The parallelism can be improved in DSM system if more than one operation is executed by different processor at the same time. So in this case replication of data is required and more than one write request is to be done. But multiple write-requests create some inconsistent result when the location is same for all write. Now it becomes important to maintain the consistency of memory. The framework described in the previous chapter solves this problem by using memory coherence protocol and memory consistency model. The framework finds the address of updates and propagates it to all the nodes which share location. By this activity, processors who are sharing same location come to know about any write-request and hence the consistency of memory is maintained.

6.1 Data Replication

Data replication can be implemented in different ways, influencing both performance and the programming model. Many systems that use replication apply an invalidation or updation scheme where the replicas are invalidated or updated when a write operation occur. We have taken updation method for the replication. With this strategy, a write method on
a replicated object is sent to all machines that contain a copy [1, 3, 8]. Then the method is applied to all copies.

6.1.1 Migration Algorithm (SRSW)

Data in the migration algorithm is sent to the location of the data access request, allowing subsequent accesses to the data to be performed locally [6, 14]. It allows only one node to access a shared data at time as shown in figure 6.1. Due to this it is also called as single-reader single writer (SRSW) algorithm. Typically, the whole page or block containing the data item transfer instead of an individual item requested. This algorithm takes advantage of the locality of reference exhibited by programs.

![Figure 6.1 Migration Algorithm](image)

The migration algorithm provides an opportunity to integrate DSM with the virtual memory provided by the operating system running at individual nodes. When the page size used by DSM is a multiple of the virtual memory page size, a locally held shared memory page can be mapped to an application’s virtual address space and accessed using normal machine instructions. On a memory access fault, if the memory address maps to a remote page, a fault-handler will migrate the page before mapping it to the process’s address space.
Upon migrating a page, the page is removed from all the address spaces it was mapped to at the previous node.

### 6.1.2 Read-Replication Algorithm (MRSW)

The migrating algorithm allows only one node at a time whereas read-replication allows multiple nodes. The algorithm allows multiple nodes to have read access and one node to read-write access, that’s why this algorithm is also called as multiple readers and multiple writers (MRSW) algorithm [2, 6]. This algorithm enhances the performance of system by allowing the multiple nodes for concurrent accessing.

![Read Replication Algorithm](image)

**Figure 6.2 Read Replication Algorithm**

The algorithm performs well when there are only read accesses but when if there is any write access it is expensive as the entire shared blocks at various nodes have to be invalidated (figure 6.2) to maintain the consistency. MRSW algorithm in DSM system is implemented with write-invalidate coherence protocol. Yet MRSW algorithm has the potential to reduce the average load of read operations when the ratio of reads to writes is large.
6.1.3 Full Replication Algorithm (MRMW)

This algorithm is the extension of MRSW algorithm. It allows multiple nodes to have read-write-access that’s why it is called multiple readers and multiple writers (MRMW) algorithm. As many nodes can write shared data concurrently so it becomes necessary to control the shared data to maintain the concurrency [4, 5, 6]. One way for maintaining concurrency is to update all the nodes about write request done by a node.

6.2 Extending the Framework and Implementation

In previous chapter we have discussed a DSM framework about maintaining the memory consistency of DSM system by using coherence protocol and memory consistency. Now in this chapter we are going to extend the said DSM framework. In this framework we have taken processor consistency as memory consistency model and write-update coherency protocol for coherency protocol.

The framework is implemented with MRMW algorithm for data replication. The processor consistency model defines the read, write and synchronization operation. The read and write operation are comprises into access operation. The all three operation are mapped to corresponding operation provided by an underlying coherency [7, 11, 12]. A write-update coherency protocol is used in the frame work to maintain memory consistency for ensuring the serialization of write operation and that any subsequent reads or writes access the update data.
6.2.1 Processor Consistency Implementation

1. How to keep track of the location of remote data

2. How to make shared data concurrently accessible at several nodes to improve system performance

3. How to overcome the communication delays and high overhead associated with execution of coherency protocols.

The first issue i.e. tracking of the location of remote data is implemented with the help of coherence. The second issue is implemented with the help of proper program order and the third issue is implemented with the help of a proper N/W topology.
The execution history of processor consistency is shown by an example in the figure 6.4. W(x) is a write operation written by a particular processor on location x and R(x) is read operation read by a processor on location x. Consider for processor P1 and P2, processor P1 write “1” to location x and processor P2 observe this write by reading 1 from x. This implies that the write access completed some time between being issued by P1 and observes by P2. Similarly it is happening to processors P2, P3 and P4.

