Chapter 1

An Introduction to Distributed Shared Memory Concepts

Editors' Introduction

A general survey of distributed shared memory (DSM) principles, algorithms, design issues, and existing systems is given in the following two papers, included in Chapter 1:

1. J. Protić, M. Tomašević, and V. Milutinović, "An Overview of Distributed Shared Memory" (*originally developed for this tutorial,* 1996).
2. B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer,* Vol. 24, No. 8, Aug. 1991, pp. 52–60.

The first research efforts in the area of distributed shared memory systems started in the mid-1980s. This interest has continued to the present, with the increasing attention of the research community. Having in mind the primary goal of providing a shared memory abstraction in a distributed memory system, in order to ease the programming, designers of the early DSM systems were inspired by the principles of virtual memory, as well as the cache coherence maintenance in shared memory multiprocessors.

On the one hand, networks of workstations are becoming more and more popular and powerful nowadays, so they represent the most suitable platform for many programmers entering the world of parallel computing. Communication speed is still the main obstacle preventing these systems from reaching the level of performance of supercomputers, but weakening of the memory consistency semantics can significantly reduce the communication needs. On the other hand, designers of shared memory multiprocessors are striving for scalability, by physical distribu-

tion of shared memory and its sophisticated organization such as clustering and hierarchical layout of the overall system. For these reasons, as the gap between multiprocessors and multicomputers (that early DSM intended to bridge) narrows, and both classes of systems seemingly approach each other in basic ideas and performance, more and more systems can be found that fit into a large family of modern DSM. In spite of many misunderstandings and terminology confusion in this area, we adopted the most general definition, which assumes that all systems providing shared memory abstraction in a distributed memory system belong to the DSM category.

The main purpose of this chapter is to elaborate the distributed shared memory concept and closely related issues. The papers in this chapter define the fundamental principles and effects of DSM memory organization on overall system performance. Insight is also provided into the broad variety of hardware, software, and hybrid DSM approaches, with a discussion of their main advantages and drawbacks. Chapter 1 serves as a base for an extensive elaboration of DSM concepts and systems in Chapters 2–6.

The paper "An Overview of Distributed Shared Memory," by Protić, Tomašević, and Milutinović, covers all of the topics incorporated in this tutorial. In the first part of the paper, an overview of basic approaches to DSM concepts, algorithms, and memory consistency models is presented, together with an elaboration on possible classifications in this area. The second part of the paper briefly describes a relatively large number of existing systems, presenting their essence and selected details, including advantages, disadvantages, complexity, and performance considerations. The DSM systems are classified according to their implementation level into three groups: hardware, software, and hybrid DSM implementations. Relevant information about prototypes, commercial systems, and standards is also provided. The overall structure of the paper is quite similar to the layout of this tutorial. The paper also presents an extensive list of references covering the area of distributed shared memory.

In the paper "Distributed Shared Memory: A Survey of Issues and Algorithms," Nitzberg and Lo introduce DSM concepts, algorithms, and some existing systems. They consider the relevant choices that a DSM system designer must make, such as structure and granularity of shared data, as well as the coherence semantics. The issues of scalability and heterogeneity in DSM systems are also discussed. The authors explain the essential features of the DSM approach—data location and access, and coherence protocol—and illustrate general principles by a more detailed description of coherence maintenance algorithms used in the Dash and PLUS systems. Other issues such as replacement strategy, synchronization, and the problem of thrashing are also mentioned. The paper provides an extensive list of DSM systems, available at the time of writing, with a brief description of their essence, and a survey of the design issues for some of them.

Suggestions for Further Reading

1. V. Lo, "Operating System Enhancements for Distributed Shared Memory," *Advances in Computers*, Vol. 39, 1994, pp. 191–237.
2. D.R. Cheriton, "Problem-Oriented Shared Memory: A Decentralized Approach to Distributed System Design," *Proc. 6th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., 1986, pp. 190–197.
3. 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.
4. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, Calif., 1990.
5. K. Hwang, *Advanced Computer Organization: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, N.Y., 1993.
6. M.J. Flynn, *Computer Architectures: Pipelined and Parallel Processor Design*, Jones and Bartlett, Boston, Mass., 1995.

An Overview of Distributed Shared Memory

Jelica Protić
Milo Tomašević
Veljko Milutinović

Department of Computer Engineering
School of Electrical Engineering
University of Belgrade
Belgrade, Yugoslavia
{jeca,etomasev,emilutiv}@ubbg.etf.bg.ac.yu

I. Concepts

A. Introduction

Significant progress has been made in the research on and development of systems with multiple processors that are capable of delivering high computing power satisfying the constantly increasing demands of typical applications. Systems with multiple processors are usually classified into two large groups, according to their memory system organization: shared memory and distributed memory systems.

In a shared memory system (often called a *tightly coupled multiprocessor*), a global physical memory is equally accessible to all processors. An important advantage of these systems is the general and convenient programming model that enables simple data sharing through a uniform mechanism of reading and writing shared structures in the common memory. Other programming models can be readily emulated on those systems. The cost of parallel software development is reduced owing to the ease of programming and portability. However, shared memory multiprocessors typically suffer from increased contention and longer latencies in the accessing of shared memory, resulting in a somewhat lower peak performance and limited scalability compared to distributed systems. In addition, the design of memory systems tends to be complex. A detailed discussion of shared memory multiprocessors is provided in [FLYNN95].

On the other hand, a distributed memory system (often called a *multicomputer*) consists of multiple independent processing nodes with local memory modules, connected by means of some general interconnection network. Distributed memory systems are claimed to be scalable, and systems with very high computing power are possible. However, communication between processes

residing on different nodes is achieved through a message-passing model, which requires explicit use of send/receive primitives; most programmers find this more difficult to achieve, since they must take care of data distribution across the system, and manage the communication. Also, process migration imposes problems because of different address spaces. Therefore, compared to shared memory systems, the hardware problems are easier and software problems more complex in distributed memory systems.

A relatively new concept—distributed shared memory (DSM), discussed in [LO94] and [PROTI96]—tries to combine the advantages of the two approaches. A DSM system logically implements the shared memory model in a physically distributed memory system. The specific mechanism for achieving the shared memory abstraction can be implemented in hardware and/or software in a variety of ways. The DSM system hides the remote communication mechanism from the application writer, so the ease of programming and the portability typical of shared memory systems are preserved. Existing applications for shared memory systems can be relatively easily modified and efficiently executed on DSM systems, preserving software investments while maximizing the resulting performance. In addition, the scalability and cost-effectiveness of underlying distributed memory systems are also inherited. Consequently, the importance of the distributed shared memory concept comes from the fact that it seems to be a viable choice for building efficient large-scale multiprocessors.

The ability to provide a transparent interface and a convenient programming environment for distributed and parallel applications has made the DSM model the focus of numerous research efforts. The main objective of current research in DSM systems is the development of general approaches that minimize the average access time to shared data, while maintaining data consistency. Some solutions implement a specific software layer on top of existing message-passing systems, while others extend strategies applied in shared memory multiprocessors with private caches, described in [TOMAS94a], [TOMAS94b], and [TARTA95], to multilevel memory systems.

Part I of this paper provides comprehensive insight into the increasingly important area of DSM. As such, Sections I.B–D cover general DSM concepts and approaches. Possible classifications of DSM systems are discussed, as well as various important design choices in building DSM systems. Section I.E presents a set of DSM algorithms from the open literature, and differences between them are analyzed under various conditions. Part II of this paper represents a survey of existing DSM systems, developed either as research prototypes or as commercial products and standards. It consists of three sections dedicated to DSM implementations on the hardware level, software level, and using a hybrid hardware/software approach. Although not exhaustive, this survey presents extensive and up-to-date information on several key implementation schemes for maintaining data in DSM systems. In the description of each DSM system, the essentials of the approach, implementation issues, and basic DSM mechanism are highlighted.

B. General Structure of a Distributed Shared Memory System

A DSM system can be generally viewed as a set of nodes or clusters, connected by an interconnection network (Figure 1). A cluster can be a uniprocessor or a

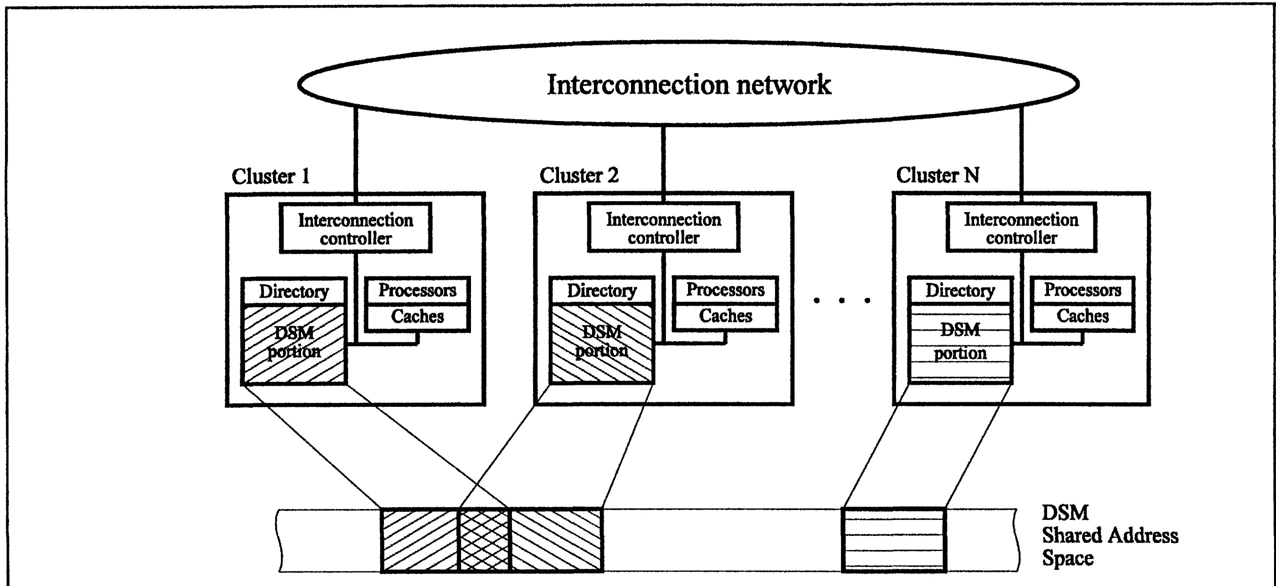

Figure 1. Structure and organization of a DSM system.

multiprocessor system, usually organized around a shared bus. Private caches attached to the processors are inevitable for reducing memory latency. Each cluster in the system contains a physically local memory module, which is partially or entirely mapped to the DSM global address space. Regardless of the network topology (for example, a bus, ring, mesh, or local area network), a specific interconnection controller within each cluster is needed to connect it into the system.

Information about states and current locations of particular data blocks is usually kept in the form of a system table or directory. Storage and organization of directory are among the most important design decisions, with a large impact on system scalability. Directory organization varies from a full-map storage to different dynamic organizations, such as single- or double-linked lists and trees. No matter which organization is employed, the cluster must provide the storage for the entire directory, or just a part of it. In this way, the system directory can be distributed across the system as a flat or hierarchical structure. In hierarchical topologies, if clusters on intermediate levels exist, they usually contain only directories and the corresponding interface controllers. Directory organization and the semantics of information kept in directories depend on the applied method for maintaining data consistency.

