and non-atomically. Hutto and Ahamad propose the use of weakly consistent memories in a DSM system to minimize the effect of network latency and to improve scalability.

John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, volume 25, pages 168–176, March 1990.

Phillip W. Hutto and Mustaque Ahamad. Weakening consistency in distributed shared memories. In *10th International Conference on Distributed Computing Systems*, pages 302–309, May 1990.

Richard J. Lipton and Jonathan S. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Sep 1988.

Kung-Lung Wu and W. Kent Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

The following three readings relate to programming in distributed systems based on the shared memory paradigm.

D. Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

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

Barbara H. Liskov and R. W. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, pages 381–404, July 1983.

The paper by Goodman, *et al.*, proposes synchronization mechanisms in SMMs, and the one by Gharachorloo, *et al.*, suggests ways to weaken the consistency requirements in SMMs.

K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *The 17th Annual International Symposium on Computer Architecture*, May 1990.

J. R. Goodman, M. K. Vernon and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. 3rd Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 64–73, April 1989.

[6] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. *Operating Systems Review*, 23(5):211–223, Dec 1989.

[7] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transaction on Computers*, C-28(9):690–691, Sep 1979.

[8] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A Hamilton, David L Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. In *IEEE Journal on Selected Areas in Communications*, volume 1, pages 842–857, November 1983.

[9] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *Proc. 17th Int'l. Symp. on Computer Architecture*, pages 27–37, May 1990.

[10] Joonwon Lee and Umakishore Ramachandran. Locks, directories, and weak coherence - a recipe for scalable shared memory multiprocessors. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. To appear. An earlier version of this paper was presented at the 1990 ISCA Workshop on Scalable Shared Memory Architectures, Seattle, WA, May 1990.

[11] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[12] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *10th International Conference on Distributed Computing Systems*, pages 468–475, May 1990.

[13] Umakishore Ramachandran, Mustaque Ahamad, and Yousef Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 160–169, August 1989.

[14] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 0ct 1987.

[15] A. Sane, K. MacGregor, and Roy H. Campbell. Distributed virtual memory consistency protocols: Design and performance. In *Second IEEE Workshop on Experimental Distributed Systems*, pages 91–96, Oct 1990.

Further Reading

In this paper, we focussed on system issues in implementing DSM. There are other orthogonal issues such as language and runtime support for distributed computations based on the DSM model; reliability and fault-tolerance in DSM-based systems; and theoretical framework for reasoning about the correctness of DSM-based computations.

Munin is a language and runtime support system that identifies different types of shared data and adapts the coherence maintenance algorithms to suit the data types used in the computation. Wu and Fuchs have proposed recoverable shared memory that allows the implementation of checkpointing and recovery algorithms for fault-tolerant computations. Lipton and Sandberg propose a scalable shared memory system that allows copies of shared memory to be updated asynchronously

6 Concluding Remarks

Distributed systems offer the potential for computing capacity far in excess of that achievable with a uniprocessor. However, programming distributed applications is that much more complex. In particular information sharing and synchronization between threads of the same computation executing on different nodes could be very difficult. Message passing could be used for this purpose, and it even fits the physical architecture of the distributed system. However, programming with the message-passing paradigm requires a major re-orientation from the style of programming one may be used to in a uniprocessor. Distributed shared memory is emerging as a viable paradigm for programming distributed computations. It gives the illusion of shared memory in a physically distributed architecture, thus making programming in such an environment that much easier. However, a judicious use of the paradigm is necessary to ensure acceptable performance given the latency for remote accesses. There are several system issues that need to be addressed in implementing a DSM system: its interaction with virtual memory management at each node, the granularity of access, the ordering of shared memory accesses, the choice of coherence protocol, and the choice of synchronization. The survey presented here of DSM systems has attempted to compare them in the light of these issues. An important observation that could be made in summary is that to make the DSM paradigm more appealing from the performance standpoint, it is necessary to devise special-purpose hardware to reduce the processing overhead associated with remote memory accesses, as well as use high-speed medium such as optical fibers to reduce the communication overhead.

References

[1] Roberto Bisiani and Alessandro Forin. Multilingual parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[2] Partha Dasgupta, Richard LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, April 1991. To appear.

[3] G. S. Delp, A. S. Sethi, and D. J. Farber. An analysis of Memnet: An experiment in high-speed shared-memory local networking. In *SIGCOMM 88 Symposium on Communications, Architectures and Protocols*, pages 165–174, Aug 1988.

[4] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *The 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[5] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. 14th Int'l. Symp. on Computer Architecture*, pages 373–382, June 1988.

communication medium. The latter relates to the cost of shipping a unit of data across the network. The processing overhead could be reduced by providing some form of hardware (firmware) support. Memnet is one such example.

The bandwidth on the communication medium limits the speed of data transfer. Until now the communication medium in local area networks has mostly been electronic, e.g., Ethernet. With a 10Mb/sec Ethernet, the bit transfer rate on the wire for 8K bytes is approximately 6 ms not accounting for overheads such as packetizing for Ethernet, etc. Optical fiber is a promising new technology for use in local area networks. It offers bandwidth in the range of Gigabits/sec, and also the potential for multiple channels on the same "wire". Such a medium is expected to reduce the latency for remote memory access significantly but would require devising new protocols for memory coherence.

