CHAPTER 2

BACKGROUND

DSM system is an architectural approach designed to overcome the scaling limitations of symmetric shared memory multiprocessors while retaining a shared memory model for communication and programming. DSM multiprocessors achieve this by using a memory that is physically distributed but logically implements a single shared address space, allowing the processor to communicate through, and share the contents of the entire memory [3, 7, 10]. DSM multiprocessors have the same basic organization as the machines in Fig 2.1.

Figure 2.1 Structure of Distributed Shared Memory
Sharing data is an essential requirement of any distributed system. Distributed system it stands for a multi-computer architecture in which each node is an independent machine connected to each other through a network. Sharing data in a multi-processor architecture is relatively easier, since all of the nodes share the same system bus and therefore have a uniform view of the physical memory [4, 23, 26]. On the other hand a multi-computer system does not enjoy such hardware privileges. So, sharing data becomes a problem which has to be tackled in the software (either inside the Operating System or as a user-level application) and not in hardware as in multi-processor systems. Traditional methods of data sharing viz. message passing via sockets are not appealing from a programmer’s perspective, in which he or she has to explicitly take care of the networking issues [6, 9, 20, 31]. A DSM provides an abstraction to the programmer of a uniform shared memory located across different machines.

Since a DSM system involves moving of data from one node to another which are on typical networks, performance is an important criterion in the design of a DSM system. Just as is the case with multi-processor systems, since same copies of data might reside on different nodes, consistency between these copies is also another major issue.

2.1 Designing Issue of DSM System

Any DSM system has to rely on the message passing technique across the network for data exchange between two computers. The DSM has to present a uniform global view of the entire address space (consisting of
physical memories of all the machines in the network) to a program executing on any machine. A DSM manager on a particular machine would capture all the remote data accesses made by any process running on that machine [24, 29, 30]. A design of a DSM would involve making following issues:

- Where, with respect to the Virtual Memory Manager does the DSM operate?
- What kind of consistency model should the system provide?
- What should be the granularity of the shared data?
- What kind of naming scheme has to be used to access remote data?

2.1.1 Relationship with Virtual Memory Manager

The issue that arises here is from the fact that the virtual memory on a particular machine and the distributed shared memory (which is nothing but an abstraction) are inherently different. But the aim is to provide a uniform view of both to the user applications.

A DSM manager can be implemented parallel to the virtual memory manger so that in case of a page fault the virtual memory manger can decide if the missing page is available locally or requires a remote access. For remote accesses the virtual memory manger can ask the DSM manager to fetch the page. Moreover the virtual memory manger also has to perform book-keeping operations like maintaining
the page table entries for the remote pages [10, 17]. Such kind of design is closest to the shared memory in multi-processor systems.

An alternative design would be in which the DSM manager is built on top of the virtual memory manager such that the virtual memory manager is not aware of different kinds of memory accesses. The language provides capabilities such that certain data segments are fetched from remote addresses while rest are accessed locally. Performance-wise, the former approach is better because it eliminates one step of writing the data into the memory buffers.

2.1.2 Choice of Consistency Model

Consistency is a very important parameter in a design of DSM. As discussed earlier, the possible replication of shared data across multiple nodes mean that different machines have different copies of the data. Thus the DSM has to maintain consistency in the state of these copies of the same data. This problem is similar to cache coherency in multi processors [1, 13, 14, 25]. Several solutions which are also prevalent in other scenarios are proposed for this which can be broadly classified as

**Write-all scheme:** In this scheme, if a data is modified at some node, the updated copy of the data is shipped to all the nodes which are currently holding a copy of that data. This scheme is simple to implement and most reliable, but it is expensive in terms of network traffic because the whole data has to be shipped across multiple clients.

**Write-invalidate scheme:** In this scheme, if a data is modified at some node, instead of sending the entire data across, the host sends an
invalidate message to all other machine holding a copy of that data. Thus this scheme puts lesser load on the network and hence performs better. Whenever a machine tries to access an invalid data, the DSM manager fetches the valid data from the host.

**Lock-based synchronization:** The above two schemes enforce “weak” consistency in the sense that they do not guarantee that events are globally ordered. With locks or some similar synchronization primitive, a release consistency can be enforced. This means that a node accessing some data can acquire a lock such that no other node can access it at that time. Once the node comes out of the critical section, it updates all the nodes which have the copy of that data. Most of the implementations use the first two schemes and usually keep synchronization separate from consistency model. In our design also we have allowed the first two schemes to be implemented. It is important to note that the choice of the consistency protocol will affect consistency of the system as well as the performance of the system. Unfortunately both these parameters act opposite to each other so there is a trade-off between performance and the consistency of the system.

