Distributed Shared Memory: Concepts and Systems

Jelica Protić, Milo Tomašević, and Veljko Milutinović
University of Belgrade

In surveying current approaches to distributed shared-memory computing, these authors find that the reduced cost of parallel software development will help make the DSM paradigm a viable solution to large-scale, high-performance computing.

Research and development of systems with multiple processors has shown significant progress recently. These systems, needed to deliver the high computing power necessary for satisfying the ever-increasing demands of today's typical applications, usually fall into two large classifications, according to their memory system organization: shared- and distributed-memory systems.

A shared-memory system[1] (often called a tightly coupled multiprocessor) makes a global physical memory equally accessible to all processors. These systems offer a general and convenient programming model that enables simple data sharing through a uniform mechanism of reading and writing shared structures in the common memory. Users can readily emulate other programming models on these systems. The programming ease and portability of these systems cut parallel software development costs. However, shared-memory multiprocessors typically suffer from increased contention and longer latencies in accessing the shared memory, which degrades peak performance and limits scalability compared to distributed systems. Memory system design also tends to be complex.

In contrast, a distributed-memory system (often called a multicomputer) consists of multiple independent processing nodes with local memory modules, connected by a general interconnection network. The scalable nature of distributed-memory systems makes systems with very high computing power possible. However, communication between processes residing on different nodes involves a message-passing model that requires explicit use of send/receive primitives. Because they must take care of data distribution across the system and manage the communication, most programmers find this process more difficult. Also, process migration imposes problems because of different address spaces. Therefore, compared to shared-memory systems, hardware problems are easier and software problems more complex in distributed-memory systems.

1063-6552/96/$4.00 © 1996 IEEE

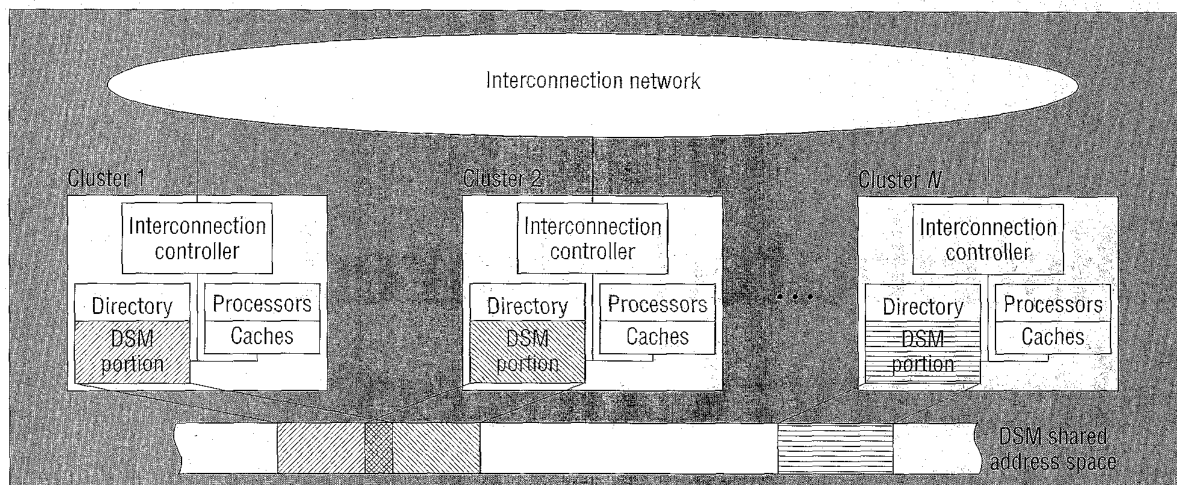

Figure 1. Structure and organization of a DSM system.

A relatively new concept—distributed shared memory[2,3]—combines the advantages of the two approaches. A DSM system logically implements the shared-memory model on a physically distributed-memory system. System designers can implement the specific mechanism for achieving the shared-memory abstraction in hardware or software in a variety of ways. The DSM system hides the remote communication mechanism from the application writer, preserving the programming ease and portability typical of shared-memory systems. DSM systems allow relatively easy modification and efficient execution of existing shared-memory system applications, which preserves software investments while maximizing the resulting performance. In addition, the scalability and cost-effectiveness of underlying distributed-memory systems are also inherited. Consequently, DSM systems offer a viable choice for building efficient, large-scale multiprocessors.

The DSM model's ability to provide a transparent interface and convenient programming environment for distributed and parallel applications have made it the focus of numerous research efforts in recent years. Current DSM system research focuses on 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. Others extend strategies applied in shared-memory multiprocessors with private caches to multilevel memory systems.[4–6]

This article reviews the increasingly important area of DSM. After first covering general DSM concepts and approaches, it surveys existing DSM systems, developed either as research prototypes or commercial products and standards. Although not exhaustive, this survey tries to provide an extensive, up-to-date overview of several key implementation schemes for maintaining data in DSM systems.

General DSM system structure

A DSM system generally involves a set of nodes or clusters, connected by an interconnection network (Figure 1). A cluster itself can be a uniprocessor or a multiprocessor system, usually organized around a shared bus. Private caches attached to the processors are virtually inevitable for reducing memory latency. Each system cluster contains a physically local memory module, which maps partially or entirely to the DSM global address space. Regardless of the network topology—bus, ring, mesh, or local-area network—a specific interconnection controller in each cluster must connect it into the system.

Information about states and current locations of particular data blocks usually resides in a system table or directory. Directory storage and organization are among the most important design decisions, greatly affecting system scalability. Directory organization varies from full-map storage to different dynamic organizations, such as single- or double-linked lists and trees. No matter the organization, the cluster provides storage either for the entire directory or for just part of it. In this way, the system directory can distribute 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.

Classifications of DSM systems

Since the first DSM research efforts in the mid-eighties, interest in this concept has grown significantly, resulting in tens of systems developed predominantly as

research prototypes. Designers of the early DSM systems were inspired by the principles of virtual memory, as well as by cache-coherence maintenance in shared-memory multiprocessors.

Networks of workstations, which are becoming increasingly popular and powerful, represent the most suitable platform for many programmers to approach parallel computing. However, communication latency, including operating system overhead and transfer time, remains the main obstacle for matching the performance of high-end machines with those network systems. Conversely, designers of shared-memory multiprocessors strive for scalability, achieved through physical distribution of shared memory and sophisticated organization of the overall system through such techniques as clustering and hierarchical layout.

Thus, as the gap between multiprocessors and multicomputers (that early DSM intended to bridge) narrows and the basic ideas and performance of both classes of systems seemingly converge, many more emerging systems fit into the large family of modern DSM. To eliminate misunderstanding, we adopt a general definition for this family, which assumes that all systems providing a shared-memory abstraction on a distributed-memory system belong to the DSM category.