5 Performance Issues

There are two approaches for a computation to access remote data: send the computation to the data, usually referred to as remote procedure call (RPC); bring the data to the computation. DSM is a facility for enabling the latter approach. RPC is attractive when the amount of data to be manipulated is large or if the data is infrequently accessed. DSM is attractive when the computation needs to frequently access the data. In this case the locality of data access may be exploited to reduce the network traffic and the associated latency. Since shared data may be accessed from several nodes, using RPC to service these requests would result in this node becoming a bottleneck. DSM provides automatic "load sharing", since it allows the data to migrate to the nodes that need to access it. It has been shown [13, 11] through simulation studies and experimentation with parallel programs that DSM is a viable paradigm for loosely-coupled systems.

To give a flavor of the performance of a DSM system, we present the page-fault service rate for remote accesses in the Clouds distributed operating system. The Clouds system is implemented on a network of Sun 3/60 workstations connected by a 10Mbit/sec Ethernet. In Clouds, servicing a page-fault (8K page) from the local disk takes approximately 6 ms while servicing a page-fault across the network takes approximately 24 ms. Out of this time, roughly 12 ms is in the message exchange protocol and the data transfer on the medium; while the remaining 12 ms is accounted for by the DSM protocol overhead.

segment without the overhead for consistency maintenance. The application may choose to receive updates to this segment asynchronously. In Mether, a process may continue to read a stale copy of a page that has since been written by another process. It is up to the application to either force updates to be propagated on writes, or request for updates to existing copies prior to reading a page. Similarly in Agora, a server process (that has the master copy) is responsible for propagating updates to cache copies at other nodes.

Ivy, Mach, Memnet, and Choices implement a write-invalidation based memory coherence protocol. Agora uses a write-update style protocol, while Clouds uses a lock-based protocol. All these systems prevent memory inconsistencies from happening (*a la* deadlock prevention in operating systems). Domain, on the other hand, uses a version-based protocol to detect inconsistencies after they have happened (*a la* deadlock detection in operating systems), and takes corrective action.

The lock-based protocol of Clouds avoids having to send invalidation messages upon writes as is the case with write-invalidation style protocols. However, in lieu of these messages, the requesting thread has to explicitly relinquish a lock that it has acquired using the discard primitive. While in terms of the number of messages (invalidations or discards) there may not be a significant difference between the write-invalidation approach and Clouds, the burden of generating these messages is spread out among the readers and the writers in Clouds while it rests completely with the writers in the write-invalidation approach. Explicit locking of segments has the advantage of reducing the thrashing effect across the network that is possible in the write-invalidation protocols for highly shared data. Choices and Mirage attempt to reduce the extent of this thrashing by allowing a processor to retain control of a page for a fixed duration of time during which it may disregard remote requests for the same page.

In systems that do not use the lock-based approach, there is need for mechanisms to synchronize access to shared data. These mechanisms are orthogonal to the DSM and are usually built at the application level. This approach has the advantage of flexibility in the choice of primitives for synchronization. Clouds, Domain, and Choices provide some form of synchronization integrated with the DSM system.

There are two aspects to the overhead associated with a DSM system: processing (software), and communication (hardware). The former deals with maintaining the state information in software for the shared memory pages as well as the overhead for the message-exchange protocol on the

on the same processor.

Data granularity (see section 2.2) has two aspects: *unit of transfer* and *unit of access and locking*. Unit of transfer refers to the amount of information shipped across the network to satisfy a remote request, while the unit of access and locking is self explanatory. One disadvantage of integrating DSM management with VM management is that the data granularity of shared memory may be dictated by the underlying VM architecture. Most VM architectures provide address mapping and protection attributes at the level of a page or multiples of a page. This feature could constrain the unit of transfer between the DSM managers to be a page, and could lead to inefficiencies if the size of the shared data structure is less than a page. For example, suppose the size of a data object is 512 bytes. In a DSM system with a page size of 8192 bytes, an access to this data object would result in the transfer of 8192 bytes, an unnecessary overhead of 1500%. This overhead could be reduced with some simple optimizations. In Clouds, for example, if the data object being shared is less than a page, then only the actual size of the data object is transferred. A similar optimization is implemented in the Mether system. In Mether, each access to shared data results in the transfer of the first 32-bytes of the page. A process may decide to fetch the entire page if it so desires by examining the first 32-bytes of the page.

Another disadvantage of such an integration (between VM and DSM) could be that the unit of access and locking may be constrained to be a page (or multiples thereof). If multiple data structures are allocated on the same page, then this constraint could lead to *false sharing*, wherein distinct private data structures appear shared due to co-location on the same page. However, the advantage of such an integration is that the hardware memory management facilities in the underlying architecture could be exploited to efficiently implement access control and locking of the shared data structures.

Most DSM systems follow a strict memory model for data accesses: a read to a memory location always returns the most recent write to that location. However, as was mentioned earlier (see section 2.3) such a strict memory model is not required to ensure program correctness. In fact, weakening the memory model could result in significant performance advantage since the DSM manager would not have to incur the overhead associated with the strict memory model. Clouds, Mether, and Agora are examples of DSM systems that have implemented weaker models of memory coherence. The weak-read mode of segment access in Clouds allows an application to acquire a

	Domain	Ivy	Clouds	Mach	Agora	Memnet	Choices	Mether
Virtual Memory	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Granularity	Page	Page	Page	Page	Data structure	32-bytes	Page	Page
Memory Model	SC	SC	SC and User-defined	SC	User-defined	SC	SC	User-defined
Coherence Protocol	Version number based	WI	LBS	WI	WU	WI	WI and LBX	WU
Synchronization	Yes	No	Yes	No	No	No	Yes	No
Dedicated Hardware	No	No	No	No	No	Yes	No	No