C. Classifications of Distributed Shared Memory Systems

Since the first research efforts in the area of DSM in the mid-1980s, the interest in this concept has increased continuously, resulting in tens of systems developed predominantly as research prototypes. Designers of the early DSM systems were inspired by the principle of virtual memory, as well as by the principle of cache coherence maintenance in shared memory multiprocessors.

On the one hand, networks of workstations are becoming more and more popular and powerful nowadays, so they represent the most suitable platform for

many programmers entering the world of parallel computing. However, communication speed is still the main obstacle preventing these systems from reaching the level of performance of high-end machines. On the other hand, designers of shared memory multiprocessors are striving for scalability, by physical distribution of shared memory and its sophisticated organization such as clustering and hierarchical layout of the overall system. For these reasons, as the gap between multiprocessors and multicomputers (that early DSM intended to bridge) narrows, and both classes of systems seemingly approach each other in basic ideas and performance, more and more systems can be found that fit into a large family of modern DSM. In spite of many misunderstandings and terminology confusion in this area, we adopt the most general definition, which assumes that all systems providing shared memory abstraction on a distributed memory system belong to the DSM category. This abstraction is achieved by specific actions necessary for accessing data from global virtual DSM address space, which can be shared among the nodes. Three important issues are involved in the performance of these actions, which bring the data to the site where accessed, while keeping them consistent (with respect to other processors); each of these actions leads to a specific classification of DSM systems:

- *How* the access actually executes → **DSM algorithm**
- *Where* the access is implemented → **Implementation level of DSM mechanism**
- *What* the precise semantics of the word *consistent* is → **Memory consistency model**

Having in mind these most important DSM characteristics, we next discuss three possible classifications of DSM approaches.

1. The First Classification: According to the DSM Algorithm

DSM algorithms can be classified according to the allowable existence of multiple copies of the same data, also considering access rights of those copies. Two strategies for distribution of shared data are most frequently applied: *replication* and *migration*. Replication allows that multiple copies of the same data item reside simultaneously in different local memories (or caches), in order to increase the parallelism in accessing logically shared data. On the other hand, migration implies a single copy of a data item that must be moved to the accessing site for exclusive use, counting on the locality of references in parallel applications. DSM algorithms can be classified as:

1. SRSW (single reader/single writer)
   a. Without migration
   b. With migration
2. MRSW (multiple reader/single writer)
3. MRMW (multiple reader/multiple writer)

Replication is prohibited in SRSW, while it is allowed in MRSW and MRMW algorithms. The complexity of coherence maintenance is strongly dependent on

the introduced classes. To adapt to the application characteristics on the basis of typical read/write patterns, while keeping the acceptable complexity of the algorithm, many solutions have been proposed. Among them, MRSW algorithms are predominant. More details on DSM algorithms are given in Section I.E.

2. The Second Classification: According to the Implementation Level of the DSM Mechanism

The level at which the DSM mechanism is implemented is one of the most important decisions in building a DSM system, since it affects both the programming and the overall system performance and cost. Possible implementation levels are:

1. Software
   a. Runtime library routines
   b. Operating system
      i. Inside the kernel
      ii. Outside the kernel
   c. Compiler-inserted primitives
2. Hardware
3. Hardware/software combination (hybrid)

Software-oriented DSM started with the idea of hiding the message-passing mechanism and providing a shared memory paradigm in loosely coupled systems. Some of these solutions rely on specific runtime libraries that are to be linked with the application that uses shared data. The others implement DSM on the level of programming language, since the compiler can detect shared accesses and insert calls to synchronization and coherence routines into the executable code. Another class of approaches finds it appropriate to incorporate a DSM mechanism into the distributed operating system, inside or outside the kernel. Operating system- and runtime library-oriented approaches often integrate the DSM mechanism with an existing virtual memory management system. However, existing DSM systems usually combine elements of different software approaches. For example, IVY is predominantly a runtime library solution that also includes modifications to the operating system, while Midway implementation is based on a runtime system and the compiler for code generation that marks shared data as dirty when written into.

Some DSM systems use dedicated hardware responsible for locating, copying shared data items, and keeping their coherence. Those solutions extend traditional caching techniques typical of shared memory multiprocessors to DSM systems with scalable interconnection networks. They can for the most part be classified into three groups according to their memory system architecture: CC-NUMA (cache coherent nonuniform memory access) architecture, COMA (cache-only memory architecture), and RMS (reflective memory system) architecture. In a CC-NUMA system, DSM address space is statically distributed across local memory modules of clusters, which can be accessed both by the local processors and by processors from other clusters in the system, although with quite different access latencies. COMA provides the dynamic partitioning of data in the form of distributed memories, organized as large second-level caches

(attraction memories). RMS architectures use a hardware-implemented update mechanism in order to propagate immediately every change to all sharing sites using broadcast or multicast messages. This kind of memory system is also called "mirror memory."

Because of possible performance/complexity trade-offs, the integration of software and hardware methods seems to be one of the most promising approaches in the future of DSM. Some software approaches include hardware accelerators for frequent operations in order to improve the performance, while some hardware solutions handle infrequent events in software to minimize complexity. An example of a hybrid solution is to use hardware to manage fixed size fine-grain data units, in combination with the coarse-grain data management in software. To gain better performance in DSM systems, recent implementations have used multiple protocols within the same system, and even integrate message passing with the DSM mechanism. To handle the complexity of recent, basically software solutions, special programmable protocol processors can also be added to the system.

While hardware solutions bring total transparency of the DSM mechanism to the programmer and software layers, and typically achieve lower access latencies, software solutions can take better advantage of application characteristics through the use of various hints provided by the programmer. Software systems are also very suitable for experimenting with new concepts and algorithms. As a consequence, the number of software DSM systems presented in the open literature is considerably higher; however, the ideas generated in software-based solutions often migrate to hardware-oriented systems. Our presentation of DSM systems in Part II follows the classification of hardware, software, and hybrid DSM implementations and elaborates extensively on these issues.

3. The Third Classification: According to the Memory Consistency Model

A memory consistency model defines the legal ordering of memory references issued by some processor, as observed by other processors. Different types of parallel applications inherently require various consistency models. Performance of the system in executing these applications is largely influenced by the restrictiveness of the model. Stronger forms of the consistency model typically increase the memory access latency and the bandwidth requirements, while simplifying programming. Looser constraints in more relaxed models that allow reordering, pipelining, and overlapping of memory consequently result in better performance, at the expense of higher involvement of the programmer in synchronizing the accesses to shared data. To achieve optimal behavior, systems with multiple consistency models adaptively applied to appropriate data types have been proposed.

Stronger memory consistency models that treat synchronization accesses as ordinary read and write operations are *sequential* and *processor consistency* models. More relaxed models that distinguish between ordinary and synchronization accesses are *weak, release, lazy release,* and *entry consistency* models. A brief overview of memory consistency models can be found in [LO94]. Sufficient conditions for ensuring sequential, processor, weak, and release memory consistency models are given in [GHARA90].

Sequential consistency provides that all processors in the system observe the same interleaving of reads and writes issued in sequences by individual proces-

sors. A simple implementation of this model is a single-port shared memory system that enforces serialized access servicing from a single first in–first out (FIFO) queue. In DSM systems, similar implementation is achieved by serializing all requests on a central server node. In both cases, no bypassing of read and write requests is allowed. Conditions for sequential consistency hold in the majority of bus-based shared memory multiprocessors, as well as in early DSM systems, such as IVY and Mirage.

Processor consistency assumes that the order in which memory operations can be seen by different processors need not be identical, but the sequence of writes issued by each processor must be observed by all other processors in the same order of issuance. Unlike sequential consistency, processor consistency implementations allow reads to bypass writes in queues from which memory requests are serviced. Examples of systems that guarantee processor consistency are VAX 8800, PLUS, Merlin, RMS, and so on.

Weak consistency distinguishes between ordinary and synchronization memory accesses. It requires that memory becomes consistent only on synchronization accesses. In this model, requirements for sequential consistency apply only on synchronization accesses themselves. In addition, a synchronization access must wait for *all* previous accesses to be performed, while ordinary reads and writes must wait *only* for completion of previous synchronization accesses. A variant of weak consistency model is used in SPARC architecture by Sun Microsystems.

Release consistency further divides synchronization accesses into acquire and release, so that protected ordinary shared accesses can be performed between acquire–release pairs. In this model, ordinary read or write access can be performed only after all previous *acquires* on the same processor are performed. In addition, *release* can be performed only after all previous ordinary reads and writes on the same processor are performed. Finally, acquire and release synchronization accesses must fulfill the requirements that processor consistency imposes on ordinary read and write accesses, respectively. Different implementations of release consistency can be found in Dash and Munin DSM systems.

An enhancement of release consistency, *lazy release consistency*, is presented in [KELEH92]. Instead of propagating modifications to the shared address space on each release (as in release consistency—sometimes called *eager release*), modifications are further postponed until the next relevant acquire. In addition, not all modifications need to be propagated on the acquire—only those associated to the chain of preceding critical sections. In this way, the amount of data exchanged is minimized, while the number of messages is also reduced by combining modification with lock acquires in one message. Lazy release consistency was implemented in the DSM system TreadMarks.

Finally, *entry consistency* is a new improvement of release consistency. This model requires that each ordinary shared variable or object be protected and associated to the synchronization variable using language-level annotation. Consequently, modification to the ordinary shared variable is postponed to the next acquire of the associated synchronization variable that guards it. Since only the changes for associated variables need be propagated at the moment of acquire, the traffic is significantly decreased. Latency is also reduced since a shared access does not have to wait for the completion of other nonrelated acquires. Performance improvement is achieved at the expense of higher programmer involvement in specifying synchronization information for each variable. Entry consistency was implemented for the first time in the DSM system Midway.

D. Important Design Choices in Building Distributed Shared Memory Systems

In addition to the DSM algorithm, implementation level of DSM mechanism, and memory consistency model, a set of characteristics that can strongly affect the overall performance of a DSM system includes:

Cluster configuration—Single/multiple processor(s), with/without (shared/private) (single/multiple level) caches, local memory organization, network interface, etc.

Interconnection network—Bus hierarchy, ring, mesh, hypercube, specific LAN, and so on

Structure of shared data—Nonstructured or structured into objects, language types, and so on

Granularity of coherence unit—Word, cache block, page, complex data structure, and so on

Responsibility for DSM management—Centralized, distributed fixed, distributed dynamic

Coherence policy—Write-invalidate, write-update, type-specific, and so on

Cluster configuration varies greatly across different DSM systems. It includes one or several (usually off-the-shelf) processor(s). Since each processor has its local cache (or even cache hierarchy) the cache coherence on a cluster level must be integrated with the DSM mechanisms on the global level. Parts of a local memory module can be configured as private or shared (mapped to the virtual shared address space). In addition to coupling the cluster to the system, the network interface controller sometimes integrates some important responsibilities of DSM management.

Almost all types of *interconnection networks* found in multiprocessors and distributed systems can also be used in DSM systems. The majority of software-oriented DSM systems are network independent, although many of them happened to be built on top of an Ethernet network, readily available in most environments. On the other hand, topologies such as a multilevel bus, ring hierarchy, or mesh have been used as platforms for some hardware-oriented DSM systems. The topology of the interconnection network can offer or restrict good potential for parallel exchange of data related to DSM management. For the same reasons, topology also affects scalability. In addition, it determines the possibility and cost of broadcast and multicast transactions, very important for implementing DSM algorithms.

