CHAPTER 6 IMPLEMENTATION RESULTS

6.1 Introduction

This section represents the outputs discussion gained by proposing a technique over the usual DSM approaches described in chapter 4 integrating variable based granularity, SDM algorithm, memory controller and coherency management. This section also describes the impact of distributed shared memory structure while selecting different parameters described in chapter 3 and 4. Results of selection appropriate granularity methods are also described here in comparison of other methodologies. How consistency mechanism implemented with proposed system and its comparisons are also discussed in details. Finally, protocol for shared data access described to make virtual address space management.

A distributed computing system is executed using several distributed archetypal where the virtual memory mechanism plays main role in the framework. This research pronounces a one novel way of structural design with software constraints of the shared memory design. It involves composite phases in basic architecture and near relationship amongst logical address organization and technique that mark virtual shared memory possible.

Figure 6.1 Sample Output of DSM System Architecture
Fig. 6.1 illustrates a sample structural view of the distributed shared memory scheme. A significant character played by latency while shopping the protocol. Here, the proposal makes use of a sequential consistency method that is the robust model which is feasible whenever there exists any delay in network negotiation. Fig. 6.2 describes the processing of a shared data item by different number of computers in network. All the data are being shared on individual sites that is accessed with nodes at the same time. Likewise, in parallel systems, database fragments are distributed over various disk. It will execute some operations like sorting, merging on all sets. So, DSM system will use all the shared memory contents parallel from all disks on a site that is the main benefit of DSM system over parallel processing system.

**Figure 6.2 Memory Operations on Shared Contents**

Sometimes when the data are less than another approach is to replicate copies of data to all disks. So, the processor inside system by getting all copies of data and they can perform the operation locally as Fig. 6.3. After acquiring individual output system will merge all the output to get final results. But, this approach incurs too much overhead in term of memory management and bandwidth. There may be also chances of redundancy and database may go to inconsistent state. After updating of memory content it has been shared to all the sites if required. In this case all systems will get copy data so processing may be faster. The better approach is to create a DSM system as Fig. 6.2 which will logically combine local memories of each node and creates virtual global address space. Individual processes access memory content of the single memory space. So, it will appear single memory space to each
user. The final result of updating of shared data items will be visible same as its all user. Processors will access shared data contents with specified protocol that makes DSM possible. The overall system structure depends on the design goal and practical environment. To construct such distributed shared memory system, it is required to consider different factors like memory structure and granularity, synchronization, coherency management, data location and access, environment, scalability and other issues.

![Diagram: Large shared data size: Limits calculation time, shortening total processing time](image)

**Figure 6.3 Local Memory Operations**

### 6.2 Performance Evaluation

Here, expense of read and write execution and disputes in their application is considered for comparison with conventional architectures. Also, the defined algorithms can be categorized by whether they replicate and/or migrate data. As many as two of the algorithms migrate data items to the node only to be accessed in an effort to deed locality within data accesses and reduce the amount of remote accesses, thereby neglecting the statement overhead. The two other algorithms replicate the data, and because of this various other read accesses can also be possible during the exact juncture with local accesses. Based on replication, the execution of DSM should mark the replication apparent to the applications. Meanwhile, the processes should be incapable to be perceived by writing and reading shared contents that each data access is not concentrated to the equal replica of the data. Rather strictly, the outcome of applications through shared items should be homogenous as if
the memory actions of all sites were executed in any sequential direction and the actions of each discrete site should be appeared chronologically in the sequence indicated by its application, in such situation the shared memory is perceived to be inconsistency. The DSM in a shared memory structure is projected to respond in such way. This characterization of consistency should not be disorganized with a stringent definition needful read accesses to fetch the value of the latest write to the similar location. This is certainly applicable to simultaneous processes running on a multiple processors, but not essential to those on shared memory computers with CPU caches and write back buffers. In case the stringent description holds, then so be the weaker explanation; but the opposite remains false.

In this discussion and investigation, there are certain statement with respect to the atmosphere with the algorithms are applied. These are illustrated here. In common, the performance of distributed applications and parallel is read out mainly by communication cost which is calculated by the fundamental hardware, consecutively. The extrapolations of our output to the other framework are reserved for the future work. Moreover, in this framework, negotiation amongst the processor is low and not reliably associated with local memory access. The multicast and broadcast intercommunication where in a distinct message can be conveyed to several sites in a single network transmission is presented. Mostly, ring and bus network fits the explanation. Here, statement is distributed shared memory architecture comprising of a cluster of sites interconnected by Ethernet i.e. a local area network.

For performance measurement, intercommunication cost is considered in the term number of packet events equal to the number of messages sent over the network. A point to point message consequently requires a single message at a packet event both at the receiving site as well as the sending site. Such packet related to a cost with either receiving or sending smaller packets. A broadcast or multicast message communication involves one message, a packet event at the receiving site along with the sending site. The shared memory model which is distributed delivers two fundamental processes to accessing the shared data contents i.e. write and read data.
data := read (address);  
write (data, address);

The write updates the contents referred by an address to the value of data, whereas the read fetches the shared memory contents referred by the address. To make things easier, the algorithms for executing shared data are hereby designated considering these two actions. Needless to say, the distinction in both semantics and syntax of these executions is probable, e.g. actions may be termed by a number of discrete names, such as in/out, fetch/store, etc. The statement is distributed programs call this purpose explicitly, while this may not be constantly remain essential with appropriate compiler support and/or operating system that the accessed shared data is constantly of a single value.