Table 1: Comparison of DSM systems

Legend:

SC : Sequential Consistency

WI : Write Invalidate WU : Write Update

LBX : Lock-based (exclusive) LBS : Lock-based (exclusive and shared)

Accesses to remote memory are referred to the DSM manager by the VM manager, which in turn satisfies the request using its own coherence protocol. Thus, a page-fault to local memory is indistinguishable from a page-fault to remote memory, insofar as a process is concerned. The difference may only be in the latency of service. On the other hand, Memnet does not support virtual memory. The shared memory in Memnet is at the physical address level and its management is similar to private caches in an SMM. References to shared memory are serviced by a Memnet device without the software overhead associated with the VM management. Clouds provides a tighter integration of the VM system and the DSM system than any of the other systems by maintaining sharing information at the thread (process) level as opposed to processor level. In fact, such an integration is essential for a system that uses a lock-based coherence protocol to assure mutual exclusion (when needed) for a thread from all other threads, including ones that execute

denying access until the lock is removed. Another variant (similar to Mirage) that Choices allows is retaining access to a page for a fixed duration of time regardless of other pending requests.

3.7 Mether

Mether [12] provides a set of mechanisms for sharing memory across the network on top of SunOS 4.0. Mether differs from most other distributed shared memory systems in that it does not provide strict memory coherence. A process can continue to write on a page without the changes being reflected in other copies of the page. The other copies of the page may be updated in one of the following three ways: The process with the consistent copy of the page may initiate the updates to be propagated to all the other copies; a process holding an inconsistent copy of the page may invalidate its copy, resulting in a page-fault the next time it tries to access that page; a process holding an inconsistent copy of the page may explicitly request a consistent copy of the page. As should be evident, the user is responsible for tailoring the consistency requirements commensurate with the needs of the application. Mether provides a set of system calls to facilitate customizing the coherence requirements.

Mether provides for data driven page-faults. In a data driven page-fault, the process that caused the page-fault is blocked. No request for servicing the page-fault is sent by the server across the network. The page-fault is serviced when another process actively sends out an update for the page that caused the fault. Thus the page-faults are completely passive. Mether defines two types of pages: a short page (32-bytes) and a full page (8192 bytes). A short page, referred to as a *subset*, corresponds to the first 32-bytes of a full page, referred to as a *superset*. A process is ready to resume execution following a page-fault, as soon as the subset of the page is made available to this node.

4 Discussion

In this section we compare the features of the DSM systems surveyed in this paper, with respect to the issues enumerated in section 2 (see Table 1). All the DSM systems presented in this paper, with the exception of Agora and Memnet have integrated the DSM management with the VM management, i.e., the DSM manager cooperates with the VM manager to service page-faults.

cache. The reserved part is the permanent residence for the portion of the global shared memory that is managed by this Memnet device, while the cache is a temporary store for chunks that have been brought from remote Memnet devices. There is a chunk status table with each Memnet device that contains an entry for the chunks that are present in its physical memory. Memnet uses a write-invalidate style of cache protocol. A chunk may be in one of three states: valid (readable but not writable), exclusive (readable and writable), and invalid.

When a reference is generated for a chunk that is not locally available, then this request is sent around the token-ring. The first node that has valid copy of this chunk responds to this request. In case the reference is a "write", all other valid copies are invalidated before the chunk is written. In this sense, Memnet treats the token-ring as a logical broadcast bus. When a chunk has to be replaced from the cache, it is written back to its designated reserved area in the appropriate Memnet device. Since the system does not explicitly provide synchronization mechanisms, they have to be implemented at the user level to protect shared accesses.

3.6 Choices

Choices [15] is an operating system architecture developed at the University of Illinois at Urbana-Champaign. It uses class hierarchies and object-oriented programming to support the building of customized operating systems for shared memory and networked multiprocessors.

The virtual memory management system of Choices is similar to that of Mach. Choices uses the idea of a memory object that is cached in physical memory. The memory object abstraction is provided by the *MemoryObject* class while the virtual memory abstraction is provided by the *Domain* class. The Domain class maintains information about the association between the virtual memory of an application and the memory objects. It provides methods to bind and release a virtual address to a memory location. Sharing of memory is achieved by mapping the same memory object into multiple Domains. Sharing across the network is achieved via the *DistributedMemoryObjectCache* class. This class is responsible for servicing page-faults on a node for shared data residing on a remote node. It communicates with its peers to maintain the consistency of shared data using a write-invalidate style of protocol similar to the distributed manager protocol of Ivy. The granularity of remote access is a page. Choices also provides for locking a page similar to the read-write mode of Clouds (see section 3.3) to guarantee atomic updates to a memory object by

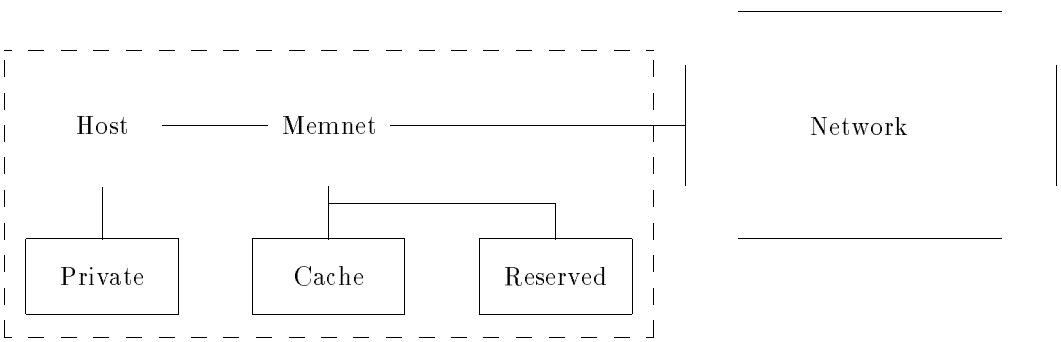

Figure 1: Memnet Architecture

potentially return stale data if the read occurs after the write is complete on the master copy and before the updates are propagated to the cache copies. The system expects that synchronization is implemented orthogonally using semaphores to guard against such stale accesses.