There are three key issues when accessing data items in the DSM address space, while keeping the data consistent:

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

DSM ALGORITHMS

The algorithms for implementing DSM deal with two basic problems:

- static and dynamic distribution of shared data across the system, to minimize access latency, and
- preserving a coherent view of shared data, while minimizing coherence-management overhead.

Two frequently used strategies for distributing shared data are *replication* and *migration*. Replication allows multiple copies of the same data item to reside in different local memories (or caches). It is mainly used to enable simultaneous accesses by different sites to the same data, predominantly when read sharing prevails.

Migration implies that only a single copy of a data item exists at any one time, so the data item must be moved to the requesting site for exclusive use. To decrease coherence-management overhead, users prefer this strategy when sequential patterns of write sharing are prevalent. System designers must choose a DSM algorithm that is 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.[7–10] Our presentation follows a classification of algorithms similar to that of Michael Stumm and Songnian Zhou.[8]

Single reader/single writer algorithms

This class of algorithms prohibits replication, while permitting but not requiring migration. The simplest DSM management algorithm is the *central server* algorithm.[8] The approach relies on a unique central server that services all access requests from other nodes to shared data, physically located on this node. This algorithm suffers from performance problems because 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 spaces onto several different servers. Simple mapping functions, such as hashing, can serve to locate the appropriate server for the corresponding piece of data.

More sophisticated SRSW algorithms also permit migration. However, only one copy of the data item can exist at any one time, and this copy can migrate upon demand. This kind of algorithm is called *hot potato*.[10] If an application exhibits high reference locality, the cost of data migration is amortized over multiple accesses, because data moves not as individual items, but in fixed-size units—blocks. The algorithm can perform well when a longer sequence of accesses from one processor uninterrupted by accesses from other processors is likely, and write after read to the same data occurs frequently. Anyway, performance of this rarely used algorithm is restrictively low, because it does not capitalize on the parallel potential of multiple read-only copies, when the read sharing prevails.

Multiple reader/single writer algorithms

The main intention of MRSW (or *read-replication)* algorithms is to reduce the average cost of read operations, counting that read sharing is the prevalent pattern in parallel applications. To this end, they allow simultaneous

local execution of read operations at multiple hosts. Only one host at a time can receive permission to update a replicated copy. A write to a writable copy increases the cost of this operation, because the use of other replicated stale copies must be prevented. Therefore, MRSW algorithms are usually invalidation-based. A great many protocols follow this principle.

Algorithms in this class differ in the allocation of DSM management responsibility. Kai Li and Paul Hudak proposed several of them.[7] Several terms need defining before we discuss those algorithms:

- *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, and
- *Copy set:* a set of all sites that have copies of the data block.

The algorithms proposed by Li and Hudak include:

(1) *Centralized manager algorithm.* All read and write requests go to the manager, which is the only site that keeps the identity of a particular data block's owner. The manager forwards the data request to the owner, and waits for confirmation from the requesting site, indicating that it received the copy of the block from the owner. For 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).

(2) *Improved centralized manager algorithm.* Unlike the original centralized manager algorithm, the owner, not the manager, keeps the copy set. The copy set goes together with the data to the new owner, which is also responsible for invalidations. Here, decentralized synchronization improves overall performance.

(3) *Fixed distributed manager algorithm.* In this algorithm, instead of centralizing the management, each site manages a predetermined subset of data blocks. The distribution proceeds according to some default mapping function. Clients can still override it by supplying their own mapping, tailored to the application's expected behavior. When a parallel program exhibits a high rate of requests for data blocks, this algorithm outperforms the centralized solutions.

(4) *Broadcast distributed manager algorithm.* This algorithm has no manager. Instead, the requesting processor sends a broadcast message to find the data block's true owner. The owner performs all actions just like the manager in the previous algorithms, and keeps the copy set. But in this approach, all processors must process each broadcast, slowing their own computations.

(5) *Dynamic distributed manager algorithm.* In this algorithm, the identity of the probable owner, not the real owner, is kept for each particular data block. All requests go to the probable owner, which usually is also the real owner. However, if the probable owner is not the real one, the algorithm forwards the request to the node representing the probable owner according to the information kept in its own table. For every read and write request, forward, and invalidation message, the probable owner field changes accordingly, to decrease the number of messages to locate the real owner. This algorithm is often called Li's algorithm. For its basic version, where the ownership changes on both read and write faults, the algorithm's performance does not deteriorate as more processors add to the system, but rather degrades logarithmically when more processors contend for the same data block.[7]

A modification of the dynamic distributed manager algorithm suggests a distribution of the copy set that should be organized as a tree, rooted at the owner site. This modification also distributes the responsibility for invalidations.[7]

Multiple reader/multiple writer algorithms

The MRMW (also called *full-replication*) algorithm allows replication of data blocks with both read and write permission. To preserve coherence, updates of each copy must distribute to all other copies on remote sites, by multicast or broadcast messages. Because this algorithm tries to minimize the cost of write access, it is appropriate for write sharing and often serves with write-update protocols. This algorithm can produce high coherence traffic, especially when update frequency and the number of replicated copies are high.

Protocols complying to the MRMW algorithm can be complex and demanding. One way to maintain data consistency is to globally sequence the write operations—to implement reliable multicast. When a processor attempts to write to the shared memory, the intended modification goes 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, it verifies the sequence number, and if not correct, it requests a retransmission.

A modification of this algorithm distributes the sequencing task.[11] In this solution, the server managing the master copy of any particular data structure sequences writes to that data structure. Although the system is not sequentially consistent in this case, each particular data structure is maintained consistently.

Avenues for performance improvement

Researchers have dedicated considerable effort to various modifications of the basic algorithms, to improve their behavior and enhance their performance by reducing the amount of data transferred in the system. Most of those ideas were evaluated by simulation studies, and some were implemented on existing prototype systems.

An enhancement of Li's algorithm called the Shrewd algorithm eliminates all unnecessary page transfers with the assistance of the sequence number per copy of a page.[10] On each write fault at a node with an existing read-only copy, the sequence number goes with the request. If this number is the same as the number kept by the owner, the requester can access the page without its transfer. This solution shows remarkable benefits when the read-to-write ratio increases.

All of Li and Hudak's solutions assume that the page transfer executes on each attempt to access a page that does not reside on the accessing site. One modification employs a competitive algorithm and allows page replication only when the number of accesses to the remote page exceeds the replication cost.[9] A similar rule applies to migration, although because only one site can have the page in this case, the condition to migrate the page is more restrictive and dependent on other sites' access pattern to the same page. The performance of these policies is guaranteed to stay within a constant factor from the optimal.

The Mirage system applies another restriction to data transfer requests, which reduces thrashing—an adverse effect occurring 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 defines a time window Δ in which the site is guaranteed to uninterruptedly possess the page after acquiring it. Users can tune the value of Δ statically or dynamically, depending on the degree of processor locality exhibited by the particular application.