<table>
<thead>
<tr>
<th>P1:</th>
<th>W(x)1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2:</td>
<td>R(x)1</td>
</tr>
<tr>
<td>P3:</td>
<td>R(x)1</td>
</tr>
<tr>
<td>P4:</td>
<td>R(x)2</td>
</tr>
</tbody>
</table>

Figure 6.4 Execution History of PC

The access diagram of all processors is shown in figure 6.5. Access ‘N’ shows the access order. e.g. A1 is the first access A2 is the second access and so on. The dashed line shows the causal link that is weak ordering. The dark line shows internal link that is strict ordering. Weak ordering means that the order may be change as per requirement. A processor writes on any location after completing all read operation. There is no total global order and partial order for the same processor. Each processor follows the internal order and order of write of same processor.
The view diagram of all processor is shown in figure 6.6, based on the access operation shown in figure 6.5. Before reading any location ‘x’ a processor must be updated of write on location ‘x’. e.g. for processor P3 it have only two read operation R(x)1 and R(x)2, so before accessing R(x)1 P3 must access W(x)1 and so on. In the view diagram it is easily shown the data replication. When it replicates data, the DSM system must keep track of replicated copies.

The logic will differ in that, after initiating updation for a write, the processor may proceed with its write and any subsequent
reads. However, a write will stall if the counter indicates that the previous write still has acknowledgements needing to be returned. Since write operations are no longer atomic, processor consistency imposes an additional constraint on the order of write sub-operations to the same location. This requirement is called the coherence requirement [10, 13].

The coherence requirement constrains all write sub-operations to the same location to complete in the same order across all memory copies. The requirement of coherence is implemented in a manner that if any processor executes a write operation, this write operation must be updated to all processor.

### 6.2.2 Write-update Coherence Protocol implementation

In the framework the write-update coherence protocol is implemented as a single linked list of nodes. The nodes where the write-update message is to be send are connected in a single linked list manner (figure 6.7).
The node which wants to write firstly it writes on the desired location on the memory and send an update to the starting node through memory. The starting node updates its copy and sends the update to the next node of the list. Now the starting node updates the next node of the linked list. Each node of the linked list is updated. Each node in the list receives the updates, updates its copy and send to next node in the list until the last node. The last node updates its copy and sends an acknowledgement to the writing node. The acknowledgement message indicates that the update has been performed.

6.3 Working

Here first we discuss the term used in the working of framework:

**Initiator:** The message updating is started from one of the processor which wants to write and move to all nodes. This processor is named as initiator.

**Page Map Table:** A page map table is created to the initiator. A page map table of processor involved in a particular process is the informational table consists of starting node and the page.

**Starting node:** The starting node is the node from where the updating of message is started. The starting node is selected by concurrency control.

**Page:** The page consists of the no. of processor sharing for a particular memory model.
**Copy list:** Copy list of each processor contains the information of master node and the reference of node connected to that processor.

**Master node:** Among all nodes one processor works as a controller called master node. The ownership manager selects one master node among all processors.

The working of the framework is described in two parts. First part finds the destination of node to propagate and incorporate the updates and the second part propagates the updates to all nodes.

### 6.3.1 Finding the destination and incorporating updates

This step finds only the processors which are sharing same memory location. So it becomes necessary to update write operation to all these processors. This is done in following steps:

**a.** Finding the Actual Update: Among all write updates, the synchronization (SYN) manager and memory manager (MM) checks and verifies the actual update.

**b.** Finding the destination: The ownership manager finds the destination of actual update with the help of processor consistency model and sends to the SYN.

