Chapter Two

Distributed Shared Memory Issues

Distributed shared memory provides a virtual machine where processes executing on all nodes share a common address space. The shared address space is physically distributed among the nodes in the system. In most implementations of DSM, the distributed address space is allowed to be replicated and cached locally by the nodes for faster access. The distribution and replication of the address space raise several issues that need to be addressed to guarantee ‘‘correct’’ execution of applications. First, since different accesses are served by different nodes, the access paths and access latencies differ from access to access. Therefore, accesses from different processes can get ordered in several different ways. Second, more than one access to the same address can be serviced concurrently. The DSM implementation should ensure that access orderings that lead to incorrect execution of the application are not possible. Specifically, concurrent accesses to the same address should be managed so as to provide the abstraction of a single common memory module.

The support of DSM involves costs in the storage overhead of tables, as well as in the performance overhead of moving data. Both these costs grow with the size of the system. Naive mechanisms for supporting DSM can load large systems excessively, making the use of DSM impractical for large systems. DSM support mechanisms must be designed such that they scale well to larger systems. The reliability of the DSM multicomputer is another important issue to be addressed. Since the distributed address space in a DSM system is shared by all processors on the system, when a single node's memory fails, a part of the shared address space is lost. Hence the failure of a single node in the system can halt the progress of the entire system (similar scenario is also possible in message-passing multicomputer systems). Error recovery mechanisms that improve the reliability of the DSM system must also be developed.

We describe these issues here in more detail, and discuss some approaches to handling these issues. Finally, we summarize the key aspects of other DSM architectures and highlight their approach to these issues.

2.1. Consistency

Processes interact with memory through read and write accesses. Write accesses modify the contents of locations in memory. Read accesses retrieve the contents of memory locations. By reading the state of memory, a process may ''observe'' the order in which the write accesses were completed by memory. For example, consider write accesses to two different memory locations issued by two processes. By reading these locations, a third process may observe (determine from its perspective) that one of these write accesses was completed before the other. This third process may thus establish (from its perspective) an ordering between the two write accesses. A *consistency model* defines the possible relationships between the order in which accesses are initiated by a process and the order in which these accesses may be observed by other processes in the system. Applications may rely on a well-defined consistency model for correctness. For example, in the implementation of mutual exclusion without hardware interlocks [Lamp79]. If memory access orderings that are not expected by the application are possible, the application can execute incorrectly (such as multiple processes entering a critical section). The implementation of an architecture must ensure that only orderings legal in the consistency model are possible during execution.

On conventional, simple uniprocessors, only one thread of computation (e.g., process) is active at any instant. This single thread of control provides a simple consistency model which is straightforward to understand and easy to implement. In this consistency model, all accesses issued to memory are totally ordered in real time, and each memory access observes the effects of all preceding accesses. Hence, all the accesses of a process in this system can be sequentially ordered.

In a DSM system, accesses can be issued to memory in parallel by multiple processes. Since these accesses can be serviced concurrently by the distributed and replicated memory, the ordering of memory accesses in such a system is not straightforward. Since multiple paths exist between processors in the multicomputer and data is dynamically replicated in DSM, a set of accesses that are serviced concurrently can be ordered in any of several possible ways. Furthermore, the mechanism used to access the distributed memory may allow different processes to observe different orderings. Hence, it is essential to establish a consistency model for this system.

Different consistency models impose different requirements on the implementation and on the application programmer or compiler. A consistency model that places strong constraints on the access ordering requires a conservative implementation, which can limit the performance of the system [Ghar91]. However, such a model can ease the programmer's (or compiler's) task of writing correct programs since the implementation has limited freedom in ordering accesses. A consistency model that is weaker provides the hardware flexibility to choose orderings that support more parallelism. This enables aggressive (potentially costlier) implementations that can support higher performance. However, the programmer will have more responsibility in writing the application such that only access orderings which lead to correct application execution are possible.

Several consistency models have been proposed in the literature. The model that is considered ‘‘intuitive’’ by most programmers is *sequential consistency* [Lamp79]. Sequential consistency derives from the execution model of a sequential system. It requires that the execution on a parallel system is equivalent to some interleaving of the execution of the parallel processes on a sequential machine. Scheurich and Dubois [Sche87] have identified conditions that are sufficient to guarantee that a system is sequentially consistent. Specifically, three conditions must hold:

1) Each processor must issue all its memory accesses in the order that they appear in the program.

2) A write access must block the processor issuing the access until it is guaranteed that no processor in the system can read an older value.