A variety of specific algorithms have been 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 in the distribution of management responsibilities. To improve performance, recent DSM implementations relax memory consistency semantics, which requires considerable modification of the algorithms and organization of directory information. Implementations of critical operations using hardware accelerators and a combination of invalidate and update methods also improve modern DSM system performance.

Implementation level of the DSM mechanism

The level where the DSM mechanism is implemented is one of the most important decisions in building a DSM system: it affects both programming and overall system performance and cost.

To achieve ease of programming, cost-effectiveness, and scalability, DSM systems logically implement the shared-memory model on physically distributed memories.[12,14] Because the DSM's shared address space distributes across local memories, a lookup must execute on each access to these data, to determine if the requested data is in the local memory. If not, the system must bring the data to the local memory. The system must also take an action on write accesses to preserve the coherence of shared data. Both lookup and action can execute in software, hardware, or the combination of both. According to this property, systems fall into three groups: software, hardware, and hybrid implementations.

The choice of implementation usually depends on price/performance trade-offs. Although typically superior in performance, hardware implementations require additional complexity, which only high-performance, large-scale machines can afford. Low-end systems, such as networks of personal computers, based on commodity microprocessors, still do not tolerate the cost of additional hardware for DSM, which limits them to software implementation. For mid-range systems, such as clusters of workstations, low-cost additional hardware, typically used in hybrid solutions, seems appropriate.

Software DSM implementations

Until the last decade, distributed systems widely employed message-passing communication. However, this appeared to be much less convenient than the shared-memory programming model because the programmer must be aware of data distribution and explicitly manage data exchange via messages. In addition,

Software DSM implementations

These fall into user-level, OS, and programming language variations.

User-level and combined software implementations

Table A and the following paragraphs summarize these implementations: IVY, Mermaid, Munin, TreadMarks, Midway, Blizzard, Mirage, Clouds, Orca, and Linda.

IVY

IVY[1] was one of the first proposed software DSM solutions, implemented as a set of user-level modules built on the top of the modified Aegis OS on the Apollo Domain workstations. IVY contains 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 OS 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 are 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 compared to the best sequential solutions for typical parallel programs. Although IVY's performance could have been improved by implementing it on the system level rather than on the user level, its most important contribution is in proving the viability of the DSM concept on real systems with parallel applications.

Mermaid

An algorithm similar to IVY's also serves in Mermaid[2]—the first system to provide DSM in a heterogeneous environment (HDSM). The prototype configuration includes Sun/Unix workstations and DEC Firefly multiprocessors. The DSM mechanism is implemented at the user level, as a library package for linking to the application programs. Minor changes to the SunOS OS kernel include setting the access permission of memory pages from the user level, as well as passing the address of a DSM

Table A. Software DSM implementations.

Implementation	Type of implementation	Type of algorithm	Consistency model	Granularity unit	Coherence policy
IVY	User-level library + OS modification	MRSW	Sequential	1 Kbyte	Invalidate
Mermaid	User-level library + OS modifications	MRSW	Sequential	1 Kbyte, 8 Kbytes	Invalidate
Munin	Runtime system + linker + library + preprocessor + OS modifications	Type-specific (SRSW, MRSW, MRMW)	Release	Variable size objects	Type-specific (delayed update, invalidate)
Midway	Runtime system + compiler	MRMW	Entry, release, processor	4 Kbytes	Update
TreadMarks	User-level	MRMW	Lazy release	4 Kbytes	Update, invalidate
Blizzard	User-level + OS kernel modification	MRSW	Sequential	32-128 bytes	Invalidate
Mirage	OS kernel	MRSW	Sequential	512 bytes	Invalidate
Clouds	OS, out of kernel	MRSW	Inconsistent, sequential	8 Kbytes	Discard segment when unlocked
Linda	Language	MRSW	Sequential	Variable (tuple size)	Implementation-dependent
Orca	Language	MRSW	Synchronization dependent	Shared data object size	Update

such systems 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 the top of message passing emerged in the mid-eighties. Generally, this can be achieved in user-level, run-time library routines, the OS, or a 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, because DSM management is usually supported through virtual memory. Thus, if the requested data is absent 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. But, parallel programs characterized with fine-grain sharing are adversely affected, because of false sharing and thrashing.

Table B. Munin's type-specific memory coherence.

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

page to its user-level fault handler. Because of the heterogeneity of clusters, in addition to data exchange, the need for data conversion arises.

For user-defined data types, besides the conversion of standard data types, the user must provide conversion routines and a table mapping data types to particular routines. Just one data type is allowed per page. Mermaid ensures the variable page size suited to data access patterns. Because Firefly is a shared-memory multiprocessor, it allows comparisons of physical versus distributed shared memory, which shows that the speedup increases less then 20% when moving from DSM to physically shared memory for up to four nodes. Because the conversion costs are substantially lower than page-transfer costs, the introduced overhead caused by heterogeneity is acceptably low—the page fault delay for the heterogeneous system and the homogeneous system with only Firefly multiprocessors is very comparable.

Munin

The Munin[3] 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 Unix system V kernel and the Presto parallel-programming environment. It is a runtime system implementation, although it also requires a preprocessor that converts the program annotations, a modified linker, some library routines, and OS support. It employs different coherence protocols well-suited to the expected access pattern for a shared-data object type (see Table B).

The programmer must provide one of several annotations for each shared object that 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 write sequences. Evaluation using two Munin representative programs (with only minor annotations) shows that their performance is less than 10% worse compared to their carefully hand-coded message-passing counterparts.

TreadMarks

Another DSM implementation that counts on significant data traffic reduction by relaxing consistency semantics according to the lazy release consistency model is TreadMarks.[4] This user-level implementation relies on Unix standard libraries for remote process creation, interprocess communication, and memory management. Therefore, no modifications to the OS kernel or particular compiler are required. TreadMarks runs on commonly available Unix systems. It employs an invalidation-based protocol that allows multiple concurrent writers to modify the page.

On the first write to a shared page, DSM software makes a copy (*twin*) for later comparison with the current copy of the page to make a *diff*—a record containing 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 the next acquire to get better performance. Experiments using DECstation-5000/240's connected by a 100-Mbps ATM network and a 10-Mbps Ethernet reported good speedups for five Splash programs.[5] Experimental results show more efficient communication interfaces can overcome latency and bandwidth limitations, thus further narrowing the gap between software DSM systems and supercomputers.

Midway

Unlike Munin, which uses various coherence protocols on a type-specific basis to implement a single consistency model (release consistency), Midway supports multiple consistency models (processor, release, and entry) that can change dynamically in the same program.[6] Midway operates 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 are done with appropriate explicit synchronization accesses, sequential consistency is also available.