### 2.1.3 Choice of Granularity

Granularity refers to the size of a data unit in which it exists in the distributed shared memory. This is an important decision which essentially governs the design of a DSM. The immediate successor of shared memory from multiprocessor world would have a page as the unit for data transfer [2, 27].
2.1.4 Naming Scheme

When a process wants to access remote data it has to know on which machine does the data reside and fetch it from there. Since all data (or at least the shared one) is visible to all the machines, there has to be a unique naming mechanism to avoid any conflicts. One possible solution is to have a logical global address space. The virtual memory manager at each node performs the translation of the logical address to get the location of the data segment on a remote machine. But such an approach would not be useful if the granularity of shared data is less than a page. In such a case the calling process will have to possess explicit knowledge of the remote location of the data which it wants to access.

2.2 Implementation of DSM System

The level where the DSM mechanism is implemented is one of the most important decisions in building a DSM system: it affects both programming and overall system performance and cost. To achieve ease of programming, cost-effectiveness, and scalability, DSM systems logically implement the shared-memory model on physically distributed memories [5, 28]. Because the DSM’s shared address space distributes across local memories, a lookup must execute on each access to these data, to determine if the requested data is in the local memory. If not, the system must bring the data to the local memory. The system must also take an action on write accesses to preserve the coherence of shared data. Both lookup and action can execute in software, hardware, or the
combination of both. According to this property, systems fall into three groups: software, hardware, and hybrid implementations. The choice of implementation usually depends on price performance trade-offs. Although typically superior in performance, hardware implementations require additional complexity, which only high-performance, large-scale machines can afford. Low-end systems, such as networks of personal computers, based on commodity microprocessors, still do not tolerate the cost of additional hardware for DSM, which limits them to software implementation. For mid-range systems, such as clusters of workstations, low-cost additional hardware, typically used in hybrid solutions, seems appropriate.

2.2.1 Software DSM Implementation

Until the last decade, distributed systems widely employed message-passing communication. However, this appeared to be much less convenient than the shared-memory programming model because the programmer must be aware of data distribution and explicitly manage data exchange via messages. A software DSM implementation emerged in the mid 1980s and is to provide the shared memory paradigm to the programmer on top of message passing. Generally, this can be achieved in user-level, run-time library routines, the OS, or a programming language [5, 15, 21, 22, 33]. Larger grain sizes on the order of a kilobyte are typical for the software DSM implementations because their DSM management is usually supported through virtual memory. Thus, if the requested data is absent in local memory, a page-fault handler will retrieve the page from either the local memory of another node, or disk of requesting node or another node. Typical
representatives of this approach are **IVY, Mermaid, Munin, Midway, TreadMarks, Orca, Linda, Mirage** and **Clouds**. IVY was one of the first proposed software DSM solutions.

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

### 2.2.2 Hardware DSM Implementation

Hardware-implemented DSM mechanisms ensure automatic replication of shared data in local memories and processor caches, transparently for software layers. This approach efficiently supports fine-grain sharing. The nonstructured, physical unit of replication and coherence is small, typically a cache line. Consequently, hardware DSM mechanisms usually represent an extension of the principles found in cache-coherence schemes of scalable shared memory architectures. This approach considerably reduces communication requirements because finer sharing granularities minimize the detrimental effects of false sharing and thrashing. Searching and
directory functions implemented in hardware are much faster than with software-level implementations, and memory access latencies decrease [5, 15, 21, 22, 32]. But hardware DSM implementation usually requires complicate design and verification for advanced coherence maintenance and latency reduction techniques, so hardware DSM is often used in high-end machines where performance is more important than cost. Typical representatives of this approach are Memnet, Dash, KSR1, DDM, SCI, Merlin and RMS. Memnet among them was one of the earliest hardware DSM systems.

2.2.3 Hybrid Level DSM Implementation

During the evolution of this field, the research community proposed numerous entirely hardware or software implementations of the DSM mechanism. However, even in entirely hardware DSM approaches, there are software-controlled features explicitly visible to the programmer for memory reference optimization-for example, prefetch, update, and deliver in Dash; and prefetch and poststore in KSR1. Many purely software solutions, however, require some hardware support such as virtual memory management hardware in IVY and ECC in Blizzard-E. As to be expected, neither the entirely hardware nor entirely software approach has all the advantages [5, 31]. Therefore, it is quite natural to employ hybrid methods, with predominantly or partially combined hardware and software elements, to balance the cost-to-complexity trade-offs. The “Hybrid-level implementations” sidebar summarizes some of these tradeoffs.
2.3 Classification of DSM System

DSM systems can be classified into three broad categories.