3.5 Memnet

Memnet [3] is a shared local area token-ring network being developed at the University of Delaware. It provides close coupling to the processors of a distributed multiprocessor system. There are three distinctive features of this project: first its allows a granularity of access (32-byte chunks) finer than a page; second it employs dedicated hardware (Memnet device) to service remote memory accesses; third it exploits the features of a special-purpose token ring network to implement a write-invalidate style of cache protocol. Given that there is an appreciable software overhead for remote access, dedicated hardware is almost a necessity to assure acceptable performance in DSM systems. The Memnet system does not support virtual memory. Therefore, the way DSM is managed on Memnet is very similar to cache management in an SMM.

Each node in the system consists of a host and a Memnet device (Figure 1). The host has access to its private memory which is inaccessible to other nodes. There is a large shared memory that is accessible from any node in the system. This shared memory is divided into 32-byte chunks and distributed among all the Memnet devices. The hardware address space seen by each host has two parts: private and shared. References to the shared part are passed to the associated Memnet device, which coordinates with other devices to resolve the references.

The physical memory associated with each Memnet is divided into two parts: *reserved* and

of another task. There are two ways of sharing memory between tasks in Mach: *copy-on-write* and *read-write*. In copy-on-write sharing, unrelated tasks share an address space without the actual data being copied. The first task that tries to write gets a copy of the shared address space. The copies become distinct and different from this point on. Read-write shared memory is created by allocating a memory region and setting its inheritance attributes to *shared*, *copy*, or *none*. Subsequent child task creation obeys the inheritance attributes specified by the parent. Pages specified as shared are physically shared between the parent and the children, i.e., there is exactly one copy of the page in the multiprocessor system. A child gets a logical copy of a page that is marked copy. Pages marked as none are not shared between the parent and the children.

Note that memory objects may be shared across the network. In this case, the physical memory is considered to be a cache for the memory objects. Associated with each memory object is a server called *pager* which manages the memory object. The pager handles any request for the memory object through a communication port. On a page-fault, the Mach page-fault handler checks whether the faulting thread has the access permissions for the page. If the permissions are correct then the page request is sent to the pager for the memory object. The pager services the request and sends back the page from the memory object. Mach provides strict memory coherence semantics using a write-invalidate approach for sharing of pages across the network.

Although Mach's shared memory semantics are geared towards managing shared memory in a tightly-coupled multiprocessor, there is nothing in the design that precludes its use in a loosely-coupled system. In fact, Agora [1] is a system that is built on top of Mach with the specific intent of providing shared memory semantics in a loosely-coupled system. The Agora system allows processes to share structured data, e.g., abstract data types across heterogeneous architectures over a local area network. Agora uses the shared memory abstraction of the Mach operating system to share data structures among processes on the same machine. It also provides simple locks to synchronize access to shared data. To provide sharing across the network, the shared data structures are stored in the shared memory of the process that created the data structure. This copy of data is called the *master copy* while a copy of the data at another node is called a *cache copy*. Data is shared using copy-on-write semantics with updates to the cache copies. Writes to shared data are done on the master copy while data is read from the cache copy. A server process running on the node with the master copy is responsible for updating the copies of the data at other nodes. A read may

is responsible for preserving the consistency of the segments that it owns. DSMC uses a lock-based protocol for coherence maintenance that unifies synchronization and transport of data. It supports both exclusive (read-write) locks as well as shared (read-only) locks for segment access. Upon a lock request, the owner DSMC encloses the requested segment (parts thereof, see section 4) in the message that grants the lock request, thus providing synchronization for free. A segment may be requested by a thread in one of the four modes: *read-only*, *read-write*, *weak-read*, and *none*. Read-only mode provides a non-exclusive lock on the segment while read-write mode provides an exclusive lock on the segment. Mode none gives exclusive access to the segment without locking the segment, i.e., any new request would result in the segment being yanked away to service the request. These three modes provide strong memory coherence, i.e., the contents of a segment are indistinguishable regardless of the node from which it is accessed. However, there are situations where such strong memory coherence may not be required (e.g. a monitoring thread that wants to "inspect" the contents of a segment). For such purposes, the weak-read mode is provided. In this mode a current copy of the segment is sent to the requester. The thread would continue to receive updates to the segment if it had specified the *update* option on the weak-read request. A thread explicitly relinquishes a lock that it has acquired for a segment by using the *discard* primitive.

3.4 Mach and Agora

Mach [14] is a multiprocessor operating system kernel developed at CMU. It provides five basic abstractions: *task*, *thread*, *port*, *message* and *memory object*. A task is an execution environment that includes a virtual address space and an access list to system resources. A thread is a basic unit of CPU scheduling and it executes in the environment provided by a task; a port is a communication channel; and a message is a typed collection of data objects. Inter-thread communication is effected using messages on the ports. A memory object is a collection of data that may be mapped into the address space of any task. It is a structured mechanism for managing virtual memory independent of the underlying architecture.

