受理
'94.12.20
電気・電子工学
専攻

301

A Study on Distributed Shared Memory in a Widely Distributed Environment

広域環境における分散共有メモリの研究

Masato Oguchi

in partial fulfillment of
the requirements for the degree of

Doctor of Engineering
at the Graduate School
of

The University of Tokyo

December 20, 1994

Advisor: **Prof. Hitoshi Aida**

Contents

List of Figures

Chapter 1

Introduction

High throughput private networks will soon be realized in a widely distributed environment in which only public networks have been constructed thus far[1][2][3]. Special functional computers, such as graphic workstations, supercomputers, database machines, and data I/O computers, are desirable to connect with each other in these private networks (Fig.1.1).

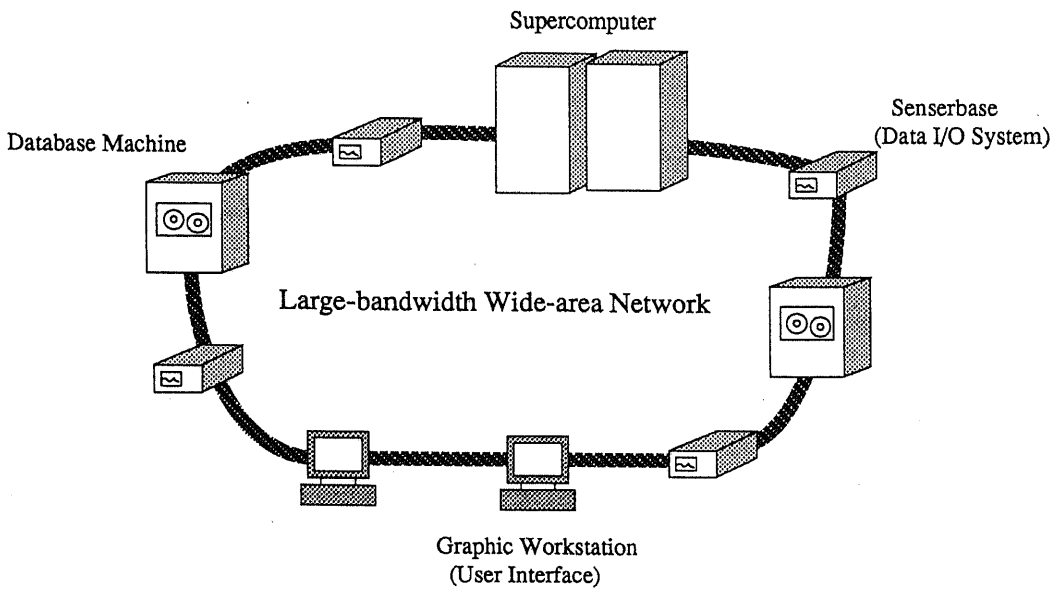


Figure 1.1: An example of a widely distributed system

For example, a wide area control system for an electric power company can be constructed on a network with sensor nodes for measuring current, high performance computers used for future estimation, database computers for storing past data, and graphic workstations for the human interface. In many cases, this broad range of computers are distributed over a wide area.

Distributed shared memory (DSM) is an attractive way to realize a system when functionally distributed computing in a widely distributed environment is required. One reason is its simplicity. Software used in functionally distributed computing systems are inevitably complicated because each module must do many different tasks. Hence DSM should have an advantage in its simple software programming,

because modules can exchange data through shared memory.

Another reason is its flexibility. In contrast to a message passing type system, a receiver of data communication is not specified in DSM. As a result, it is easy to add or delete some functional modules to change the system's functions. This is one great advantage of DSM because the flexibility of the system configuration and/or applications is important for distributed computing, as opposed to the case of parallel processing which requires fixed calculation in most cases. For example, in the case of the electricity control system described above, sensor nodes may sometimes be added or replaced. If the system was constructed using message-passing, such modifications would be troublesome. Moreover, monitoring of the system may want to examine data in the system arbitrarily during operation. This kind of unscheduled data transfer is most easily implemented by DSM.

On the other hand, a disadvantage of DSM is said to be its lack of scalability. As far as functionally distributed computing is concerned, however, this does not seem to be a significant problem, because it is rare to realize a functionally distributed computing system with a great number of nodes.

Studies of DSM – one famous model is shared virtual memory proposed by Li[4] – include various aspects such as system architectures and memory consistency problems[5][6]. These analyses however, have been made only with a network constructed over a small area. DSM in a widely distributed environment differs from the local case, because it is impossible to hide network latency even if a large bandwidth network is used. Even if a very efficient wide area solution for keeping the contents of memory consistent is used[7][8], this latency itself cannot be overcome.

The objective of this study is to clarify the characteristics of DSM in a widely distributed environment. Analytical evaluation of DSM in such environment is given at first. Synchronization matters of DSM are also discussed. Then an innovated model of DSM suitable for such an environment is proposed and evaluated. Finally, quantitative evaluation of DSM, using hardware memory system, is shown.

The structure of this dissertation is as follows.

First, DSM models in a network environment proposed until now are reviewed in Chapter 2. Various models of DSM are proposed and discussed so far. One famous model is shared virtual memory, and the majority of DSM models belongs to this type. Replicated shared memory is another DSM model, which has characteristics almost opposite to those of shared virtual memory. These two models are compared in Chapter 3 by analytical evaluation. Two benchmark programs, whose synchronization behavior is completely different from each other, are used in order to highlight the nature of DSM in a widely distributed environment.

In the case of replicated shared memory, data written to shared memory is not updated simultaneously at all nodes. Although this lazy updating makes the system performance increase, this brings about memory coherency problems. Unfortunately it is difficult for replicated shared memory to implement the synchronization methods using shared memory directly, such as Test&Set. Hence some other efficient synchronization methods are needed. In Chapter 4, some semaphore mechanisms, suitable for replicated shared memory, are proposed and evaluated. Conditions for replicated shared memory to keep release consistency is also clarified.

Replicated shared memory is considered to be suitable for a widely distributed environment. This method, however, is considered to have some problems. In order to overcome these problems , an alternative method to realize replicated shared memory is proposed in Chapter 5. Multi-thread programming executed on multiprocessor and ATM network is used to realize this mechanism.

It is difficult to evaluate DSM models by software simulation, especially in the case of a widely distributed environment in which networks have a great role in terms of performance. This is because a distributed computing system is a complicated combination of each computer node and a network system so that they are hard to describe by simple event-driven or trace-driven simulation. Therefore, SCRAMNet^{TM}, whose operation is based on replicated shared memory itself, is used for an evaluation of DSM models in Chapter 6. This quantitative evaluation com-

pares shared virtual memory with replicated shared memory including an improved model proposed in Chapter 5.

Finally, the conclusions are given in Chapter 7.

Chapter 2

Distributed Shared Memory Models

2.1 Message passing and distributed shared memory

Distributed computing can be classified in various way. One typical classification, which is based on how data is exchanged among nodes, is as follows.