- Page-based DSM – in which the unit of data sharing is a memory page.
- Shared variable based DSM – in which the unit of data sharing is a variable.
- An object based DSM – in which the unit of data sharing is an object.

The choice of objects as units of granularity over a page or shared variables is because of the modularity and flexibility offered by objects. Moreover, objects eliminate false sharing. Another reason is that integration of object based DSM with the object oriented languages is easy to achieve. The design also provides flexibility in terms of consistency models used by allowing the user applications to specify under what consistency scheme they want a particular object to be shared.

2.3.1 Page Based DSM System

As the name suggests, this DSM has a memory page as the unit of data sharing. Page based DSM closely emulate the shared memory in multiprocessor sphere. The entire address space (spread over all the computers) is divided into pages. Whenever the virtual memory manager (VMM) finds a request to an address space which is not local, it asks the DSM manager to fetch that page from the remote machine. Such kind of page fault handling is simple and similar to what is done
for local page faults. To improve performance, most implementations do replication of the pages so that same page does not have to be transferred again and again. This especially improves performance for read only pages. Keeping multiple copies of same page lead to the issues of consistency between these copies. If pages are not replicated, achieving sequential consistency is not difficult. But for replicated copies page based DSM generally follow the same scheme as cache coherency schemes for shared memory in multiprocessors [2, 5, 15]. Naming issues in page based DSM are usually addresses by having a global address space and the virtual memory manger identifying page faults as local or remote. The virtual memory manger itself or a separate DSM manager then fetches the remote page by translating the logical address.

The disadvantage of a page based DSM is that it is not very efficient performance-wise. Most of the time the actual data being shared is much smaller than the page size hence there is lot of wastage in terms of shared space. Moreover it also leads to false sharing. Most of the programming languages organize data into smaller units like a package or object or some data structure. Hence it would make more sense to share data in these units which is the idea behind the next two types of implementations.

2.3.2 Shared Variable Based DSM System

Shared variable based DSM present a more structured approach towards sharing of data between multiple machines. The basic idea is to let the applications decide as to which variables are to be shared.
The DSM manager will maintain a database of the shared variables present in the logical shared memory. The primary difference from the page based approach is that each shared variable is individually managed thereby eliminating the possibility of false sharing. This approach expects the programmer to explicitly declare which variables are to be shared globally. These variables are then stored by the DSM manager of that machine. Any process on a different machine can access this variable by requesting through that machine’s DSM manager [2, 5, 15]. The advantage which the shared variable DSM holds is that it provides a more structured way of sharing data between multiple computers. Most of the implementations favor the write invalidate scheme because it ensures weak consistency without affecting the performance too much.

2.3.3 Object Based DSM System

The final type of DSM is the object-based DSM in which an object is the unit of data sharing. The primary difference of this approach from the shared variable based approach is that in object based DSM all data is encapsulated as objects. Thus the programmer does not have to explicitly declare the objects which he/she wants to share globally. Moreover, since the data is accessible only through the methods defined, this scheme provides protection for the data which is not found in the shared variable approach. Each object contains the actual data and methods which can be invoked to obtain or set the data. Note that the direct access of data is not allowed. Instead the process can invoke the methods available for that object [2, 5, 31]. Depending on the location of the object with reference to the calling process, the
method can be invoked locally or remotely. The runtime and/or the Operating System take care of invoking the method without the application being aware of the object’s location. This approach enjoys the advantages of the shared variable approach in terms of managing each shared object separately. The consistency schemes can be applied similarly to the shared variable DSM.

The disadvantage of this approach is that all the data is forced to be encapsulated with objects, thereby putting an extra overhead in invoking methods to get or set the actual data. This overhead would be completely unnecessary for data which is not being shared. Hence this approach can be thought of as a trade-off between ease of programming and performance. One thing to note here is that object based DSM and shared variable based DSM are language provided features whereas the page based DSM requires the support of Operating System.

### 2.4 Challenges in DSM Architecture

The challenges in DSM system can be categorized into three parts based on the basic design choices of a DSM i.e. Implementation level, Memory Consistency Model and Communications.

#### 2.4.1 Implementation Level

As reported in there exists a number of alternatives for determining the level of abstraction where a DSM implementation is to be deployed:
from systems that maintain consistency entirely in hardware to those that exist entirely in software. Considering our requirement of a middleware solution spanning a heterogeneous environment, we cannot guarantee homogenous hardware support.

Software DSM systems can be split into three classes: page-based; variable-based; object-based. In each of these approaches our concern is where, and how, transparency of remote access is introduced:

**Page-based** - uses the memory management unit (MMU) to trap remote access attempts.

**Variable-based** - requires custom compilers to add special instructions to program code in order to detect remote access.