Structure of shared data represents the global layout of shared address space, as well as the organization of data items in it. Hardware solutions always deal with nonstructured data objects, while some software implementations tend to use data items that represent logical entities, in order to take advantage of the locality naturally expressed by the application.

Granularity of coherence unit determines the size of data blocks managed by coherence protocols. The impact of this parameter on overall system performance is closely related to the locality of data access typical for the application. In general, hardware-oriented systems use smaller units (typically cache blocks), while some software solutions, based on virtual memory mechanisms, organize data in

larger physical blocks (pages), counting on coarse-grain sharing. The use of larger blocks results in saving space for directory storage, but it also increases the probability that multiple processors will require access to the same block simultaneously, even if they actually access unrelated parts of that block. This phenomenon is referred to as *false sharing*. This can cause *thrashing*—a behavior characterized by extensive exchange of data between sites competing for the same data block.

Responsibility for DSM management determines which site must handle actions related to the consistency maintenance in the system; the management can be centralized or distributed. Centralized management is easier to implement, but the central manager represents a bottleneck. The responsibility for distributed management can be defined statically or dynamically, eliminating bottlenecks and providing scalability. Distribution of responsibility for DSM management is closely related to the distribution of directory information.

Coherence policy determines whether the existing copies of a data item being written to at one site will be updated or invalidated at other sites. The choice of coherence policy is related to the granularity of shared data. For very fine-grain data items, the cost of an update message is approximately the same as the cost of an invalidation message. Therefore, the update policy is often used in systems with word-based coherence maintenance. On the other hand, invalidation is largely used in coarse-grain systems. The efficiency of an invalidation approach increases when the read and write access sequences to the same data item by various processors are not highly interleaved. The best performance can be expected if coherence policy dynamically adapts to the observed reference pattern.

E. Distributed Shared Memory Algorithms

The algorithms for implementing distributed shared memory deal with two basic problems: (1) static and dynamic distribution of shared data across the system, in order to minimize their access latency, and (2) preserving a coherent view of shared data, while trying to keep the overhead of coherence management as low as possible. Replication and migration are the two most frequently used policies that try to minimize data access time, by bringing data to the site where they are currently used. Replication is mainly used to enable simultaneous accesses by different sites to the same data, predominantly when read sharing prevails. Migration is preferred when sequential patterns of write sharing are prevalent in order to decrease the overhead of coherence management. The choice of a suitable DSM algorithm is a vital issue in achieving high system performance. Therefore, is must be well adapted to the system configuration and characteristics of memory references in typical applications.

Classifications of DSM algorithms and the evaluation of their performance have been extensively discussed in [LIHUD89], [STUM90], [BLACK89], and [KESSL89]. This presentation follows a classification of algorithms similar to the one found in [STUM90].

1. Single Reader/Single Writer Algorithms

Single reader/single writer (SRSW) algorithms prohibit the possibility of replication, while the migration can be, but is not necessarily, applied. The simplest algorithm for DSM management is the *central server* algorithm [STUM90]. The

approach is based on a unique central server that is responsible for servicing all access requests from other nodes to shared data, physically located on this node. This algorithm suffers from performance problems since the central server can become a bottleneck in the system. Such an organization implies no physical distribution of shared memory. A possible modification is the static distribution of physical memory and the static distribution of responsibilities for parts of shared address space onto several different servers. Some simple mapping functions (for example, hashing) can be used to locate the appropriate server for the corresponding piece of data.

More sophisticated SRSW algorithms additionally allow for the possibility of migration. However, only one copy of the data item can exist at any one time and this copy can be migrated on demand. In [KESSL89] this kind of algorithm is referred to as *Hot Potato*. If an application exhibits high locality of reference, the cost of data migration is amortized over multiple accesses, since data are moved not as individual items, but in fixed size units—blocks. It can perform well in cases where a longer sequence of accesses from one processor uninterrupted with accesses from other processors is likely to happen, and write after read to the same data occurs frequently. In any case, the performance level of this rarely used algorithm is restrictively low, since it does not take advantage of the parallel potential of multiple read-only copies, in cases when read sharing prevails.

2. Multiple Reader/Single Writer Algorithms

The main intention of multiple reader/single writer (MRSW) (or *read-replication*) algorithms is to reduce the average cost of read operations, by counting on the fact that read sharing is the prevalent pattern in parallel applications. To this end, they allow read operations to be simultaneously executed locally at multiple hosts. Permission to update a replicated copy can be given to only one host at a time. On the occurrence of write to a writable copy, the cost of this operation is increased, because the use of other replicated stale copies must be prevented. Therefore, the MRSW algorithms are usually invalidation based. Protocols following this principle are numerous.

A variety of algorithms belong to this class. They differ in the way the responsibility for DSM management is allocated. Several MRSW algorithms are proposed in [LIHUD89]. Before discussing those algorithms, the following terms must be defined:

Manager—The site responsible for organizing the write access to a data block

Owner—The site that owns the only writable copy of the data block

Copy set—A set of all sites that have copies of the data block

A list of algorithms proposed by Li and Hudak includes the following.

a. **Centralized Manager Algorithm.** All read and write requests are sent to the manager, which is the only site that keeps the identity of the owner of a particular data block. The manager forwards the request for data to the owner, and waits for confirmation from the requesting site, indicating that it received the copy of the block from the owner. In the case of a write operation, the manager also sends invalidations to all sites from the copy set (a vector that identifies the current holders of the data block, kept by the manager).

b. **Improved Centralized Manager Algorithm.** Unlike the original centralized manager algorithm, the owner, instead of the manager, keeps the copy set in this version of the centralized algorithm. Copy set is sent together with the data to the new owner, which is also responsible for invalidations. In this case, the overall performance can be improved because of the decentralized synchronization.

c. **Fixed Distributed Manager Algorithm.** In the fixed distributed manager algorithm, instead of centralizing the management, each site is predetermined to manage a subset of data blocks. The distribution is done according to some default mapping function. Clients are still allowed to override it by supplying their own mapping, tailored to the expected behavior of the application. When a parallel program exhibits a high rate of requests for data blocks, this algorithm performs better than the centralized solutions.

d. **Broadcast Distributed Manager Algorithm.** There is actually no manager in the broadcast distributed manager algorithm. Instead, the requesting processor sends a broadcast message to find the true owner of the data block. The owner performs all actions just like the manager in previous algorithms, and keeps the copy set. The disadvantage of this approach is that all processors must process each broadcast, slowing down their own computations.

e. **Dynamic Distributed Manager Algorithm.** In the dynamic distributed manager algorithm, the identity of the probable owner, not the real owner, is kept for each particular data block. All requests are sent to the probable owner, which is also the real owner in most cases. However, if the probable owner does not happen to be the real one, it forwards the request to the node that represents probable owner according to the information kept in its own table. For every read and write request, forward, and invalidation messages, the probable owner field is changed accordingly, in order to decrease the number of messages to locate the real owner. This algorithm is often called the Li algorithm. For its basic version, where the ownership is changed on both read and write fault, it is shown in [LIHUD89] that the performance of the algorithm does not worsen as more processors are added to the system, but rather degrades logarithmically when more processors contend for the same data block.

A modification of the dynamic distributed manager algorithm, also proposed in [LIHUD89], suggests a distribution of the copy set, which should be organized as a tree rooted at the owner site. This is a way to distribute the responsibility for invalidations, as well.

3. Multiple Reader/Multiple Writer Algorithms

The multiple reader/multiple writer (MRMW) algorithm (also called the *full-replication* algorithm) allows the replication of data blocks with both read and write permission. To preserve coherence, updates of each copy must be distributed to all other copies at remote sites, by multicast or broadcast messages. This algorithm tries to minimize the cost of write access. Therefore, it is appropriate for write sharing and it is often used with write-update protocols. This algorithm

can produce high coherence traffic, especially when the update frequency and the number of replicated copies are high.

Protocols complying to the MRMW algorithm can be complex and demanding. One possible way to maintain data consistency is to sequence the write operations globally, in order to implement reliable multicast. When a processor attempts to write to the shared memory, the intended modification is sent to the sequencer. The sequencer assigns the next sequence number to the modification and multicasts the modification with this sequence number to all sites having the copy. When the modification arrives at a site the sequence number is verified, and if it is not correct a retransmission is requested.

A modification of this algorithm proposed in [BISIA88] distributes the task of sequencing. In this solution, writes to any particular data structure are sequenced by the server that manages the master copy of that data structure. Although the system is not sequentially consistent in this case, each particular data structure is maintained in a consistent manner.

4. Avenues for Performance Improvement

Considerable effort has been dedicated to various modifications of the basic algorithms, in order to improve their behavior and gain better performance by reducing the amount of data transferred in the system. Most of these ideas have been evaluated by simulation studies, and some of them have been implemented in existing prototype systems.

An enhancement of Li's algorithm (named the Shrewd algorithm) is proposed in [KESSL89]. It eliminates all unnecessary page transfers with the assistance of the sequence number per copy of a page. On each write fault at a node with a previously existing read-only copy, the sequence number is sent with the request. If this number is the same as the number kept by the owner, the requester will be allowed to access the page without its transfer. This solution shows remarkable benefits when the read-to-write ratio increases.

All solutions presented in [LIHUD89] assume that a page transfer is performed subsequent to each attempt to access a page that does not reside on the accessing site. A modification proposed in [BLACK89] employs a competitive algorithm and allows page replication only when the number of accesses to the remote page exceeds the replication cost. A similar rule is applied to migration, although the fact that, in this case, only one site can have the page makes the condition to migrate the page more restrictive and dependent on the other site's access pattern to the same page. The performance of these policies is guaranteed to stay within a constant factor from the optimal.

Another restriction to data transfer requests is applied in the system Mirage, in order to reduce thrashing—an adverse effect that occurs when an alternating sequence of accesses to the same page issued by different sites makes its migration the predominant activity. The solution to this problem is found in defining a time window Δ in which the site is guaranteed to uninterruptedly possess the page after it has acquired it. The value of Δ can be tuned statically or dynamically, depending on the degree of processor locality exhibited by the particular application.

There are a variety of specific algorithms implemented in existing DSM systems, or simulated extensively using appropriate workload traces. Early DSM implementations found the main source of possible performance and scalability

improvements in various solutions for the organization and storage of system tables, such as copy set, as well as the distribution of management responsibilities. In striving to gain better performance, recent DSM implementations have relaxed memory consistency semantics, so the algorithms and the organization of directory information must be considerably modified. Implementations of critical operations using hardware accelerators and a combination of invalidate and update methods also contribute to the better performance of modern DSM systems.

II. Systems

A. Introduction

A distributed shared memory system logically implements a shared memory model on physically distributed memories, in order to achieve ease of programming, cost-effectiveness, and scalability [PROTI96]. Basic DSM concepts are discussed extensively in Part I of this paper. Part II represents a wide overview of existing systems, predominantly developed as research prototypes. Since the shared address space of DSM is distributed across local memories, on each access to these data a lookup must be performed, in order to determine if the requested data is in the local memory, and if not, an action must be taken to bring it to the local memory. An action is also needed on write accesses in order to preserve the coherence of shared data. Both lookup and action can be performed in software, hardware, or the combination of both. According to this property, systems are classified into three groups: *software*, *hardware*, and *hybrid* implementations. The choice of implementation level usually depends on price/performance trade-offs. Although typically superior in performance, hardware implementations require additional complexity, allowable only in high-performance large-scale machines. Low-end systems, such as networks of personal computers, based on commodity microprocessors, still do not tolerate cost of additional hardware for DSM, and are limited to software implementation. For the class of mid-range systems, such as clusters of workstations, low-cost additional hardware, typically used in hybrid solutions, seems to be appropriate.