3) A read access must block the processor until a value is received for the read and it can be guaranteed that no processor in the system can receive an older value.

The sequential consistency model is conceptually simpler than weaker consistency models such as *weak ordering* [Dubo86, Adve90]. In the weak ordering model, all synchronization accesses must be identifiable by the hardware, and the hardware need only guarantee that all synchronization accesses can be ordered sequentially. By allowing a processor to continue without blocking on non-synchronization accesses, a weakly ordered system can support better performance than the sequentially consistent system. However, the execution of an application on the weakly ordered system will appear sequentially consistent only if all shared variables in the program are accessed within synchronization constructs that are visible to the hardware.

2.2. Coherency

To support higher performance, multiprocessor systems (including DSM multicomputers) allow the same shared data to be cached by multiple processors. Since several copies of a memory location exist, the memory system can service several accesses to the same location at the same time. The memory system is defined to be *coherent* if all write operations to any given memory location are observed by all processors in the system in the same order [Arch87]. Coherency allows a system to provide the abstraction of a single master copy while servicing multiple accesses concurrently.

The replication of data enables several processors to concurrently operate on the same memory location with low latency. However, it allows the possibility that the system may associate more than one value with a memory location at the end of an execution. In addition, the value at the end of execution may not be the value written by the ‘‘last’’ write to the location. A coherence protocol maintains coherence by managing all the copies such that the effect of the operations on the several copies are equivalent to a single sequence of operations on one master copy.

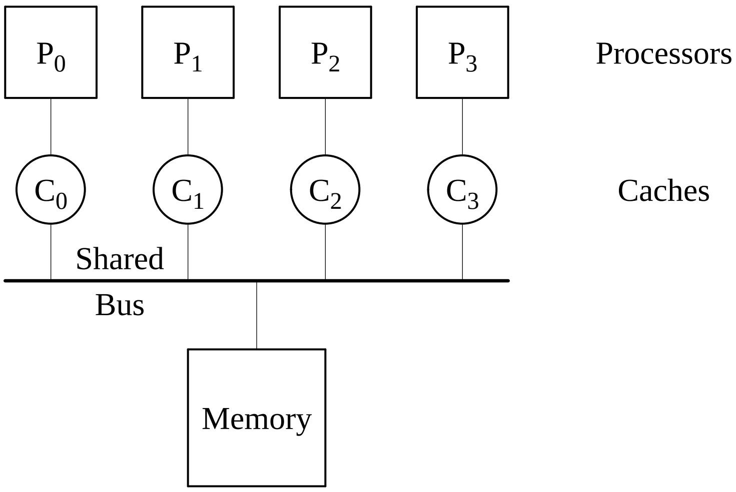

Figure 2.1: System with shared bus for snoop-based coherence protocols.

Maintaining *cache coherence* is a critical requirement in shared memory multiprocessor architectures [Cens78, Arch85, Agar88, Li89b]. Cache coherence protocols can be divided into two classes: *snoop-based coherence protocols* [Katz85, Good83] and *directory-based coherence protocols* [Tang76, Agar88]. Snoop-based coherence protocols are suitable only for systems where processors share a common communication medium such as a bus (Figure 2.1). Each processor cache in the system can snoop on all traffic on the bus and is required to take appropriate actions for addresses that are present in the cache.

In order to implement snoop-based coherence protocols, each processor cache maintains the status of its blocks as shared or unshared. When a processor writes to a shared block, the write is propagated on the bus. Upon seeing the write bus transaction, all other processors that have a copy of the block either invalidate their copy of the block (in an invalidation protocol), or update their copy of the block (in an update protocol). Upon a read miss, the cache issues the access on the bus. The requested block is provided by the main memory, if the main memory has the latest value, and by another cache, if the main memory's value is stale. Snoop-based cache protocols are not suitable for large systems since the shared bus can only support a limited number of processors (about 30) without degrading performance.