**Object-based** - special programming language features are required to determine when a remote machine’s memory is to be accessed.

Although variable-based may be possible as a certain degree of platform independence is provided, the compile time requirements restrict the ability to introduce new types during runtime. This leaves the possibility of object based solutions. Even though this approach appears to be tightly coupled to a programming language, it still affords a degree of platform independence beyond that offered by page-based systems. In addition, if the supported language allows the introduction of new types during runtime, then such a quality may be introduced into the DSM.
2.4.2 Memory Consistency Model

A memory consistency model may be considered a contract between those elements of a system that access memory and the memory itself. Choosing an appropriate memory consistency model is a trade-off between minimizing access order constraints and the complexity of the programming model: assuring strict consistency of the DSM for all accesses is achieved at the expense of performance as increased message passing coupled with the locking of resources is required [25, 27]. For the type of DSM that we are concerned with the single, most important, design choice affecting scalability is where, physically, to store memory. If such a storage space was consigned to a single location there is a greater potential for memory contention issues to arise, commonly resulting in bottlenecks. Furthermore, data may be geographically separated from an accessing process to such an extent that latency of message exchange may be sufficiently high as to hinder performance.

To afford scalable solution for DSM a compromise must be reached regarding the consistency of memory against the performance incurred from using such memory. One design option would be to replicate shared memory across the DSM, affording local access when appropriate, while seeking to maintain a degree of consistency across replicas to ensure successful application operation [2, 5, 13].

For example, distributed transactions may be employed to ensure state changes of one or more memory replicas are achieved in a consistent manner. Alternatively, group communication protocols
offering total ordering and atomic multicast may be used to ensure all state changes are viewed in a mutually consistent way. As these techniques themselves come with a performance cost, they must be used only when needed (i.e., during state change events).

### 2.4.3 Communication Channel

In a DSM where nodes and processes are separated over some geographic distance the choice of communication medium is limited. In fact, to ensure availability of such a system for the widest audience a developer must rely on standard protocols such as those governing public access network traffic (e.g., TCP/IP for the Internet). As existing middleware provides a convenient, easy to use, communication abstraction for developers over such protocols, it would be folly not to exploit such middleware in DSM. As RPC is the primary method for enacting communication within existing middleware, one must consider RPC as a suitable communication mechanism on which to construct DSM. Using RPC requires a communication stream between sender and receiver to be initialized and maintained either throughout a call or for as long as RPC participants hold references to each other (usually sender holding reference to receiver). This approach to tightly coupled communication is satisfactory for small numbers of participants but does not scale to support hundreds and thousands of participants.

However, in a DSM system it is quite conceivable that hundreds, if not thousands, of clients may at some point in an application’s execution require access to a memory location.
Alternatively, a client may require access to many hundreds, if not thousands, of memory locations. RPC used in such a manner is not scalable, as the management of connections at both client and server side would be a substantial drain on available processing resources. This scalability problem has been tackled by middleware developers via abstracting away the one-to-one communication model of RPC in favor of a many-to-many solution. This is achieved by providing messaging services that decouple sender from receiver (e.g., sender does not know who is receiving its messages.

2.5 Memory Coherence in DSM Architecture

In parallel computer systems however there are two or more processors working at the same time, so there is the possibility that the processors will all want to process the same value at the same time. Provided none of the processors updates the value then they can share it indefinitely, but as soon as one updates the value, the others will be working on an out-of-date copy. Some scheme is required to notify all processors of changes to shared values; such a scheme is known as a "memory coherence protocol". Coherence defines the behaviour of reads and writes to the same memory location [1, 12, 14, 16].

There are two type coherence protocols one is write-invalidate and other is write-update. Write-invalidate protocol invalidates all copies of operation if any processor wants to writes. Write-update protocol updates all writes to all processor if any processor wants to write.
2.5.1 Write-Invalidate Coherence Protocol

In Write-invalidate protocol a processor perform an invalidate bus transaction before writing the data in order to ensure that it has the only valid copy of the data block.

![Diagram of Write-Invalidate Protocol](image)

**Figure 2.2** Write-Invalidate Protocol

When a write operation occurs, the write-invalidate strategies signal the holders of a shared block to invalidate its copy as shown in the fig2.2. It sends the invalidate message to all processor even if they may no longer to use the data.

2.5.2 Write-update Coherence Protocol

Write-update protocol immediately put writes performed by a processor on the bus, thus updating all copies of the data as shown in fig.2.3. In write-update protocol the updation is started from the block master processor and proceeds to the copy list chain. The write
operation is completed when the last processor sends an acknowledgement to the originator of operation.