B. Software Distributed Shared Memory Implementations

Until the last decade distributed systems widely employed the message-passing communication paradigm. However, it appeared to be much less convenient than the shared memory programming model since the programmer must be aware of data distribution and explicitly manage data exchange via messages. In addition, they introduce severe problems in passing complex data structures, and process migration in multiple address spaces is aggravated. Therefore, the idea of building a software mechanism that provides the shared memory paradigm to the programmer on top of message passing emerged in the mid-1980s. Generally, this can be achieved in user-level runtime library routines, the operating system, or the programming language. Some DSM systems combine the elements of these three approaches. Larger grain sizes (on the order of a kilobyte) are typical for software solutions, since DSM management is usually supported through a virtual memory mechanism. It means that if the requested data are not present in local memory, a page fault handler will retrieve the page either from the local

memory of another cluster or from disk. Coarse-grain pages are advantageous for applications with high locality of references and also reduce the necessary directory storage. On the other hand, parallel programs characterized by fine-grain sharing are adversely affected, owing to false sharing and thrashing.

Software support for DSM is generally more flexible than the hardware support and enables better tailoring of the consistency mechanisms to the application behavior. However, in most cases it cannot compete with hardware implementations in performance. Apart from trying to introduce hardware accelerators to solve the problem, designers also concentrate on relaxing the consistency model, although this can put an additional burden on the programmer. The fact that research and experiments can rely on widely available programming languages and operating systems on the networks of workstations resulted in numerous implementations of software DSM.

1. User-Level and Combined Software Distributed Memory System Implementations

IVY [LI88] is one of the first proposed software DSM solutions, implemented as a set of user-level modules built on top of the modified Aegis operating system on the Apollo Domain workstations. IVY is composed of five modules. Three of them, from the client interface (*process management, memory allocation,* and *initialization*), consist of a set of primitives that can be used by application programs. *Remote operation* and *memory mapping* routines use the operating system low-level support. IVY provides a mechanism for consistency maintenance using an invalidation approach on 1-Kbyte pages. For experimental purposes, three algorithms for ensuring sequential consistency were implemented: the improved centralized manager, the fixed distributed manager, and the dynamic distributed manager. Performance measurements on a system with up to eight clusters have shown linear speedup in comparison with the best sequential solutions for some typical parallel programs. Although IVY performance could have been improved by implementing it at the system level rather than at the user level, its most important contribution was in proving the viability of the DSM concept in real systems with parallel applications.

A similar DSM algorithm is also used in Mermaid [ZHOU90]—the first system to provide a DSM paradigm in a heterogeneous environment (HDSM). The prototype configuration includes the SUN/Unix workstations and the DEC Firefly multiprocessors. The DSM mechanism was implemented at the user level, as a library package which is to be linked to the application programs. Minor changes to the SunOS operating system kernel included setting the access permission of memory pages from the user level, as well as passing the address of a DSM page to its user-level fault handler. Because of the heterogeneity of clusters, in addition to data exchange, the need for data conversion also arises. Besides the conversion of standard data types, for user-defined data types conversion routines and a table for mapping data types to particular routines must be provided by the user. A restriction is that just one data type is allowed per page. Mermaid ensures the variable page size that can be suited to data access patterns. Since the Firefly is a shared-memory multiprocessor, it was possible to compare physical versus distributed shared memory. The results showed that the speedup is increased far less than 20 percent when moving from DSM to physically shared memory for up to four nodes. Since the conversion costs are found to be substan-

tially lower than page transfer costs, it was concluded that the introduced overhead caused by heterogeneity was acceptably low—page fault delay for the heterogeneous system was comparable to that of the homogeneous system with only Firefly multiprocessors.

The Munin [CARTE91] DSM system includes two important features: type-specific coherence mechanisms and the release consistency model. The 16-processor prototype is implemented on an Ethernet network of SUN-3 workstations. Munin is based on the Stanford V kernel and the Presto parallel programming environment. It can be classified as a runtime system implementation, although a preprocessor that converts the program annotations, a modified linker, some library routines, and operating system support are also required. It employs different coherence protocols well suited to the expected access pattern for a shared data object type (Figure 2). The programmer is responsible for providing one of several annotations for each shared object, which selects appropriate low-level parameters of coherence protocol for this object. The data object directory is distributed among nodes and organized as a hash table. The release consistency model is implemented in software with delayed update queues for efficient merging and propagating of write sequences. Evaluation using two representative Munin programs (with only minor annotations) shows that their performance is less than 10 percent worse compared to their carefully hand-coded message passing counterparts.

Another DSM implementation that counts on significant reduction of data traffic by relaxing consistency semantics according to the lazy release consistency model is TreadMarks [KELEH94]. This is a user-level implementation that relies on Unix standard libraries in order to accomplish remote process creation, interprocess communication, and memory management. Therefore, no

Data object type	Coherence mechanism
Private	None
Write-once	Replication
Write-many	Delayed update
Results	Delayed update
Synchronization	Distributed locks
Migratory	Migration
Producer-consumer	Eager object movement
Read mostly	Broadcast
General read-write	Ownership

Figure 2. Munin's type-specific memory coherence.

modifications to the operating system kernel or particular compiler are required. TreadMarks runs on commonly available Unix systems. It employs an invalidation-based protocol, which allows multiple concurrent writers to modify the page. On the first write to a shared page, DSM software makes a copy *(twin)* that can later be compared to the current copy of the page in order to make a *diff*—a record that contains all modifications to the page. Lazy release consistency does not require *diff* creation on each release (as in the Munin implementation), but allows it to be postponed until next acquire in order to obtain better performance. Experiments were performed using DECstation-5000/240s connected by a 100-Mbps ATM network and a 10-Mbps Ethernet, and good speedups for five SPLASH programs were reported. Results of experiments have shown that latency and bandwidth limitations can be overcome using more efficient communication interfaces.

Unlike Munin, which uses various coherence protocols on a type-specific basis, Midway [BERSH93] supports multiple consistency models (processor, release, entry) that can be dynamically changed within the same program, in order to implement a single consistency model—release consistency. Midway is operational on a cluster of MIPS R3000-based DEC stations, under the Mach OS. At the programming language level, all shared data must be declared and explicitly associated with at least one synchronization object, also declared as an instance of one of Midway's data types, which include locks and barriers. If the necessary labeling information is included, and all accesses to shared data done with appropriate explicit synchronization accesses, sequential consistency can also be achieved. Midway consists of three components: a set of keywords and function calls used to annotate a parallel program, a compiler that generates code that marks shared data as dirty when written to, and a runtime system that implements several consistency models. Runtime system procedure calls associate synchronization objects to runtime data. The control of versions of synchronization objects is done using the associated timestamps, which are reset when data are modified. For all consistency models, Midway uses an update mechanism. Although less efficient with the Ethernet connection, Midway shows close to linear speedups of chosen applications when using the ATM network.

Blizzard is another user-level DSM implementation that also requires some modifications to the operating system kernel [SCHOI94]. It uses Tempest—a user-level communication and memory interface that provides mechanisms necessary for both fine-grained shared memory and message passing. There are three variants of this approach: Blizzard-S, Blizzard-E, and Blizzard-ES. The essence of Blizzard-S, an entirely software variant, is the modification of executable code by inserting a fast routine before each shared memory reference. It is intended for state lookup and access control for the block. If the state check requires some action, an appropriate user handler is invoked. Blizzard-E, on the other hand, uses the machine's memory ECC (error correction code) bits to indicate an *invalid* state of the block by forcing uncorrectable errors. However, a *read-only* state is maintained by enforcing read-only protection at page level by the memory management unit (MMU). Otherwise, *read-write* permission is assumed. The third variant, Blizzard-ES, combines the ECC approach of Blizzard-E for read instructions, and software tests of Blizzard-S for write instructions. Performance evaluation of the three variants for several shared memory benchmarks reveals that Blizzard-S is the most efficient (typically within a factor of two). When compared to a hardware DSM implementation with fine-grain

access control (the KSR1 multiprocessor), the typical slowdown of Blizzard is severalfold depending on the application.

2. Operating System Software Distributed Shared Memory Implementations

In Mirage [FLEIS89], coherence maintenance is implemented inside the operating system kernel. The prototype consists of VAX 11/750s connected by Ethernet network, using the System V interface. The main contribution introduced by Mirage is that page ownership can be guaranteed for a fixed period of time, called *time window* Δ. In this way thrashing is avoided, and inherent processor locality can be better exploited. The value of the Δ parameter can be tuned statically or dynamically. Mirage uses the model based on page segmentation. A process that creates a shared segment defines its size, name, and access protection, while the other processes locate and access the segment by name. All requests are sent to the site of segment creation, called the *library site* (Figure 3) where they are queued and sequentially processed. The *clock site*, which provides the most recent copy of the page, is either a writer or one of the readers of the requested page, since the writer and the readers cannot possess copies of the same page simultaneously. Performance evaluation of the worst case example, in which two processes interchangeably perform writes to the same page, has shown that the throughput increase is highly sensitive to the proper choice of the parameter Δ value.

Clouds [RAMAC91] is an operating system that incorporates software-based DSM management and implements a set of primitives either on top of Unix, or in the context of the object-based operating system kernel Ra. Clouds is implemented on SUN-3 workstations connected via Ethernet. The distributed shared memory consists of objects, composed of segments, that have access attributes: *read-only, read-write, weak-read*, or *none*. Since the *weak-read* mode allows the node to obtain a copy of the page with no guarantee that the page will not be modified during read, the memory system behavior of Clouds without any specific restrictions leads to inconsistent DSM. Fetching of segments is based on *get*

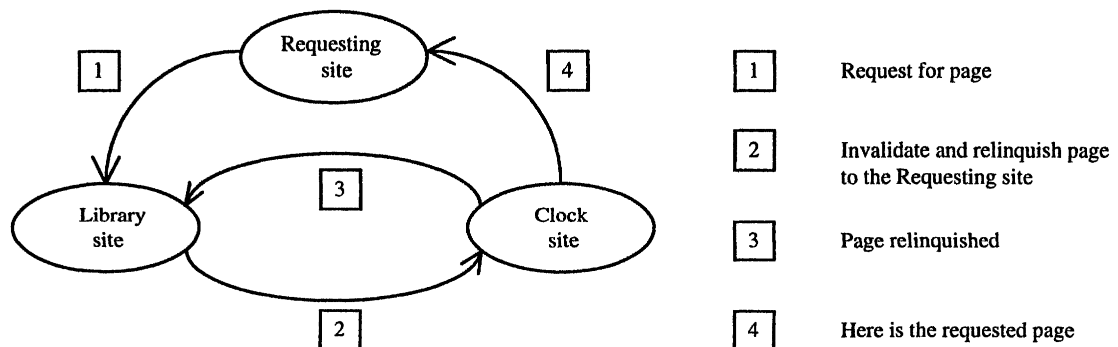

Figure 3. Write request for a page in Mirage.

and *discard* operations provided by a distributed shared memory controller (DSMC). This software module also offers P and V semaphore primitives as separate operations. The DSMC is, therefore, a part of the Clouds operating system, but implemented outside its kernel Ra. It is invoked by a DSM partition that handles segment requests from both Ra and user objects, and determines whether the request for segment should be satisfied locally by disk partition, or remotely by the distributed shared memory controller. Both DSM and DSMC partitions are also implemented on top of Unix, with minor changes due to the operating system dependencies.