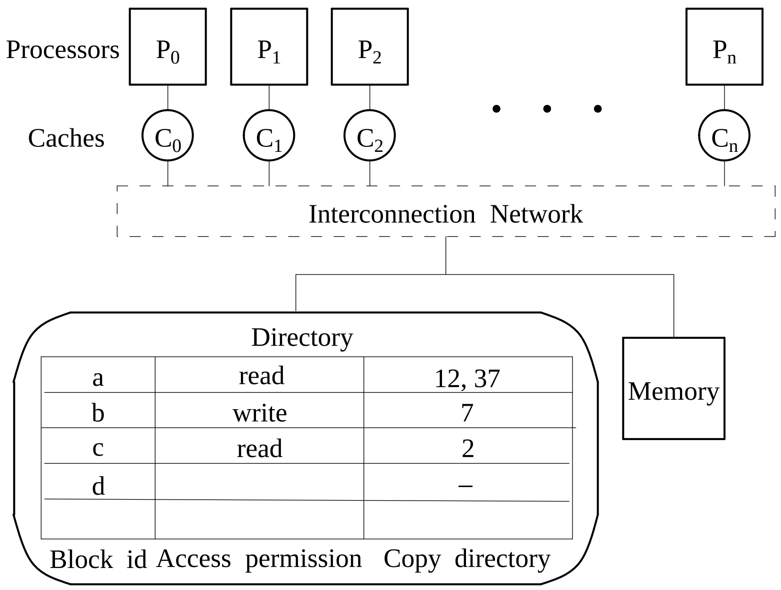

Figure 2.2: Centralized directory for directory-based coherence protocol.

Directory-based protocols [Tang76, Cens78, Agar88, Li89b] do not rely on a broadcast network such as a shared bus, and, hence are more suitable for large systems. Due to the absence of a snooping medium, directory-based protocols use a directory which maintains, for each block, the state of the block as shared or exclusive, and the set of nodes that have copies of the block (we call this part of the directory the *copy directory*). In the absence of a bus, point-to-point messages are used for communication amongst the nodes and the directory. Upon a write to a shared block, the directory is looked up to determine the nodes that have copies of the block. These copies are invalidated (invalidation-based) or updated (update-based) to perform the write. The directory can be centralized at the memory (Figure 2.2) or at a specific node. The broadcast network (bus) is a performance bottleneck which is removed when we use a directory. However, the centralized directory is a performance and reliability bottleneck.

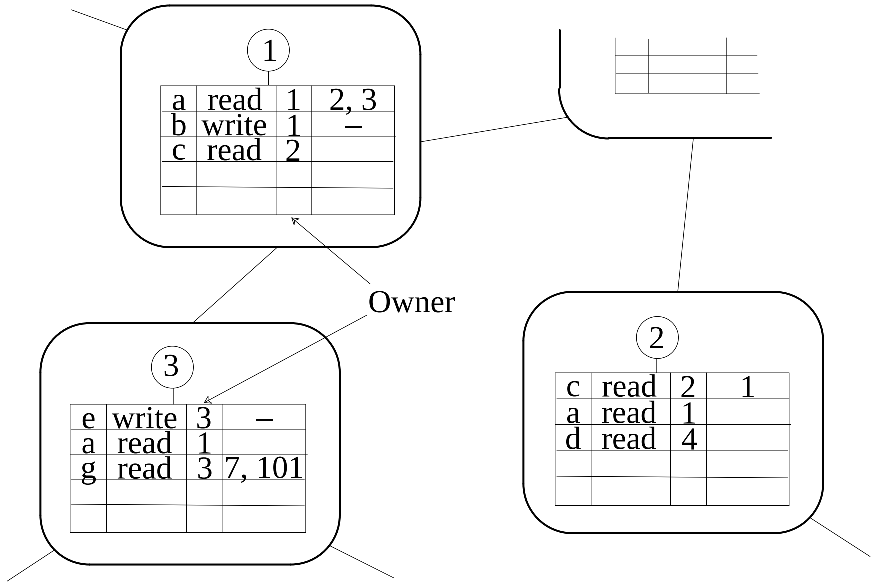

Figure 2.3: Distributed directory for directory-based coherence protocol.

Distributed directory-based coherence protocols alleviate the problem of a central directory becoming a bottleneck by distributing the directory among the nodes in the system. Li et al. [Li89b] have explored several different ways to distribute the directory. With the distributed fixed manager protocol, each node is statically assigned a set of blocks to manage. A node maintains the directory for the blocks that it manages (Figure 2.3). In the distributed dynamic manager protocol, the manager of a block is dynamically determined. Typically, the node that last wrote the block maintains and manages the directory for that block. Since a block's manager is not fixed in the distributed dynamic manager protocol, an additional mechanism to locate the current manager of blocks is needed.

While directory-based coherence protocols scale better to large systems than snoop-based coherence protocols, they are more complex to design and verify. In a snoop-based system, the shared bus can serve as a mutual-exclusion lock. Since only one processor can access the lock (bus) at a time, all accesses are effectively ordered sequentially. In a system where processors are interconnected by a network, multiple paths are available. Consequently, a directory-based coherence protocol implementation for such a system must address issues of message races, deadlocks etc.