A Write-invalidate protocol is easier to implement and cause less network traffic during the invalidation, but it is inefficient if many processors frequently access the object. Where as Write-update protocol cause more bus traffic in the network but allows multiple writes in a single bus transaction. A comparative analysis of message passing in Write-invalidate and Write-update coherence protocol is given in table 2.1.

**Table 2.1** Massages Passing in Write-Invalidate/Update Coherence Protocol

<table>
<thead>
<tr>
<th>Mode</th>
<th>Outgoing</th>
<th>Incoming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-invalidate</td>
<td>read(data)[wb]</td>
<td>read(data)</td>
</tr>
<tr>
<td></td>
<td>read(data)</td>
<td>read(data)[wb]</td>
</tr>
<tr>
<td></td>
<td>send(data)</td>
<td>Invalidate</td>
</tr>
<tr>
<td>Write-update</td>
<td>read(data)[wb]</td>
<td>read(data)</td>
</tr>
<tr>
<td></td>
<td>update-ack</td>
<td>read(data)[wb]</td>
</tr>
<tr>
<td></td>
<td>send update(data)</td>
<td>update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>update[wb]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>update-ack</td>
</tr>
</tbody>
</table>

wb: write back, ack: acknowledgement
In write-invalidate protocol the processor invalidate itself by reading data comes from other processor and pass the message to next processor. Whereas in write-update protocol the processor update itself whatever the change occur in write operation and send the message to next processor for updation.

2.6 Memory Consistency in DSM Architecture

On a DSM system the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values. The programming interfaces to DSM systems may differ in a variety of respects.

![Memory Consistency in DSM System](image)

*Figure 2.4 Memory Consistency in DSM System*

The memory consistency model refers to how updates to distributed shared memory are rejected to the processes in the system. The most
The intuitive model of DSM is that a read should always return the last value written unfortunately the notion of the last value written is not well defined in a distributed system [8, 11]. The consistency model of a DSM system specifies the ordering constraints on concurrent memory accesses by multiple processors, and hence has fundamental impact on DSM systems programming convenience and implementation efficiency. In Figure 2.4, the memory system of DSM system consists of local MCS processes at each processor $p_i$, which use local memory and communication with each other using a message passing communication system.

As the memory consistency model influences many aspects of system design, including the design of programming languages, compilers, and the underlying hardware, in order to enhance performance, multiprocessors tend to implement sophisticated memory structures. These memories may replicate data through constructs such as caches and write buffers. Furthermore, the time required to access a data object may vary between processes and between objects. Any of these architectural features allow processes to have inconsistent views of memory, which, in turn, can result in unexpected program outcomes. Under these circumstances the DSM system may use the time, the processor and data to propagate updates to process and objects. The use of time updates on a shared data objects by one processor are made visible to the all processor at the time when the data object is to be read by other processor. The use of processor updates on a shared data objects by one processor to other processor which is the next one to read the shared data object. The use of data propagates to all processor which is sharing the data.
Generally memory consistency models for a DSM system specify how memory behaves with respect to read and write operation from multiple processors. But synchronization of process is also needed. So the memory consistency model of DSM system can be categorized into parts one which is based on only read and write memory operation called as uniform memory model and the other which is based on synchronization operation also called hybrid model. The synchronization operations are mapped to corresponding operations provided by concurrency control.

2.6.1 Classification

The memory consistency model of DSM system is based on the ordering of writing to different memory locations as well as accessing the different memory location. The ordering may be strict, weak or relaxed ordering. Due to this ordering the DSM system can be classified as Strong, Weak and Relaxed Model.

2.6.1.1 Strong Memory Consistency

The strong or strict consistency model requires that any read on a data item returns a value corresponding to the most recent write on that data item. This is what one would expect from a single program running on a uniprocessor. A problem with strong consistency is that the interpretation of ‘most recent’ is not clear in a distributed data store. A strong interpretation requires that all clients have a notion of an absolute global time. It also requires instant propagation of writes to all replicas. Due to the fact that it is not possible to achieve absolute global time in a distributed system and
the fact that communicating between replicas can never be instantaneous, strong consistency is difficult to implement in a distributed memory. In a DSM system, storage accesses are strongly ordered if

1. Accesses to global data by any one processor are initiated, issued and performed in program order, and if

2. At the time when a STORE on global data by processor 1 is observed by processor K, all accesses to global data performed with respect to 1 before the issuing of the STORE must be performed with respect to K.

