On the Synchronization Mechanisms in Distributed Shared Memory Systems

Mahendra Ramachandran and Mukesh Singhal

Department of Computer and Information Science
The Ohio State University, Columbus, Ohio 43210-1277
email: {ramach,singhal}@cis.ohio-state.edu

Abstract

Distributed Shared Memory (DSM) is the implementation of the shared memory programming paradigm on a distributed memory (or multicomputer) system. Programming multicomputer systems using Distributed Shared Memory as the programming model is appealing because it combines the performance advantage of distributed memory systems and the ease of programming of shared memory systems. In DSM systems, cooperating tasks communicate with one another through shared variables. Thus, DSM systems must provide synchronization mechanisms to Coordinate concurrent access to these shared variables. In this paper we describe and classify the synchronization mechanisms supported by several Distributed Shared Memory systems. We classify these systems according to whether they are hardware or software based, whether the mechanism is integrated into the system or not and whether implementation of the synchronization mechanism is centralized or distributed.

Key phrases: Distributed Shared Memory, Distributed Memory Architectures, Operating Systems, Synchronization.

1 Introduction

The development of large application software as well as tools such as compilers is considerably more advanced in shared memory multiprocessors than in distributed memory architectures such as hypercubes and meshes. This is primarily due to the ease of programming of the shared memory paradigm. There are comparatively fewer program development tools available for the message passing paradigm. Yet, from the architectural standpoint, distributed memory systems are more attractive because they perform better, are cheaper to build with off-the-shelf components, and are more scalable than shared memory multiprocessors. Thus, it is appealing to provide the shared memory paradigm on distributed memory systems. This has motivated research in Distributed Shared Memory systems (DSM).

The issue of supporting shared memory on a network of processors, each with its own memory unit, is a complex one. It involves providing a single, logically shared, address space while the underlying architecture is composed of multiple, physically distributed, disjoint address spaces. To reduce the data access overhead incurred in fetching data across the network, DSM systems group shared data into larger units such as segments or pages. These pages are then cached onto processors on demand similar to the manner in which pages are brought into main memory in traditional virtual memory systems [19]. However, unlike in virtual memory, the pages in DSM systems are resources that are competed for by concurrent processes executing on different processors.

Program execution efficiency is improved by replicating the shared pages and placing copies of the page on the processors. While this replication allows concurrent local access to the pages by the processes, the copies become outdated when one of the pages is modified by its processor; this problem is similar to the cache-coherence problem in multiprocessors [10, 29]. Maintaining the coherence of shared data in DSM systems has drawn a lot of attention [24]. A coherence maintenance protocol ensures that replicated data is updated in an application transparent manner, however, it is insufficient to ensure the correct execution of a parallel program.

In a concurrent programming environment, it is essential to provide mechanisms through which processes can access shared data for communication and synchronization in a coordinated manner. In shared memory systems, these synchronization mechanisms are based on atomic access to shared variables (e.g., semaphores and monitors). In DSM systems, if the variables used for synchronization were placed in the shared address space as other variables, thrashing of the shared page would result in significant performance degradation.

In this paper, we examine synchronization mechanisms provided by a variety of distributed shared memory systems, both software and hardware based. We classify the synchronization mechanisms supported in existing DSM systems according to three criteria: software versus hardware, centralized versus distributed, and a mechanism that is integrated into the DSM versus one which is not. We review synchronization mechanisms provided in existing DSM systems and categorize the systems according to the developed classification.

The paper is organized as follows: In Section 2, we review basic synchronization mechanisms and present a characterization and classification of the mechanisms supported by various DSMs. In Section 3, we describe the synchronization mechanism implemented in specific systems. We conclude the paper in Section 4.

2 Preliminaries

Most distributed shared memory systems incorporate a variety of synchronization mechanisms whose existence can be traced back to traditional operating systems. For completeness, a brief description of these mechanisms is presented in this section. We then present a taxonomy of synchronization mechanisms in the DSM systems.