(Continued on page 70)

Software support for DSM is generally more flexible than hardware support and enables better tailoring of the consistency mechanisms to the application behavior. However, it usually cannot compete with hardware implementations in performance. Apart from introducing hardware accelerators to solve the problem, designers also concentrate on relaxing the consistency model, although this can put an additional burden on the programmer. Because research can rely on widely available programming languages and OSs on the networks of workstations, numerous implementations of software DSM have emerged.

The "Software DSM implementations" sidebar describes some of the better-known representations.

Hardware-level DSM implementations

Hardware-implemented DSM mechanisms ensure automatic replication of shared data in local memories and processor caches, transparently for software layers. This approach efficiently supports fine-grain sharing.

(Continued from page 69)

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 time stamps, which reset when data is modified. For all consistency models, Midway uses an update mechanism. Although less efficient with an Ethernet connection, Midway shows close-to-linear speedups of chosen applications when using an ATM network.

Blizzard

Another user-level DSM implementation that also requires some modifications to the OS kernel, Blizzard uses Tempest—a user-level communication and memory interface that provides mechanisms necessary for both fine-grained shared memory and message passing.[7] It comes in three variants: BlizzardS, Blizzard-E, and Blizzard-ES. Blizzard-S, an entirely software variant, is essentially the modification of executable code by the insertion of 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 invokes.

Blizzard-E uses the machine's memory error correction code bits to indicate the block's invalid state by forcing uncorrectable errors. However, this version maintains the read-only state by enforcing read-only protection on the page level by the memory-management unit. Otherwise, it assumes read-write permission.

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 hardware DSM implementation with fine-grain access control—the KSR1 multiprocessor—Blizzard's typical slowdown is several times, depending on the application.

OPERATING SYSTEM SOFTWARE IMPLEMENTATIONS

Mirage implements coherence maintenance inside the OS kernel.[8] The prototype consists of VAX 11/750s connected by Ethernet network, using the System V interface. Mirage's main contribution is in guaranteeing page ownership for a fixed period of time, called the time window, D. This technique avoids thrashing and better exploits inherent processor locality. The value of D 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.

In Mirage, all requests go to the site of the segment creation, called the library site (see Figure A), where they queue and process sequentially. The Clock site, which provides the page's most recent copy, is either a writer or one of the readers of the requested page, because the writer and the readers cannot process the copies of the same page simultaneously. Performance evaluation on 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 proper choice of the parameter D value.

Clouds, an OS that incorporates software-based DSM management, implements a set of primitives either on top of Unix, or in the context of the object-based OS kernel Ra.[9] Clouds was implemented on Ethernet-connected Sun-3 workstations. The DSM consists of objects composed of segments that have access attributes: read-only, read-write, weak-read, or none. Because the weak-read mode allows the node to get a copy of the page with no guarantee that the page will not be modified during read, memory system behavior of Clouds without any specific restrictions leads to inconsistent DSM. Fetching of segments relies on get and discard operations provided by a DSM controller.

This software module also offers P and V semaphore primitives as separate operations. The DSMC is, therefore, a part of the Clouds OS, 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 also reside on top of Unix, with minor changes caused by the OS dependencies.

The nonstructured, physical unit of replication and coherence is small, typically a cache line. Consequently, hardware DSM mechanisms usually represent an extension of the principles found in cache-coherence schemes of scalable shared-memory architectures. This approach considerably reduces communication requirements, because finer sharing granularities minimize the detrimental effects of false sharing and thrashing. Searching and directory functions implemented in hardware are much faster than with software-level implementations, and memory-access latencies decrease. However, advanced coherence-maintenance and latency-reduction techniques usually complicate design and verification. Therefore, hardware DSM is often used in high-end machines where performance is more important than cost.

See the "Hardware DSM implementations" sidebar for a description of three especially interesting groups of hardware DSM systems.

Hybrid-level DSM implementations

During the evolution of this field, the research community proposed numerous entirely hardware or software implementations of the DSM mechanism. 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; and

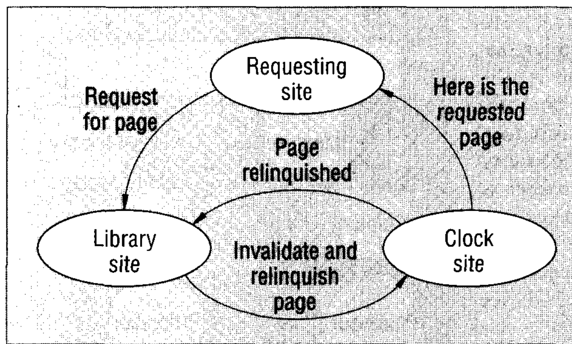

Figure A. Write request for the page in Mirage.

Programming language implementations

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.[10] Linda, an architecture-independent language, provides several special language operators for dealing with such distributed data structures, like inserting, removing, and reading tuples. It avoids the consistency problem: a tuple must be removed from the tuple space before an update, and the modified version is reinserted.

By its nature, the Linda environment offers possibilities for process decoupling, transparent communication, and dynamic scheduling. Linda offers replication for problem partitioning. Linda was implemented on shared-memory machines (Encore Multimax, Sequent Balance) as well as on loosely coupled systems (S/Net, Ethernet network of MicroVaxes).

Henri E. Bal and Andrew S. Tannenbaum extensively discuss software DSM implementations.[11] They propose a new model of shared data objects (passive objects accessible through predefined operations), used in Orca language for distributed programming. The distributed implementation relies on selective replication, migration, and an update mechanism. Different variants of the update mechanism are available, depending on the type of communication provided by the underlying distributed system (point-to-point messages, reliable, and unreliable multicast messages). Orca is predominantly intended for application programming.

References

1. K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. Int'l Conf. Parallel Processing*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 94–101.
2. S. Zhou, M. Stumm, and T. McInerney, "Extending Distributed Shared Memory to Heterogeneous Environments," *Proc. 10th Int'l Conf. Distributed Computing Systems*, CS Press, 1990, pp. 30–37.
3. 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, 1991, pp. 152–164.
4. P. Keleher et al., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. Usenix Winter Conf.*, Usenix Assoc., Berkeley, Calif., 1994, pp. 115–132.
5. J.P. Singh, W.-D. Weber, and A. Gupta, "Splash: Stanford Parallel Applications for Shared Memory," Tech. Report CSL-TR-91-469, Stanford Univ., Stanford, Calif., 1991.
6. B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon, "The Midway Distributed Shared Memory System," *Compcon 93*, CS Press, 1993, pp. 528–537.
7. I. Schoinas et al., "Fine-Grain Access Control for Distributed Shared Memory," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 1994, pp. 297–306.
8. B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proc. 14th ACM Symp. Operating System Principles*, ACM Press, 1989, pp. 211–223.
9. 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.
10. S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer*, Vol. 19, No. 8, May 1986, pp. 26–34.
11. H.E. Bal and A.S. Tannenbaum, "Distributed Programming with Shared Data," *Int'l Conf. on Computer Languages*, CS Press, 1988, pp. 82–91.

prefetch and poststore in KSR1. Many purely software solutions, however, require some hardware support—such as virtual memory management hardware in IVY and ECC in Blizzard-E. As to be expected, neither the entirely hardware nor entirely software approach has all the advantages. Therefore, it is quite natural to employ hybrid methods, with predominantly or partially combined hardware and software elements, to balance the cost-to-complexity trade-offs.