- Message passing
- Distributed shared memory (DSM)

Although it is difficult to define each model precisely because both of them have a lot of variation, their features are as follows in general terms.

In the case of message passing, data is transferred explicitly from sender to receiver. That is, sender process must know where data should be sent to, when communication is in need. In this method, only necessary data is transferred, and only designated node receives it. As a result, an efficient system can be constructed because waste communication is avoided and strict synchronization among processes can be realized. Furthermore, good scalability is achieved in this method.

In the case of DSM, on the other hand, receiver of a message is not designated in data communication. Namely, data is written to shared memory, if other nodes need it. In other words, shared memory realizes implicit communication among nodes. This mechanism yields simplicity and flexibility, as is mentioned in previous chapter.

In most DSM models, all nodes do not share a memory actually. They employ some kinds of techniques such as clustering, paging, and caching, so that contents of memory can be shared among nodes.

2.2 Distributed shared memory models

Among the various kinds of DSM[5][6], one of the most famous models is shared virtual memory proposed by Li[4]. Although many varieties of shared virtual memory

have been studied since then, almost all of their basic functions can be considered an adaptation of the paging mechanism to a network environment. That is to say, each node has only a part of the shared memory contents, and when a pagefault occurs, it brings in the contents of pages it needs from other nodes through the network.

In the case of shared virtual memory, a large amount of data may be sent through the network when a pagefault occurs, even if only a small amount of data is needed. Also a page in need is sent only on demand, which means the page acquisition operation is only initiated when the page is required, so performance must decrease if communication among nodes takes a long time.

Replicated shared memory, another kind of DSM, has characteristics almost opposite to those of shared virtual memory. Different from the case of shared virtual memory, each node has the entire contents of shared memory which are updated by broadcast. As a result, the memory access time of replicated shared memory is usually constant in contrast to the case of shared virtual memory which needs pagefault or invalidation time in some cases. As far as data transfer is concerned, the amount of data transferred during each pagefault in shared virtual memory is large because the data transfer unit is usually one page, but the number of pagefaults is relatively small. In the case of replicated shared memory, the number of data transferred at one time is only 1 word, but data transfer is performed many times because each write access to shared memory causes a broadcast.

As a result of these characteristics, replicated shared memory is considered to be suitable for functionally distributed computing in a wide area in which unscheduled memory access has a great meaning. Conceptual presentation of this idea is shown in Figure 2.1. In the case of shared virtual memory, although the average memory access time is short while the number of unscheduled memory access is small, the access time must become longer as the number of unscheduled access increases. In the case of replicated shared memory, on the other hand, the average memory access time remains almost the same even if the number of unscheduled memory access increases, but the access time itself is fairly large. Thus it is desirable to realize an

ideal model suitable for functionally distributed computing in a wide area, which combines the strong points of the two models. A candidate for this model will be mentioned later in this dissertation.

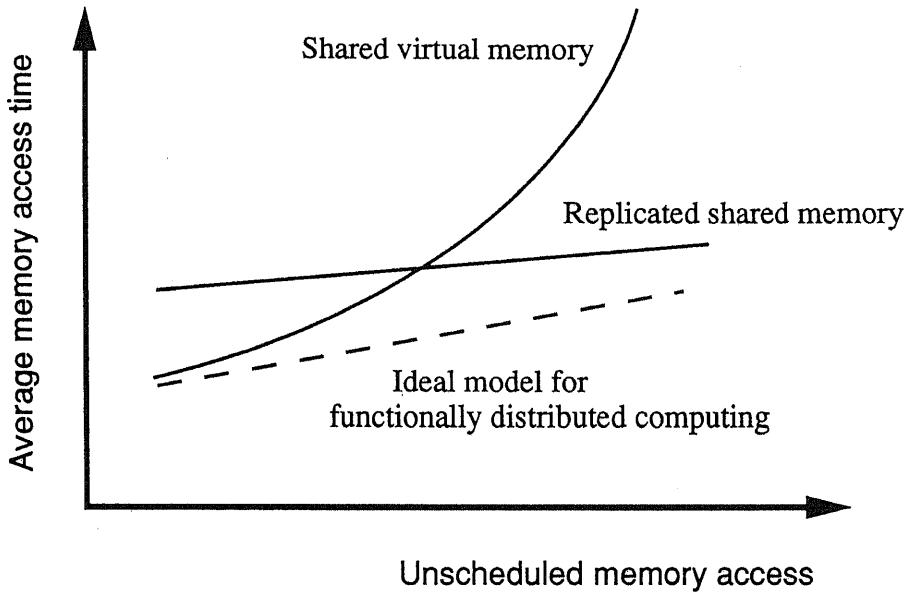


Figure 2.1: Conceptual figure describing characteristics of each model

2.3 Comparison of distributed computing systems

Distributed computing systems are evaluated according to various kinds of criteria. Typical examples of criteria are as follows.

1. Scalability

   scalability in terms of number of nodes.

2. Necessity of special hardware

   necessity of hardware support other than normal machine memory and communication equipments.

3. Flexibility of system configuration

   flexibility in the case of modification or expansion of the system.

4. Efficiency in fixed applications

   efficiency in routine work execution.

5. Flexibility for spontaneous usage applications

   flexibility in the case of non-routine work is dispatched.

6. Efficiency of data transfer

   efficiency about how much of data is transferred in vain.

Here, three typical types of distributed computing models, message passing, shared virtual memory, and replicated shared memory, are evaluated according to these criteria (Table 2.1).

Table 2.1: Comparison of distributed computing systems

	MP	SVM	RSM
1.Scalability	+	0	0
2.Hardware	+	+	−
3.Flexibility of system configuration	−	0	+
4.Efficiency of fixed applications	+	0	0
5.Flexibility for spontaneous usage applications	−	0	+
6.Efficiency of data transfer	+	0	0

MP : Message Passing

SVM : Shared Virtual Memory

RSM : Replicated Shared Memory

2.4 Related works

2.4.1 Plus

Plus[9] is a hardware-based DSM system studied at Carnegie Mellon University. In general, synchronizations and accessing remote data must be realized efficiently in DSM in order to improve performance. Hence Plus has developed a peculiar protocol to support this efficiency, and this protocol is implemented on specialized hardwares. A Plus node is shown in Figure 2.2.