2.1 Basic Synchronization Mechanisms

Semaphores

The concept of semaphores was proposed by Dijkstra to solve a variety of synchronization problems, namely, mutual exclusion, K-ary mutual exclusion, and process synchronization in the uniprocessor system [9]. A semaphore is a synchronization variable which is initialized to one for a binary semaphore or greater than one for a resource counting semaphore.

Two operations, P and V, are defined on a semaphore as follows:

```
P(S):  if (S ≥ 1) then
         S := S - 1;
       else
         block the process on
         the semaphore queue;

V(S):  if (semaphore queue is not empty) then
         de-queue a process;
       else
         S := S + 1;
```

Eventcounts

The eventcount synchronization mechanism was proposed by Reed and Kanodia [28] as an alternative to the mutual exclusion type of synchronization provided by mechanisms such as binary semaphores and monitors [18]. An eventcount is a monotonically increasing integer variable. Whenever a process needs to signal the occurrence of an event to the other processes in the system, it does so by incrementing the eventcount variable. To increment the *eventcount* variable, a process executes the *advance(eventcount)* primitive.

A process can observe the current state of the variable *eventcount* by executing the *read(eventcount)* primitive. If a process must wait for the variable *eventcount* to reach a particular value, the process need not repeatedly perform *read* operations in a loop.

Instead, it can block and wait for the variable to reach the specified *value* by executing the *await(eventcount, value)* primitive.

<u>Locks</u>

In locking mechanism, a lock is associated with every shared resource. A process gets mutually exclusive access to a resource by acquiring the lock on the resource. This is done by performing an *acquire* operation on the particular lock variable. When the process succeeds in acquiring the lock, it is the sole possessor of the resource until it relinquishes the lock. This is done by performing a *release* operation. When a lock is released, any process waiting to acquire the lock is granted.

<u>Barriers</u>

A barrier is a synchronization mechanism that allows multiple processes to synchronize together at certain points during program execution [30]. This allows multiple processes to complete a phase of computation, synchronize to ensure that all other processes have also completed the phase, and then proceed to the next phase of computation. When each process arrives at a barrier, it blocks until all other processes have arrived at the barrier. At that point, all the blocked processes are unblocked and proceed with their computations.

2.2 A Taxonomy of Synchronization Mechanisms in DSM Systems

Synchronization mechanisms provided in existing DSMs can be classified along three criteria.

1. It may be a hardware or a software based solution.

2. A mechanism may be implemented in a centralized or a distributed manner.
3. A mechanism may be integrated into the DSM system or non-integrated.

We next discuss these criteria in detail.

<u>Hardware versus Software</u>

A DSM system can be implemented either in hardware or software. Early DSMs were software based and were implemented on multicomputers whose architecture provided no support for a shared memory environment [19, 23, 27, 3]. Increasingly, support for DSM is being provided in hardware for efficiency [2, 11, 12, 1, 17]. This has enabled the development of distributed memory computers with hardware support for shared memory.

Software implemented systems are designed to provide the DSM on existing distributed memory systems. These systems were implemented either as an entire operating system or as a library of user callable routines and a runtime system which ran on top of the operating system [27, 19]. The libraries provide the primitives supported by the DSM and the runtime system implements the mechanisms that access the shared data and perform synchronization. While such systems serve as valuable testbeds to validate a particular mechanism, the overhead incurred is high. This overhead can be reduced by incorporating these mechanisms in the kernel [22]. The architecture on which software based DSMs are implemented typically provide rudimentary synchronization (e.g., lock-unlock, semaphores or just a hardware Test&Set) on a per-node basis. It is the responsibility of the DSM to provide global synchronization mechanisms to synchronize processes across the network.