An address space consists of a collection of memory mappings between a task and memory objects. A task may modify its address space by allocating and de-allocating a region of virtual memory. A task may also set protection attributes and specify inheritance of a region of virtual memory. It could create and manage a memory object that maybe mapped into the address space

returns the value of the eventcount. Await primitive suspends the calling process until the eventcount value reaches a specified value. Advance primitive increments the value of the eventcount by one and wakes up any sleeping processes. Any process may use an eventcount after the eventcount has been initialized.

An extension of Ivy's memory coherence protocol has been implemented in the Mirage system at UCLA [6]. It allows a reader or a writer of a page to retain access to a page for a fixed duration of time regardless of the pending requests.

3.3 Clouds

Clouds [2] is an on going distributed operating system project at Georgia Tech implemented on a network of Sun-3 workstations. One of the distinctive features of Clouds is its separation of two notions that have been traditionally intertwined in most operating systems, namely, address space and computation. The former is specified by objects and the latter by threads in Clouds. An object is a passive entity (i.e. there is no process associated with it) that is part of a global name space. It specifies a distinct virtual space that is unique in the entire distributed system. The object encapsulates data that can be manipulated only from within the object. There are *entry points* in the object that are invocable from other objects. The entry points contain code for manipulating the data in the object, and may themselves invoke entry points in other objects. To allow concurrent execution of more than one computation in the same object, shared-memory style synchronization primitives are provided by the operating system.

A thread is an active entity that provides the notion of a computation. It executes in the context of an object. During the course of execution, a thread may invoke entry points in other objects. Thus a thread is not associated with a single address space. Further, since these objects may not all be at the same node, a thread may span machine boundaries during the course of execution.

The collection of objects in Clouds represents a distributed shared global virtual space. A thread traverses the address spaces of the objects that it invokes during its execution. Objects are composed of segments that form the basic unit of sharing. Each segment may be composed of one or more pages. Pages are the units of distribution. There is an entity, Distributed Shared Memory Controller (DSMC) [13] at each node that owns and maintains the segments that are created in the node. The DSMC provides a set of primitives for segment access and transport, and

mode. Ivy uses multiple-readers/single-writer memory semantics. In this approach, all read-only copies of a page are invalidated when any processor attempts to write to the page. Three different flavors of the invalidation scheme have been implemented in Ivy. In the central scheme, a central manager maintains a table to keep track of the locations for all the pages. On a page-fault, the faulting processor asks the central manager for a copy of the page. The central manager then asks the *owner* of the page to send a copy of the page to the faulting processor. A node is said to be the owner of a page if it was the last node that modified the page. The second approach, fixed distribution scheme, is similar to the centralized scheme except that each node is assigned a predetermined set of pages to manage. A mapping function is used to perform this distribution. This scheme avoids the single site bottleneck of the centralized scheme. The third scheme is the dynamic distributed manager algorithm that keeps track of the ownership for all the pages. This is done by adding a field called *probOwner* in each page table entry at all the processors. The probOwner field is used as a hint to locate the true owner of a page. A processor sends the request for a page to the node indicated in the probOwner field of the page. If the node that receives the page request is not the current owner of the page, it forwards the request to the node indicated in its page table. Initially, the probOwner field is set to some default value in all the processors. The probOwner field is updated whenever a processor receives an invalidation request, a processor relinquishes the ownership of a page on a read or write page-fault, or a processor forwards the page-fault request to another node.

On a read page-fault, the processor locates the owner of a page and sends a request to the owner of the page. The owner maintains a set of all nodes that have a readonly copy of the page in a *copyset*. The owner adds the faulting processor to the copyset of the page and sends a copy of the page to the faulting processor. On a write page-fault, the processor locates the owner of the page and sends a request to the owner of the page. The owner of the page sends the page and its copyset to the faulting processor. The faulting processor sends an invalidation message to all the processors in the copyset of the page. When all the invalidation requests have been acknowledged, the faulting processor restarts the blocked process.

Ivy provides synchronization mechanisms based on the primitives (*eventcounts*) provided by the underlying operating system. An eventcount supports four operations: *init*(count), *read*(count), *await*(count, value),and *advance*(count). Init primitive initializes an event count. Read primitive

version of the page; otherwise a write concurrency violation has occurred. A write request to a page updates both the home node's and the requesting node's time stamp for the object. The system also provides primitives to flush stale pages of cached object, inquiring current version number of an object, and sending back modified pages of a cached object.

The higher level provides an object locking mechanism. Several types of lock modes are provided including a multiple-readers/single-writer lock. Lock and unlock requests for remote objects are always sent to the home node. A lock request that is granted returns the current version number of the lock. This information is used to remove stale pages from the requesting node's main memory. The unlock operation forces modified pages back to the home node before the lock is released. In Domain, lock requests are not enqueued; if the lock is currently in use, then the requester is denied access to the lock and would have to retry later.

3.2 Ivy

Ivy [11] is a distributed shared memory system implemented on Apollo workstations interconnected by a token-ring network. It provides a shared virtual address space similar in concept to the Domain system with the difference that the granularity of access is a physical page in Ivy as opposed to an object in Domain.