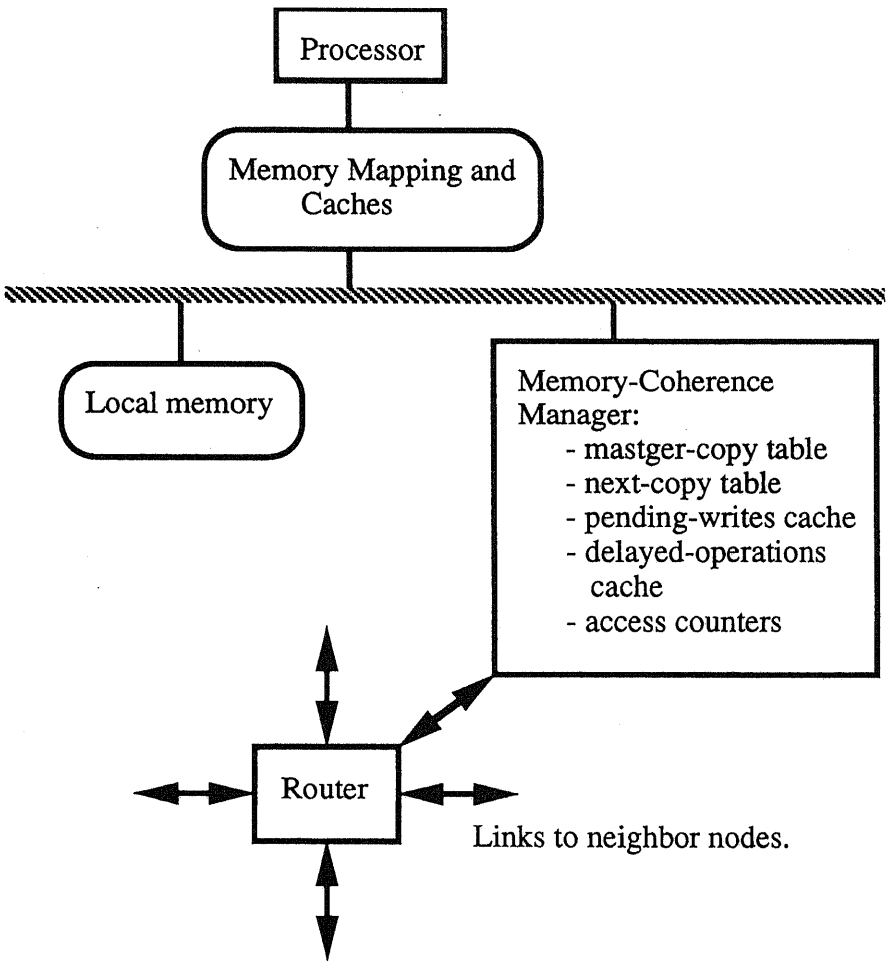


Figure 2.2: A Plus node

Memory and caching mechanisms

Plus uses a non-demand, write-update coherence protocol for the replicated data. The unit of replication is a page (4Kbytes), although the unit of memory access and coherence maintenance is one (32bit) word. Pages are replicated by the request of software, but the coherence manager hardware is aware of this replication and automatically keeps copies coherent.

A virtual page corresponds to a list of physical pages replicated on different nodes. The first item of this list is called the master copy. A node maps each virtual page to the most convenient physical copy, i.e. the closest copy. The global address of a physical page is a (node-id, page-id) pair and is generated directly by the memory-mapping mechanism of the processor.

On each node, the replication structure is made visible to the coherence manager via the master and nextcopy tables, which are maintained by the operating system. For each locally replicated physical page, the master table identifies the global physical page address of the master copy, and the next-copy table identifies the successor, if any, of the local copy along the copy-list.

In the case of read operations, the node-id field of the translated physical address determines which node is addressed, and the page-id field specifies the page within that node. If the local node is indicated, the local memory is read. Otherwise, the coherence manager sends a read request to its counterpart in the specified remote node, waits for the response, and passes the returned data to the processor.

On the other hand, writes are always performed first on the master copy and then propagated down the ordered copy-list. This insures that all copies eventually contain the same data when all writes issued by all processors have completed.

Write operations do not block the issuing processor while they propagate through the copies and a processor can issue several writes before blocking. However, reading a location that is currently being written blocks until the write completes. This is achieved by remembering the address of incomplete write operations in the pending-

writes cache of the coherence manager and guarantees strong ordering within a single processor independently of replication. There is no such guarantee with respect to another processor.

Synchronization

Plus provides various read-modify-write operations such as xchng, cond-xchng, fetch-and-add, queue, dequeue, and min-xchng. Like writes, these operations take effect at all copies of the addressed location, beginning with the master and propagating down the copy-list. However, the master, in addition to executing the operation atomically and forwarding update requests to the next copy, also returns the old contents of memory to the originating node. Since this result always has to come from the master copy, there can be a substantial delay between the initiation of the operation and the availability of its result.

Plus allows the user or the compiler to hide this latency by separating the initiation of an operation from the checking of its result. These operations are called delayed operations, since the execution of the operation overlaps regular processing, as is the case of delayed branches. The processor can continue with normal instruction execution in the meantime. Delayed synchronization allows two latency-avoidance techniques: software pipelining and context switching.

2.4.2 Munin

Munin[10] is developed in Rice University. Munin is an epoch-making model because even though it does not use specialized hardware unlike Plus, it realizes loose memory consistency (release consistency) by software.

Multiple consistency protocols

Several studies of shared memory parallel programs have indicated that no single consistency protocols is best suited for all parallel programs. Furthermore, within

a single program, different shared variables are accessed in different ways and a particular variable's access pattern can change during execution[11].

Hence Munin allows a separate consistency protocols for each shared variable, tuned to access pattern of that particular variable. Moreover, the protocol for a variable can be changed over the course of the execution of the program. Munin uses program annotations provided by the programmer to choose a consistency protocol suited to the expected access pattern of each shared variable.

Munin's consistency protocols are derived by varying eight basic protocol parameters:

Invalidate or Update? (I) This parameter specifies whether changes to an object should be propagated by invalidating or by updating remote copies.

Replicas allowed? (R) This parameter specifies whether more than one copy of an object can exist in the system.

Delayed operations allowed? (D) This parameter specifies whether or not the system may delay updates or invalidations when the object is modified.

Fixed owner? (FO) This parameter directs Munin not to propagate ownership of the object. The object may be replicated on reads, but all writes must be sent to the owner, from where they may be propagated to other nodes.

Multiple writers allowed? (M) This parameter specifies that multiple threads may concurrently modify the object with or without intervening synchronization.

Stable sharing pattern? (S) This parameter indicates that the object has a stable access pattern, i.e., the same threads access the object in the same way during the entire execution of the program. If a different thread attempts to access the object, Munin generates a runtime error. For stable sharing patterns, Munin always sends updates to the same nodes. This allows updates to be propagated to nodes prior to these nodes requesting the data.

Flush changes to owner? (Fl) This parameter directs Munin to send changes only to the object's owner and to invalidate the local copy whenever the local thread propagates changes.

Writable? (W) This parameter specifies whether the shared object can be modified. If a write is attempted to a non-writable object, Munin generates a runtime error.

Sharing annotations are added to each shared variable declaration, to guide Munin in its selection of the parameters of the protocol used to keep each object consistent. While these annotations are syntactically part of the variable's declaration, they are not programming language types, and as such they do not nest or cause compile-time errors if misused. Incorrect annotations may result in inefficient performance or in runtime errors that are detected by the Munin runtime system. The annotations are as follows:

Read-only objects are the simplest form of shared data. Once they have been initialized, no further updates occur. Thus, the consistency protocol simply consists of replication on demand. A runtime error is generated if a thread attempts to write to a read-only object.