3. Programming Language Concepts for Software Distributed Shared Memory

An architecture-independent language, Linda, is introduced in [AHUJA86]. Distributed shared memory in Linda is organized as a "tuple space"—a common pool of user-defined tuples (basic storage and access units consisting of data elements) that are addressed by logical names. Linda provides several special language operators for dealing with such distributed data structures, such as inserting, removing, and reading tuples, and so on. The consistency problem is avoided since a tuple must be removed from the tuple space before an update, and a modified version is reinserted again. By its nature, the Linda environment offers possibilities for process decoupling, transparent communication, and dynamic scheduling. Linda offers the use of replication as a method for problem partitioning. Linda is implemented on shared memory machines (Encore Multimax, Sequent Balance) as well as on loosely coupled systems (S/Net, an Ethernet network of MicroVAXes).

Software DSM implementations are extensively elaborated on in [BAL88]. A new model of shared data objects is proposed (passive objects accessible through predefined operations), and used in the Orca language for distributed programming. The distributed implementation is based on selective replication, migration, and an update mechanism. Different variants of update mechanism can be chosen, depending on the type of communication provided by the underlying distributed system (point-to-point messages, reliable multicast and unreliable multicast messages). Orca is predominantly intended for application programming.

Name and reference	Type of implementation	Type of algorithm	Consistency model	Granularity unit	Coherence policy
IVY [LI88]	User-level library + OS modification	MRSW	Sequential	1 Kbyte	Invalidate
Mermaid [ZHOU90]	User-level library + OS modifications	MRSW	Sequential	1 Kbyte, 8 Kbytes	Invalidate
Munin [CARTE91]	Runtime system + linker + library + preprocessor + OS modifications	Type-specific (SRSW, MRSW, MRMW)	Release	Variable-size objects	Type-specific (delayed update, invalidate)
Midway [BERSH93]	Runtime system + compiler	MRMW	Entry, release, processor	4 Kbytes	Update
TreadMarks [KELEH94]	User level	MRMW	Lazy release	4 Kbytes	Update, invalidate
Blizzard [SCHOI94]	User-level + OS kernel modification	MRSW	Sequential	32–128 bytes	Invalidate

Mirage [FLEIS89]	OS kernel	MRSW	Sequential	512 bytes	Invalidate
Clouds [RAMAC91]	OS, out of kernel	MRSW	Inconsistent, sequential	8 Kbytes	Discard segment when unlocked
Linda [AHUJA86]	Language	MRSW	Sequential	Variable (tuple size)	Implementation dependent
Orca [BAL88]	Language	MRSW	Synchronization dependent	Shared data object size	Update

[AHUJA86] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer,* Vol. 19, No. 8, May 1986, pp. 26–34.

[BAL88] H.E. Bal and A.S. Tanenbaum, "Distributed Programming with Shared Data," *Proc. Int'l Conf. Computer Languages '88*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 82–91.

[BERSH93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon, "The Midway Distributed Shared Memory System," *Proc. COMPCON '93,* IEEE CS Press, Los Alamitos, Calif., 1993, pp. 528–537.

[CARTE91] J.B. Carter, J.K. Bennet, and W. Zwaenepoel, "Implementation and Performance of Munin," *Proc. 13th ACM Symp. Operating Systems Principles,* ACM Press, New York, N.Y., 1991, pp. 152–164.

[FLEIS89] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proc. 14th ACM Symp. Operating System Principles,* ACM Press, New York, N.Y., 1989, pp. 211–223.

[KELEH94] P. Keleher et al., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. USENIX Winter 1994 Conf.*, 1994, pp. 115–132.

[LI88] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. 1988 Int'l Conf. Parallel Processing*, Penn State Press, University Park, Pa., 1988, pp. 94–101.

[RAMAC91] U. Ramachandran and M.Y.A. Khalidi, "An Implementation of Distributed Shared Memory," *Software Practice and Experience,* Vol. 21, No. 5, May 1991, pp. 443–464.

[SCHOI94] I. Schoinas et al., "Fine-Grain Access Control for Distributed Shared Memory," *Proc. 6th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, N.Y., 1994, pp. 297–306.

[ZHOU90] S. Zhou, M. Stumm, and T. McInerney, "Extending Distributed Shared Memory to Heterogeneous Environments," *Proc. 10th Int'l Conf. Distributed Computing Systems,* IEEE CS Press, Los Alamitos, Calif., 1990, pp. 30–37.

C. Hardware-Level Distributed Shared Memory Implementations

A hardware-implemented DSM mechanism ensures automatic replication of shared data in local memories and processor caches, transparently for software layers. This approach efficiently supports fine-grain sharing. The nonstructured, physical unit of replication and coherence is small, typically cache line. Consequently, a hardware DSM mechanism usually represents an extension of the principles found in cache coherence schemes of scalable shared-memory architectures. Communication requirements are considerably reduced, since detrimental effects of false sharing and thrashing are minimized with finer sharing granularities. Searching and directory functions implemented in hardware are much faster compared to the software-level implementations, and memory access latencies are decreased. However, advanced techniques used for coherence maintenance and latency reduction usually make the design complex and difficult to verify. Therefore, hardware DSM is often used in high-end machines where performance is more important than cost.

According to the memory system architecture, three groups of hardware DSM systems are regarded as especially interesting:

CC-NUMA—Cache coherent nonuniform memory access architecture

COMA—Cache-only memory architecture

RMS—Reflective memory system architecture

1. CC-NUMA Distributed Shared Memory Systems

In a CC-NUMA system (Figure 4), the shared virtual address space is statically distributed across local memories of clusters, which can be accessed both by the local processors and by processors from other clusters in the system, although with quite different access latencies. The DSM mechanism relies on directories with organization varying from a full map to different dynamic structures, such as singly or doubly linked lists and trees. The main effort is to achieve high performance (as in full-map schemes) and good scalability provided by reducing the directory storage overhead. To minimize latency, static partitioning of data should be done carefully, so as to maximize the frequency of local accesses. Performance indicators are also highly dependent on the interconnection topology. The invalidation mechanism is typically applied in order to provide consistency, while some relaxed memory consistency model can be used as a source of performance improvement. Typical representatives of this type of DSM approach are Memnet, Dash, and SCI.

Memnet (MEMory as NETwork abstraction)—a ring-based multiprocessor—is one of the earliest hardware DSM systems [DELP91]. The main goal was to avoid costly interprocessor communication via messages and to provide an abstraction of shared memory to applications directly from the network, without kernel OS intervention. The Memnet address space is mapped onto the local memories of each cluster *(reserved area)* in an NUMA fashion. Another part of each local memory is the *cache area,* which is used for replication of 32-byte blocks whose reserved area is in some remote host. The coherence protocol is

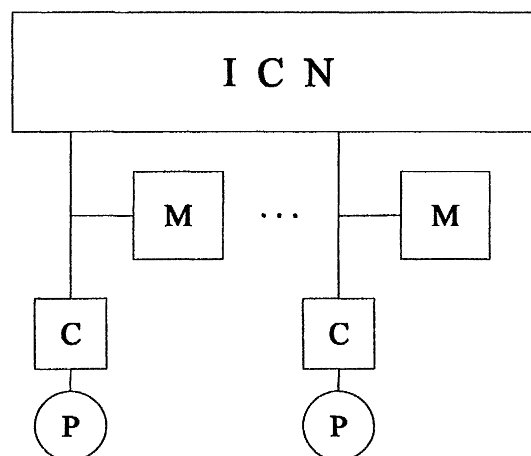

Figure 4. CC-NUMA memory architecture. P, Processor; C, cache; M, memory; ICN, interconnection network.

implemented in hardware state machines of the Memnet device in each cluster—a dual port memory controller on its local bus, and an interface to the ring. On a miss in local memory, the Memnet device sends an appropriate message that circulates on the ring. The message is inspected by each Memnet device on the ring in a snooping manner. The request is satisfied by the nearest cluster with a valid copy, which inserts requested data in the message before forwarding. The write request to a nonexclusive copy results in a message that invalidates other shared copies as it passes through each Memnet device that has a valid copy of that block. Finally, the message is received and removed from the ring by the interface of the cluster that generated it.

Dash (Directory Architecture for SHared memory), a scalable cluster multiprocessor architecture using a directory-based hardware DSM mechanism [LENOS92], also follows the CC-NUMA approach. Each four-processor cluster contains an equal part of the overall system's shared memory (*home* property) and corresponding directory entries. Each processor also has a two-level private cache hierarchy where the locations from other clusters' memories *(remote)* can be replicated or migrated in 16-byte blocks (unlike Memnet, where a part of local memory is used for this purpose). The memory hierarchy of Dash is split into four levels: (1) processor cache, (2) caches of other processors in the local cluster, (3) home cluster (cluster that contains directory and physical memory for a given memory block), and (4) remote cluster (cluster marked by the directory as holding the copy of the block). Coherence maintenance is based on a full-map directory protocol. A memory block can be in one of three states: *uncached* (not cached outside the home cluster), *cached* (one or more unmodified copies in remote clusters), and *dirty* (modified in some remote cluster). In most cases, owing to the property of locality, references can be satisfied inside the local cluster. Othewise, a request is sent to the home cluster for the involved block, which takes some action according to the state found in its directory. Improved performance is achieved by using a relaxed memory consistency model—release consistency, as well as some memory access optimizations. Techniques for reducing memory latency, such as software-controlled prefetching, update, and deliver operations, are also used in order to improve performance. Hardware support for synchronization is also provided.

Memory organization in an SCI-based CC-NUMA DSM system is similar to Dash and data from remote memories can be cached in local caches. Although the IEEE P1596 Scalable Coherent Interface (SCI) [JAMES94] represents an interface standard, rather than a complete system design, among other issues, it defines a scalable directory cache coherence protocol. Instead of centralizing the directory, SCI distributes it among those caches that are currently sharing the data, in the form of doubly linked lists. The directory entry is a shared data structure that may be concurrently accessed by multiple processors. The home memory controller keeps only a pointer to the head of the list and a few status bits for each cache block, while the local cache controllers must store the forward and backward pointers, and the status bits.

A read miss request is always sent to the home memory. The memory controller uses the requester identifier from the request packet to point to the new head of the list. The old head pointer is sent back to the requester along with the data block (if available). It is used by the requester to chain itself as the head of the list, and to request the data from the old head (if not supplied by the home cluster). In the case of write to a nonexclusive block, the request for the ownership is

also sent to the home memory. All copies in the system are invalidated by forwarding an invalidation message from the head down the list, and the requester becomes the new head of the list. However, the distribution of individual directory entries increases the latency and complexity of the memory references. To reduce latency and to support additional functions, the SCI working committee has proposed some enhancements, such as converting sharing lists to sharing trees, request combining, support for queue-based locks, and so on.

2. COMA Distributed Shared Memory Systems

Cache-only memory architecture (Figure 5) uses local memories of the clusters as huge caches for data blocks from virtual shared address space (attraction memories). There is no physical memory home location predetermined for a particular data item, and it can be replicated and migrated in attraction memories on demand. Therefore, the distribution of data across local memories (caches) is dynamically adaptable to the application behavior. The existing COMAs are characterized by hierarchical network topologies that simplify two main problems in these types of systems: location of a data block and replacement. They are less sensitive to static distribution of data than are NUMA systems. Owing to its cache organization, attraction memories are efficient in reducing capacity and conflict miss rates. On the other hand, the hierarchical structure imposes slightly higher communication and remote miss latencies. A somewhat increased storage overhead for keeping the information typical of cache memory is also inherent to COMA systems. The two most relevant representatives of COMA systems are KSR1 and DDM.