2.3. System Reliability through Fault Tolerance

Components of any system are inherently susceptible to faults. The failure of these components can lead to the failure of the system. Fault tolerance is an approach that improves system reliability by allowing the system to operate correctly despite the failure of some of its components. Fault tolerance mechanisms are essential for meeting the reliability requirements of critical special-purpose applications. Fault tolerance may also be necessary for general-purpose applications which require large multicomputer systems to continuously operate correctly for hundreds of hours. Large systems may be composed of several thousand chips, which have individual transient failure rates of 1000 - 6000 per billion part hours. If the entire system can fail due to the failure of a single chip, the system will have a mean time between failures of only tens of hours. Hence, without fault tolerance, large systems may be unable to meet their reliability requirements.

Error recovery techniques implement fault tolerance by restoring the system to a valid state following an error. In error recovery based on *checkpointing* and *rollback* [Rand78, Bari83, Tami84, Koo87, Borg83, Powe83, John87, Stro85, John88], an error-free state of the system (checkpoint) is periodically saved to stable storage. Upon an error in the system, the system recovers to a correct state by restoring (rollback) a previously checkpointed state. In a system where processes do not communicate with each other, process states are not dependent on each other. Hence, processes can be checkpointed and recovered independently in such systems.

When processes communicate with each other (as in DSM multicomputer systems), their states are interdependent. In such systems, checkpointing and rollback must guarantee that process states are always *consistent* [Merl78, Chan85] with each other. For example, in a message-passing system, process states are inconsistent if one process state indicates that a specific message was received while the states of other processes indicate that the message was never sent. If processes are checkpointed independently ignoring their interactions, recovery is entirely responsible for ensuring that the set of states to which the processes are rolled back is consistent. In such cases, recovery can lead to the domino effect [Rand78], where processes are rolled back to successively older checkpoints in the search for a consistent state, possibly even to their initial state. The domino effect can be avoided by checkpointing processes at each interprocess communication. But, this approach is expensive. *Message logging* and *coordinated checkpointing* are two approaches that avoid the domino effect without incurring this high cost.

Message logging schemes [Borg83, Powe83, John87, John88, Stro85, Sist89] checkpoint process states periodically, and log all interprocess messages in sequence. A failed process is recovered to a consistent state by restoring its checkpointed state and replaying its logged messages. Coordinated checkpointing [Bari83, Tami84, Koo87] is based on checkpointing ‘‘together’’ consistent states of all the processes that have interacted since their last checkpoint. Recovery restores a consistent state by rolling back all processes that interacted with the failed process.

Both message logging and coordinated checkpointing were first proposed in the context of message-passing multicomputers. Message logging has the advantage that processes can checkpoint and recover independently. However, coordinated checkpointing incurs less overhead since messages are not logged [Fraz89]. In addition, unlike message logging, coordinated checkpointing does not require deterministic process execution.

Error recovery mechanisms have also been proposed for shared memory multiprocessor systems with broadcast interconnection networks [Bern88, Wu90b, Ahme90]. The Sequoia multiprocessor [Bern88] maintains a consistent state in failure-free main memory and periodically checkpoints the system through operating-system controlled cache flushes. The system tolerates single failures by invalidating the cache blocks, and re-executing from the main memory state. The checkpoint-rollback mechanisms presented in [Wu90b] and [Ahme90] assume that the processor caches are reliable. In [Wu90b], independent checkpoint and rollback are facilitated by checkpointing a processor before it provides modified data to another processor. At checkpoint, processor registers and modified cache blocks are flushed to memory. Processors can recover independently by invalidating the cache contents and restoring the processor registers. Ahmed et al. [Ahme90] present schemes implementing coordinated checkpoint and rollback which take advantage of the ability to snoop in a bus-based shared-memory multiprocessor. The scheme uses three special bus lines which identify when data is shared, when a checkpoint is to be taken, and when processors should roll back. The line indicating shared data is used to identify interprocessor communication. The other two special bus lines are used to force coordinated checkpoint and recovery, respectively, of the processors that have communicated.