Consequently, the semantics of the memory access utilities can go outer side of those presented by conventional memory systems and can comprise dequeueing operations or atomic engineering or not to mention an underlying database operation. The kind of shared data can be accessed can also differ and contain for example byte, byte array, integers, or more complex user defined data types. For instance, a programming language called Linda directly cares a shared data by providing three core operations along with a shared tuple space: wherein read reads an available data item named tuple from the tuple space, out adds a fresh tuple and in reads and subsequently eliminates an existing tuple. The tuple of Linda is generally considered lesser by store than by the content.

In chapter 5, all DSM algorithms confirm consistency in distributed virtual memory. On the other hand, their efficiency described is delicate to the access characteristics of the application and data granularity. Here, consider as

\( p \): The handling cost of sending or receiving a shared data packet events that comprises feasibility for data reading, data coping and interrupt handling
overhead. The packet may be short. Typical values for system ranges from one to a number of milliseconds.

\( P \): The handling cost of sending or receiving data block; which is similar to \( p \) but significantly higher. For 8KB block, where often multiple data items require to send, typical value ranges from 20 to 40 ms. For this analytic relationship amongst \( p \) and \( P \) are significant relatively than entire values.

\( N \): No of computers contributed in shared distributed memory.

\( r \): Read/write ratio. For example, there is only individual write an action for each read on average.

\( f \): Utilized in the migration algorithm, it is the possibility of an access fault on the non replicated block. This is identical to the opposite of the average number of sequential access to a block by a distinct process, marking an access to the similar block, producing fault, \( f \) characterizes the locality of data access for the migration algorithm.

\( f' \): In read replication algorithm, the possibility of an access fault on the replicated data block is utilized. This is equivalent to the opposite of the average figure of sequential access to a block by a single process, which enables an access to the same block, producing fault, \( f' \) characterizes the local data access for the migration algorithm.

This section identifies the influences in data access cost and investigate characteristic that have important bearing on the presentation of the algorithm. Based on certain modest analytical comparison of the absolute benefits of the algorithms in an effort to disclose the essential association amongst access patters of the applications and the distributed shared memory algorithms that are expected to generate improved performance for them. To emphasis upon the needed performance characteristics of the algorithms as well as to make this analysis simpler certain assumption are made. Server bottleneck is not serious enough to cause significant delay in remote access of data by any node.
1. Server bottleneck is not serious enough to cause significant delay in remote access of data by any node.

2. The volume of message traffic will not source a network bottleneck. Here, message passing costs are \( p \) and \( P \).

3. As compared to remotely available data cost, the cost of locally available data cost is negligible.

4. Message transmission is expected to be reliable, so the expense of retransmission is not gained.

To associate the efficiency of distributed shared memory algorithms it’s required to express a performance degree. If the data access involves more than one remote node, then message passing cost of local and remote both are involved. Using these constraints and statement described above, average efficiency of each algorithm is calculated as below.

\[
\text{Client Server } E_c = \left( 1 - \frac{1}{N} \right) \times 4p \tag{1}
\]

\[
\text{Migration } E_m = f \times (4p + 2P) \tag{2}
\]

\[
\text{Read Replication } E_{rr} = f' \times [4p + 2P + \frac{Np}{r+1}] \tag{3}
\]

\[
\text{Full Replication } E_{fr} = \left( \frac{1}{r+1} \right) \times (N + 2) \times p \tag{4}
\]

\[
\text{SDM Algorithm } E_{sdm} = \left( \frac{1}{r} \right) \times (N + 2) \times p \tag{5}
\]

All of these equations have two components, where in the main component is left side of ‘*’, which is a possibility of remote access to shared items. The second is the average cost to access data items. The average cost of accessing the data items is same as the product of these two components since the local cost data is taken as negligible. Probability of read access is equivalent to the possibility of write access for SDM algorithm. The related value of these write is constantly a message from the resident site 2 packet events, which is followed by all other nodes (N). Here, \( 1/r \) occurs probability.
of write faults that occurs while write access to shared data items. These two faults rates having greater impact on the presentation of corresponding distributed shared memory (DSM) algorithms and more difficult to assess because it’s vary from application to application.

6.3 Result on Comparative Investigation

The above discussion organizes to create some match-wise evaluations of the algorithm’s performance to demonstrate circumstances wherein one algorithm may outperform the other. All relationships are created by associating the average expenses of the two algorithms concern, to develop a curve beside which they produce related performance. Define as the equal performing curve, this curve divides the parameter space into two halves in a way that in each half of a one of the algorithm will act upon better than another. For instance, is the subsequent comparisons of read replication with migration, the expression on the right side of the figure is the equal performing curve Cm=Crr. Meanwhile, each of the cost formulations comprises packet cost p, wherein only the ratio amongst and P matters in the subsequent comparative analysis. The cost of P/p is supposed to be 20. Based on this statement, this will be capable to mark certain common comments on performance.