Initially \( X = Y = 0 \)

\[
\begin{array}{ccc}
    P1 & P2 & P3 \\
    X = 1 \\
    x_2 = X \\
    y_2 = Y \\
    Y = 1 \\
    x_3 = X
\end{array}
\]

Result: \( x_2 = 1, y_2 = 0, x_3 = 0 \)

Figure 2.5 Strong Ordering

\( X \) and \( Y \) are shared variables initially present in processor caches with value 0. \( x_2 \) and \( y_2 \) are local variables, possibly registers, belonging to \( P2 \), and \( x_3 \) is local to \( P3 \). Let all the accesses be performed in program order. It is possible for the write on \( X \) to be propagated to \( P2 \) before \( P3 \), and before the write on \( Y \) is propagated to \( P2 \). Thus reads issued by \( P2 \) can return a 1 for \( X \) and a 0 for \( Y \), making it appear that \( X \) was written before \( Y \). \( P3 \) can still
return a 0 for X, making it appear that Y was written before X. Therefore, there does not exist any total ordering of memory accesses for this execution. However, none of the sufficient conditions for strong ordering (or concurrent consistency) are violated.

2.6.1.2 Weak Memory Consistency

The weakest form of consistency only guarantees that accesses to a given memory location are strongly ordered. Thus writes to different memory locations may be observed to occur in different order by different processors. Each read of a memory location is guaranteed to obtain the most recently written value. In practice, this is of course inadequate, since no guarantee of synchronization is possible with such a system. Thus some stronger form of ordering is provided conditionally, only for certain variables. These typically are explicit synchronization variables, such as that specified in a Test-and-Set instruction. An implementation technique for guaranteeing consistency at a specified point is the fence primitive. When a fence operation is initiated by a processor, execution is blocked until all pending write (and possibly, read) operations have completed. If this operation is applied at the time of a synchronization operation only it is know as a weakly ordered system. The use of a fence can potentially result in higher performance because write operations emanating from a single processor can be overlapped with reads and ensuing writes, except when a fence is encountered. Also, truly parallel writes can occur. Unfortunately, it is necessary for the programmer to state explicitly
whenever a fence must be inserted, though such a requirement can be inferred, for example, whenever a Test-and-Set instruction is encountered. Problems arise, of course, when such points are not explicitly identified.

2.6.1.3 Relaxed Memory Consistency

A model that is close to strong consistency, but that is possible to implement in a distributed data store, is the relaxed consistency model. In this model the requirement of absolute global time is dropped. Instead all operations are ordered according to a timestamp taken from the invoking client’s loosely synchronized local clock. Relaxed consistency requires that all operations be ordered according to their timestamp. This means that all operations are executed in the same order at all replicas. Note that although it is possible for a distributed data store to implement this model, it is still very expensive to do so. For this reason relaxed consistency is rarely implemented.

2.6.2 Type of Memory Consistency Model

DSM allows processes to assume a globally shared virtual memory even though they execute on nodes that do not physically share memory. The DSM software provides the abstraction of a globally shared memory in which each processor can access any data item without the programmer having to worry about where the data is or how to obtain its value. In contrast in the native programming model on networks of workstations message passing the programmer must
decide when a processor needs to communicate with whom to communicate and what data to send. For programs with complex data structures and sophisticated parallelization strategies, this can become a daunting task. On a DSM system, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values. In addition to ease of programming, DSM provides the same programming environment as that on hardware shared memory multiprocessors, allowing programs written for a DSM to be ported easily to a shared memory multiprocessor. Porting a program from a hardware shared memory multiprocessor to a DSM system may require more modifications to the program because the much higher latencies in a DSM system put an even greater value on locality of memory access.

The programming interfaces to DSM systems may differ in a variety of respects. We focus here on memory structure and memory consistency model. An unstructured memory appears as a linear array of bytes whereas in a structured memory processes access memory in terms of objects or tuples. The memory model refers to how updates to shared memory are rejected to the processes in the system. The most intuitive model of shared memory is that a read should always return the last value written; unfortunately, the notion of the last value written is not well defined in a distributed system. A more precise notion is sequential consistency whereby the memory appears to all processes as if they were executing on a single multiprogrammed processor. Basically there are two categories of memory consistency model: Uniform and Hybrid Model.
2.6.2.1 Uniform Memory Model

A uniform memory model consider only read & write memory operation to define consistency condition. It is the combination of strong & relaxed memory model. Uniform memory model generally follow the strong ordering but in extant it gives some relaxation in ordering of the read operation. The followings are model of uniform memory model

i) Atomic consistency (AC)  
ii) Sequential consistency (SC)  
iii) Casual consistency (CC)  
iv) Processor consistency (PC)  
v) PRAM  
vi) Cache consistency  
vii) Slow memory

The first two AC & SC is the strong consistency whereas the other one are relaxed consistency. The PRAM model is evolved by combining the processor consistency as well as casual consistency and slow memory model is the combination of PRAM and Cache Coherence.