The "Hybrid-level implementations" sidebar summarizes some of these tradeoffs.

Memory consistency models

The memory consistency model defines the legal ordering of memory references issued by a processor, as observed by other processors.[2,14] Different types of parallel applications inherently require various consistency models. The model's restrictiveness largely influences system performance in executing these applications. Stronger forms of the consistency model typically increase memory access latency and bandwidth requirements, while simplifying programming. Looser constraints in more relaxed models, which allow memory reordering, pipelining, and overlapping, consequently improve performance, at the expense of higher programmer involvement in synchronizing shared data accesses. For optimal behavior, systems with multiple consistency models adaptively

Hardware DSM implementations

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

- cache coherent nonuniform memory architectures (CC-NUMA),
- cache-only memory architectures (COMA), and
- reflective memory system (RMS).

CC-NUMA DSM SYSTEMS

A CC-NUMA system (see Figure B) statically distributes the shared virtual address space across local memories of clusters, which both local processors and processors from other clusters in the system can access, 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, to maximize the frequency of local accesses. Performance indicators also depend highly on the interconnection topology. The invalidation mechanism is typically applied to provide consistency, while some relaxed memory consistency model can serve as a source of performance improvement. Typical representatives of this approach are Memnet, Dash, and SCI (see Table C.)

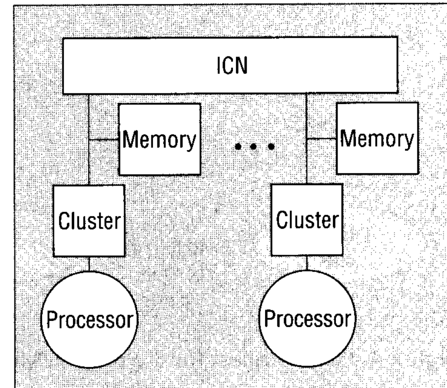

Figure B. CC-NUMA memory architecture.

Memnet

This ring-based multiprocessor—Memory as Network Abstraction—was one of the earliest hardware DSM systems.[1] The main goal was to avoid costly interprocessor communication via messages and to provide an abstraction of shared memory to an application directly by the network, without kernel OS intervention. The Memnet address space maps onto the local memories of each cluster (the reserved area) in a 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 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, which circulates on the ring. Each Memnet device on the ring inspects the message in a snooping manner. The nearest cluster with a valid copy satisfies the request by inserting requested data in the message before forwarding. The write request to a nonexclusive copy results in a message

Table C. Hardware DSM implementations.

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

applied to appropriate data types have recently emerged.

Stronger memory consistency models that treat synchronization accesses as ordinary read and write operations are *sequential* and *processor* consistency. More relaxed models that distinguish between ordinary and synchronization accesses are *weak*, *release*, *lazy release*, and *entry* consistency.

Sequential consistency mandates that all a system's processors observe the same interleaving of reads and writes issued in sequences by individual processors. A simple implementation of this model is a single-port

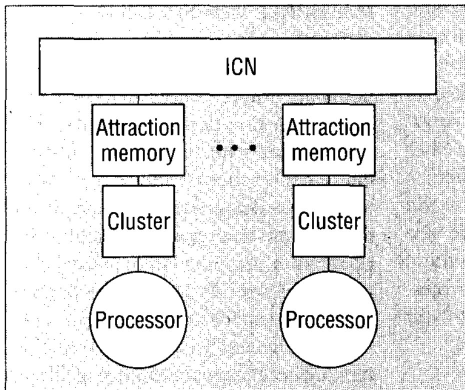

Figure C. COMA memory architecture.

that invalidates other shared copes as it passes through each Memnet device having a valid copy of that block. Finally, the interface of the cluster that generated the message receives and removes it from the ring.

Dash

Directory Architecture for Shared Memory, a scalable cluster multiprocessor architecture using a directory-based hardware DSM mechanism.[2] Each 4-processor cluster contains an equal part of the overal 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:

- processor cache,
- caches of other processors in the local cluster,
- home cluster (the cluster that contains directory and physical memory for a given memory block),
- remote cluster (the 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), or dirty (modified in some remote cluster). Usually, because of the property of locality, references can be satisfied in the local cluster. Otherwise, a request goes to the home cluster for the involved block, which takes some action according to the state found in its directory. A relaxed memory consistency model—release consistency—improves performance, as do memory-access optimizations. Techniques for reducing memory latency, such as software controlled prefetching, update and deliver operations, also improve performance. Dash provides hardware support for synchronization.

SCI

Memory organization in an Scalable Coherent Interface-based CC-NUMA DSM system is similar to Dash, and data from remote memories can be cached in local caches. However, the IEEE P1596 SCI represents an interface standard, rather than a complete system design.[3] Among other issues, it defines a scalable directory cache-coherence protocol. Instead of centralizing the directory, SCI distributes it among those caches currently sharing the data, in the form of doubly linked lists. The directory entry is a shared data structure that multiple processors may concurrently access. 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 goes back to the requester along with the data block (if available). The requester uses it 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 a write to a nonexclusive block, the request for the ownership also goes 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, and support for queue-based locks.

COMA DSM SYSTEMS

The COMA architecture (see Figure C) uses local memories of the clusters as huge caches for data blocks from virtual shared address spaces (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 and caches adapts dynamically to the application's behavior.

COMA architectures have hierarchical network topologies that simplify two main problems in this type of system: locating a data block and replacement. They are less sensitive to static distribution of data than are NUMA archi-

(Continued on page 74)

shared-memory system that enforces serialized access servicing from a single first-in, first-out (FIFO) queue. DSM systems achieve a similar implementation by serializing all requests on a central server node. Neither case allows bypassing of read and write requests from the same processor. 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 different processors can see memory operations need not be identical, but all processors must observe the

(Continued from page 73)

tectures. Because of their cache organization, attraction memories efficiently reduce capacity and conflict miss rates. But, the hierarchical structure imposes slightly higher communication and remote-miss latencies. Somewhat increased storage overhead for keeping the information typical for cache memory is also inherent to the COMA architecture. The two most relevant representatives of COMA systems are the KSR1 and DDM.