Recently, considerable effort has been directed into building large-scale distributed memory multiprocessors with hardware support for shared memory. These DSM systems provide mechanisms for accessing the distributed memory as if it were a single address space. A variety of architectures exist. In some systems, local memory of a processor is used as its cache and the local memories of the rest of the system are viewed as the global memory for that processor [11, 1, 6]. Other systems support a hierarchical view of memory similar to the cache-main-secondary memory hierarchy in traditional operating systems. In these systems, the memory hierarchy is divided into memory that is local to a processor, local to a cluster of processors, and global to the entire network of processors. Most of the systems provide some form of synchronization beyond the traditional per-processor synchronization found in multicomputers. Some systems provide global synchronization and others a restricted or hierarchical synchronization. We will describe these synchronization mechanisms when we describe some representative systems in Section 3.

Centralized versus Distributed

A synchronization mechanism can be implemented either in a centralized or a distributed manner. In centralized schemes, a particular node in the system is defined to be the central manager node and all requests for synchronization must be sent to this node. An advantage of a centralized scheme is the ease of implementation of the synchronization mechanism. Unlike in distributed schemes, there is no need to locate the owner of the synchronization variable. However, a centralized scheme will result in a bottleneck at the manager node when the access frequency of synchronization variables is high [25]. It also makes the manager the single point of failure in the system.

It is possible to improve on the centralized scheme by using more than one manager in the system. In this scheme, known as the static (or fixed) distributed scheme, there are multiple managers in the system, each of which manages a subset of the synchronization variables in the system. While this helps reduce the congestion inherent with a single manager, it would degenerate to the performance of the single manager scheme if high access frequencies occur to one or more synchronization variables belonging to one manager. This is because the static distributed manager scheme is still a centralized scheme.

In distributed schemes, the idea of a manager is no longer applicable; instead the node that currently holds the synchronization variable is considered the holder or owner [16]. When the variable is migrated to another node, the ownership of the synchronization variable is transferred to that site as well. Since the ownership of the resource changes dynamically, a requesting site may not know the identity of the current owner of the resource. In order for a site to send its request message to the owner, it may broadcast the message to all the sites in the network, or it may multicast the request to a subset of the sites or it may send the message to the site which it considers to be the owner. Broadcasting has the advantage of being able to deliver the request with just one message hop. However it will generate a lot of network traffic which would have a negative impact on synchronization delay. Multicasting, while not generating excessive message traffic as with a broadcast, may still require additional messages if none of the sites that received the message is the owner. Lastly, by sending a request message to the probable owner, it is possible to get a request granted with the fewest total number of messages generated. However, it is possible that ownership has moved to another site. In such a case the site receiving the request must forward the request message to a more recent owner of the variable until it eventually arrives at the current owner. These techniques have been used in locating other resources (including DSM pages) in distributed systems and are not unique to synchronization variables. Fowler presents a comprehensive exposition on locating objects in a distributed environment [16].

Integrated versus Non-integrated Synchronization Mechanisms

A DSM system supports an integrated synchronization mechanism if the programming model (or the communication model supported by it) defines the synchronization available for user processes. If a a synchronization mechanism is implemented similar to the way other shared data access is implemented on a particular DSM, then the DSM supports an integrated mechanism for synchronization. For instance, if a DSM system supports shared pages as the mechanism through which global data is shared, and it supports semaphores in shared semaphore pages (possibly with different semantics) then we claim that the semaphore mechanism is integrated with this DSM. Another example is a hardware-based DSM which provides caching of shared data and synchronization variables, perhaps with different cache-coherence protocols, provides an integrated synchronization mechanism.

A non-integrated synchronization mechanism is one whose choice was not influenced by the programming model, or the by manner in which other shared data is implemented by the DSM system. One synchronization mechanism can be replaced with another with little or no effect on the programming model or there is no resemblance between the programming environment and the synchronization mechanism. For instance, suppose the DSM which supports shared pages described above provides static distributed semaphores, then there is a clear distinction between shared data and the semaphore variables supported by this DSM. Another example of a non-integrated mechanism is a hardware-based DSM like the one described above which allocates all semaphore variables to one processor and all **P** and **V** operations are executed by this processor in a centralized manner.