2.6.2.2 Hybrid Memory Model

A hybrid model consider read, write as well as synchronization operation. Processes want to restrict the order on which memory
operation should be performed. Using this fact, hybrid memory model guarantee that processors only have a consistent view of the memory at synchronization time. This allows a great overlapping of basic memory accesses that can potentially lead to considerable performance gain. Following memory consistency model of distributed shared memory have taken for hybrid model:

i) Weak consistency model (WC)

ii) Release consistency model (RC)
    a) Eager release consistency (ERC)
    b) Lazy release consistency (LRC)

iii) Entry consistency (EC)

iv) Scope consistency (ScC)

v) View based consistency (VC)

Defining the Hybrid model is more complex then the uniform memory model because of memory operation, order of operation & relate to operation.

2.7 DSM System – Some Implementation

A DSM system must automatically transform shared memory access into interprocessor communication. This requires algorithm to locate and accesses shared data, maintain coherence and replicate data. The implementation of DSM system depends upon the algorithm chosen, coherence protocol and consistency model. The algorithms may by Single
Read Single Write (SRSW), Multiple Read Single Write (MRSW), and Multiple Read Multiple Write (MRMW).

Table 2.2  Consistency and Coherency implementation

<table>
<thead>
<tr>
<th>System Name</th>
<th>Type of Algorithm</th>
<th>Coherence protocol</th>
<th>Consistency Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVY</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>Mermaid</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>Munin</td>
<td>MRSW/MRMW</td>
<td>Delayed update, Invalidate</td>
<td>Release</td>
</tr>
<tr>
<td>Midway</td>
<td>MRMW</td>
<td>Update</td>
<td>Processor, Release</td>
</tr>
<tr>
<td>Bilzzard</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>Mirage</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>Linda</td>
<td>MRSW</td>
<td>Implementation Dependent</td>
<td>Sequential</td>
</tr>
<tr>
<td>Memnet</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>DASH</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Release</td>
</tr>
<tr>
<td>SCI</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>KSR1</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>DDM</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>Merlin</td>
<td>MRMW</td>
<td>Update</td>
<td>Processor</td>
</tr>
<tr>
<td>RMS</td>
<td>MRMW</td>
<td>Update</td>
<td>Processor</td>
</tr>
<tr>
<td>Flash</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Release</td>
</tr>
<tr>
<td>Plus</td>
<td>MRMW</td>
<td>Update</td>
<td>Processor</td>
</tr>
<tr>
<td>Alewife</td>
<td>MRSW</td>
<td>Invalidate</td>
<td>Sequential</td>
</tr>
<tr>
<td>Galactica Net</td>
<td>MRMW</td>
<td>Update/Invalidate</td>
<td>Processor, Multiple</td>
</tr>
</tbody>
</table>

Table 2.2 shows comparative analysis of implementation of some popular DSM system based on Algorithm, Consistency model and
Coherency Protocol. Mostly system which uses single write operation uses sequential consistency for its strict ordering. As invalidate protocol invalidates all writes of all processor when any processor executes writes operation. So for single write operation the invalidate protocol is very much suitable. For multiple write operation update protocol is used as instead of invalidating all writes, update protocol updates all processor about the write operation.

By using data replication and write-update coherence protocol the parallelism of DSM system can be improved. Replication of data multiple can be done if there are multiple write operation in the system. So MRMW algorithm is used for data replication. For MRMW algorithm Write-update protocol is used because it provides multiple writes on a single bus. Here we will discuss some DSM system which is implemented by Processor Consistency, MRMW algorithm and Write-Update Coherence protocol.

2.7.1 Midway

Midway supports multiple consistency models (processor, release and entry) that can change dynamically in the same program. It consists of three components: a set of keywords and function calls used to annotate a parallel program, a compiler that generates code that marks shared data a dirty when written to, and a runtime system that implements several consistency models. For all consistency models, Midway uses an update mechanism [5, 15].
**Issues of importance**

Multiple consistency models that can be dynamically changed may be very convenient for some applications with specialized requirements for consistency semantics closely related to particular algorithm. Anyway, entry consistency may be hard to use, and it can be very complicated to port programs with this model, since consistency requirements should be well understood and made explicit by the programmer.

### 2.7.2 Merlin

Merlin (Memory Routed Logical Interconnection Network) represents a reflective memory based interconnection system using mesh topology with low latency memory sharing on the word basis. Merlin transmits the word packet through the interconnection network to all shared copies in other local memories. It supports two types of sharing in H/W: synchronous which updates to the same region are routed through a specific canonical cluster and rapid in which updates are propagated individually by the shortest routes [5, 15].