KSR1

The KSR1 multiprocessor represents one of the early attempts to make DSM systems available on the market.[4] 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, which is organized as a hierarchy with directories on intermediate levels. This 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 in a particular local cache are exclusive (only valid copy), nonexclusive (owner; multiple copies exist), copy (nonowner; valid copy), and invalid (not valid, but allocated subpage). Besides these usual states, KSR1 provides the atomic state for synchronization. Locking and unlocking the subpage are achieved with special instructions. As in all architectures with no main memory where all data are stored in the 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 cause longer latencies.

DDM

The Data Diffusion Machine prototype is made of 4-processor clusters with an attraction memory and an asynchronous split-transaction bus.[5] Attaching a directory on top of the local DDM bus, to enable its communication with a higher-level bus of the same type, allows a large system with directory- and 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.

A snoopy write-invalidate coherence protocol 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 correspond to invalid, exclusive, and valid, typical for the snoopy protocols. Replacing the dirty state is 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 go 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.

REFLECTIVE MEMORY DSM SYSTEMS

Reflective memory systems have a hardware-implemented update mechanism at a 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 (see Figure D). 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. To keep it updated in a nondemand, anticipatory manner, each write to an address in this shared address space in a cluster propagates through a broadcast or multicast to all other clusters where the same address is mapped into.

The processor does not stall on writes, and computation overlaps with communication. This is a source of performance improvement typical for relaxed memory consistency models. Also, there is no contention and long latencies as in typical shared-memory systems, because unrestricted access to shared data and simultaneous accesses to local copies are ensured. But, all reads from the shared memory are local, with a deterministic access time. The principle of this DSM mechanism closely resembles the write-update cache-coherence protocols. Typical reflective memory systems are RMS and Merlin.

RMS

Several systems with different clusters and network topologies apply reflective memory. Because broadcast is the most appropriate mechanism for updating replicated segments, the shared-bus topology is especially convenient for the

sequence of writes issued by each processor in the same sequence. 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, and RMS.

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 to synchronization accesses. A synchronization access also must wait for all previous accesses to execute, while ordinary reads and writes must wait only for completion of previous synchronization accesses. Sun's Sparc architecture uses a variant of the weak consistency model.

Release consistency further divides synchronization accesses to acquire and release, so that protected ordinary shared accesses can execute between acquire-release pairs. In this model, ordinary read or write access

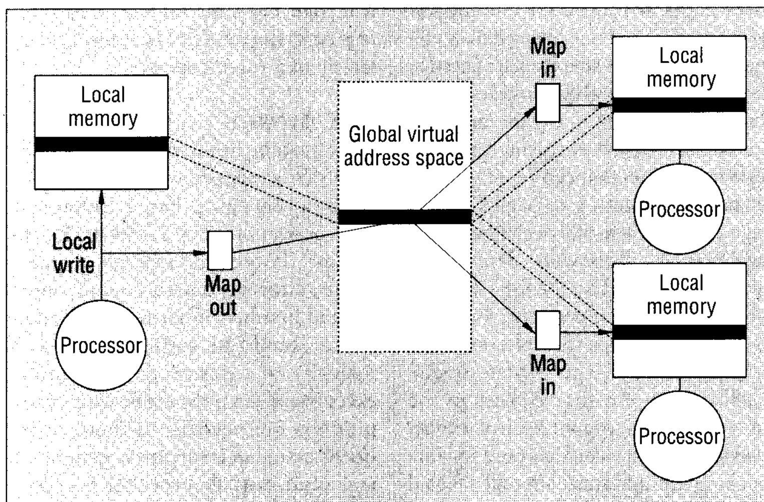

Figure D. Reflective memory DSM architecture.

reflective memory architecture. The Encore Computer Corporation developed a number of bus-based RMSs for a wide range of applications—for example, the Encore Infinity.[6]

These systems typically consist of a lower number of minicomputer clusters connected by the RM bus, a write-only bus because traffic on it only consists of word-based distributed write transfers (address + value of the data word). Later enhancements (Memory Channel) also allow for block-based updates. The replication unit is a 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 map to different addresses in each cluster. Therefore, the translation map tables are provided separately for the transmit (for each block of the local memory) and receive (for each block of the reflected address space) sides.

Merlin

Although very convenient for broadcasting, bus-based systems are notorious for their restricted scalability. Hence, the 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.[7]

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 transmits the word packet through the interconnection network to all shared copies in other local memories. Merlin supports two types of sharing 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.

References

1. G. Delp, D. Farber, and R. Minnich, "Memory as a Network Abstraction," *IEEE Network*, July 1991, pp. 34–41.
2. D. Lenoski et al., "The Stanford DASH Multiprocessor," *Computer*, Vol. 25, No. 3, Mar. 1992, pp. 63–79.
3. D.V. James, "The Scalable Coherent Interface: Scaling to High-Performance Systems," *Compcon 94*, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 64–71.
4. S. Frank et al. "The KSR1: Bridging the Gap Between Shared Memory and MPPs," *Compcon 93*, CS Press, 1993, pp. 285–294.
5. E. Hagersten, A. Landin, and S. Haridi, "DDM—A Cache-Only Memory Architecture," *Computer*, Vol. 25, No. 9, Sept. 1992, pp. 44–54.
6. S. Lucci et al., "Reflective-Memory Multiprocessor," *Proc. 28th IEEE/ACM Hawaii Int'l Conf. System Sciences*, CS Press, 1995, pp. 85–94.
7. C. Maples and L. Wittie, "Merlin: A Superglue for Multicomputer Systems," *Compcon 90*, CS Press, 1990, pp. 73–81.

can execute only after all previous acquires on the same processor execute. In addition, a release can execute only after all previous ordinary reads and writes on the same processor execute. Finally, acquire and release synchronization accesses must fulfill the requirements that processor consistency puts on ordinary read and write accesses. The Dash and Munin DSM systems exhibit different implementations of release consistency.

Lazy release consistency is an enhancement of release consistency.[15] Instead of propagating modifications to the shared address space on each release (like in release consistency—sometimes called *eager consistency*), modifications wait until the next relevant acquire. Also, not all modifications must propagate on the acquire, but only those associated with the chain of preceding synchronization operations on that specific lock. This minimizes the amount of data exchanged, while also reducing the number of messages by combining modification with lock acquires in one message. The Treadmarks DSM system implements lazy release consistency.

Hybrid DSM implementations

Table D provides an overview of the hybrid-level DSM implementations.