6.3.1 Migration Vs. Read Replication

The single variance amongst these dual algorithms is that replication, which is utilized in the read replication algorithm to permit included reads on various computers lacking block actions; however claiming multicasting invalidation requirements upon updates. In case the expense of a block transmission is considerably greater with regards to a smaller message, then the curves for diverse values of N and r cluster strictly organized and are absolutely near to the f = f' line. Excitingly, as shown in Fig. 6.4 the invalidation movement does not have a strong impact on the algorithm’s absolute performances. Normally, read replication efficiently decreases the block fault rate, since in divergence to the migration algorithm, incorporated read accesses to the similar block will no longer effect faults, so the value of ‘f’ is smaller than f. And for that reason,
one can anticipate read replication to beat migration for a massive popularity of applications. Here,

\[ f = f'[1 + \frac{N}{44(r+1)}] \]  

Figure 6.4 Performance Evaluation: Migration Vs. Read Replication

6.3.2 Central Server Vs. Read Replication

The central server besides read replication algorithm evaluation depicted in Fig. 6.5. Obvious according to a number of sites the equal performing curve is practically flat. Henceforth, locality of access is the significant attention in selecting amongst the both algorithms. Furthermore, the encouragement of the read/writes ratio is also nominal.

Figure 6.5 Performance Evaluation: Central Server Vs. Read Replication
Here, read replication algorithms promising to perform for several applications. A block fault level is 0.07 characteristically which is 14 accesses amongst faults is calculated very frequent. That is defined by

\[ f' = 4 \times \frac{(1 - \frac{1}{N})}{44} + \frac{N}{r+1} \]  

[7]

6.3.3 Read Replication Vs. Full Replication

Replication methodology used by both the algorithms. Since all data is replicated at all machines the full replication algorithm is not vulnerable to poor locality. Other side multicast cost increases with N. Several copies are preserved even for updates as full replication algorithm is further aggressive in distributed framework. Based on several aspects including the read/write ratio, the degree of replication, and the degree of locality attainable of read replication of this two algorithm’s comparative performance demonstrated in Fig. 6.6. Hence, for larger structures and when update frequency is higher than full replication performs unwell (when r is lower). Here,

\[ f' = \frac{N+2}{N+44(r+1)} \]  

[8]

![Figure 6.6 Performance Evaluation: Read Replication vs. Full Replication](Image)
6.3.4 Central Server Vs. Full Replication

Here, two different exercises performed in supportive to share data in both the algorithms. The central server algorithm is absolutely centralized while full replication is distributed and replicated. The aggressive replication of full replication appears to be profitable, as long as the read/write ratio is five or higher for N values of up to approximately 20. The curve shown in Fig. 6.7 excluding for smaller values of N, is practically linear. The full replication algorithms is fastener for a very large number of replication and frequent updates to replicated data. In case of medium scale system the selection goes to the modest central server algorithm.

$$r = \frac{1}{4} \left[ (N - 1) + \left( \frac{3}{N-1} \right) \right]$$  \[9\]

![Performance Comparison](image)

**Figure 6.7 Performance Evaluation: Central Server Vs. Full Replication**

6.3.5 SDM Vs. Full Replication

Entirely, distributed shared memory (DSM) algorithms pronounced in the section 5.3 are pessimistic in that they confirm a priori that a processor can access shared data only before the shared space is and will continue consistent. Instead of locating a sequence number for each individual write execution a sequence number is achieved in a series of sequential writes by a single process. Here, the SDM algorithm that is enthusiastic in that it governs
a posteriori whether a process has accessed inconsistent data, in which instance that process is rolled back to a preceding consistent state. This algorithm developed from endeavors to improve the performance of the full replication algorithm by combining various write execution into a one statement packet as performance depicted in Fig. 6.8. If the updated shared data is not accessed before it touches a remote site, temporary inconsistencies will not a problem. Understandably, this may lead to inconsistent copies of data, since writes are through the local copy and only later transferred in batch to other sites.

![Performance Comparison](image)

**Figure 6.8 Performance Evaluation: SDM Vs. Full Replication**

With SDM algorithm is appropriate to establish the shared data accesses into transactions, where each operation contains of several read and write actions bracketed by an initiate transaction and close a transaction. Then, a conflict has encountered, in which circumstance, one of the processes will have to roll back. To roll back a process, all accesses to whole shared data are logged. To preserve the manageability of the size of these logs and the actions on them powerfully. When close transaction is performed, a distinctive breach permitted sequence number for the transaction is demanded from a memory controller. This sequence number controls the demand in which transactions
are to be committed. A transaction Tx with sequence number Sn is aborted if any simultaneously performed transaction with a sequence number less than Sn has updated any of the shared data that transaction Tx accessed. Otherwise, the transaction is completed and its revisions to shared data are transferred to all other sites. These transactions will never have to roll back. Obviously, the performance of this enthusiastic SDM algorithm will be subject to the access characteristic of the application and the application’s use of transactions. It will also outperform the read replication algorithm for those applications that suffer from thrashing, meanwhile the optimistic algorithm associates shared memory accesses at the per shared data content level as disparate to a per data block level for which efficiency defined as an equation (5). The logs of a transaction can then be rejected. If rollbacks are irregular, it will easily outperform the core full replication algorithm. The main disadvantage of this algorithm, nevertheless, is the detail that the shared memory is no longer transparent to the application because the memory accesses must be structured into a transaction and because roll backs must be behavior controlled by the application.