**c.** Incorporating updates to local copies: The updates are incorporated into local copies with the help of MM.
6.3.2 Propagating updates to all nodes

After incorporating updates to local copies now write update coherence protocol propagate the write updates to all nodes with the help of communication service. The complete activity of updating messages to all nodes is done in following steps:

a. The concurrency control will select one processor as an initiator.

b. The synchronization manager creates a write update message. The message contains the location of write as well as sequence number.

c. The sequence number is necessary because more than one message is flowing.

d. Each time before accepting message the node will check the sequence number. If they are in the order, the node will accept the message otherwise refuse. It is done to maintain the consistency.

e. The initiator processor will check its page-map table and sends write message to the starting node as mentioned in the page-map table.

f. The starting node accepts the message. Now it will check its copy list to find the master node and send the message to master node.

g. The master node accepts the message and checks its copy list to find the address of next node where it has to send this message.

h. Now this activity is repeated until the initiator processor will get the acknowledgement.

For proper implementation of the framework, we have designed some algorithm on the basis of the above working. Let there
are N processor in the system, among N processor some processor want to read and some want to write on a particular memory module. The main motive of the algorithm is to update the processors sharing a particular memory module about write operation. This can be completed in to the following algorithms:

- Algorithm for creating the message body of write operation.
- Algorithm to create the page map table consists of staring node and page.
- Algorithm to create the copy list for each processor except initiator.
- Algorithm to send and accept the message by the nodes.
- Algorithm for Message updating
- Algorithm for acknowledging the initiator about completion of updation.

Following are the terms used in our algorithm, these terms may be used as variable:

\[ P: \text{Processor} \]
\[ N: \text{Total No. of Processor in the system} \]
\[ X: \text{No. of processor among N which want to read only} \]
\[ Y: \text{No. of processor among N which want to write only} \]
\[ PR_{x+y}: \text{Process which is executed by X & Y Process} \]
\[ Yi: \text{Processor which is the initiator among Y and starts write operation} \]
\[ L: \text{Location where Yi wants to write} \]
CP: Page of page map table; give the no of active processor involved in PR_{x+y}

MN: Master node

SN: Starting Node among CP processor

NAi: Address of next processor of process Pi

MSG: Message which is to be sent to the processor

6.3.3 Algorithm for creation of page map table

The creation of page table consists of two steps:

**Step 1:** Counting number (CP) of processor to be updated.
Function Count() will count the number of processor involved for executing process PR_{x+y}.

**Step 2:** Finding the Starting Node (SN) among CP nodes.
Function SN(SYN) assigns the starting node. That will be done by the synchronization manager of the framework:

Page-map-table (CP, SN)
{
    CP = Count();
    SN = SN(SYN);
}

Count( )
{
    P=0;
    for i←1 to N
if (Pi is about to execute process PR_{x+y})
    \[ P \leftarrow P+1; \]
else
    \[ P \leftarrow P; \]
return(P);

\}

\}
i=i+1;

\}
SN(SYN)
\{
i=0;
while (Pi is executing process which is a cooperating process)
    no-operation;
SYN = Pi;
return(SYN);
i=i+1;
\}

6.3.4 Algorithm for creation of copy list

The copy list of a particular processor is created in two steps:

**Step 1:** Finding the information about the master node (MN)
The ownership manager will find the master node among CP processor. This could be done by the help of ownership manager.

**Step 2:** Finding the address \( NA_i \) of next processor. \( NA_i \) gives the address by which \( i^{th} \) processor is connected.
Copy-list (MN, NAi)
{
    MN = Ownership(OM);
    NAi = Addr(OM);
}
Ownership(OM)
The ownership manager will select the master node.
{
    return(address of master node);
}
Addr(OM)
The ownership manager will find the address of node connected to a particular processor.
{
    return(address of Pi+1 connected to Pi);
}

6.3.5 Algorithm for writing by Initiator processor

Initiator processor Yi writes value V on location L and this information will be stored in message body MSG.

Message(Yi)
{
    W(L)V ← Yi;
    MSG← W(L)V;
    Return(MSG);
}
6.3.6 Algorithm for sending and accepting message

Send_{AB}(MSG)
{
    Processor A will send the message (MSG) to the processor B;
}

Accept_C(MSG)
{
    Processor C accept message MSG;
}

6.3.7 Algorithm for updating message

Updating nodes about the write message is done by sending message to each node. This consists of two steps:

Step 1: Updating the starting node (SN):
Firstly the message will be sent to the starting node. The information about the starting node will be found by SN.