In Ivy, a process address space is divided into two parts: a private part and a shared part. The private part is local to a process and cannot be accessed by any other process. The shared part is implemented using shared virtual memory. A process may access any memory location of the shared virtual memory through the shared part of its address space. To manage the shared virtual memory, each node has a memory mapping manager. The memory mapping manager implements the mapping between the physical memory of the processor and the shared virtual memory address space. The memory mapping manager at each node treats the physical memory as a cache for the shared virtual memory and is responsible for maintaining the shared virtual memory coherent at all times. The shared virtual memory is implemented at the processor-level: Thus once a page of the shared virtual memory is made available at a node by its memory mapping manager, it becomes accessible to all processes that execute at this node.

Ivy uses a write-invalidate type of coherence protocol to manage its shared virtual memory. The virtual memory is partitioned into pages. Individual pages can exist in read-only, write, or nil

3 Case studies

In this section, we review several DSM systems with respect to the design issues presented in the previous section.

3.1 Apollo Domain

Apollo Domain [8] system is one of the earliest systems that employs DSM paradigm to assure consistency of shared objects in a local area network of personal workstations and data servers. It provides an integrated distributed environment with each node possessing a high degree of autonomy with additional system mechanisms that permit cooperation and sharing among the nodes. The Domain system allows users to name and access all objects in a transparent manner by having a distributed object storage system (OSS). The OSS is a flat address space of objects addressed by unique identifiers (UID's). The distributed OSS allows objects to be accessed from any node in the network. Processes could potentially access all objects by presenting their UID's and mapping the object into their respective address spaces.

The OSS is implemented in two layers. The first layer provides access to local objects that reside on the same node as the faulting process. The second layer provides access to remote objects in a transparent manner: On a page-fault, the OSS determines if the access is to a local object. If so then the object is read from the local disk and mapped into the process' address space. If the object is not locally available then it is located using an object locating service. After the object is located, the specific page is requested and mapped into the process' address space.

To assure consistency of replicated copies of an object a two-level approach is adopted. The lower level detects concurrency violations using a time-stamp based version number scheme for each object. The time-stamp corresponds to the time the object was last modified. Every node remembers the version number for all remote objects whose pages it has encached in its main memory. Every time an object is read from another node, its version number is returned with it. If it is the only page of the object encached in this node, its version number is remembered. If not, the returned version number must match the remembered version number for the object; otherwise a read concurrency violation has occurred. Every time a page of an object is written back to its home node, the current version number is sent with the write request and an updated version number is returned. The home node only accepts the page if the write occurs on a current

free with this approach.

These three categories of cache protocols may be likened to deadlock prevention in an operating system, in that they prevent memory consistency violations from ever happening. It is possible to take a more liberal approach (similar to deadlock detection) and allow memory consistency violations to occur but have mechanisms in the system to detect such violations and take corrective action when they occur (see section 3.1).

In distributed systems, the number of messages is a measure of protocol performance. From this standpoint, the lock-based policy is expected to out-perform the other two, since coherence is maintained commensurate with the semantics of sharing in the computation. Moreover, since locking could be integrated with the data transfer there is no need for any additional mechanisms for providing mutual exclusion for shared write accesses. In both write-broadcast and write-invalidate policies there is a need to provide synchronization mechanisms on top of the coherence policy to assure mutual exclusion for multiple nodes requesting to write to the same page. However, lock-based policy has its drawbacks: In particular it does not have the generality of the other two policies. By decoupling memory coherence and synchronization, it is possible to devise synchronization mechanisms independent of the coherence policy. The lock-based policy requires explicit directives from the system software to know the semantics of sharing, while the other two do not require any such directives.

2.5 Synchronization

Another issue is the way interprocess communication is achieved in such systems. Extending the analogy of shared memory multiprocessors to DSM, it would seem that shared-memory style of synchronization would be expected in DSM systems as well. However, the granularity of accesses in DSM systems precludes using true shared memory style of synchronization such as Test-and-Set on arbitrary memory locations. One possibility is to combine synchronization with sharing as has been suggested in some multiprocessor cache protocols [9]. Another possibility to have an orthogonal set of primitives to achieve synchronization. This latter approach is attractive since there could be situations where there may be very little sharing of data but independent computation may have to synchronize with one another. Some systems provide semaphore operations or lock operations in addition to the shared memory primitives.

It is possible to devise distributed versions of cache coherence protocols to maintain the consistency of distributed shared memory. However, such implementations would suffer to some degree due to the mismatch in the capabilities of their intended environments and distributed systems. The definition of coherence that works well for shared memory multiprocessors is not appropriate in a distributed environment since there is no "system bus" to impose a total order on the memory operations that are performed by all the processors. Further, while invalidation of cached copies of data is a viable approach in multiprocessors (with a system bus) it is expensive in a distributed system due to the cost of the invalidation messages. Invalidation involves at least sending a multicast message to all the nodes that have a read-copy of the data. Achieving reliable delivery of such multicast messages is expensive in a distributed system.

In reality, memory coherence and process synchronization are closely inter twined. The ability of a process to read or write shared data is invariably acquired through some synchronization method. Absence of synchronization implies that any arbitrary ordering of simultaneous accesses to a shared location should yield valid results for a given computation. Therefore, in such cases there is no coherence requirement. In fact, it would be reasonable to argue that memory coherence should be defined in conjunction with process synchronization.

Not surprisingly, the solutions that have been suggested in the DSM environment are similar to the ones in the SMM caches. Broadly, these solutions may be classified into three categories:

1. **Write-broadcast policy**: This is the most intuitive scheme wherein a write to a location is sent to all the nodes that have a copy, thus updating all the copies. While this scheme is simple to implement, it is expected to generate considerable network traffic.

