ping-Pong effect, Mirage adds a dynamically tunable parameter to the coherence protocol. This parameter determines the minimum amount of time (Δ) a page will be available at a node. For example, if a node performed a write to a shared page, the page would be writable on that node for Δ time. This solves the problem of having a page stolen away after only a single request on a node can be satisfied. Because Δ is tuned dynamically on the basis of access patterns, a process can complete a write run (or read run) before losing access to the page. Thus, 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.

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 heavily access shared data. For better performance, some systems provide specialized synchronization primitives along with DSM. Clouds provides semaphore operations by grouping semaphores into centrally managed segments. Munin supports the synchronization memory type with distributed locks. Plus supplies a variety of synchronization instructions, and supports delayed execution, in which the synchronization can be initiated, then later tested for successful completion. Dubois, Scheurich, and Briggs[12] discuss the relationship between coherence and synchronization.

Memory management can be restructured 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 allocate memory from the global buffer space only when the private buffer is empty.

Research has shown distributed shared memory systems to be viable. The systems described in this article demonstrate that DSM can be implemented in a variety of hardware and software environments: commercial workstations with native operating systems software, innovative customized hardware, and even heterogeneous systems. Many of the design choices and algorithms needed to implement DSM are well understood and integrated with related areas of computer science.

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 and the advantages of a smaller unit of sharing. However, the performance results to date are preliminary. Most systems are experimental or prototypes consisting of only a few nodes. In addition, because of the dearth of test programs, most studies are based on a small group of applications or a synthetic workload. Nevertheless, research has proved that DSM effectively supports parallel processing, and it promises to be a fruitful and exciting area of research for the coming decade. ■

Acknowledgments

This work was supported in part by NSF grant CCR-8808532, a Tektronix research fellowship, and the NSF Research Experiences for Undergraduates program. We appreciate the comments from the anonymous referees and thank the authors who verified information about their systems. Thanks also to Kurt Windisch for helping prepare this manuscript.

References

1. M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer*, Vol. 23, No. 5, May 1990, pp. 54-64.

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

3. B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proc. 14th ACM Symp. Operating System Principles*, ACM, New York, 1989, pp. 211-223.

4. D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 148-159.

5. G. Delp, *The Architecture and Implementation of Memnet: A High-Speed Shared Memory Computer Communication Network*, doctoral dissertation, Univ. of Delaware, Newark, Del., 1988.

6. R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared-Memory System," *Proc. 17th Int'l Symp. Computer Archi-*

tecture, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 115-124.

7. U. Ramachandran and M.Y.A. Khalidi, "An Implementation of Distributed Shared Memory," *First Workshop Experiences with Building Distributed and Multiprocessor Systems*, Usenix Assoc., Berkeley, Calif., 1989, pp. 21-38.
8. N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course*, MIT Press, Cambridge, Mass., 1990.
9. K. Li and R. Schaefer, "A Hypercube Shared Virtual Memory System," *Proc. Int'l Conf. Parallel Processing*, Pennsylvania State Univ. Press, University Park, Pa., and London, 1989, pp. 125-132.
10. S. Zhou et al., "A Heterogeneous Distributed Shared Memory," to be published in *IEEE Trans. Parallel and Distributed Systems*.
11. J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. 1990 Conf. Principles and Practice of Parallel Programming*, ACM Press, New York, N.Y., 1990, pp. 168-176.
12. M. Dubois, C. Scheurich, and F.A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, Vol. 21, No. 2, Feb. 1988, pp. 9-21.

Bill Nitzberg is a PhD student in the Department of Computer and Information Science at the University of Oregon. In the AT&T-sponsored ACM International Programming Contest, Nitzberg was a member of the 1990 team and coached the 1991 team, which placed eighth and sixth, respectively.

Nitzberg received a BS in mathematics and an MS in computer science, both from the University of Oregon. He is a member of ACM.

Virginia Lo is an assistant professor in the Department of Computer and Information Science at the University of Oregon. Her research interests include distributed operating systems and the mapping of parallel algorithms to parallel architectures.

Lo received a BA from the University of Michigan, an MS in computer science from Pennsylvania State University, and a PhD in computer science from the University of Illinois at Urbana-Champaign. She is a member of the IEEE, the IEEE Computer Society, and the ACM.

Readers can reach the authors at the Department of Computer Science, University of Oregon, Eugene, OR 97403; e-mail [last name]@cs.uoregon.edu.

Charles F. Goldfarb, IBM Almaden Research Center

HyTime is being developed as an ment, plus the large capital and orga- and SPDL. Some industry standards