3 Synchronization Schemes in Existing DSM Systems

In this section, we discuss synchronization schemes in a set of representative DSM systems. We first describe synchronization schemes for software-based systems followed those for hardware-based systems. For each system, a brief description of the system is first presented, followed by the synchronization mechanism supported by the system.

3.1 Software Implementations

Software based DSM implementations are available on a variety of platforms. Some systems have been built on mini-computers connected by Ethernet; Mirage, for instance, is built on a network of Vax 11/750 computers [15]. Many others are built on a network of workstations or even personal computers, e.g., Mether and Mirage+ [23, 14]. Since most software implementations are academic research projects, the choice of the platforms have been dictated by the existing infrastructure. Nevertheless, the performance results reported give us insight on the viability of DSM systems on such platforms. A few systems are built on existing multicomputers such as the Intel iPSC/2 hypercube [20].

<u>IVY</u>

The IVY system, one of the early DSM systems, is implemented on a network of Apollo workstations connected via Ethernet[21]. IVY supports shared pages which are replicated on the nodes of the system to reduce latency. Strict consistency is used to maintain the coherence of the replicated pages [19].

In the IVY system, eventcounts are used for synchronization. The choice was motivated by the fact that the Aegis operating system, on top of which IVY was implemented, already provided eventcounts. In an earlier version of IVY, IVY I, eventcounts were implemented using remote procedure calls. All eventcount variables were placed in the local (non-shared) memory of a processor and operations on eventcount variables were sent as remote calls to this processor. A variation to this scheme is a static distributed technique [21]. The RPC based implementation is essentially a centralized scheme which is not integrated with the programming model supported by IVY.

IVY II supports an integrated synchronization mechanism where the eventcount variables were located in the shared pages of the shared virtual memory space. It is a dynamic distributed scheme; the pages containing the eventcount variables migrate among the processors where they are used. To ensure the atomicity of the eventcount operations, the shared page containing the eventcount variables is kept on a processor by locking it (referred to as "wiring" the page [21]) during the operation. Li claims that this mechanism not only has cleaner semantics than the RPC version, but also is more efficient than the RPC implementation of IVY I when there are multiple processes per processor. Since the page containing the eventcount is cached on the processor where it is accessed, the accesses by the multiple processes would be local to that processor, thus yielding better performance. However, if different eventcount variables on the same shared page are frequently accessed on different processors, thrashing of the page may result due to false sharing.

Clouds

Clouds is a DSM system developed at Georgia Tech on a network of workstation connected via Ethernet [26]. It supports an object-based model of programming and allows for the migration of objects and other shared data to the processors where they are needed. The object model of computation integrates the synchronization mechanism by providing atomicity in the form of remote procedure calls. An object in Clouds is composed of shared-data segments which can be migrated among processors as well. Clouds ensures strict consistency of these segments by combining the locking and unlocking the data segment with the accessing and releasing operations, respectively.

In order to support inter-process synchronization that is independent of the RPC mechanism, semaphores are supported in the kernel of the Clouds system [27]. The kernel, known as Ra, groups the semaphores into semaphore segments which reside on a processor that created them. The P and V operations are performed on that processor in a centralized manner. The semaphore segment also provides linked-lists to queue blocked P requests. An alternative, proposed in [27], is to move the entire segment of semaphores to the requesting site. This alternative would improve the performance for semaphore access if multiple processes reside on a processor; however, false sharing of the semaphore segment by processes on different processors can increase overheads.

Munin and TreadMarks

The Munin system, developed at Rice University, has been implemented on a network of workstations interconnected via Ethernet [4]. The system consists of a runtime system which interacts with a modified version of the **V** kernel operating system [5]. Unlike IVY and Clouds, Munin's design philosophy is that different shared data require varying degrees of consistency and the performance can be improved by using a spectrum of consistency maintenance protocols for these shared variables. In order to do so, Munin defines nine shared-data types, each with it own coherence protocol [3]. Munin supports an integrated synchronization mechanism; one of the shared-data types is the synchronization type. This data type is implemented as a distributed lock. Aside from locks, centralized implementations of barriers and conditional variables are also provided in Munin [4]. A condition variable makes it possible to implement synchronization based on Monitors [18]. Processes block by executing a *wait* primitive on Condition variable.