2. **Write-invalidate policy**: In this scheme instead of updating, the first write to a location causes invalidation of all the other copies. The traffic on the network would depend on the level of sharing with this scheme. A major disadvantage of this scheme is the unnecessary invalidations that would have to be sent to nodes that are no longer interested in a particular location even though they may have it cached.

3. **Lock-based policy**: In this approach, a request to read or write an item is preceded by a lock-request for the same item, and succeeded by a lock-release. By combining the lock-request (synchronization) and the request for the item, it is possible to provide synchronization for

overly restrictive memory model. This fact was first observed by Dubois, Scheurich and Briggs [4], who proposed *weak ordering* as an alternative to sequential consistency. Weak ordering requires that memory accesses from a process are performed in program order; synchronization operations are globally performed before allowing a process to continue; and all shared data accesses from a process are globally performed before issuing a synchronization operation. It is possible to weaken the ordering constraint further by distinguishing between synchronization operations that acquire and release rights to access shared data [10]. Of course, all such weakening of the memory model assumes that it is possible to distinguish between two types of memory accesses: synchronization and read-write data. If such a distinction cannot be made then a conservative approach (such as sequential consistency) may be the only way to assure program correctness. Recently, there have been proposals for hardware support to make this distinction possible in SMMs [9]. DSM, on the other hand, is usually a software abstraction. Therefore it is quite straightforward (with support from the compiler and/or operating system) to make this distinction possible and weaken the memory model.

2.4 Choice of Protocol

DSM assumes that all memory is globally shared. This assumption requires that independent computations started at different nodes see a consistent view of the shared memory. To facilitate this view would require a coherence scheme. Consistency maintenance of distributed shared memory is similar to cache coherence in multiprocessors. Shared memory multiprocessors such as Encore's Multimax, consist of several processors connected to a common shared memory via a system bus. A main memory cache is associated with each processor to help reduce the traffic to the shared memory. Multiprocessor cache consistency protocols ensure the following memory coherence constraint: a read operation performed by a processor returns the most recent value written into that location (by any processor). This criterion is appropriate in a shared memory multiprocessor since the system bus (a broadcast medium) serializes the memory operations of all the processors. The cache coherence algorithms that have been proposed for multiprocessors are viable since the cost (measured in circuit complexity as well as time) of implementing them in hardware is a small fraction of the total system cost. Further, bus-based multiprocessors usually have the ability to invalidate (or update) all cached copies in one atomic bus cycle.

memory management at each node. This requirement often forces the granularity of access to be an integral number of the fundamental unit of memory management (usually a page). However, it is possible to reduce the network latency by transferring the page partially (see section 4). The key point to note is that the data granularity has to be sufficiently high to make the DSM paradigm viable.

2.3 Memory Model

In a uniprocessor, correctness of execution is ensured by preserving the order of memory references generated by a processor. Lamport [7] has proposed *sequential consistency* as a memory model for ordering shared memory accesses to ensure correct multiprocessor execution. In this model, the order of memory references generated by an individual processor's preserved, while the global order of memory references from all the processors is an arbitrary interleaving of the individual processors' reference streams that preserves the order of references emanating from each processor. Essentially, sequential consistency ensures that the view of the memory is consistent at all times from all the processors.

Insofar as correctness of multiprocessor execution is concerned, only the ordering of the shared memory references is of interest. Shared memory accesses may be categorized into three types:

- shared code,
- synchronization variables, and
- shared data variables.

Shared code is always read-only (barring self-modifying code which is not in vogue anymore), and hence is always consistent. On the other hand, synchronization variables require that memory consistency be strictly preserved. Shared data variables normally require strict consistency as well. However, several applications exist wherein the program correctness would not be compromised even if there are temporary inconsistencies in the view of the shared memory as seen by distinct processes. Examples of such applications include asynchronous and/or iterative algorithms. Moreover, if the programs are written to obey some synchronization paradigm such as lock/unlock, and semaphore P/V, then ensuring a consistent view of shared data may be deferred to synchronization points in the program without compromising program correctness. Thus sequential consistency is an

a remote memory access request

- the software overhead involved in the communication subsystem (i.e. the basic transport protocol) for effecting the inter-node message communication to service the request
- the speed of the communication medium (i.e. hardware).

2.2 Granularity

There are two dimensions to granularity: *computation granularity* and *data granularity*. The former deals with the amount of computation a process has to do between synchronization points in a multi-process computation. The latter deals with the amount of shared information processed during this computation phase.

Eggers and Katz [5] define "write-run" as a sequence of reads and writes by a given processor following an initial write executed by the same processor to a given shared memory location before an external read by a different processor to that shared memory location occurs. Write run in some sense captures the computation granularity in a multi-process computation. In an SMM system, write-runs of representative applications may range from a few to a few tens of references. However, in a DSM system, write-runs of a few hundred instructions would be more appropriate given the latency for remote accesses (see section 5).