Step 2: Updating master node and other:
After sending message to the starting node now the message will sent to the master node. The information about the master node will be found by MN of copy list.

Update( )
{
    Message(Yi);
    Page-map-table (SN, CP);
Copy-list (MN, NAi);

Send_{Yi}SN(MSG)
{
Accept_{SN}(MSG);
SN ← MSG;
}

Send_{SN}MN(MSG)
{
Accept_{MN}(MSG);
MN ← MSG;
}

Send_{MN}NAi (MSG)
{
Accept_{NAi} (MSG);
NAi ← MSG;
NAi_{i+1} = NAi+1;
while(NAi_{i+1} = Yi)
{
Send_{NAi} NAi_{i+1} (MSG)
{
Accept NAi_{i+1} (MSG);
NAi_{i+1} ← MSG;
}
NAi = NAi_{i+1};
}
}
exit;
Acknowledged

Yi will be acknowledged that the updating message to all processors is completed.

6.4 Functional representation of Algorithms

Algorithm ‘E’ is the main algorithm in which all algorithms are encapsulated. A functional representation of the entire algorithm is shown in figure 2.

![Functional Representation of Algorithm](image-url)

**Figure 6.8** Functional Representation of Algorithm
This representation gives the complete working of the framework. The main function update() has three sub functions all three are executed simultaneously for multiple writing. The multiple writes is the combination of writes operation generated by different processor. The numbers of writes depend upon the number of memory module used in the DSM system. Each write operation is updated to all the processors which is reading particular memory module only. By this activity the memory consistency of the framework can be maintained. As the algorithm considers only the group of processors which is sharing same memory location, so each processor have replica of the page. At a result it gives the better data replication. But for this data must be properly sequenced or controlled to ensure consistency. In our framework we concentrate on the consistency assurance only. In our framework the consistency is maintained by the processor consistency model with write-update coherence protocol in this manner that their read operations occur locally whereas the write operations are broadcast over the bus.

6.5 Degree of Parallelism

The degree of parallelism (P) of proposed framework depends upon the degree of data replication (D).

\[ P \propto D \quad \text{Eq.(6.1)} \]

Data replication in the framework is implemented by MRMW algorithm so the degree of data replication depends upon the total time (T) to complete the updates of all write messages in per unit time i.e.
Updating a message includes time to write a message by the initiator processor as well as sending this message to different nodes and writing on a particular node. So the total time taken ‘T’ for updating all write messages will depend upon time ‘\(T_w\)’ to write a message by an initiator processor and the time to update a message ‘\(T_u\)’ to a particular node. So

\[
T = T_w + T_u \tag{6.3}
\]

### 6.5.1 Time to update one write

Let \(T_{m,i}\) is the time to write a message on a node ‘i’, \(T_{n,i}\) is the network response time by a node ‘i’ and \(T_{s,i}\) is the time to send a write message from one node to another node. Let there are ‘n’ nodes in the system for which write message is to update then \(T_u\) for one write will be given by

\[
T_{u,1} = \sum_{i=1}^{n} (T_{m,i} + T_{n,i} + T_{s,i}) \tag{6.4}
\]

### 6.5.2 Total time to update all write

Total time spent for updating nodes about writes to another node will be given by:

\[
T = (T_w + \sum_{i=1}^{n} T_{m,i}) + \sum_{i=1}^{n-1} (T_{s,i} + T_{n,i}) \tag{6.5}
\]
Let that $T_{m,i}, T_{n,i}, T_{s,i}$ is same for all nodes respectively so $T$ will be:

$$T = (T_w + n \times T_M) + n \times (T_N + T_S) \quad Eq.(6.6)$$

Let the hit access ratio for the system is $\alpha$ so number of misses will be $(1 - \alpha)$, so total time taken in updating process among all shared nodes will be

$$T = \frac{a \times \{(T_w + n \times T_M) + n \times (T_N + T_S)\}}{(1 - a)} \quad Eq.(6.7)$$