In order to implement error recovery in a distributed system, interprocess communication must be identified. In a message-passing system, processes communicate through readily identifiable messages. The task of identifying interprocess communication is more difficult in a DSM system, where processes communicate through accesses to the shared address space. Since, unlike bus-based multiprocessors, DSM multicomputers do not communicate through broadcasts, they cannot use snooping to identify interprocess communication. Unlike message-passing systems and bus-based shared memory multiprocessor systems, DSM multicomputers employ a coherence directory to manage the address space. This coherence directory is also susceptible to node failures. Since the coherence directory state is not a part of the state of any application process, the directory state will not be recovered by the mechanisms recovering application process states. Hence, error recovery in DSM multicomputers must incorporate additional mechanisms to handle the failure of the distributed directory.

2.4. Storage Overhead for Coherency Management in DSM Systems

The support of a shared address space on a distributed memory multicomputer requires the maintenance of several tables and directories, as well as the use of a coherence protocol. Tables are required to identify which virtual addresses are mapped to physical memory. For the coherence protocol, coherency directories must maintain the state of blocks (shared, exclusive etc.) in the system as well as the location of copies of these blocks.

Conventional single-level tables incur a very high storage overhead in mapping the large virtual address spaces of large systems. These tables maintain an entry for every virtual page in the system. Hence, their size grows with the size of the virtual address space rather than the size of the physical memory. The size of these tables can be much larger than the physical memory in any node.

A related issue which is also important is the granularity at which shared data is managed. Larger granules require smaller mapping tables. Hence, with larger granules, larger address spaces can be mapped for a given mapping table size. However, since large granules contain more variables, more processors can concurrently require access to a single granule (possibly for different variables). This can lead to increased contention for granules between multiple nodes (false sharing). Also, larger granules incur higher network transfer latency.

Another important design issue is the organization of the copy directory, which maintains the location of copies. If the copy directory maintains the identities of all nodes that possess copies, its size grows linearly with the number of nodes in the system. In large systems, the storage available per node may not be large enough to accommodate the copy directories of all blocks managed by the node. Hence, this straightforward copy directory organization may not be scalable to larger systems. While the copy directory can be eliminated by using broadcasts to invalidate (or update) the block at a write, this will degrade performance since broadcasts on point-to-point networks are usually expensive.

In summary, with a straightforward implementation, the size of the tables and directories required to manage the shared address space increases in proportion to the size of the address space and the number of nodes. However, the sizes of tables and directories that can be maintained is limited by the storage available per node. Since the storage available per node may not grow proportionately with the size of the system (the number of nodes and the size of the address space), the size of the system where DSM can be supported effectively may be limited. Thus there is a need for DSM support mechanisms that can scale up to large systems without requiring significant increases in size of mapping and coherence tables on each node.

2.5. Implementations of DSM Systems

The idea of implementing shared memory in a system with physically distributed memory is not new. Several implementations of distributed shared memory have been reported in the literature. Some implementations support DSM primarily through software, while others provide significant hardware support in the implementation. We summarize the key aspects of some of these implementations here.

The Cm* multiprocessor [Full78] was probably the first system where several processors shared memory that was physically distributed. In the Cm*, when a processor referenced a location that is not available locally, a message is sent to the remote processor cluster where the memory being addressed is physically connected. However, data from the remote memories were not cached locally. While the absence of caching avoided the coherency problem, the remote accesses resulted in poor performance. More recently, several research efforts have investigated DSM as a vehicle for scalable high-performance.

Kai Li [Li86b] proposed the implementation of shared virtual memory on a loosely-coupled network of workstations. In this shared virtual memory, the virtual memory is shared by all workstations and the physical memory in each workstation serves as a cache of this virtual memory. A prototype shared virtual memory system, Ivy [Li88], was implemented, in software, on a network of Apollo workstations under the Aegis operating system. Coherence is maintained in Ivy using directory-based coherence protocols with invalidation. The prototype was used to investigate three directory-based protocols — with centralized managers, with distributed fixed managers, and with distributed dynamic managers. Memory is mapped and maintained coherent in Ivy at the granularity of 1 K byte pages. Sequential consistency is enforced by implementing atomic writes. Memory and process management are both handled in software. Process synchronization is achieved using the underlying eventcount mechanism in the Aegis operating system.

Clouds [Rama89] uses a distributed shared memory abstraction to develop an object-based distributed operating system. The distributed shared memory constitutes the virtual address space of all objects in the system. Each thread of computation in the object-based system progresses as a series of object invocations, each invocation executing in the virtual address space of the object invoked. The global shared memory abstraction provides a convenient mechanism to map the object's address into the address space of the executing thread. Clouds implements the distributed shared memory abstraction in software. The coherence protocol in Clouds differs from coherency protocols used in other DSM systems in two key aspects. First, the coherence protocol does not satisfy write requests by invalidating (or updating) read copies. Instead, the writing thread is suspended until threads with read copies voluntarily discard their copies. The Clouds kernel forces the discard of memory segments that are read-write shared when a critical section is exited. It also uses a deadlock detection algorithm to detect deadlocks due to undiscarded memory segments. Second, the synchronization mechanism is unified with the coherence scheme by transferring a memory segment only when the associated semaphore operation succeeds. The Clouds system also permits memory segments to be acquired without enforcing strong memory coherence.