When a process executes the *signal* primitive, one of the blocked processes is unblocked and allowed to proceed. The definition of the *signal* primitive in Munin is different; all the blocked processes are unblocked together.

Locks are implemented using a variation of the dynamic distributed scheme. When a processor needs the lock, it sends a request to the lock's owner. The lock owner may have passed the lock to another processor, in which case it will forward the request message to that processor. This forwarding will result in the request eventually arriving at the current owner of the lock. When the current owner receives the request, it either grants the lock if it is free or queues the request if the lock is currently in use. If the owner already has another request queued for the lock, it sends the request to that site to be queued there. In this manner, a distributed queue is created on the network with every waiting processor queuing the processor that is to receive the lock after it.

TreadMarks, also developed at Rice University, supports locks and barriers for synchronization in a manner similar to Munin [7]. Unlike Munin, however, this system uses a single memory consistency model called lazy release consistency [7], which allows differing the propagation of data updates (or invalidations) until processes synchronize. In order to facilitate this delayed propagation, the lock request must be timestamped by the sending processor. This timestamp is used by the lock holder to determine which data element's updates/invalidates must be forwarded along with the lock. This synchronization mechanism is unusual in that the coherence maintenance protocol is combined into the synchronization mechanism and not vice-versa.

Mirage

The Mirage system is implemented in the kernel of a modified Locus operating system on a group of Vax 11/750 computers connected by Ethernet [15]. More recently, a modified version of the system, Mirage+, has been ported to a network of personal computers running AIX [14]. Mirage and Mirage+ support paged-segmentation where the unit of data migration or distribution is the page. The coherence protocol used is similar to IVY with one important distinction, the use of a variable parameter Δ to reduce the effects of thrashing. When a page is cached on a processor, it is not swapped out of the processor until Δ time units have expired.

Synchronization in Mirage is provided as semaphores, implemented in the Locus kernel [13]. This mechanism, is integrated into the Locus kernel. The Locus system was designed to provide reliability in a distributed system. The Locus semaphore mechanism is a fault-tolerant to processor failures. The semaphores are grouped into *semaphore sets*. Operations are performed atomically on an entire semaphore set in a centralized manner by the manager of the set. The manager also maintains *undo logs* for each process that performs operations on a semaphore set. If a process is aborted or the site where it resides crashes, the manager uses the undo log to clean up any half-finished semaphore operations.

Shiva

Shiva, a DSM with a coherence protocol similar to IVY, has been developed on the Intel iPSC/2 hypercube multicomputer [20]. The prototype was implemented on NX/2, the minimal operating system executed on each node of the iPSC/2.

The system provides binary semaphores. These semaphores are accessed by using send/receive primitives to the central semaphore controller. It is also possible to implement the mechanism using the dynamic distributed technique to cache the semaphore on the processor performing P and V operations [20]. Since only the semaphore being accessed is cached, this implementation avoids the false sharing problem inherent in mechanisms which cache a group of semaphores. On the other hand, this mechanism is not integrated into the DSM system.

3.2 Hardware Solutions

Early hardware implementations of DSM consisted of a group processors or workstations connected via a network with a specially designed network interface. For instance, MemNet was composed of processors connected to a token-ring by an interface device called a MemNet device [8]. This device was responsible for performing remote access requests on behalf of its processor as well as servicing the remote requests of other processors in the system.