Let

$$T_W = T_w + n \times T_M$$

$$T_U = n \times (T_N + T_S)$$

So

$$T = \frac{a \times (T_W + T_U)}{(1 - a)} \quad Eq.(6.8)$$

The hit access ratio of the system is given by

$$a = \frac{I}{I + R}$$

Where ‘R’ is read write ratio. The read write ‘R’ is given by

$$R = \frac{w}{r}$$

Where ‘r’ is number of read and ‘w’ is number of write. So the hit access ratio will be
\[
a = \frac{r}{r + w} \quad \text{Eq.(6.9)}
\]

So by eq. (8) and eq. (9) Total time taken ‘T’

\[
T = \frac{r \times (T_W + T_U)}{w}
\]

As the network time for updating a message ‘T_U’ depends upon the network structure and writing time for a message ‘T_W’ will be same for all writes and in the read operation there in no involvement of N/W as it is one to one communication, so we can say that the total access time taken in updating a write message will be inversely proportional to no. of write.

\[
T \propto \frac{1}{w}
\]

So in the proposed framework when numbers of write operation increases, the access time to update all these write message in a particular moment will be decreased. By eq. (6.1) & (6.2) when access time (T) decreased the degree of parallelism increases or we can say that degree of parallelism increases when number of write increases. i.e.

\[
P \propto w
\]

If the number of write operation increases it means that more write operation is performed simultaneously in the system and cause more data replication. More data replication causes more parallelism in the system.
6.6 Performance Analysis

In other perspective when considering ‘T_U’ we see that it will depend upon the number of node ‘n’ for which the write messages are to be updated. So all ‘n’ nodes must be updated about all write operations. If there are ‘w’ writes then these writes will be updated to ‘n’ nodes which increases network access time by ‘nw’. So framework load ‘\(\rho\)’ will also be increased by ‘nw’ times. i.e.

\[
\rho \propto nw
\]

or

\[
\rho = \frac{\alpha nw}{r}
\]

and by eq. (9)

\[
\rho = \frac{nw}{r+w} \quad \text{Eq. (10)}
\]

If there is any write operation done on some location by any processor then all nodes which are sharing this location must be updated about writes. The number of shared nodes can be fixed for all time or it may be change as per requirement of a process. As above we have discussed that the degree of parallelism increases by increasing the number of writes, so the load will also depend upon the number of writes. Although reads does not affect degree of parallelism but it affects the framework load. Now we analysis the framework load ‘\(\rho\)’ with different parameters. The main parameters which affect the framework load of the proposed framework are the number of nodes and number of writes.
**Case 1:** if the number of shared nodes are increasing whereas number of reads and number of writes are constant then the framework load increases with increasing the number of nodes (figure 6.9)

![Figure 6.9 Framework load by number of nodes](image)

**Case 2:** if the number of writes is increasing whereas number of reads and number of nodes are constant then framework load increases constantly with increasing the number of writes (figure 6.10).

![Figure 6.10 Framework load by number of writes](image)

**Case 3:** if the number of reads is increasing whereas number of writes and number of nodes are constant then the framework load is near about constant with increasing the number of reads (figure 6.11).
**Figure 6.11** Framework load by number of reads

**Case 4: (a)** if number of reads, nodes and writes all are increasing then the framework load initially increases but after some moment in time the framework load becoming constant or decreasing (figure 6.12 Read Load).

**Case 4: (b)** But if number of reads and writes are increasing only then the framework load initially increases but after some moment in time the framework load becoming constant or decreasing (figure 6.12 R/W Load)

**Figure 6.12** Framework load with all

By all cases it is found that the load of the framework increases more when there are more nodes for updating the write message. The
number of write also increases the load but as compare to the number of nodes it is less. The best result comes with case 3 and 4.

1. By the case 3 the load decreases but it is not possible because it decreases the degree of parallelism and by which it contradicts the motive of our framework. So this case is not perfect for the framework.

2. By the case 4(a) the framework load increases but after some time it decreases. It may be the best case for our framework but it may decrease the processor utilization because it fixes the number of nodes for a particular location. So this case is also not suggested for the framework.

3. By the case 4(b) the framework load increases rapidly but after some time the framework load become constant and no affect arises due to the increments of nodes, read and write. This case gives higher degree of parallelism for the framework and keeping the performance load less. So this case is very much appropriate for the framework.

So we can say that the proposed system performs well when there are more writes as well as more reads and shared nodes.
REFERENCE