The two another pairs not yet measured are summarized as follows. The evaluation amongst the central server and migration resembles that amongst the central server and read replication, with a somewhat flat curve beneath f = 0.09. The comparison amongst migration and full replication exposes no clear defeater, as in the circumstance of read replication versus full replication, with the key determining parameter being S and r. Thus, except the block fault rate is very higher, migration achieves better. Based on the evaluations above, we can make a few comments. The central server algorithm is simple to implement and may be adequate for uncommon accesses to shared data, particularly if the read/write ratio is lower (that is, a higher degree of accesses are writes). This is frequently the situation with locks, as will be discussed additional below. The fault rate of the simple migration algorithm may increase due to interleaved accesses if distinct data content that ensue to be in the similar block are accessed by different sites. However, locality of reference and a higher block hit ratio is existing in a wide range of applications, making block migration and replication beneficial. It thus does not discover locality to
Implementation Results

Hemant D. Vasava

its full level. The full replication algorithm is appropriate for small scale replication besides infrequent updates. In difference, the read replication algorithm is frequently a good compromise for many applications. The central server and full replication algorithms share the properties of insensitivity to access locality, so they may outperform the read replication algorithm if the application demonstrations low access locality. Significantly, severe performance problems with algorithms that exchange large data blocks is block thrashing. For migration, it takings the procedure of transferring data back and forth in quick sequence when interleaved data accesses are prepared by two or more sites. For read replication, it takes the procedure of blocks to read only authorizations being repetitively invalidated soon after they are replicated. Such conditions, designated a poor site locality in references. Where SDM algorithm better tailored consistency mechanism by using shared variable as granularity. For many applications, shared data can be allocated and the computation can be partitioned to minimize thrashing. Application controlled locks can also be used to suppress thrashing. In either case, the complete transparency of the distributed shared memory is compromised somewhat. Many variations to the core distributed shared memory algorithms available and its comparison with SDM algorithm described in Table 6.1. Several of them improve performance for specific applications by elevating for particular memory behaviors, normally by attempting to decrease the amount of communication since expenses are conquered by communication costs.

Here, we represented principally interesting variants and also presented how applications can support expand performance by governing the activities of the shared memory distributed algorithm. The major enhancements to performance can possibly be attained by reducing consistency restrictions, somewhat we do not reflect here. Table 6.1 describes the comparison of DSM algorithm based on type of data structure, it uses runtime, space, number of CPU (10), shared data access and speed. These goals have been largely met to acquire improved reliability and performance. The inspiration for this strategy was to implement with shared variable and data structure, its SDM algorithm, memory controller concept and specific concerns, kernel programming, interprocess statement and protocol involved. So, first any
process that is part of distributed shared memory discover require a data unit in its own local memory. If the shared data item not resides to the local memory node, then the remote access required. As shared data items are in the form of shared variable which are programmed using global names memory controller of system manage look up for remote data. The request of remote data handled by the memory controller of the system. Lookup having all shared data list that can be accessed through requesting site with the help of the memory controller. Then the normal memory access is utilized to manage data. The DSM system will outperform with the variable based data organization and make data consistent after completion of the specified transaction.

Table 6.1 Comparison of DSM Algorithms

<table>
<thead>
<tr>
<th>Method</th>
<th>Data Location</th>
<th>Memory Access</th>
<th>Efficiency</th>
<th>Cost</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Server</td>
<td>Server</td>
<td>R- Return data, W- Update/AKN</td>
<td>65 %</td>
<td>Moderate</td>
<td>Bottleneck at server</td>
</tr>
<tr>
<td>Migration</td>
<td>Server</td>
<td>R/W data migrated to local machine</td>
<td>70 %</td>
<td>Costly</td>
<td>Thrashing, False sharing</td>
</tr>
<tr>
<td>Read Replication</td>
<td>Distributed</td>
<td>R-Multiple, W- Single at time</td>
<td>76 %</td>
<td>Moderate</td>
<td>Single read</td>
</tr>
<tr>
<td>Full Replication</td>
<td>Distributed</td>
<td>R/W- Multiple with constraints</td>
<td>80 %</td>
<td>Costly</td>
<td>Access control required</td>
</tr>
<tr>
<td>SDM Algorithm</td>
<td>Distributed</td>
<td>Multiple R/W</td>
<td>92 %</td>
<td>Low</td>
<td>upgradable programming (Mngt. by memory controller)</td>
</tr>
</tbody>
</table>

Hemant D. Vasava
6.4 Result on Granularity Mechanism

It states in the data block of shared memory, that is, to the unit of sharing and the unit of data transfer across the network. When a network block fault occurs probable entities are a few variables, objects, tuple, pages or data structures. Choosing appropriate block size is a significant task of the design of a DSM system, since block size is typically a degree of the granularity of parallelism discovered and the amount of network traffic produced by network block faults. The structure of the shared memory space of a DSM system is usually dependent on the type of program that the DSM system is projected to support. Structure discusses to the layout of the shared data in memory. In a DSM system that permits the replacement of the shared data item, copies of shared data items may concurrently be existing in the main memories of a number of machines. In this situation, the core problem is to resolve the memory coherence difficult that contracts with the consistency of a piece of shared data underlying in the main memories of two or more nodes. This difficulty is similar to that which rises with conventional caches, in actual with multi cache patterns for shared memory multiple computers.