Migratory A single thread performs multiple accesses to the object, including one or more writes, before another thread accesses the object[12]. Such an access pattern is typical of shared objects that are accessed only inside a critical section. The consistency protocol for migratory objects is to migrate the object to the new thread, provide it with read and write access (even if the first access is a read), and invalidate the original copy. This protocol avoids a write miss and a message to invalidate the old copy when the new thread first modifies the object.

Write-shared objects are concurrently written by multiple threads, without the writes being synchronized, because the programmer knows that the updates

modify separate words of the object. However, because of the way that objects are laid out in memory, there may be false sharing. False sharing occurs when two shared variables reside in the same consistency unit, such as a cache block or a virtual memory page. In system that do not support multiple writers to an object, the consistency unit may be exchanged between processors even though the processors are accessing different objects.

Producer-consumer objects are written (produced) by one thread and read (consumed) by one or more other threads. The producer-consumer consistency protocol is to replicate the object, and to update, rather than invalidate, the consumer's copies of the object when the object is modified by the producer. This eliminates read misses by the consumer threads. Release consistency allows the producer's updates to be buffered until the producer releases the lock that protects the objects. At that point, all of the changes can be passed to the consumer threads in a single message. Furthermore, producer-consumer objects have stable sharing relationship, so the system can determine once which node need to receive updates of an object, and use that information thereafter. If the sharing pattern changes unexpectedly, a runtime error is generated.

Reduction objects are accessed via Fetch_and_Φ operations. Such operations are equivalent to a lock acquisition, a read followed by a write of the object, and a lock release. An example of a reduction object is the global minimum in a parallel minimum path algorithm, which would be maintained via a Fetch_and_min. Reduction objects are implemented using a fixed-owner protocol.

Result objects are accessed in phases. They are alternately modified in parallel by multiple threads, followed by a phase in which a single thread accesses them exclusively. The problem with treating these objects as standard write-shared objects is that when the multiple threads complete execution, they

unnecessarily update the other copies. Instead, updates to result objects are sent back only to the single thread that requires exclusive access.

Conventional objects are replicated on demand and are kept consistent by requiring a writer to be the sole owner before it can modify the object. Upon a write miss, an invalidation message is transmitted to all other replicas. The thread that generated the miss blocks until it has the only copy in the system. A shared object is considered conventional if no annotation is provided by the programmer.

The combination of protocol parameter settings for each annotation is summarized in Table 2.2.

Table 2.2: Munin annotations and corresponding protocol parameters

Sharing Annotation	Parameter Settings							
	I	R	D	FO	M	S	Fl	W
Read-only	N	Y	-	-	-	-	-	N
Migratory	Y	N	-	N	N	-	N	Y
Write-shared	N	Y	Y	N	Y	N	N	Y
Producer-Consumer	N	Y	Y	N	Y	Y	N	Y
Reduction	N	Y	N	Y	N	-	N	Y
Result	N	Y	Y	Y	Y	-	Y	Y
Conventional	Y	Y	N	N	N	-	N	Y

In Munin project, another looser memory consistency model, called lazy release consistency, is proposed[13]. In this protocol, all data modified before release is issued is merged together, and transmitted to a next node which issues an acquire request. It is mentioned that the number of protocol messages and data can be reduced dramatically by using this method. Furthermore, a lazy hybrid protocol which combines lazy update protocol and lazy invalidate protocol is also proposed[14].

2.4.3 Gigabit network project

AURORA

AURORA project[2], one of NREN (National Research and Education Network), is focusing on a supporting technology for gigabit networks. A configuration of AURORA testbed is shown in Figure 2.3.

AURORA project's goal is roughly divided into following three categories:

- Exploration of Network Technology Alternatives
- Investigation of Distributed System / Application Interface Paradigms
- Experimentation with Gigabit Network Applications

In the first category, Bellcore's ATM-based Sunshine switch and IBM's PTM-based plaNET switch were developed. Transmission protocols higher than transport level, host interfaces, and interconnections between ATM and PTM networks were also studied.

In the second category, the main theme is "memory as a network abstraction, not an I/O interface"[15]. DSM has been studied for this theme. Memnet and Mether are results of these researches.

In the third category, some network applications such as video-conferencing, multimedia teleconferencing, presentations of high-resolution images, and so on are included.