More recent implementations encompass processor networks where the nodes themselves are clusters or multiprocessors. The DASH system, for instance, is a two dimensional mesh where the nodes are Silicon Graphics 4D/240 bus-based shared memory multiprocessors [11]. The network consists of a pair of meshes, one for sending remote requests and the other to receive replies. DASH uses a directory-based scheme to maintain the coherence of shared data. This scheme ensures that updates to synchronization variables (locks) are propagated to a cluster of nodes which has cached the lock variable when it is released. Dash also provides the *fetch and increment* and *fetch and decrement* atomic operations. These are performed at the memory locations of the synchronization variables without caching them on the processors. Thus, Dash supports an integrated, dynamic distributed mechanism as well as a non-integrated, static distributed mechanism.

Some systems implement the synchronization mechanism in software even though the data consistency is maintained in hardware. For instance, the FLASH system, which is designed to test a variety of coherence maintenance protocols [12], implements synchronization primitives at the operating system level.

We next describe a few representative systems.

Plus

The Plus system, a hardware implementation of a DSM, is composed of a group of Plus nodes, connected via a "fast interconnection network" [2]. Each Plus node consists of a processor with its cache, memory (for local and replicated global data), and a Memory Coherence Manager (MCM) connected by a bus. Shared data in Plus are grouped into pages, which are replicated on the nodes where they are required. The MCM on each node ensures that when a data element is written to by the processor, the update is propagated to all the other nodes which have cached that page. A subsequent read on that data element by the processor is blocked until its MCM receives confirmation that the propagation of updates has completed. However, writes to other data elements can proceed concurrently. Sequential consistency, if necessary, is ensured by using the user callable *write-fence* operation to inhibit concurrent writes from the same processor.

Synchronization in the Plus system is through semaphores. In keeping with the design philosophy of distinguishing between the issuing of updates and the reading of the updated data location, Plus supports the concept of "*delayed operations*". In this approach, when a processor issues a **P** operation, it need not block and wait for the operation to complete; instead, it can perform other computations that do not depend on the synchronization request. The processor can read the result of the synchronization operation after a certain delay from its local memory (where it would be cached by the MCM). This is similar to the delayed branch technique used in programming RISC architectures. Judicious placement of the synchronization request and read operations in the program allows for concurrency in the computation and synchronization. This helps reduce the effect of the synchronization delay on the performance of the program. Since the synchronization operations are performed at the node where the synchronization variable is located, this mechanism uses a static distributed scheme.

Willow

The architecture of Willow is tree-based hierarchical multiprocessor [1]. All intermediate nodes are memory modules and each leaf node of the tree is a processor module. Both types of modules (memory and processor) have caches associated with them. The connection between a node and its children is through a shared bus. Thus at the lowest level are groups of bus-based shared memory multiprocessors with four processors and a memory module.

Synchronization is provided in the form of *Conditional Test&Set* operations. Every cache in the system has a separate unit which is reserved for synchronization variables. When a processor performs the *Conditional Test&Set* operation on a synchronization variable in its cache, the *set* phase of the operation attempts to write through to all replica of the synchronization variable. The operation is completed only after the successful completion of this phase. If multiple processors attempt to perform the operation on the same variable, the one that acquires the shared bus at the highest level of the tree common to them succeeds in completing the operation. The other processors invalidate the result of the conditional test phase of the operation and wait for the synchronization variable's value to be reset to zero. The ideal case in this scheme is when multiple processors accessing a synchronization variable belong to the same multiprocessor since this would require the use of the bus in that multiprocessor. In the worst case, if two processors at the two extreme leaves of the tree contend for the same variable, the synchronization must be handled by the bus at the root of the tree. This scheme is a distributed one which has been integrated into the hardware of the system.

Convex Exemplar

The Exemplar architecture, developed by Convex Corp. [6], is a hierarchical distributed memory architecture. It is composed of clusters called *hypernodes*. Each hypernode has eight processors and shared memory connected via a cross-bar. Each processor has a private portion of that shared memory for its own exclusive use as well. The hypernodes are connected into a two-dimensional torus network. A part of each hypernode's memory, the global memory, is accessible to other hypernodes.