**Issues of importance**

Merlin is a simple and efficient solution for DSM. Only about 10% of Merlin interface is processor specific, so it can be implemented for a wide variety of commercially available processors. In most cases, communications overlap computation, since all copied words are routed by interface without any host cycle. Although intended to provide the shared memory paradigm in heterogeneous environments,
Merlin does not resolve the problem of format conversions for different representations of data, and it exclusively uses static information about data sharing, independently to dynamic application behavior.

2.7.3 RMS

RMS consists of a lower number of minicomputer clusters connected by the RM bus, a write-only bus because traffic on it only consists of word based distributed write transfers (address + value of the data word). Later enhancements (Memory Channel) also allow for block-based updates. The replication unit is an 8-KB segment. Segment can be treated as windows that can be open or closed. A replicated segment can map to different addresses in each cluster [5, 15].

Issues of importance

The RMS provides advantages of the symmetric shared memory models, without incurring the usual access latencies. However, this mechanism is not sequentially consistent and an explicit synchronization is needed. The parts of address space which are configured as the reflected memory have to be disabled for placing in the cache.

2.7.4 Plus

A typical hybrid approach achieves data replication and migration from a shared virtual and migration from a shared virtual address space
across the cluster in software, while implementing coherence management in hardware. In plus software handles data placement and replacement in local memory. However memory coherence for replicated data resides on the 32-bit word b a nondemand, write-update protocol implemented in H/W. replicated instances of a page are chained into an ordered singly linked list, headed with master copy, to ensure the propagation of updates to all copies [2, 5, 15]. To optimize the synchronization, it provides a set of specialized interlocked read-modify-write operation called delayed operations.

**Issues of importance**

There is no overhead in referencing the remote memory locations. The memory consistency protocol is carried out independently, since it is implemented in hardware. Performance evaluation shows that the concept of replicated data (as a consequence of write-update mechanism) increases the processor utilization substantially, due to the decrease in read latency. For a higher level of replication, the system can get saturated with frequent updates, slowing down the useful computation.

### 2.7.5 Galactica Net

It replicates pages from the virtual address space on demand under control of virtual memory software, implemented in the Mach OS. It also provides hardware support for a virtual memory mechanism, realized through a block transfer engine that can rapidly transfer pages in reaction to page faults. A page can be in one of three states: read
only, private, and update, denoted by tables maintained by the OS [5, 15]. A write-update protocol, implemented entirely in hardware, keeps the coherence for writeable shared pages (update mode). All copies of a shared page in the update state are organized in a virtual sharing ring-a linked list used for forwarding of updates.

**Issues of importance**

Since the unit of sharing is the page, the only special hardware required on the node is the standard memory management hardware. Because of the update mechanism, for some applications, broadcast of excessive updates can produce a large amount of traffic in the system.

2.8 **Our Approach**

In the research we have given an extensive view of different memory architecture used in parallel computer, mainly we have covered the following architecture

- Shared memory
  - Distributed Shared Memory (DSM)
  - Virtual Shared Memory (VSM)
  - Shared Virtual Memory (SVM)
- Distributed Memory

After analyzing in details of all above memory architecture, we choose DSM for our research work because it combines the advantages of
Shared memory as well as the Distributed Memory. The performance and the programmability of the DSM system depend upon the Memory Consistency Model and Memory Coherency. As very immense scope of Memory Consistency Model, we choose to continue our research work with it.

The consistency model of a DSM system specifies the ordering constraints on concurrent memory accesses by multiple processors. Lots of Consistency Model are defined by a wide variety of source including architecture system, application programmer etc. We go through near about the entire memory consistency models of DSM architecture. We analyzed that the memory models can be combined in two ways, uniform memory model which consider only ‘Read’ & ‘Write’ Memory operation and hybrid model which also consider synchronization operation. For uniform memory model Atomic, Sequential, Casual, Processor, PRAM, and Slow memory consistency model of DSM System is taken and for Hybrid Model we have taken take Weak, Entry, Release, Scope & View Consistency model of DSM System.

In our research we design a structural memory model of uniform memory models as well as hybrid memory model in a unified framework and define each consistency model according to our designed structural memory model with respect to the DSM system. We compare and relate each consistency model and analyze the performance of our proposed model.

For maintaining the memory consistency in DSM System, we propose a framework with the help of write-update coherence protocol.
The framework provides high degree of data replication. For achieving high degree of data replication we implement MRMW algorithm and processor consistency model into our proposed framework. Due to this the framework increases the degree of parallelism. The research also includes the memory management technique for paging of framework. Implementing DSM framework with paging scheme leads to the false sharing and high cost associated with virtual memory operation. Our research analyzes the effect of granularity and finds the solution of false sharing. We also analysis the different overheads of DSM framework with respect to the page-size and virtual memory operation.
REFERENCE