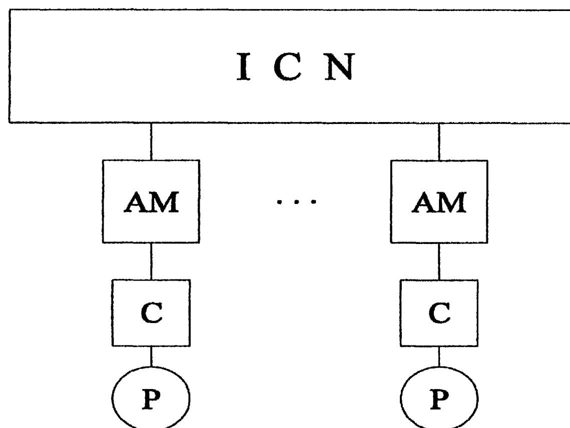

Figure 5. Cache-only memory architecture (COMA). P, Processor; C, cache; AM, attraction memory; ICN, interconnection network.

The KSR1 multiprocessor represents one of the early attempts to make DSM systems available on the market [FRANK93]. It consists of a ring-based hierarchical organization of clusters, each with a local 32-Mbyte cache. The unit of allocation in local caches is a page (16 Kbytes), while the unit of transfer and sharing in local caches is a subpage (128 bytes). The dedicated hardware responsible for locating, copying, and maintaining coherence of subpages in local caches is called the ALLCACHE engine, and it is organized as a hierarchy with directories on intermediate levels. The ALLCACHE engine transparently routes the requests through the hierarchy. Missed accesses are most likely to be satisfied by clusters on the same or next higher level in the hierarchy. In that way, the ALLCACHE organization minimizes the path to locate a particular address. The coherence protocol is invalidation based. Possible states of a subpage within a particular local cache include the following: *exclusive* (only valid copy), *nonexclusive* (owner; multiple copies exist), *copy* (nonowner; valid copy), and *invalid* (not valid, but allocated subpage). Besides these usual states, an *atomic* state is provided for synchronization purposes. Locking and unlocking the subpage are achieved by special instructions. As in all architectures with no main memory, where all data are stored in caches, the problem of the replacement of cache lines arises. There is no default destination for the line in the main memory, so the choice of a new destination and the directory update can be complicated and time consuming. Besides that, propagation of requests through hierarchical directories is responsible for longer latencies.

The DDM (Data Diffusion Machine) is another COMA multiprocessor [HAGER92]. The DDM prototype is made of four-processor clusters with an attraction memory and an asynchronous split-transaction bus. Attaching a directory on top of the local DDM bus, to enable its communication with a higher level bus of the same type, is the way DDM builds a large system with directory/bus-based hierarchy (as opposed to the KSR1 ring-based hierarchy). The directory is a set-associative memory that stores the state information for all items in attraction memories below it, but without data. The employed coherence protocol is of the snoopy write-invalidate type, which handles the attraction of data on read, erases the replicated data on write, and manages the replacement when a set in an attraction memory is full. An item can be in seven states; three of them correspond to Invalid, Exclusive, and Valid (typical for the snoopy protocols), while the state Dirty is replaced with a set of four transient states needed to remember the outstanding requests on the split-transaction bus. Transactions that cannot be completed on a lower level are directed through the directory to the level above. Similarly, the directory recognizes the transactions that need to be serviced by a subsystem and routes them onto the level below it.

3. Reflective Memory Distributed Shared Memory Systems

Reflective memory systems are DSM systems with a hardware-implemented update mechanism designed for fine data granularity. The global shared address space is formed out of the segments in local memories, which are designated as shared, and mapped to this space through programmable mapping tables in each cluster (Figure 6). Hence, the parts of this shared space are selectively replicated ("reflected") across different clusters. Coherence maintenance of shared regions is based on the full-replication, MRMW algorithm. Each write to an address in this shared address space in a cluster is propagated using a broadcast or mul-

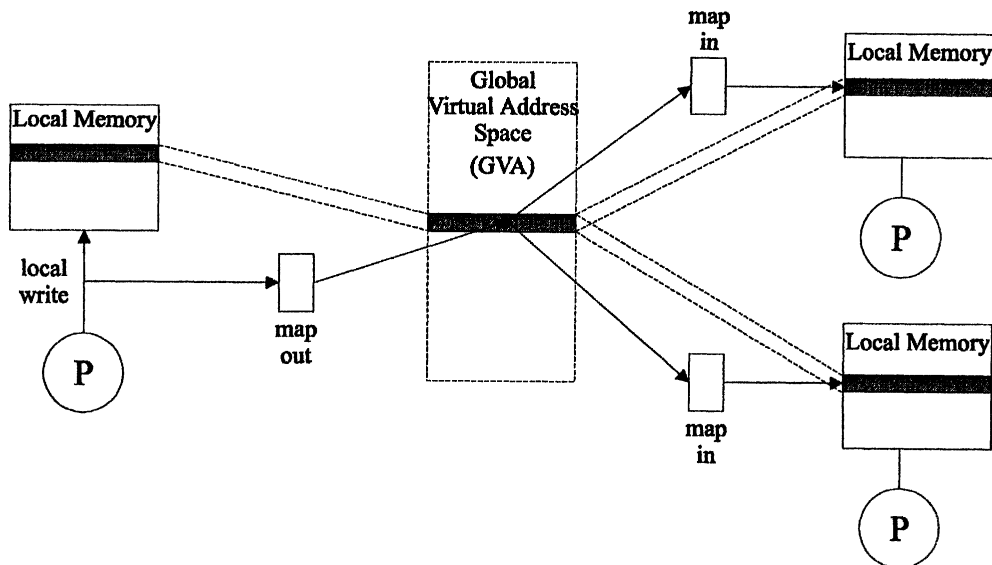

Figure 6. Reflective memory system (RMS) architecture.

ticast mechanism to all other clusters to which the same address is mapped, to keep it updated in a nondemand, anticipatory manner. The processor does not stall on writes, and computation is overlapped with communication. This is the source of performance improvement typical of relaxed memory consistency models. Also, there is no contention and long latencies as in typical shared memory systems, since unrestricted access to shared data and simultaneous accesses to local copies are ensured. On the other hand, all reads from the shared memory are local, with a deterministic access time. The principle of this DSM mechanism is similar to the write-update cache coherence protocols. Typical reflective memory systems are RMS and Merlin.

The reflective memory concept is applied in some existing systems with different clusters and network topologies. Since broadcast is the most appropriate mechanism for updating of replicated segments, the shared bus topology is especially convenient for the reflective memory architecture. A number of bus-based reflective memory systems (RMS), for example, the Encore Infinity [LUCCI95], have been developed by the Encore Computer Corporation for a wide range of applications. These systems typically consist of a lower number of minicomputer clusters connected by an RM bus—a "write-only" bus since traffic on it consists only of word-based distributed write transfers (address + value of the data word). Some later enhancements also allow for block-based updates (memory channel). The unit of replication is an 8-Kbyte segment. Segments are treated as "windows" that can be open (mapped into reflective shared space) or closed (disabled for reflection and exclusively accessed by each particular cluster). A replicated segment can be mapped to different addresses in each cluster. Therefore, the translation map tables are provided separately for the transmit (for each block of local memory), and receive (for each block of reflected address space) sides.

Although very convenient for broadcasting, bus-based systems are notorious for their restricted scalability. Hence, Merlin (MEmory Routed, Logical Interconnection Network) represents a reflective memory-based interconnection system using mesh topology with low-latency memory sharing on the word basis [MAPLE90]. Besides user-specified sharing information, OS calls are necessary to initialize routing maps and establish data exchange patterns before program execution. The Merlin interface in the host backplane monitors all memory changes, and on each write to the local physical memory mapped as shared it makes a temporary copy of the address and the written value noninvasively. Instead of broadcast as in RMS, multicast is used to transmit the word packet through the interconnection network to all shared copies in other local memories. Two types of sharing are supported in hardware: synchronous (updates to the same region are routed through a specific canonical cluster) and rapid (updates are propagated individually by the shortest routes). This system also addresses the synchronization, interrupt, and lock handling integrated with reflective memory sharing. Merlin also provides a support for heterogeneous processing.

A similar principle is employed even in multicomputers to minimize message-passing overhead, for example, in the SHRIMP multicomputer [BLUMR94]. A virtual memory-mapped network interface implements an "automatic update" feature in hardware. On analyzing communication patterns, it was noticed that most messages from a send buffer are targeted to the same destination. Therefore, after some page (send buffer) is mapped out to some other's cluster memory (receive buffer), each local write (message) to this page is also immediately propagated to this destination. There are two implementations of automatic updates: single-write and block-write. In this way, passing the message avoids any software involvement.

Name and reference	Cluster configuration	Network	Type of algorithm	Consistency model	Granularity unit (bytes)	Coherence policy
Memnet [DELP91]	Single processor, Memnet device	Token ring	MRSW	Sequential	32	Invalidate
Dash [LENOS92]	SGI 4D/340 (4 PEs, 2-L caches), local memory	Mesh	MRSW	Release	16	Invalidate
SCI [JAMES94]	Arbitrary	Arbitrary	MRSW	Sequential	16	Invalidate
KSR1 [FRANK93]	64-bit custom PE, I+D caches, 32-Mbyte local memory	Ring-based hierarchy	MRSW	Sequential	128	Invalidate
DDM [HAGER92]	4 MC88110s, 2-L caches, 8- to 32-Mbyte local memory	Bus-based hierarchy	MRSW	Sequential	16	Invalidate
Merlin [MAPLE90]	40-MIPS computer	Mesh	MRMW	Processor	8	Update
RMS [LUCCI95]	1–4 processors, caches, 256-Mbyte local memory	RM bus	MRMW	Processor	4	Update

[DELP91] G. Delp, D. Farber, and R. Minnich, "Memory as a Network Abstraction," *IEEE Network*, July 1991, pp. 34–41.

[FRANK93] S. Frank, H. Burkhardt III, and J. Rothnie, "The KSR1: Bridging the Gap between Shared Memory and MPPs," *Proc. COMPCON '93*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 285–294.

[HAGER92] E. Hagersten, A. Landin, and S. Haridi, "DDM—A Cache-Only Memory Architecture," *Computer*, Vol. 25, No. 9, Sept. 1992, pp. 44–54.

[JAMES94] D.V. James, "The Scalable Coherent Interface: Scaling to High-Performance Systems," *Proc. COMPCON '94: Digest of Papers*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 64–71.

[LENOS92] D. Lenoski et al., "The Stanford Dash Multiprocessor," *Computer*, Vol. 25, No. 3, Mar. 1992, pp. 63–79.

[LUCCI95] S. Lucci et al., "Reflective-Memory Multiprocessor," *Proc. 28th IEEE/ACM Hawaii Int'l Conf. System Sciences*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 85–94.

[MAPLE90] C. Maples and L. Wittie, "Merlin: A Superglue for Multicomputer Systems," *Proc. COMPCON '90*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 73–81.