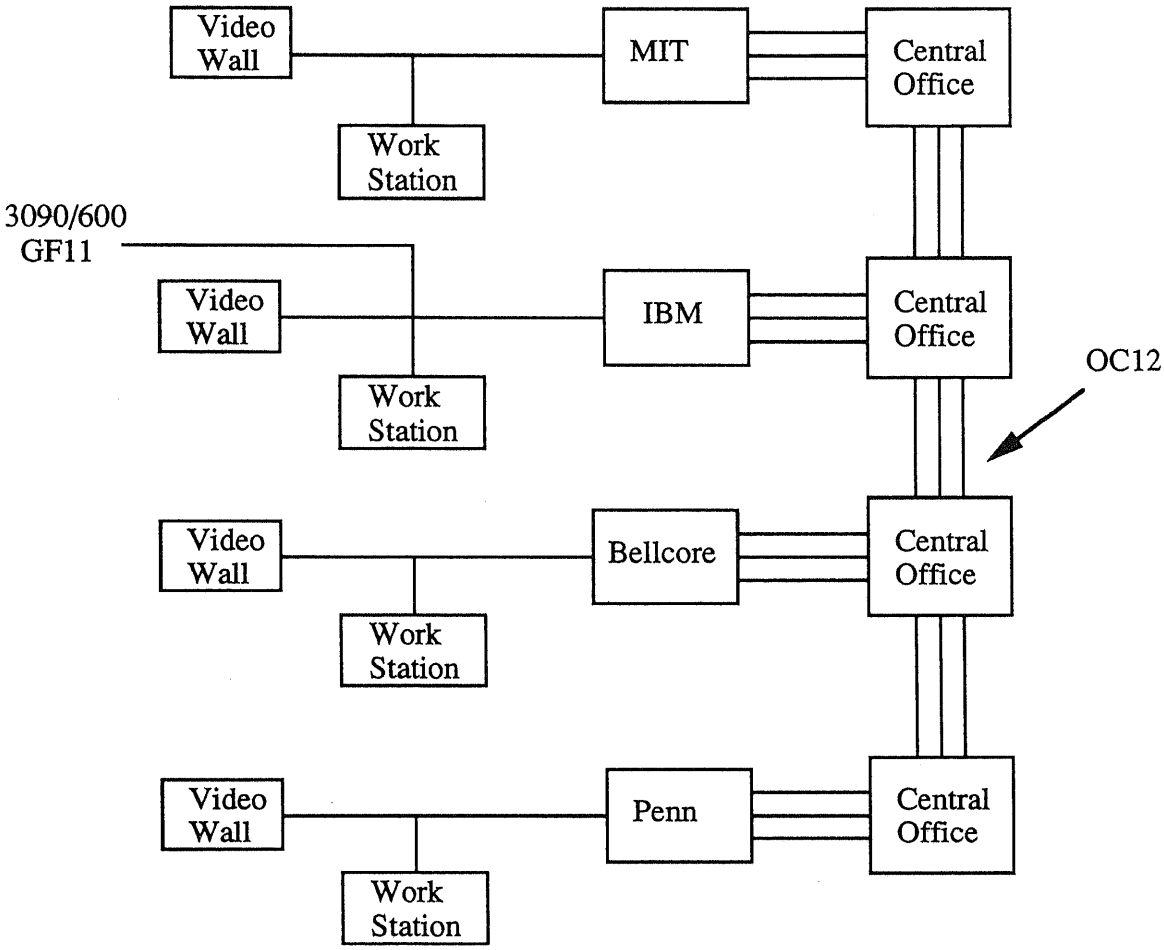


Figure 2.3: AURORA testbed topology

CapNet

CapNet[7] is proposed as a DSM model suitable for a WAN environment. The protocol itself, single writer, multiple readers, write-invalidation protocol is employed, is not peculiar unlike some DSM models mentioned so far, but adapting to a WAN environment is deeply considered in CapNet.

There are number of crucial differences between WANs and LANs as follows:

- WANs have much larger latency (the round-trip time required to send data

and receive a response)

- WANs cannot effectively support broadcast
- WANs have traditionally been bandwidth constrained

The scheme proposed in CapNet is to integrate the network and the operating system software. This is done by augmenting the packet switches with information necessary for locating pages. By distributing the page table into the network switches, the network can route a page request to the owner directly. Under this scheme, a memory request is routed according to its virtual address, which is shared by the participants in an area of DSM. Each switch has its own version of the page table and each page table entry contains an identifier for an outgoing hop leading to the owner of the page. (This table resembles a routing table.) Entries in the page table are updated as pages are transferred to satisfy write requests. Therefore, the current location of a page is always accurate, and it is maintained by the network fabric. A host requesting a shared page may generate a page request. The request is sent to the network; the network locates the page and transfers it to the requester. The whole process takes only two messages, and no broadcasting is required. Address information in the network is self-maintaining. It is possible that there are more than one outstanding request for a page. These requests can be held in a FIFO queue appended to the page. When the page is transferred, the queue of requests is transferred with it.

2.4.4 Memory consistency theories

As far as theoretical studies of memory consistency problem are concerned, various kinds of researches - one famous research is a proposition of release consistency model[21] - have been made until now. Some related works are reviewed in this subsection.

Sequential consistency

Sufficient condition for sequential consistency is discussed in [24].

Condition : Sequential consistency

> Sequential consistency is satisfied in any system if an access may not be performed with respect to any processor until the previous access by the same processor has been globally performed and if accesses of each individual processor are globally performed in program order.

In [23], on the other hand, looser condition than the latter one is proved to be sufficient for sequential consistency, by using directed graphs.

Definition : Access graph

> The Access graph is a directed graph of accesses in a multiprocessor with the arcs defined by the following relations:
>
> - Accesses from the same processor are ordered in program order.
> - Two writes to the same address from different processors are ordered.
> - A read R is preceded by the write that is source of the value read by R.
> - A write W, that follows another write Ws to the same address, is ordered after a read R if Ws is source of R.

Theorem : Sequential consistency

> Sequential consistency is maintained in a multiprocessor if the Access graph is an acyclic graph.

That is to say, it is no need to wait until previous accesses are globally performed in every cases, if an appropriate order is kept.

Race-free network

In the case of a consistency problem in multiprocessor, memory access order in a multi-stage network must be discussed. Hence so called race-free network model is proposed and sufficient condition for sequential consistency is discussed.

Definition : Race-free network

- A race-free network is a network with the topology of any acyclic undirected network graph.
- Transactions propagate on the arcs in the network without the possibility of overtaking each other.
- Transactions may be buffered in the network nodes but buffers must maintain a strict FIFO order between transactions.

Condition : Sequential consistency, RFN

In a system with a race-free network sequential consistency is maintained if:

- A reading processor may continue execution when the read has been performed.
- A writing processor may continue execution when the write has been performed with respect to its root node, and when all writes performed with respect to the root of this item previous to and including this write, also have been performed with respect to all nodes in the path from the root down to the processor.

In [23], some schemes to keep memory consistency in multiprocessor is discussed by using theorems and conditions mentioned above.

Memory consistency problems will be discussed again in Chapter 4.

Chapter 3

Analytical Evaluation of DSM in a Widely Distributed Environment

3.1 Various kinds of DSM models

3.1.1 Model I : Shared virtual memory

The first model we will discuss in this chapter is that of shared virtual memory. This model works as shown in Fig.3.1.