Plus

A typical hybrid approach achieves data replication and migration from a shared virtual address space across the clusters in software, while implementing coherence management in hardware. Plus is such a system.[1] In Plus, software handles data placement and replacement in local memories in units of 4-Kbyte pages. However, memory coherence for replicated data resides 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, to ensure the propagation of updates to all copies. Because a relaxed consistency model is assumed, writes are nonblocking, and the *fence* operation is available to the user for explicit strong ordering of writes. To optimize the synchronization, Plus provides a set of specialized interlocked read-modify-write operations called *delayed* operations. Plus hides their latency by splitting them into *issue* and *verify* phases, thus allowing them to proceed concurrently with regular processing.

Galactica Net

Galactica Net solves some DSM issues in a manner similar to Plus. [2] It replicates pages from the virtual address space on demand under control of virtual memory software, implemented in the Mach OS. It also provides hardware support for a virtual memory mechanism, realized through a block transfer engine, that 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 maintained by the OS. A write-update protocol, implemented entirely in hardware, keeps the coherence for writeable shared pages (*update* mode).

All copies of a shared page in the update state 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 each cluster's network interface, which are also maintained by software. Therefore, write references to pages in update state 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. Also, the unit of sharing is quite large, and false sharing effects can degrade performance. Upon recognizing an actual reference pattern, Galactica Net can dynamically switch from the 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 to prevent the unnecessary traffic to unused copies.

MIT Alewife

This system implements the LimitLESS directory protocol, which represents a hardware-based coherence scheme supported by a software mechanism.[3] To reduce storage requirements, directory entries contain only a limited number of hardware pointers, which should be sufficient in a vast majority of cases. Software handles exceptional circumstances, when more pointers are needed. In those infrequent cases, an interrupt is generated, and a full-map directory for the block is emulated in software. A fast-trap mechanism supports this feature, and a multiple-context concept hides the memory latency. This approach's main advantage is that the applied directory coherence protocol is storage-efficient, while performing about as well as the full-map directory protocol.

Flash

Unlike Alewife, the Flash multiprocessor implements the memory coherence protocol in software, but shifts its execution burden from the main processor to an auxiliary protocol processor—Magic (Memory and General Interconnection Controller).[4] This specialized programmable controller efficiently executes protocol actions in a pipelined manner, avoiding context switches on the main processor. Other systems, such as the

Table D. Hybrid-level DSM implementations.

Name	Cluster configuration + network	Type of algorithm	Consistency model	Granularity unit	Coherence policy
Plus	M88000, 32K cache, 8-32M local memory, mesh	MRMW	Processor	4 Kbytes	Update
Galactica Net	4 M88110s, 2-L caches 256M local memory, mesh	MRMW	Multiple	8 Kbytes	Update/ invalidate
Alewife	Sparcle PE, 64K cache, 4M local memory, CMMU, mesh	MRSW	Sequential	16 bytes	Invalidate
Flash	MIPS T5, I+D caches, Magic controller, mesh	MRSW	Release	128 bytes	Invalidate
Typhoon	SuperSparc, 2-L caches, NP controller	MRSW	Custom	32 bytes	Invalidate custom
Shrimp	16 Pentium PC nodes, Intel Paragon routing network	MRMW	AURC, scope	4 Kbytes	Update/ invalidate
Hybrid DSM	Flash-like	MRSW	Release	Variable	Invalidate

network interface processor (NP) in Typhoon,[5] follow this approach, which also ensures great flexibility in experimenting and testing. The NP processor uses a hardware-assisted dispatch mechanism to invoke a user-level procedure to handle an event.

Shrimp

The Shrimp (Scalable High-Performance Really Inexpensive Multiprocessor) multicomputer also uses reflective memory.[6] The virtual memory-mapped network interface implements an automatic update in hardware. In this system, after a page (send buffer) maps out to another page's cluster memory (receive buffer) by the OS, each local write (message) to this page also immediately propagates to the destination automatically by hardware. The *automatic update release consistency* (AURC) approach keeps only one copy of a page consistent using fine-grain automatic updates, while keeping other copies consistent using an invalidation-based software protocol. Another solution for implementing DSM on the Shrimp multicomputer uses the innovative concept of scope consistency, representing a compromise between entry and lazy release consistency.

Other hybrid approaches

To improve the performance, a hybrid approach called *cooperative shared memory* uses programmer-supplied annotations.[7] Programmers identify 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 if misapplied), but reduce unintended communication caused by thrashing and false sharing. Cooperative prefetch can also serve 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.

One hybrid DSM protocol[8] combines 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 must explicitly identify the *regions*—coarse-grain data structures. Then, usage annotations—for example, *BeginRead/EndRead, BeginWrite/EndWrite*—identify program segments that safely reference the data from a certain region (without modification from other processors). Library routines invoked by these annotations maintain coherence of annotated data. A directory-based hardware protocol manages nonannotated data coherence. Both the protocol's software and hardware components use the invalidation policy. A variable-size coherence unit of the software part of the protocol eliminates 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. As in Midway, Munin, and CICO, the main disadvantage is the burden put on the programmer to insert the annotations, although this may not be so complicated because this data-use information is naturally known.

Finally, because message-passing and shared-memory machines have been converging recently, efforts are ongoing to integrate these two communication paradigms in a single system. In addition to the just-discussed coherence protocol, Alewife also allows explicit sending of messages in a shared-memory program. Messages are delivered via an interrupt and are dispatched in software. Besides the Dash-like software-implemented directory cache-coherence protocol, Flash also provides hardware support for message passing with low overhead. Flash gives the user accesses to block transfer without sacrificing protection.

Typhoon is a proposed hardware implementation especially suited for the Tempest interface—a set of user-level mechanisms that can modify the semantics and performance of shared-memory operations. Tempest consists of four types of these 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 a part of the 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.

References

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

2. A. Wilson, R. LaRowe, and M. Teller, "Hardware Assist for Distributed Shared Memory," *Proc. 13th Int'l Conf. on Distributed Computing Systems*, CS Press, 1993, pp. 246–255.

3. D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Software-Extended Coherent Shared Memory: Performance and Cost," *Proc. 21th Ann. Int'l Symp. Computer Architecture*, CS Press, 1994, pp. 314–324.

4. J. Kuskin et al., "The Stanford Flash Multiprocessor," *Proc. 21th Ann. Int'l Symp. Computer Architecture*, CS Press, 1994, pp. 302–313.

5. S. Reinhardt, J. Larus, and D. Wood, "Tempest and Typhoon: User-Level Shared Memory," *Proc. 21th Ann. Int'l Symp. Computer Architecture*, CS Press, 1994, pp. 325–336.

6. L. Iftode, J. Pal Singh, and K. Li, "Scope Consistency: A Bridge Between Release Consistency and Entry Consistency," to be published in *Proc. Eighth Ann. Symp. Parallel Algorithms and Architectures*, CS Press, 1996.

7. M. Hill, J. Larus, and S. Reinhardt, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors," *ACM Trans. Computer Systems*, ACM Press, New York, 1993, pp. 300–318.