<table>
<thead>
<tr>
<th>Item</th>
<th>Objects</th>
<th>Shared Variables/Data Structures</th>
<th>Pages/Chunks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium</td>
<td>Network</td>
<td>Network</td>
<td>Network</td>
</tr>
<tr>
<td>Transfer Unit</td>
<td>Objects</td>
<td>Variables</td>
<td>Pages</td>
</tr>
<tr>
<td>Data Migration done by</td>
<td>System Software</td>
<td>System Software</td>
<td>System Software</td>
</tr>
<tr>
<td>Shared Logical Address Space?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Possible Memory Operations</td>
<td>Common</td>
<td>Read/Write</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Object Oriented Implementation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Is Remote Access Possible in</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>hardware?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Possibility of unattached memory?</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Address Translation by?</td>
<td>Runtime System</td>
<td>Runtime System or Language</td>
<td>Operating System</td>
</tr>
</tbody>
</table>
Meanwhile, different memory coherence protocols build different conventions and tradeoffs, the selection is commonly dependent on the configuration of memory access. In addition, synchronization primitives, such as event count, lock and semaphores are required to synchronize simultaneous accesses to shared data. In a DSM system, simultaneous accesses to shared data may be produced, therefore, a memory coherence protocol is not only adequate to preserve the consistency of shared data. Data locality and access to share data in a DSM system, it should be potential to locate and fetch the data accessed by a consumer process. Therefore, a DSM system must develop some arrangement of data block locating methods in order to serve network data block faults to encounter the need of the memory coherence semantics presence used. Replacement approach refers, if the local memory of a site is full, a cache miss at that node indicates not only a raise of the accessed data block from a remote node but also a replacement. That is, a block of the local memory must be switched by the new data block.

Consequently, a cache replacement policy is also necessary in the design of a distributed shared memory. Table 6.2 describes compassion of all the data organization which depends upon project requirements. Here, variable based granularity better provides shared memory mechanism as the data are organized in the form of variables. It is having number of benefits in comparison with mentioned granularity mechanisms. Table 6.2 also illustrated the mechanism required to manage shared variable based granularity. With specified SDM algorithm and its architecture provides benefits as started below. But, programmer of DSM system must aware of data items being shared and its limited.

- Provides large memory space.
- False sharing and thrashing can be excluded.
- The MMU can be used if we place each shared variable on an own page.
- Consistency can be applied easily.
- Classes of variables can be defined easily.
- Unshared variable separation.
- Only a limited number of variables are shared.
If shared contents is not accessible, then it will negotiate to the memory controller for remote data on another machine and it will negotiate for remote content through mapping and access it like own memory space. This selection of distributed shared memory structure will eliminate thrashing, false sharing and page fault impediments.

In the case of pages, if a process cannot preserve its minimum requisite number of frames, then it must be swapped out, freeing up frames for other processes. This is a middle level of CPU scheduling. But what about a process that can keep least, but cannot keep all of the frames that it is presently using on a regular basis? In this situation, it is enforced to page out pages that it will require again in the very near future, leading to big numbers of page faults. A process that is spending more time paging than achieving is said to be thrashing as shown in Fig. 6.9. Primary, process scheduling structures would control the level of multiprogramming permitted based on CPU utilization, count in more processes when CPU utilization was low.

The difficulty is that when memory occupied up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would low, producing the schedule to enhance in even more processes and intensifying the problem. Ultimately, the system would fundamentally grind to a halt.

![Figure 6.9 Thrashing Outcome](image-url)
Local page replacement strategies can avoid one thrashing process from placing pages far from other processes, but it still inclines to obstruct the I/O queue, thereby decelerating down any other process that needs to do even a little bit of paging or any other I/O for that difficulty. To stop thrashing we must deliver processes with as several frames as they really need right now, but how do we know what that is? The locality method records that processes classically access memory references in a given locality, making lots of references to the similar general area of memory before transferring periodically to a new locality, as shown in Fig. 6.9 above. If we could just remain as several frames as are involved in the current locality, then page faulting would arise primarily on alterations from one locality to another. E.g. when one function exits and another is called. That will also hide data association like message passing method.

6.5 Consistency Analysis

The output of any operation is the identical as if the execution of entirely processors was performed in certain sequential order and the execution of each individual processor seems in this sequence in the order stated by its program. This description revenues is that while processes run in parallel on diverse computers or even in pseudo parallel on a timesharing system any valid interleaving is adequate characteristic, but all processes must see the identical sequence of memory references. A memory in which one process (or processor) understands one interleaving and other process realizes an altered one is not a sequentially consistent memory. Remember that, nothing is explained about time; that is, there is no reference to the most current location.

In this situation, a process, perceives writes from all processes, but only its private reads. Consider following example of Fig. 6.10 than what the output will be? While strict consistency is the best programming method, it is nearly difficult to implement in a distributed computing system. Furthermore, experience describes that developer can often achieve quite well with weaker methods. For illustration, all reference books on operating systems discuss
the mutual exclusion and critical sections. This conversation always contains the limitation that properly written in parallel programs such as the producer, consumer problem should not create any rules about the relative speeds of the processes or how their declarations will interleave in time.