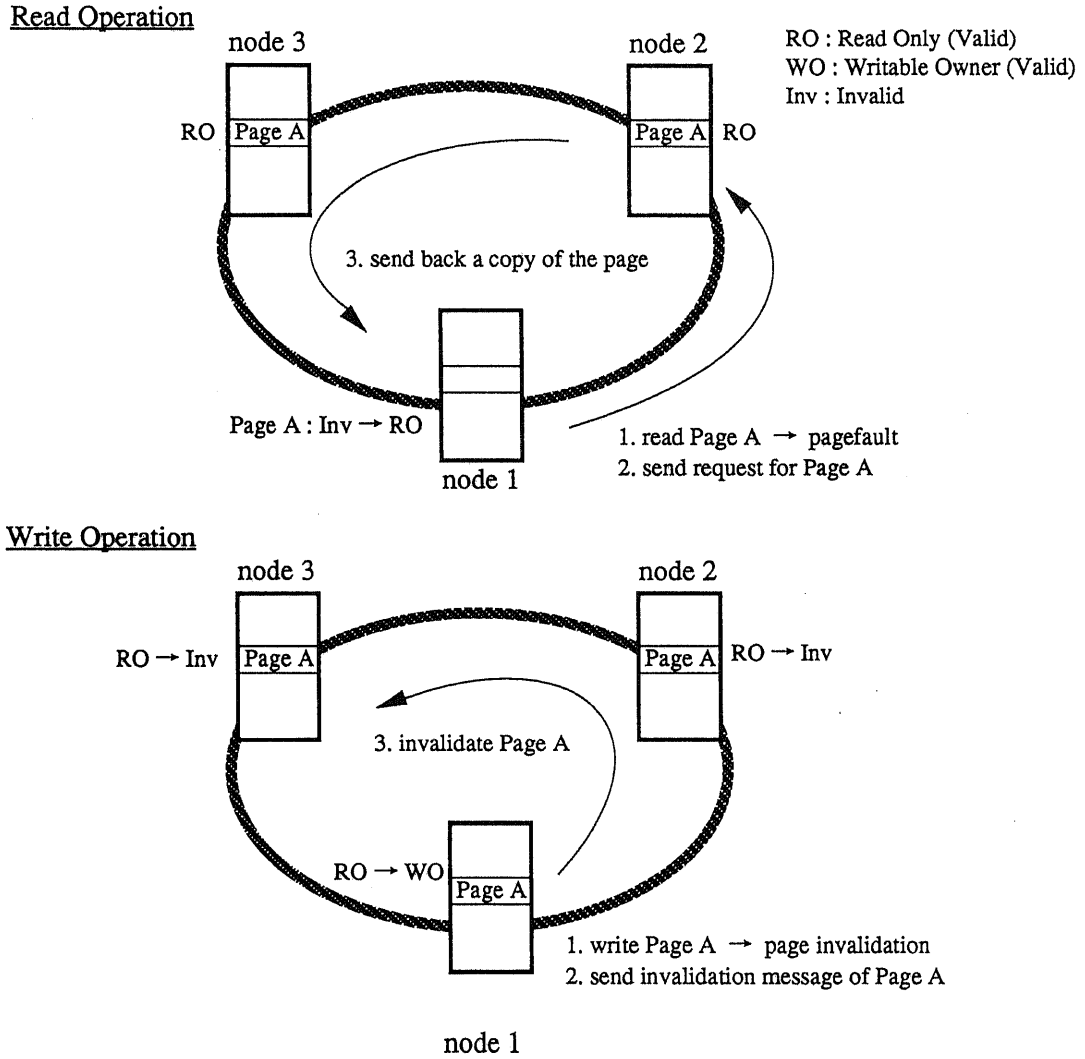


Figure 3.1: Basic Operations of Shared Virtual Memory

In this model, each node has several copies of shared pages, and when a node

comes to a pagefault, it sends a request for that page to the other nodes, who send back the page contents through the network. The method for deciding which node has which copies and how to communicate with that node will not be discussed here because it seems to have little influence on our discussion. If a node already has a copy, or if it has received a copy from another node, it can access the copied page just the same way as it accesses pages of normal machine memory (which we will call local memory). When a node tries to write on a page of shared memory, on the other hand, it sends a message to the other nodes in order to invalidate copies of the same page.

Since shared virtual memory has been actively studied prior to now, a lot of techniques have been proposed to improve the performance. For example, Plus[9] uses write-update protocol. Munin[10] employs various data types and coherence protocols including a write-update protocol associated with each variable, which can be designated by the application programmer. Such techniques seem to be effective for routine calculations whose data flow can be predicted in advance. In the case of functionally distributed computing however, unexpected operation may sometimes be executed, as is mentioned in previous chapters. Thus these techniques are not considered to work effectively for functionally distributed computing, so we adopt the simple write-invalidate model as a representative of shared virtual memory.

As shared data can be accessed like the contents of local memory, it is cached so that they are processed quickly. We assume that memory size is large enough such that there is no need to swap out the contents of memory. Memory allocation and replication or invalidation of its contents are performed by the memory management system, extended for shared virtual memory.

Here, we assume that each node has to acquire a semaphore before it begins to execute a critical section, and must release the semaphore after the critical section is finished. This operation must be done not only in a write access but also in a read access to keep the shared memory contents consistent.

In a widely distributed environment, pagefaults and invalidation of pages have

a greater meaning than in a local environment. This is because a large amount of latency is unavoidable when communicating through the network, even if only a small amount of data has to be sent. We will divide this model into two sub-models, based on the operation for page invalidation.

Model Ia : Basic model

This is a basic model of shared virtual memory. A write operation to shared memory in this model is realized as follows: if a node is not the writable owner of a page, it sends an invalidation message of that page to the other nodes. After invalidation of all copies of the page is completed, the page may be written to as if it were a local memory page.

Model Ib : Page allocation optimized model

As we assume that a semaphore mechanism should be used to execute critical sections, there is no need to wait for completion of invalidation of all the copies if page allocation is optimized considering the relation between pages and semaphores. This is because memory consistency problems, such as a page being accessed by another node during its invalidation, cannot happen, if page allocation is optimized. We define such a model in which page allocation is optimized so that a node trying to write to a page in shared memory does not wait until invalidation of all copies are finished. As far as a pagefault is concerned, it works just the same way as Model Ia, because a node which comes to pagefault has to wait until the data arrives anyway.

When the shared virtual memory model is discussed, a page lock mechanism is adopted in some cases in order to prevent false-sharing. Model Ib is considered to be equivalent to such a model in terms of system performance, because a page will not be accessed by other nodes if a semaphore is acquired in Model Ib, which is the same situation as acquiring a lock in the page lock method. As a result, there is no need to wait for completion of invalidation in both cases.

3.1.2 Model II : Replicated shared memory

In the case of shared virtual memory, a large amount of data needs to be sent through the network whenever a pagefault occurs, even if only a small amount of data is needed. Also a page in need is sent only on demand, which means the page acquisition operation is only initiated when the page is required, so performance must decrease if communication among nodes takes a long time.

We have proposed and discussed how to realize replicated shared memory (called broadcast memory with FIFO), whose characteristics are considered to be almost opposite to those of shared virtual memory. Replicated shared memory, as shown in Fig.3.2, works as follows.

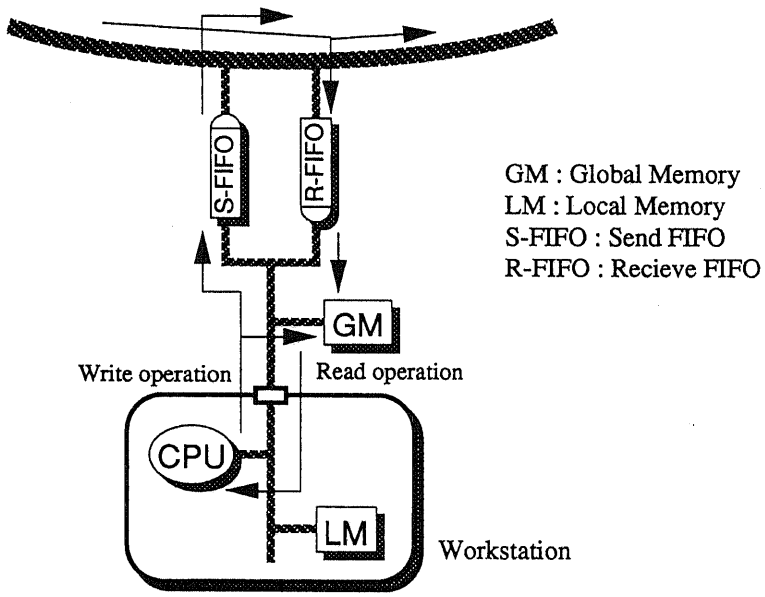


Figure 3.2: Basic Operations of Replicated Shared Memory at Each Node

Each node has its own copy of the entire contents of shared memory stored in a so-called global memory. A read access to shared memory is performed by reading the contents of its own global memory. When a write access to shared memory is performed, the written data is broadcast to all other nodes on the network to

update their own copies. The data transfer unit is a 32bit word. Each bit of written data is first stored in a FIFO buffer and sent to the network later, so that the node can resume its work without waiting for acquisition of the network. A semaphore mechanism should be used in critical sections in order to keep memory consistent[27][39][33].