D. Hybrid Level Distributed Shared Memory Implementations

During the evolution of this field, numerous entirely hardware or software implementations of DSM mechanism were proposed. However, even in entirely hardware DSM approaches, there are software-controlled features explicitly visible to the programmer for memory reference optimization (for example, prefetch, update, and deliver in Dash; prefetch and poststore in KSR1). On the other side, many purely software solutions require some hardware support (for example, virtual memory management hardware in IVY; error correction code in Blizzard-E). As can be expected, neither approach has all the advantages. Therefore, it seemed quite natural to employ hybrid methods, with predominantly or partially combined hardware and software elements, in order to balance the cost/complexity trade-offs. A solution implemented in some industrial computers provided a DSM-like paradigm in software executed by a microcontroller located on a separate communication board [PROTI93].

One of the typical hybrid approaches is to achieve replication and migration of data from a shared virtual address space across the clusters in software, while their coherence management is implemented in hardware. PLUS is an example of such a system [BISIA90]. In PLUS, software is responsible for data placement and replacement in local memories in units of 4-Kbyte pages. However, memory coherence for replicated data is maintained on the 32-bit word basis by a nondemand, write-update protocol implemented in hardware. Replicated instances of a page are chained into an ordered, singly linked list, headed with the *master* copy, in order to ensure the propagation of updates to all copies. Since a relaxed consistency model is assumed, writes are nonblocking, and the *fence* operation is available to the user for explicit strong ordering of writes. In order to optimize the synchronization, PLUS provides a set of specialized interlocked read-modify-write operations called *delayed* operations. Their latency is hidden by splitting them into *issue* and *verify* phases, and allowing them to proceed concurrently with regular processing.

Some solutions to DSM issues in another hybrid DSM system—Galactica Net [WILSO94]—are similar to those applied in PLUS. Pages from virtual address space are replicated on demand under control of virtual memory software, implemented in the Mach operating system. In addition, there is hardware support for the virtual memory mechanism, realized through a block transfer engine, which can rapidly transfer pages in reaction to page faults. A page can be in one of three states: *read-only, private,* and *update,* denoted by tables main-

tained by the OS. The coherence for writable shared pages (*update* mode) is kept by a write-update protocol implemented entirely in hardware. All copies of a shared page in *update* mode are organized in a *virtual sharing ring*—a linked list used for forwarding of updates. Virtual shared rings are realized using update routing tables kept in the network interface of each cluster, and also maintained by software. Therefore, write references to pages in update mode are detected by hardware and propagated according to the table. Because of the update mechanism, for some applications, broadcast of excessive updates can produce a large amount of traffic. Besides that, the unit of sharing is quite large, and false sharing effects can adversely affect performance. On recognizing an actual reference pattern, Galactica Net can dynamically switch from a hardware update scheme to software invalidate coherence (another hybrid and adaptive feature), using a competitive protocol based on per-page update counters. When remote updates to a page far exceed local references, an interrupt is raised, and the OS invalidates this page and removes it from its sharing ring in order to prevent the unnecessary traffic to unused copies.

MIT's Alewife is a specific hybrid system that implements the LimitLESS directory protocol. This protocol represents a hardware-based coherence scheme supported by a software mechanism [CHAIK94]. Directory entries contain only a limited number of hardware pointers, in order to reduce the storage requirements—the design assumes that it is sufficient in the vast majority of cases. Exceptional circumstances, when more pointers are needed, are handled in software. In those infrequent cases, an interrupt is generated, and a full-map directory for the block is emulated in software. A fast trap mechanism provides support for this feature, and a multiple context concept is used for hiding the memory latency. The main advantage of this approach is that the applied directory coherence protocol is storage efficient, while performing about as well as the full-map directory protocol.

Unlike Alewife, the basic idea behind the FLASH multiprocessor is to implement the memory coherence protocol in software, but to move the burden of its execution from the main processor to an auxiliary protocol processor—MAGIC (Memory And General Interconnection Controller) [KUSKI94]. This specialized programmable controller allows for efficient execution of protocol actions in a pipelined manner, avoiding context switches on the main processor. This approach, which also ensures great flexibility in experimenting and testing, is followed in some other systems, for example the network interface processor (NP) in Typhoon [REINH94]. The NP uses a hardware-assisted dispatch mechanism to invoke a user-level procedure to handle some events.

To improve performance, a hybrid approach called *cooperative shared memory* is based on programmer-supplied annotations [HILL93]. The programmer identifies the segments that use shared data with corresponding *Check-In* (exclusive or shared access) and *Check-Out* (relinquish) annotations, executed as memory system directives. These performance primitives do not change program semantics (even misapplied), but reduce unintended communication caused by thrashing and false sharing. Cooperative *prefetch* can also be used to hide the memory latency. The CICO programming model is completely and efficiently supported in hardware by a minimal directory protocol Dir_1SW. Traps to the system software occur only on memory accesses that violate the CICO.

A hybrid DSM protocol presented in [CHAND93] tries to combine the advantages of a software protocol for coarse-grain data regions and a hardware

coherence scheme for fine-grain sharing in a tightly coupled system. The software part of the protocol is similar to Midway. The programmer is expected to identify explicitly the *regions*—coarse-grain data structures. Usage annotations (for example, *BeginRead/EndRead, BeginWrite/EndWrite*) are then provided to identify program segments where the data from a certain region are safely referenced (without modification from other processors). Coherence of annotated data is kept by library routines invoked by these annotations. Coherence of nonannotated data is managed by means of a directory-based hardware protocol. Both software and hardware components of the protocol use the invalidation policy. The variable-size coherence unit of the software part of the protocol eliminates the problem of false sharing, while reducing remote misses by efficient bulk transfers of coarse-grain data and their replication in local memories. The protocol is also insensitive to initial data placement. Just like in Midway, Munin, and CICO, the main disadvantage is the burden put on the programmer to insert the annotations, although this appears to be not so complicated since this information about the data usage is naturally known.

The implementation of automatic update release consistency (AURC) turns Shrimp [BLUMR96] into an efficient hybrid DSM system. In this approach, only one copy of a page is kept consistent, using fine-grain automatic updates performed by hardware, after necessary software mappings. All other copies are kept consistent using an invalidation-based software protocol. An AURC refinement called *scope consistency* [IFTOD96] represents a successful compromise between entry and lazy release consistency.

Finally, since message passing and shared memory machines have been converging recently, some efforts have been made to integrate these two communication paradigms within a single system (Alewife, Cray T3D, FLASH, Typhoon). In addition to the above-mentioned coherence protocol, Alewife also allows explicit sending of messages in a shared memory program. Messages are delivered via an interrupt and dispatched in software. Cray T3D is also a physically distributed memory machine with hardware-implemented logically shared memory. It also integrates message passing extensively supported by DMA. Besides the Dash-like software-implemented directory cache coherence protocol, FLASH also provides message passing with low overhead, owing to some hardware support. Accesses to block transfer are allowed to the user without sacrificing protection, while the interaction of message data with cache coherence is ensured. Typhoon is a proposed hardware implementation especially suited for the Tempest interface—a set of user-level mechanisms that can be used to modify the semantics and performance of shared memory operations. Tempest consists of four types of user-level mechanisms: low-overhead messages, bulk data transfers, virtual memory management, and fine-grain access control. For example, user-level transparent shared memory can be implemented using *Stache*—a user library with Tempest fine-grain access mechanisms. *Stache* replicates the remote data in part of a cluster's local memory according to a COMA-like policy. It maps virtual addresses of shared data to local physical memory at page granularity, but maintains coherence at the block level. A coherence protocol similar to LimitLESS is implemented entirely in software.

Name and reference	Cluster configuration and network	Type of algorithm	Consistency model	Granularity unit	Coherence policy
PLUS [BISIA90]	M88000, 32-Kbyte cache, 8- to 32-Mbyte local memory, mesh	MRMW	Processor	4 Kbytes	Update
Galactica Net [WILSO94]	4 M88110s, 2-L caches, 256-Mbyte local memory, mesh	MRMW	Multiple	8 Kbytes	Update/Invalidate
Alewife [CHAIK94]	Sparcle PE, 64-Kbytes cache, 4-Mbyte local memory, CMMU, mesh	MRSW	Sequential	16 bytes	Invalidate
FLASH [KUSKI94]	MIPS T5, I+D caches, MAGIC controller, mesh	MRSW	Release	128 bytes	Invalidate
Typhoon [REINH94]	SuperSPARC, 2-L caches, NP controller	MRSW	Custom	32 bytes	Invalidate custom
Hybrid DSM [CHAND93]	FLASH-like	MRSW	Release	Variable	Invalidate
Shrimp [IFTOD96]	16 Pentium PC nodes, Intel Paragon routing network	MRSW	AURC, Scope	4 Kbytes	Update/Invalidate

[BISIA90] R. Bisani and M. Ravishankar, "PLUS: A Distributed Shared-Memory System," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 115–124.

[CHAIK94] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Software-Extended Coherent Shared Memory: Performance and Cost," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 314–324.

[CHAND93] R. Chandra et al., "Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols," CSL-TR-93-597, Stanford University, Stanford, Calif., Dec. 1993.

[IFTOD96] L. Iftode, J. Pal Singh, and K. Li, "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," *Proc. 8th Ann. Symp. Parallel Algorithms and Architectures,* 1996, pp. 277–287.

[KUSKI94] J. Kuskin et al., "The Stanford FLASH Multiprocessor," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 302–313.

[REINH94] S. Reinhardt, J. Larus, and D. Wood, "Tempest and Typhoon: User-Level Shared Memory," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 325–336.

[WILSO94] A. Wilson, R. LaRowe, and M. Teller, "Hardware Assist for Distributed Shared Memory," *Proc. 13th Int'l Conf. Distributed Computing Systems,* IEEE CS Press, Los Alamitos, Calif., 1993, pp. 246–255.

E. Conclusion

This survey provides extensive coverage of relevant topics in an increasingly important area—distributed shared memory computing. A special attempt has been made to give a broad overview of various approaches, presented according to the implementation level of the DSM mechanism. Because of the combined advantages of the shared memory and distributed systems, DSM approaches appear to be a viable step toward large-scale high-performance systems with reduced cost in parallel software development. In spite of this, the building of successful commercial systems that follow the DSM paradigm is still in its infancy; and research prototypes still prevail. Therefore, the DSM field remains a very active research area. Some of the promising research directions include (1) improving the DSM algorithms and mechanisms, and adapting them to the characteristics of typical applications and system configurations, (2) synergistic combining of hardware and software DSM implementations, (3) integration of the shared memory and message-passing programming paradigms, (4) creating new and innovative system architectures (especially in the memory system), (5) combining multiple consistency models, and so on. From this point of view, further

investments in exploring, developing, and implementing DSM systems seem to be quite justified and promising.

References

[BLACK89] D.L. Black, A. Gupta, and W. Weber, "Competitive Management of Distributed Shared Memory," *Proc. COMPCON '89*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 184–190.

[BLUMR94] M. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proc. 21st Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1994, pp. 142–153.

[FLYNN95] M.J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design,* Jones and Bartlett Publishers, Boston, Mass., 1995.

[GHARA90] K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1990, pp. 15–26.

[HILL93] M. Hill, J. Larus, and S. Reinhardt, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors," *ACM Trans. Computer Systems*, Nov. 1993, pp. 300–318.

[KELEH92] P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. 19th Ann. Int'l Symp Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1992, pp. 13–21.

[KESSL89] R.E. Kessler and M. Livny, "An Analysis of Distributed Shared Memory Algorithms," *Proc. 9th Int'l Conf. Distributed Computing Systems,* IEEE CS Press, Los Alamitos, Calif., 1989, pp. 498–505.