![Diagram](image)

**Figure 6.10 Example to Illustrate Consistency Mechanism**

A = 1, C = 1?  →  Yes, this is what one would expect.

A = 0, C = 1?  →  Yes, if st to B overtakes the st to A on interconnect toward P3.

A = 0, C = 0?  →  Yes, if st to B overtakes the st to A on interconnect toward P3.

A = 1, C = 0?  →  Yes, if the st to A overtakes the st to C from the same processor.

Including on two procedures within one process to occur so speedily that the other process will not be capable to do somewhat in between is looking for trouble. As an alternative, the reader is qualified to program in such a mode that the exact sequence of statement execution (at point, memory references) does not difficulty. Accommodating this argument in detail, means culture to live with a weaker memory method. With some exercise, most parallel programmers are capable to adjust to it. When the sequence of events is important, semaphores or other synchronization actions should be utilized. Sequential consistency is a somewhat weaker model than
strict consistency. It was first defined by Lamport, who thought that a sequentially consistent memory is one that fulfills the condition. That time does not show a role can be seen. A memory performing as shown in Fig. 6.11 is sequentially consistent, even though the first read done by P₂ returns the original value of 0 instead of the new value of 1.

![Figure 6.11 Example of Some Valid Consistency Ordering of Execution](image)

To create this fact more clear, let us consider the example of Fig. 6.12. Here, the code for three processes that running in parallel on three different computers. All three machines share the identical sequential consistent shared memory, and all have access to the shared variables x, y and z. All declarations are presumed to be atomic. Several interleaved operation sequences are probable. From a memory reference viewpoint, a transfer should be perceived as a write, and a print declaration should be understood as a simultaneous read of its two parameters. With six autonomous statements, there are potentially 6! possible operation sequences, although some of these violate the execution order. Consider the 120 (5!) sequences that initiated with x=1. Half of these have print (x, z) before y=1 and thus
violate program sequence. Half similarly have print (x, y) before z=1 and also violate program sequence. Only 1/4 of the 120 sequences or 30 are valid.

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>x = 1;</td>
<td>y = 1;</td>
<td>y = 1;</td>
<td></td>
</tr>
<tr>
<td>print(y, z);</td>
<td>y = 1;</td>
<td>z = 1;</td>
<td>x = 1;</td>
<td></td>
</tr>
<tr>
<td>y = 1;</td>
<td>print(x, z);</td>
<td>print(x, y);</td>
<td>z = 1;</td>
<td></td>
</tr>
<tr>
<td>print(x, z);</td>
<td>print(y, z);</td>
<td>print(x, z);</td>
<td>print(x, z);</td>
<td></td>
</tr>
<tr>
<td>z = 1;</td>
<td>z = 1;</td>
<td>x = 1;</td>
<td>print(y, z);</td>
<td></td>
</tr>
<tr>
<td>print(x, z);</td>
<td>print(x, y);</td>
<td>print(y, z);</td>
<td>print(x, y);</td>
<td></td>
</tr>
<tr>
<td>Prints: 001011</td>
<td>Prints: 101011</td>
<td>Prints: 010111</td>
<td>Prints: 111111</td>
<td></td>
</tr>
<tr>
<td>Signature: 001011</td>
<td>Signature: 101011</td>
<td>Signature: 110101</td>
<td>Signature: 111111</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.12 Execution Sequence of Shared Data

Additional 30 legal sequences are probably starting with y=1 and another 30 can begin with z=1, for a total of 90 valid operation sequences. Four of these are described in Fig. 6.11. Four valid operation sequences for the program of consistency method. That may be vertical axis, increasing downward in Fig. 6.11 the three processes are executed in sequence, first P1, then P2, then P3. The other three cases establish different, but similarly legal, interleavings of the declarations in time. All of the three processes print two variables. Meanwhile the only values each variable can yield on being the initial value (0), or the given value (1), each process yields a 2 bit string. The numbers afterward prints are the authentic results that seem on the output device.

If we concatenate the result of P1, P2, and P3 in that sequence, we obtain a 6 bit string that characterizes a specific interleaving of declarations and thus memory references. This is the string recorded as the signature in Fig. 6.12. This will describe each organization by its signature rather than by its printout. Not all 64 signature forms are permissible. As an insignificant example, 000000 is not allowed, because that would indicate that the print statements ran earlier the project statements, violating the need that the statements are
executed in program sequence. An additional refined example is 001001. The first two bits, 00, mean that y and z were both 0 when P1 did its printing. This condition arises only when P1 performs both statements before P2 or P3 starts. The next two bits, 10, mean that P2 must run after P1 has started but before P3 has started. The last two bits, 01, mean that P3, must finish before P1 starts, but we have already realized that P1 must go first. Therefore, 001001 is not permitted. Simply, the 90 altered valid declaration orderings yield a variety of different program outputs less than 64, though, that are permitted under the statement of sequential consistency. The agreement amongst the memory and software here is that the software must agree all of these as valid outputs. In other statements, the software must agree the four results shown in Fig. 6.11 and all the other valid outcomes as proper responses, and must work properly if any of them arises.

An application that workings of certain of these outcomes and not for others violates the agreement with the memory and is incorrect. A sequentially consistent memory can be developed on a DSM or a multicomputer system that reproduces writable data by confirming that no memory action is started until all the previous ones have been finished. In a system with an effective, perfectly orderly, reliable broadcast method, for example, all shared variables could be collected together on one or more pages, and executions to the shared pages could be broadcast. The exact sequence in which the operations are interleaved does not substance as long as all processes approve on the directive of all operations on the shared memory.