8. R. Chandra et al., "Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols," Tech. Report No. CSL-TR-93-597, Stanford Univ., Stanford, Calif., 1993.

Finally, entry consistency also improves 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 of the ordinary shared variable waits until the next acquire of the associated synchronization variable that guards it. Because only the changes for a subset of shared variables protected by the particular synchronization variable must propagate at that moment, the traffic significantly decreases. Latency also falls because a shared access does not have to wait on the completion of other unrelated acquires. Performance improves at the expense of higher programmer involvement in specifying synchronization information for each variable. The Midway DSM system first implemented entry consistency.

Important design choices in building DSM systems

In addition to the DSM algorithm, implementation level of DSM mechanism, and memory consistency model, characteristics that strongly affect overall DSM system performance include cluster configuration, interconnection network, structure of shared data, granularity of coherence unit, responsibility for the DSM management, and coherence policy.

Cluster configuration

Varying greatly across different DSM systems, cluster configuration includes one or several usually off-the-shelf processors. Because each processor has its own local cache (or even cache hierarchy), cache coherence on the cluster level must be integrated globally with the DSM mechanism. 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 important DSM management responsibilities.

Interconnection networks

Almost all types of interconnection networks found in multiprocessors and distributed systems will work in DSM systems. Most software-oriented DSM systems are network independent, although many are built on top of Ethernet, readily available in most environments. But, topologies such as multilevel buses, ring hierarchies, or meshes have served as platforms for hardware-oriented DSM systems. The interconnection network's topology can offer or restrict a good potential for parallel exchange of data related to the DSM management. For the same reasons, it also affects scalability. In addition, it determines the possibility and cost of broadcast and multicast transactions, which is very important for implementing DSM algorithms.

Shared data structure

The 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 software implementations tend to use data items that represent logical entities, to take advantage of the locality naturally expressed by the application.

Coherence unit granularity

The granularity of the coherence unit determines the size of the data blocks managed by coherence protocols. This parameter's affect on the overall system performance relates closely to the locality of data access typical for the application. In general, hardware-oriented systems use smaller units (typically cache blocks), while software solutions, based on virtual memory mechanisms, organize data in larger physical blocks (pages), counting on coarse-grain sharing. In a phenomenon called *false sharing*, the use of larger blocks saves space for directory storage, but 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 can cause thrashing.

DSM management responsibility

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

Coherence policy

The coherence policy determines whether the existing copies of a data item being written to at one site will be updated or invalidated on the other sites. The choice of the coherence policy relates to the granularity of shared data. For very fine-grain data items, an update message

costs approximately the same as an invalidation message. Therefore, systems with word-based coherence maintenance often use the update policy, but coarse-grain systems largely use invalidation. An invalidation approach's efficiency grows when the read and write access sequences to the same data item by various processors are not highly interleaved. The best performance comes when the coherence policy dynamically adapts to observed reference patterns.

Because of the combined advantages of the shared-memory and distributed systems, DSM approaches appear to be a viable solution for large-scale, high-performance systems with a reduced cost of parallel software development. However, efforts to build successful commercial systems that follow the DSM paradigm are still in their infancy, so research prototypes still prevail. Therefore, DSM remains a very active research area. Promising research directions include

- improving DSM algorithms and mechanisms, and adapting them to the characteristics of typical applications and system configurations,
- synergistic combining of hardware and software DSM implementations,
- integrating shared-memory and message-passing programming paradigms,
- creating new and innovative system architectures (especially in the memory system), and
- combining multiple-consistency models.

From this point of view, further investments in exploring, developing, and implementing DSM systems seem to be quite justified.

ACKNOWLEDGMENTS

This work was partly supported by National Science Foundation of Serbia and National Technology Foundation of Serbia. We also want to thank Vojislav Protić for his help in providing up-to-date literature, and Liviu Iftode, who kindly provided some of his most recent papers.

REFERENCES

1. M.J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design,* Jones and Bartlett, Boston, 1995.
2. V. Lo, "Operating Systems Enhancements for Distributed Shared Memory," *Advances in Computers,* Vol. 39, 1994, pp. 191–237.
3. J. Protić, M. Tomašević, and V. Milutinović, "A Survey of Distributed Shared Memory Systems," *Proc. 28th Ann. Hawaii Int'l Conf. System Sciences,* IEEE Computer Society Press, Los Alamitos, Calif., 1995, pp. 74–84.
4. 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.
5. 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.
6. 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,* CS Press, 1995, pp. 272–282.
7. 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.
8. M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer,* Vol. 23, No. 5, May 1990, pp. 54–64.
9. D.L. Black, A. Gupta, and W. Weber, "Competitive Management of Distributed Shared Memory," *Compcon 89,* CS Press, 1989, pp. 184–190.
10. R.E. Kessler and M. Livny, "An Analysis of Distributed Shared Memory Algorithms," *Proc. Ninth Int'l Conf. Distributed Computing Systems,* CS Press, 1989, pp. 498–505.
11. R. Bisiani and A. Forin, "Multilanguage Parallel Programming of Heterogeneous Machines," *IEEE Trans. Computers,* Vol. 37, No. 8, Aug. 1988, pp. 930–945.
12. J. Protić, M. Tomašević, and V. Milutinović, "A Survey of Distributed Shared Memory: Concepts and Systems," Tech. Report No. ETF-TR-95-157, Dept. of Computer Engineering, Univ. of Belgrade, Belgrade, Yugoslavia, 1995.
13. J. Protić, M. Tomašević, and V. Milutinović, "Tutorial on Distributed Shared Memory: Concepts and Systems," CS Press, to be published in 1996.
14. K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* CS Press, 1990, pp. 15–26.
15. P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. 19th Ann. Int'l Symp. Computer Architecture,* CS Press, 1992, pp. 13–21.

Jelica Protić is on the faculty of the School of Electrical Engineering, University of Belgrade, where she received her BSc and MSc in computer engineering. She is currently working toward her PhD in the field of DSM. Her research interests are in computer architecture, distributed systems, and performance analysis. She can be reached at jeca@ubbg.etf.bg.ac.yu.

Milo Tomašević is on the faculty of the School of Electrical Engineering, University of Belgrade, where he received his BSc in electrical engineering and MSc and PhD in computer engineering. His research interests are computer architectures, multiprocessor systems, and distributed shared-memory systems. He can be reached at etomasev@ubbg.etf.bg.ac.yu.

Veljko Milutinović is on the faculty of the School of Electrical Engineering, University of Belgrade, where he received his BSc in electrical engineering and MSc and PhD in computer engineering. He was a coarchitect of one of the first 200-MHz microprocessors, and is active in parallel and distributed processing. He has published over 40 IEEE journal papers. He can be reached at Dalmatinska 55, 11000 Belgrade, Yugoslavia; emilutiv@ubbg.etf.bg.ac.yu.