Hardware synchronization is provided as semaphores and barriers. To perform these operations a variety of memory access instructions are provided, namely, *fetch*, *fetch and clear*, *fetch and increment*, *fetch and decrement*, and *load and clear*. The first four instructions are performed atomically at the semaphore variable's memory location while the last one results in caching the semaphore variable on the requesting processor. All the fetch instructions return the current value of the semaphore variable to the requesting processor. While the first primitive does not alter the value of the variable, the remaining three will clear it, increment it by one, or decrement it by one, respectively. The *load and clear* instruction caches the variable on the processor and clears the variable's memory location. This enables subsequent accesses by the processor to be performed locally without network traffic. However, if another processor tries to access the variable, the cached variable must be flushed. Implementation of the P operation using the various *fetch and Op* primitives would result in the processor repeatedly testing the status of the variable and thus causing excessive traffic on the network. This is essentially a static distributed scheme since all accesses are directed to the synchronization variable's memory location.

4 Discussion and Conclusion

In distributed shared memory programming model, cooperating processes access shared variables, and coordinating this access is important for the correct execution of a parallel program. Synchronization mechanisms provide the support for this coordination. Synchronization mechanisms are supported by all DSM systems, irrespective of the programming model and the coherence protocol used.

In this paper we examined the synchronization mechanisms in a variety of DSM systems. We classified the mechanisms according to whether the mechanism is implemented in hardware or software, whether the mechanism uses a centralized or distributed scheme, and lastly whether the mechanism is integrated into the DSM system or not. We summarize the classification of the synchronization mechanisms in Table 1.

In software based systems, the synchronization mechanism is implemented either by the runtime system or the kernel. It is implemented either in a centralized, static distributed or dynamically distributed manner. If it is dynamically distributed, only one copy of the synchronization variable is maintained and this copy is migrated (like a token) among the processors in the system.

In hardware based DSM systems, the mechanism is typically implemented with hardware support; a notable exception is the FLASH system in which it is implemented by the operating system kernel. The distributed schemes in such systems take advantage of the hardware support to cache the synchronization variable at the processors and rely on cache-coherence protocols to maintain the consistency of the replicated copies. Some hardware based systems implement the mechanism in a static distributed manner to eliminate the need for maintaining multiple cache-coherence protocols (one for synchronization variables and another for other shared data). Such systems use *fetch&Op* type

Table 1: A Summary of Synchronization Mechanisms

DSM System	Implementation	Mechanism	Integration
Clouds	Software	Centralized	Both
Dash	hardware	Static Distributed	Non-Integrated
Exemplar	hardware	Static Distributed	Non-Integrated
Flash	hardware (software mech.)		
IVY	Software	Centralized & Distributed	Non-integrated
Mirage	Software	Centralized	Non-integrated
Munin	Software	Distributed	Integrated
Plus	hardware	Static & Dynamic Distributed	both
Shiva	Software	Distributed	Non-integrated
TreadMarks	Software	Distributed	Integrated
Willow	hardware	Static Distributed	Integrated

of operations to reduce the number of messages generated by processors accessing the synchronization variable.

Future work in synchronization mechanisms in DSM systems include, a comprehensive performance study of the various synchronization mechanisms, synchronization mechanisms for DSM systems in hetrogenous environments, and reliable synchronization mechanisms for DSM systems that provide fault-tolerance.

References

[1] J. Bennett, S. Dwarkadas, J. Greenwood, and E. Speight. “Willow: A Scalable Shared Memory Multiprocessor ”. In *Proceedings of Supercomputing ’92*, pages 336–345, Nov 1992.

[2] R. Bisiani and M. Ravishankar. “Plus: A Distributed Shared-Memory System”. In *International Symp. on Computer Architecture*, pages 115–124, May 1990.

[3] J. Carter, J. Bennett, and W. Zwaenepoel. “Implementation and Performance of Munin”. In *ACM Symp. on Operating System Principles*, pages 152–164, October 1991.