Several prescribed structures have been proposed for conveying sequential consistency and other techniques. Let us briefly consider the example of a system. In this method, the order to read and write executions of the process i is designated by Si (the history of Pi). Here, two such sequences, S1 and S2 for P1 and P2, separately as follows:

\[
S1=W(X)1 \\
S2=R(X)0 \ R(X)1
\]
The set of all such sequences is called $S$. To acquire the relative order in which the executions seem to be executed, we must merge the execution strings in $S$ into a single string $S_q$, in which each operation appearing in $S$ appears in $S_q$ once. Automatically, $S_q$ provides the order that the execution would have been carried out had there been an only centralized memory. All legal values for $S_q$ must follow two constraints:

1. Memory coherence must be appreciated.
2. Program order must be preserved.

The first limitation means that if a read or write access $A$, seems before another access $B$, in one of the strings in $S$, $A$ must also seem before $B$ in $S_q$. If this construction is true for all pairs of actions, the resulting $S_q$ will not illustrate any execution in a directive that violates any of the programs. The second restriction, called memory coherence, revenues that a read to some place $x$, must always response the value most newly written to $x$, that is, the value $v$ written by the most recent $W(X)v$ before the $R(X)$. Memory coherence observes in isolation, each location and the sequence of actions on it, without respect to other locations. Consistency, in contrast, agreements with writing to diverse locations and their sequence.

There is only one permitted value of $S_q$:

$$S_q = R(X)0 \ W(X)1 \ R(X)1$$

For more complicated examples, there might be several permitted values of $S_q$. The characteristic of a program is supposed to be correct if its execution sequence corresponds to some legal value of $S_q$. Though, sequential consistency is a developer friendly model, it has a severe performance problem. It shows that if the read time is $r$, the write time is $w$, and the nominal packet transfer time between sites is $t$, then it is always true that $r+w\geq t$. In other words, for any sequentially consistent memory, changing the protocol to increase the read performance makes the write performance worse, and vice
versa. For this reason, researchers have explored other weaker models. In the following sections discussed some terminology.

**Issue:** Memory operation leaves the processor and becomes visible to the memory subsystem.

**Performed:** Memory operation appears to have taken place

- Performed w.r.t. processor X: as far as processor X can tell. E.g., a store S by processor Y to variable A is performed w.r.t. processor X if a subsequent load by X to A returns the value of S (or the value of a store later than S, but never a value older than that of S). A load L is performed w.r.t. processor X if all subsequent stores by any processor cannot affect the value returned by L to X.

- Globally performed or complete: performed w.r.t. to all processors. E.g., a store S by processor Y to variable A is globally performed if any subsequent load by any processor to A returns the value of S.

- X consistent execution: result of any execution that matches one of the possible total orders (interleaving) as defined by model X “Result of an execution”: Values returned by the reads.

```plaintext
Initially flag1 = flag2 = 0

P1                  P2

flag1 = 1            flag2 = 1
if(flag2 == 0)       if(flag1 == 0)
{                    
  /* critical*/      /* critical*/
}                    }
```

**Figure 6.13 Consistency Violation**
If write buffering is used each buffer can write to flag1 and flag2 respectively, and go ahead with the read operation. This will cause both the reads flag1 and flag2 to return 0, cause both P1 and P2 goes to critical section as shown in Fig. 6.13.

Initially flag1 = flag2 = 0

P1
a1:flag1 = 1
a2:if(flag2 == 0)
    { /* critical*/ }

P2
b1:flag2 = 1
b2:if(flag1 == 0)
    { /* critical*/ }

In this example, does not matter if (a1,a2) and (b1,b2) perform out-of-order, as long as b2 sees a1, both processors can't enter critical section at the same time.

Figure 6.14 Case I: Conflict Ordering

Conflict ordering is occurring in consistency mechanism while access done by different processors illustrated in Fig. 6.13 and Fig 6.14 where execution may not go to the critical section. In Fig. 6.13 illustrated memory conflict ordering which does not matter if (a1, a2) and (b1, b2) perform out of order. As long as b2 sees a1, both processors cannot enter the critical section at the same time. As described in Fig. 6.15 does not matter if (a1, a2) and (b1, b2) perform out of order. As long as a2 sees b1, both processors cannot enter the critical section at the same time.

Comparison with a single shared memory system than it also gives few shortcomings like become slower to access and it is requiring additional concurrency mechanism. Still, both are diverse strategic factors which may have pros and cons. It depends upon its main structural requirement. Write
Initially flag1 = flag2 = 0

P1

b1: flag2 = 1
b2: if(flag1 == 0)
   {
       /* critical*/
   }

a1: flag1 = 1

a2: if(flag2 == 0)
   {
       /* critical*/
   }

In this example, does not matter if (a1,a2) and (b1,b2) perform out-of-order, as long as a2 sees b1, both processors can’t enter critical section at the same time.