Replicated shared memory is realized using a bus external to the computer. Global memory is connected to the external bus. We have developed and evaluated a prototype of this bus-connected replicated shared memory[25]. Recently, ring-connected replicated shared memory systems using optical fibers have been commercially sold, such as SCRAMNet™ produced by SYSTRAN Corp.

The major differences between shared virtual memory and replicated shared memory are as follows. The memory access time of replicated shared memory is usually constant because shared data is almost always in global memory [1], in contrast to the case of shared virtual memory which needs pagefault or invalidation time in some cases. In other words, replicated shared memory might succeed in hiding network latency, which is a significant problem, especially in a widely distributed environment. The access time of replicated shared memory is in the order of ten times as long as that of shared virtual memory because the contents of global memory cannot be cached in order to keep memory consistent. As far as data transfer is concerned, the amount of data transferred during each pagefault in shared virtual memory is large because the data transfer unit is one page, but the number of pagefaults is relatively small. In the case of replicated shared memory, the number of data transferred at one time is only one word, but data transfer is performed many times because each write access to global memory causes a broadcast. Only one-to-one communication is required in shared virtual memory, while one-to-all broadcast is required in replicated shared memory.

[1] In the case of the network being congested, the FIFO might become full and write accesses to global memory could stall. In this paper, we assume the bandwidth of the network is sufficiently large and the access time to global memory is constant.

3.1.3 Model III : Hybrid model

In a widely distributed environment, replicated shared memory should be effective in some cases because each node has no need to bring data from distant nodes on demand as is done in shared virtual memory. However, using the external bus on each access to global memory could be too time consuming.

We propose a hybrid model which combines the strong points of the two models. It works as follows.

Each node has a global memory which broadcasts written data automatically, just like replicated shared memory. Use of a semaphore mechanism is required for accessing critical sections in this global memory like other models. When the contents of global memory are accessed, the page is brought into local memory, that is, a node brings shared data from its own global memory, rather than other nodes' local memory when a pagefault occurs. This operation can be done concurrently with the following executions after the first data is accessed, if a DMA controller is used. In this case, the best performance can be expected for the hybrid model. If a DMA controller is not used, on the other hand, the following executions must be suspended until all contents of the page are brought into local memory. This is considered to be the worst case of the hybrid model.

Once the page is read into local memory, its data can be accessed as if it was local data. When a barrier or a semaphore release point is reached, the page is written back to global memory, which will be broadcast to other nodes (only the modified part of the page should be broadcast).

3.2 An evaluation method for comparing models

3.2.1 An outline of the evaluation method

In order to evaluate the models mentioned in the previous section, we analyzed traces of evaluation programs. We estimated the execution time of programs using

expressions of the number of execution steps expressed as functions of parameters such as dimensions of the problem. These expressions were calculated from assembler (we used SPARC's) derived from body part of evaluation programs. We also concerned about accesses to shared memory, pagefaults, synchronizations, and so on.

This method, compared with trace-driven simulation which counts the number of execution steps automatically, needs more time to analyze a program, but the calculation time itself is quite short once analysis is finished. In general, evaluation of parallel processing takes a very long time. Moreover, to follow traces of a program takes a lot more time than just executing it, so trace-driven simulation requires a very long time, especially for a large problem size. Overall, analyzing traces has the advantage that it shows whole constitution of programs, so that it is able to handle various kinds of models flexibly. We will show in this section how we have analyzed programs and handled each model in this evaluation.

3.2.2 Some matters to be considered when analyzing programs

Some implementation-dependent matters

In this evaluation, only the body of the evaluation programs are treated. That is, the following programs are neglected: allocating and initializing shared memory and semaphores, forking and allocating processes, outputting results, and so on. Acquisition and release of semaphores, however, are considered. We will mention them later.

Treatment of non-deterministic control structures

"For" statements can be analyzed and expressed as functions of parameters explicitly, but some "if" clauses cannot determine their structures until they are executed. They are decided by the probabilities expressed by parameters, initial values of a

problem, and execution timing. We coped with them using an average value, or adopting a majority case value neglecting exception cases. Also we assume that system and initial values of problems are symmetric so that executions of the program are not biased to a special case.

Execution time of each step

We assume that executions on local memory are pipelined, so one instruction is executed in one clock cycle time. Pipeline hazards caused by branch instructions are considered to be resolved by nop operations inserted behind these instructions. Namely, the execution will resume 2 clocks later.

We assume a 25MHz machine clock, and hence local memory access time, which is equal to one clock cycle time, is 40ns. Access time to global memory, on the other hand, is assumed to be 1000ns. This value was decided by referring to SCRAMNet™'s access time. This is the "int" variable's access time, so access time for a "double" variable is twice as long as this value.

3.2.3 Relationship between pages and data

In Model I and Model III, shared data is assumed to be ideally mapped on shared memory. That is, variables accessed in different critical sections are placed in different pages. Also variables accessed in the same critical sections are placed in the same page, if possible. Pagesize is assumed to be 1Kbyte.

If variables accessed in different critical sections are mapped onto the same page, a copy of the page might be invalidated by other nodes while it is in use, even if a semaphore is acquired. This does not cause memory consistency problem, but a false-sharing state occurs. On the other hand, if variables accessed in the same critical sections are not mapped onto the same page, the number of pagefaults would increase. Variables used in normal sections of each node are also mapped onto different pages so as not to cause false-sharing.

3.2.4 Network model

DSM models mentioned in the previous section do not define any kind of network topology. This means DSM can be realized on any kind of network only if one-to-one and/or one-to-all communication is supported. In this paper, we suppose a single-direction, multi-access, ring-connected type network, such as a single-direction slotted ring or register insertion ring. One of the reasons why we chose this topology is that a ring-connected type network is used in various kinds of environment, ranging from local area to wide area. Another reason is that this topology is suitable for evaluation because features of the network can be represented by its length, and memory consistency problems are fairly simple. Such simplicity makes the characteristics of the memory models highlighted.

We assume an optical fiber network, and define round-trip delay of the network according to the next equation, referring to SCRAMNet™'s case[18].

$$delay[ns] = 500 \times nproc + 5 \times L$$

Nproc stands for the number of nodes and L stands for network length[m]. We neglect queuing delay here, because, as we can see in the results afterwards, the average communication rate is several megabit per second at most, while several hundreds megabit or gigabit per second network is assumed in this evaluation.