Both the Paradigm [Cher91] and the DASH [Leno90, Ghar90, Leno93] projects implement DSM on a hierarchical system organized as a set of processor clusters interconnected by a switching network. Each cluster comprises several processors that communicate over a shared bus. The virtual memory in Paradigm is file-based and allows replication of files. The file copies are maintained coherent by software using operating system supported remote procedure calls. At a lower level, Paradigm also supports several levels of caching which are maintained coherent by a directory-based protocol implemented in hardware.

The DASH multiprocessor implements DSM in hardware. The 64-processor prototype connects 16 4-processor bus-based clusters through a 2-D mesh. Coherence is maintained at the cache-line size of 16 bytes, using a snoop-based coherence protocol within each cluster, and using a distributed directory-based protocol across clusters [Leno90]. The directory for any block is maintained in a fixed *home node* of the block. The home node maintains a full-map copy directory (a bit-map with one bit for every node in the system) as well as storage for the block. The Dash system implements a weak model of consistency called release consistency [Ghar90]. This model allows memory accesses of a processor to complete out of order with respect to another processor. The ordering of accesses is only preserved before ‘‘release’’ synchronization operations, and other explicit ordering operations (‘‘fence’’). These operations are visible to the hardware, and the hardware blocks the processor for pending accesses only at such operations.

The Data Diffusion Machine (DDM [Hage92]) implements DSM on a system with a hierarchy of buses through extensive hardware support. In this system, the processor-memory nodes are connected to the lowest level buses. Each processor’s memory is organized as a cache which can dynamically store different sections of the address space. The higher level buses contain only directory information. The directory at each hierarchy tracks information about all hierarchies directly below it. Thus, the directory is distributed through the hierarchy. Coherence is maintained at the cache-line size of 16 bytes, using snoop-based protocols in each bus. A hierarchical search algorithm is used to locate any data item, since no specific manager is assigned for any item. A similar architecture, based on a hierarchy of buses, is also proposed in [Wils87]

KSR1 [Burk92] is a commercial machine built by Kendall Square Research implementing DSM in hardware. KSR1 is similar to the DDM [Hage92] in that the processor's memories are organized as a cache. KSR1 is organized as a hierarchy of rings with the processors connected to the lowest level ring. The KSR1 manages its memory hierarchically by mapping and allocating memory in 16 KB pages, and allowing data to be shared and maintained coherent in 128 byte sub-pages. Unlike the DDM, each page in the KSR1 is associated with an home node, which maintains storage for the page. A hierarchical search algorithm is used to locate any sub-page, as in DDM.

The Alewife [Agar95] architecture supports DSM on a machine with up to 512 nodes are interconnected in a 2-D mesh. A 32-node prototype of this architecture has been constructed. Alewife implements DSM primarily in hardware with some software assist. The architecture uses a distributed directory-based coherence protocol to maintain each 16-byte memory line coherent. Alewife is similar to DASH [Leno93] in that each memory block is assigned a *home node* which maintains the directory and storage for the block. A distinguishing feature of Alewife is that the copy directory is extended in software [Chai91]. The software-extended scheme LimitLESS effectively implements a full-map copy directory by maintaining the copy directory in software when the number of copies exceeds the small number of entries (two to five) in the hardware copy directory. Alewife implements the sequential consistency model. The hardware provides support for fine-grain synchronization using full/empty tag bits with each memory word, as well as for passing messages between processes. Multithreading and software prefetching are used to hide the latency of remote accesses.

The FLASH [Kusk94] architecture being designed at Stanford builds on their earlier DASH [Leno93] architecture. The integration of support for message passing is a key additional goal for the FLASH architecture. FLASH uses a 2-D mesh interconnection of nodes (not clusters). The DSM architecture in FLASH retains most of the features from DASH. A key difference is the use of dynamic pointer allocation in the copy directory storage. Unlike DASH, FLASH does not maintain the location of copies in each directory entry. Rather, a common store is used to maintain all the copy directories. FLASH also differs slightly from DASH in the low-level implementation of the coherence protocol.