Figure 6.15 Case II: Conflict Ordering

Figure 6.16 Consistency based on Semantics vs. Efficiency
invalidate versus write update protocol based distributed shared memory is the simplest to practice and has no configurable settings. Because clients with the shared memory protocol can only associate with all sites successively on different network, it is useful for most database action. Use the shared memory protocol for troubleshooting when you uncertain the other protocols are organized incorrectly.

Finally, Fig. 6.16 describes consistency based semantics again efficiency. Here, as implementation goes to complex structure, its efficiency increases. All the mentioned consistency models can be implemented with distributed system. But, all having some advantages and drawbacks. More better to obey system requirement goals and its environment.

6.6 Communication Policy

In a high speed local area network, transmission control protocol/internet protocol (TCP/IP) sockets besides named pipes clients are equivalent with regard to performance. However, the performance change amongst the TCP/IP sockets besides named pipes clients becomes specious with slower networks, such as across dial up or wide area networks (WANs). This is because of the diverse behaviors the interprocess communication methodology communicating amongst peers. For named pipes, network communications are usually more cooperative. A peer does not send data until another peer requests for it exhausting a read command. A network reads usually contains a sequence of peek named pipes messages earlier it starts to read the data. These can be more expensive in a slow network and reason unnecessary network traffic, which in fit disturbs other network clients. It is also significant to simplify if you are speaking about network pipes or local pipes. If the server program is running locally on the machine that is running an instance of SQL server, the local named pipes protocol is an alternative. A local named pipe runs in kernel manner and is very fast.

Mostly, applications running on client machines create desires to a code often called as server program, running on a server. They comprise networking facilities delivered by the transport layer, which is part of the internet software
stack, often called TCP/IP stack. The transport layer contains two kinds of protocols, transport control protocol (TCP) and user datagram protocol (UDP). The most broadly used programming interfaces for these protocols are sockets. TCP is a connection oriented protocol that offers a reliable flow of data between two computers. An example program that uses such services are HTTP, FTP, and Telnet. UDP is a protocol that directs independent packets of data, called datagrams, from one computer to another with no guarantees about arrival and sequencing. Example programs that use such services contain clock server and ping. The TCP and UDP protocols use ports to map incoming data to a specific process running on a computer. The port is denoted by a positive 16 bit integer value. Some ports have been reserved to support common/well known services:

- telnet 23/tcp
- ftp 21/tcp
- smtp 25/tcp
- http 80/tcp, udp
- https 443/tcp, udp
- login 513/tcp

For TCP/IP sockets, data transfers are more efficient and have less overhead. Data transmissions can also take benefit of TCP/IP socket performance improvement methods such as windowing, delayed acknowledgements, and so on. This can be very supportive in a slow communication. Subject to the type of applications, such performance variances can be important. TCP/IP sockets also provision a backlog queue. This can deliver a restricted smoothing effect related to named pipes that could lead to pipe busy faults when you are trying to associate to SQL server. Usually, TCP/IP is favored in a slow dial up network, local or wide area network, whereas named pipes can be a good selection when network speed is not the matter, as it offers more functionality, ease of use, and configuration opportunities. The protocol must be enabled on both the client and server to work as illustrated in Fig.6.17. The
server can listen for requests on all permitted protocols at the identical time. Client machines can take one or try the protocols in the order listed in server manager. Here, in DSM system all client machines are connected with this socket mechanism. Server memory controller listens all the request from a client node for remote data access. That will be more efficient for DSM system.

![Figure 6.17 Sockets Mechanism](image)

**Figure 6.17 Sockets Mechanism**

The memory controller can listen all the request at the same time and respond to the appropriate client site. So memory controller requires to run first on the specific port address. If the system is local than local host is necessary to the run client application which use a specific port number on which memory controller running. The client machine will connect with memory controller using the port and IP address. All computers on the internet is recognized by a unique, 4 byte IP address. This is characteristic written in dotted quad presentation like 128.250.25.158 where each byte is an unsigned value between 0 and 255. This illustration is clearly not user friendly because it does not express us anything about the content and then it is hard to remember. Hence, IP addresses are mapped to names like www.google.com, which are easier to remember. Internet supports name servers that translate these
names to IP addresses. In overall, each machine only has one internet address. However, machines often essential to communicate and deliver more than one type of service or to talk to multiple sites/computers at a time. For instance, there may be several web connections, FTP sessions and chat applications all running at the similar time. To differentiate these services, a perception of ports, a logical access point, characterized by a 16 bit integer number is used. That revenue, each service offered by a machine is uniquely identified by a port number.

All internet packets encompass both the port number and the destination site address on that site to which the request/message has to be supplied. The host computer messages the packets it receives to application by looking at the port numbers identified within the packets. That is, an IP address can be thought of as a house address when a letter is sent through snail/post mail and port number as the name of a definite individual to whom the letter has to be distributed. Sockets deliver an interface for programming systems at the transport layer. Network communication with sockets is very much similar to execution file I/O. In fact, socket handles are preserved like a file handle. The streams used in file I/O executions are also appropriate to socket based I/O.

Socket based transmission is autonomous of a programming language used for developing it. That means, a socket program written in the Java language can communicate with a program written in non Java (say C++ or C) socket program. An application runs on an explicit machine and has a socket that is appropriate to a precise port. The server listens to the socket for a client to prepare a connection request as Fig. 6.17. If all goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket consequently a different port number so that it can continue to listen to the original socket for connection requests while serving the connected client.