[LIHUD89] 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.

[LO94] V. Lo, "Operating System Enhancements for Distributed Shared Memory," *Advances in Computers,* Vol. 39, 1994, pp. 191–237.

[PROTI93] J. Protić and M. Aleksić, "An Example of Efficient Message Protocol for Industrial LAN," *Microprocessing and Microprogramming,* Vol. 37, Jan. 1993, pp. 45–48.

[PROTI95] J. Protić, M. Tomašević, and V. Milutinović, "A Survey of Distributed Shared Memory: Concepts and Systems," Technical Report No. ETF-TR-95-157, Department of Computer Engineering, Univ. Belgrade, Belgrade, Yugoslavia, July 1995.

[PROTI96] J. Protić, M. Tomašević, and V. Milutinović, "Distributed Shared Memory: Concepts and Systems," *Parallel & Distributed Technology*, Vol. 4, No. 2, 1996, pp. 63–79.

[STUM90] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer,* Vol. 23, No. 5, May 1990, pp. 54–64.

[TARTA95] I. Tartalja and V. Milutinović, "A Survey of Software Solutions for Maintenance of Cache Consistency in Shared Memory Multiprocessors," *Proc. 28th Ann. Hawaii Int'l Conf. System Sciences,* IEEE CS Press, Los Alamitos, Calif., 1995, pp. 272–282.

[TOMAS94a] M. Tomašević and V. Milutinović, "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors, Part 1 (Basic Issues)," *IEEE MICRO,* Vol. 14, No. 5, Oct. 1994, pp. 52–59.

[TOMAS94b] M. Tomašević and V. Milutinović, "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors, Part 2 (Advanced Issues)," *IEEE MICRO,* Vol. 14, No. 6, Dec. 1994, pp. 61–66.

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.

Reprinted from *Computer*, Vol. 24, No. 8, Aug. 1991, pp. 52–60.
Copyright © 1991 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

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.

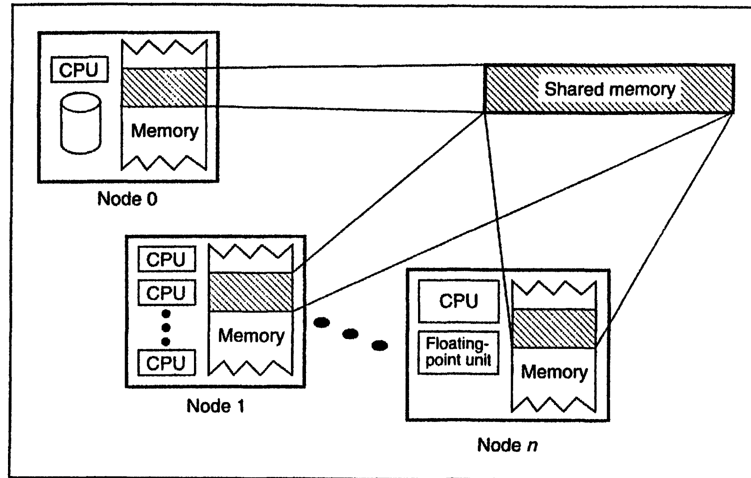

Figure 1. Distributed shared memory.

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.

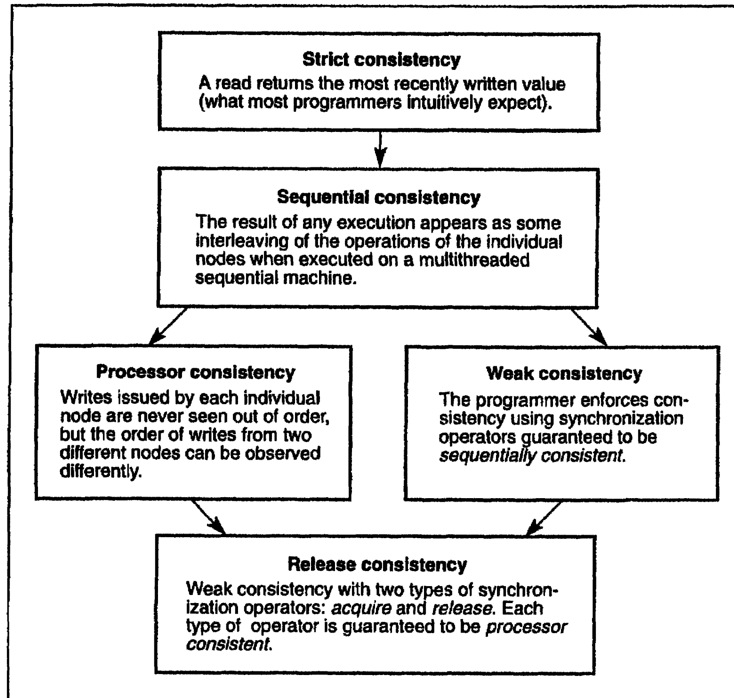

Figure 2. Intuitive definitions of memory coherence. The arrows point from stricter to weaker consistencies.

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 Tanenbaum, 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

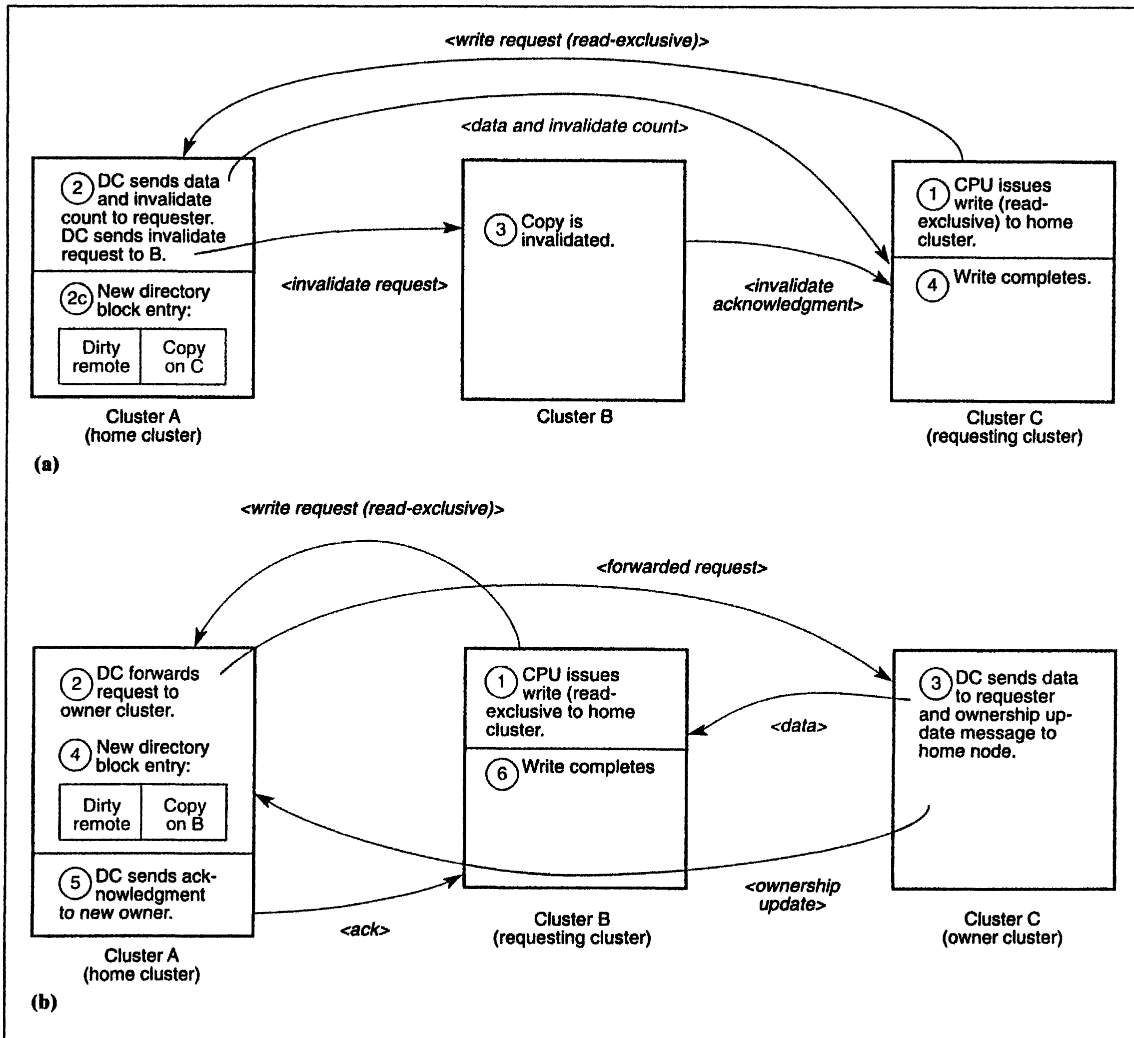

Figure 3. Simplified Dash write-invalidate protocol: (a) Data is shared remote; (b) data is dirty remote (after events depicted in Figure 3a). (DC stands for directory controller.)

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-

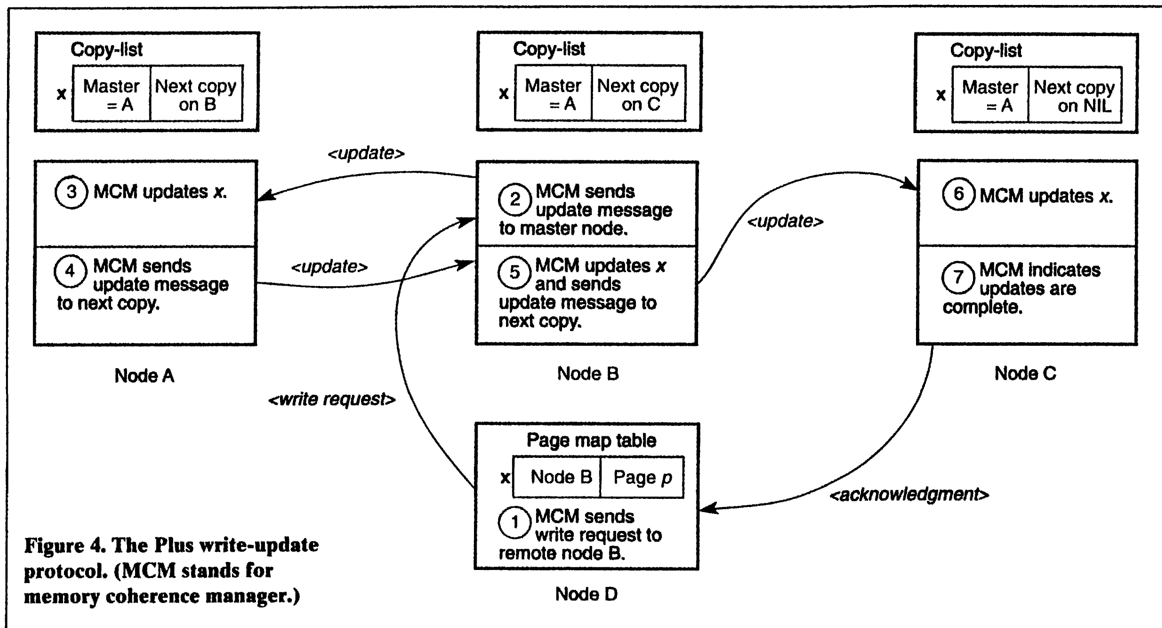

Figure 4. The Plus write-update protocol. (MCM stands for memory coherence manager.)

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 turther 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.