Another distinction between the SMM and the DSM is in the data granularity of accesses that are practical in the two. In a uniprocessor memory hierarchy, the processor to cache transfer is in the tens of nanoseconds, and the cache to main memory transfer is in the hundreds of nanoseconds, and the main memory to disk transfer is in the order of milliseconds. Correspondingly, the granularity of transfer that makes feasible sense are: byte or word between the processor and the cache, a block of several bytes between the main memory and the cache, and a page ranging from 512 bytes to a several Kilo bytes between the main memory and the disk. DSM systems add a new dimension to the memory hierarchy, namely remote memory access across the network. This dimension is the one that is of interest to this survey. The choice of the network plays a big role in determining the latency. Nevertheless, as we pointed out in the previous subsection, independent of this choice there is a fixed software overhead to be incurred depending on the choice of the data transfer protocol on the network. Moreover, such remote memory accesses need to be integrated somehow with the

2 Issues in Designing a DSM System

In designing a DSM system, an important point to note is that the nodes (compute and data servers) do not physically share memory. Message exchange across the network is the only mechanism for information transfer among the nodes. Thus each node is an autonomous entity managing its local resources such as processor(s), and physical memory. The DSM paradigm presents a unified view of the address space as seen by independent computations executing on different nodes. This global notion has to be integrated with the local notions such as virtual memory management at each node, and context switching. In the following subsections we discuss the issues that have to be addressed in designing and implementing a DSM system.

2.1 Virtual Memory and DSM

DSM is not true shared memory as is the case in shared memory multiprocessors. Thus remote memory accesses have to be reconciled with the memory management at each node. Of course, if the basic machine architecture does not support virtual memory then the solution could perhaps be simpler. However, if the basic architecture supports virtual memory then the DSM management and Virtual Memory (VM) management have to be integrated. In particular, the local memory at each node may be considered simply a cache of a global address space that spans the entire network. The DSM and VM management at each node would have to cooperate to ensure that the semantics implemented by the DSM manager and the VM manager are not compromised. The normal VM chores such as page replacement, swapping, and flushing have to be done in consideration with the DSM algorithms. Similarly, in satisfying a remote memory request the DSM would have to consult the VM manager to get a page frame, etc. Upon release of a page the DSM has to instruct the VM manager to invalidate page table entries, and take other related actions.

The effectiveness of the DSM paradigm depends crucially on how quickly a remote memory access request is serviced, and the computation is allowed to continue, which in turn depends on several factors:

- the speed at which the VM system detects that a memory access fault (i.e. a page-fault), or a prefetching request entails a remote access
- the software overhead involved in the DSM protocol (i.e. coherence mechanism) for servicing

1 Introduction

Traditionally, shared memory and message-passing have been the two paradigms used for interprocess communication and synchronization in multi-process computations. The duality between the two paradigms for structuring computations is well-known. Nevertheless, shared memory has been an appealing paradigm from the point of view of programming ease even in distributed systems as evidenced by the programming environments that have been proposed based on this paradigm. However, when such a programming paradigm is implemented on top of a distributed message-based operating system, there is a mismatch between the communication abstraction provided by the operating system and the one required by the programming environment. This mismatch leads to poor performance. Therefore, it is not surprising that several researchers [11, 13, 8] have suggested system architectures that provide the abstraction of shared memory in a physically non-shared (distributed) architecture. We refer to this abstraction as *Distributed Shared Memory* (DSM).

A second motivation for DSM arises from the structure of current distributed computing environments. A typical distributed computing environment consists of *compute servers* (workstations) and *data servers* (file servers) interconnected by a local area network. In such an environment, there are two chores to be performed to execute a computation. The first chore involves selecting a compute server, which is a scheduling decision best left to a high-level policy making entity. The second one involves bringing the code and data from the data server to the selected compute server before executing the computation. This chore requires a remote paging facility. If sharing of data is coupled with this remote paging, it could be seen that DSM presents itself as a natural facility for combining the two.

On the surface a distributed system structured with the DSM paradigm may seem no different from a shared memory multiprocessor (SMM) in the semantic sense. However, when we look at the issues to be resolved from the point of view of designing a DSM system the differences become apparent. In this paper we confine ourselves to system issues that arise in access and transport of data as they apply to DSM systems. There are a number of other orthogonal issues (see Further Reading) namely, object and process migration, reliability, availability, and fault-tolerance that are equally important. But these issues are not the focus of this paper.

A Survey of Distributed Shared Memory in Loosely-coupled Systems*

Ajay Mohindra *Umakishore Ramachandran*

GIT-CC-91/01

January, 1991

College of Computing

Georgia Institute of Technology

Atlanta, Ga 30332

Ph: (404) 894-5136

Fax: (404) 853-9378

e-mail: rama@cc.gatech.edu

Abstract

While the duality between message-passing and shared memory for interprocess communication is well-known, the shared memory paradigm has drawn considerable attention in recent times even in distributed systems. Distributed Shared Memory (DSM) is the abstraction for supporting the notion of shared memory in a physically non-shared (distributed) architecture. It gives a unified view for managing the resources in the distributed system.

The two main issues to be addressed in DSM design and implementation are maintaining the consistency of the shared data across the network without incurring a high overhead, and integrating the DSM mechanisms with the local memory management at each node. Further, programming in this environment requires some judicious trade-off between programming ease and performance: while DSM may provide the abstraction of shared memory, it is not *true* shared memory both from the point of view of the semantics and the cost of shared memory access. The main focus of this paper is to identify the issues in the design of DSM systems, present a comprehensive survey of the DSM mechanisms that have been proposed to date, and discuss these mechanisms in the light of the design issues.

Keywords: Distributed shared memory, Shared virtual memory, Memory coherence protocols, Synchronization, Local area network, Loosely-coupled systems.

*This work is funded in part by an NSF grant CCR-8619886 and an NSF PYI Award MIP-9058430 for U. Ramachandran.