[4] John Carter. *“Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency”*. PhD thesis, Rice University, Sep 1993.

[5] David R. Cheriton and Willy Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[6] Convex Pres, Richardson, TX. *"Convex Exemplar Architecture"*, Nov 1993.

[7] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. "Software Versus Hardware Shared-memory Implementation: A Case Study". In *21st Annual Int. Symp. on Computer Architecture*, pages 106–117, April 1994.

[8] G. Delp, D. Farber, R. Minnich, J. Smith, and I. Tam. "Memory as a Network Abstraction". In T. Cassavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 409–423. IEEE, 1994.

[9] E. W. Dijkstra. "Cooperating Sequential Processes". In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, 1968.

[10] M Dubois, C Scheurich, and F. Briggs. "Synchronization, Coherence and Event Ordering in Multiprocessors". *IEEE Computer*, pages 9–21, February 1988.

[11] D. Lenoski et. al. "The Stanford Dash Multiprocessor". *IEEE Computer*, 25(3):63–79, March 1992.

[12] J. Kuskin et. al. "The Stanford FLASH Multiprocessor". In *International Symp. on Computer Architecture*, pages 302–313, April 1994.

[13] B. Fleisch. "Distributed System V IPC in Locus: A design and Implementation Retrospective". In *Sigcomm '86 Symposium*, pages 386–396, 1986.

[14] B. Fleisch, R. Hyde, and N. Juul. "Mirage+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers". *Software - Practice and Experience*, 1994. To appear.

[15] B. Fleisch and G. Popek. "Mirage: A Coherent Distributed Shared Memory Design". In *Proceedings of the eleventh ACM Symp. on Operating System Principles*, pages 211–223, December 1989.

[16] Robert Fowler. *"Decentralized Object Finding Using Forwarding Addresses"*. PhD thesis, University of Washington, 1985.

[17] G. Hermannsson and L. Wittie. "Optimistic Synchronization in Distributed Shared Memory". In *International Conference on Distributed Computing Systems*, 1994.

[18] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". *Communications of the ACM*, 17(10):549–557, October 1974.

[19] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems". *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[20] K. Li and R. Schaefer. "A Hypercube Shared Virtual Memory System". In *International Conference on Parallel Processing*, pages I125–I132, 1989.

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

[22] V. Lo. "Operating Systems Enhancements for Distributed Shared Memory". In M. Yovits, editor, *Advances in Computers*, pages 191–237. Academic Press, 1994.

[23] R. Minnich and D. Farber. "The Mether System: Distributed Shared Memory for SunOS 4.0 ". In *Proceedings of the Summer 1989 Usenix Conference*, pages 51–60, Summer 1989.

[24] B. Nitzberg and V. Lo. "Distributed Shared Memory: A Survey of Issues and Algorithms". *IEEE Computer*, pages 52–60, August 1991.

[25] M. Ramachandran and M. Singhal. "Decentralized Semaphore Support in a Virtual Shared Memory System". *Journal of Supercomputing*, 1994. To appear.

[26] U. Ramachandran, M. Ahamad, and Y. Khalidi. "Coherence of Distributed Shared Memory Unifying Synchronization and Data Transfer". In *International Conference on Parallel Processing*, pages 160–169, aug 1989.

[27] U. Ramachandran and Y. Khalidi. "An Implementation of Distributed Shared Memory". *Software - Practice and Experience*, 21(5):443–464, May 1991.

[28] D. Reed and R. Kanodia. "Synchronization with Eventcounts and Sequencers". *Communications of the ACM*, Feb 1979.

[29] Per Stenstrom. "A Surevey of Cache Coherence Schemes for Multiprocessors". *IEEE Computers*, 23(6):12–24, June 1990.

[30] P. Tang and P.-C. Yew. "Processor Self-scheduling for Multiple-nested Parallel Loops". In *International Conference on Parallel Processing*, pages 528–535, 1986.