Semaphore acquisition requests, barrier synchronizations, pagefaults, and invalidation of pages must go around the network in any case. We use round-trip delay time as an approximation for these values. In a widely distributed environment, delay time becomes so large that this should be an appropriate approximation.

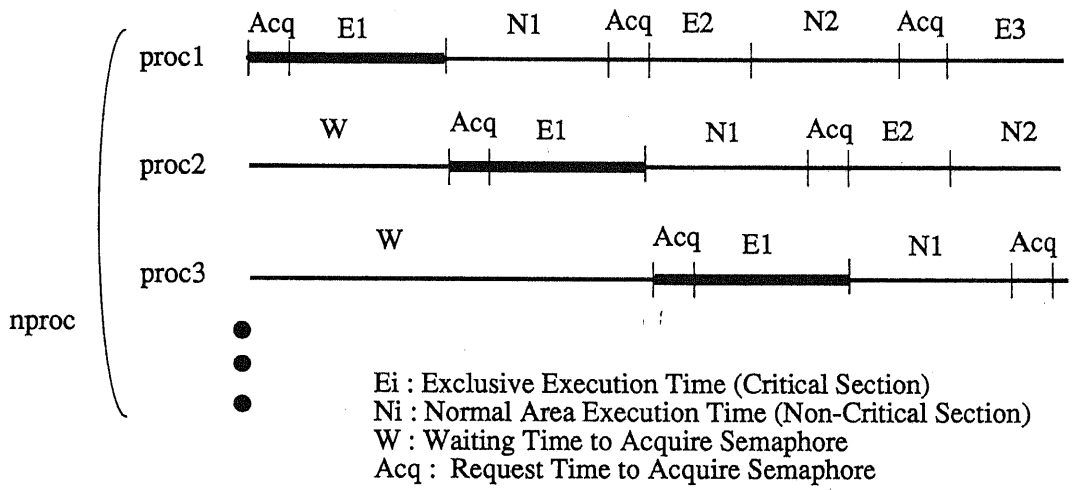
3.2.5 Exclusive execution

No collision will occur during pagefaults or invalidation of pages because all data are assumed to be mapped ideally so that all requests will succeed. In the case of acquiring semaphores, however, we should consider not only request time but also

waiting time if the semaphore has already been acquired by another node. If the execution time of critical and normal sections are constant, and jobs are allocated to each node evenly, the waiting time to acquire the semaphore is determined by the longest execution time of critical sections, as shown in Fig.3.3.

$$W = nproc \times (Acq + Max\{E_i\}) - \sum(Acq + E_i) - N$$

The waiting time is zero if the right hand side of above equation is negative.



$$W + \Sigma(Acq + Ei) + N = nproc \times (Acq + Max\{Ei\})$$

$$(N = \Sigma Ni)$$

$$\therefore W = nproc \times (Acq + Max\{Ei\}) - \Sigma(Acq + Ei) - N$$

Figure 3.3: Exclusive Execution Timing Chart

In the case of releasing semaphores, no memory consistency problem should happen if a release message will not overtake data written beforehand. Thus we assume that releasing semaphores requires only the as much time as that of normal global memory access.

3.3 Comparison of models with an evaluation program using barrier synchronizations

3.3.1 Evaluation program

We chose a Gaussian elimination method for solving simultaneous equations as an evaluation program which uses barrier synchronizations. The Gaussian elimination method is composed of three parts: finding a pivot row, forward elimination, and backward substitution. The forward elimination part, which occupies the majority of time in execution, is parallelized in this program.

The number of execution steps allocated to each node is not equal so that execution time of the whole program is determined by the execution time of the node allocated the largest number of jobs. Barrier synchronizations are used for absorbing these differences so as to prevent the so-called race condition. Exclusive executions are not required, on the other hand, because no collisions occur when shared variables are being updated in critical sections.

Of course it is not realistic to execute this problem in a distributed computing environment, because such a problem is suited to calculation on parallel computers. Unfortunately, a well-recognized benchmark program for distributed computing is not exist. Hence we chose this Gaussian elimination method as a benchmark program, because the behavior of Gaussian elimination method is considered to be similar to that of applications in distributed computing realized in a widely distributed environment. Namely, such applications do not perform exclusive executions frequently: most of their synchronizations are low-cost primitives such as a barrier synchronization, and those frequencies are not high.

3.3.2 Evaluation results

The execution times for each model versus network length L are shown in Fig.3.4. Here, the problem size (dimensions of equations), n is 1000 and the number of nodes,

nproc is 32.

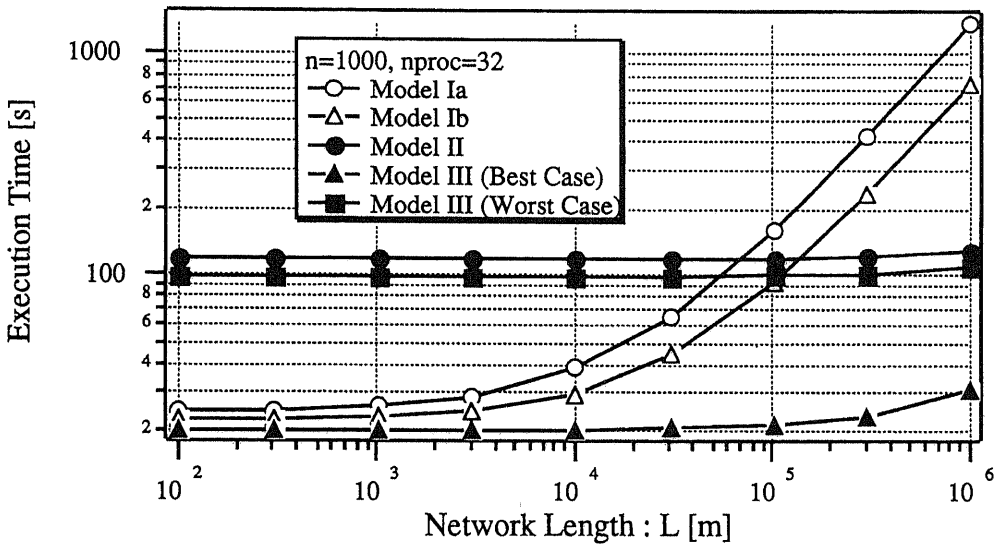


Figure 3.4: Execution Time of the Gaussian Elimination Method

When the network length is short, the execution time of replicated shared memory (Model II) is longer than that of shared virtual memory (Model Ia and Model Ib). This comes from the difference in memory access time. When network length becomes long, the execution time for shared virtual memory increases dramatically, while the execution time for replicated shared memory remains almost the same. As a result, the relation between the execution times for two models is reversed.

As network length varies, execution time of a barrier synchronization, a pagefault, and a page invalidation also changes. The replicated shared memory model uses barrier synchronizations but has nothing to do with pagefaults and page invalidation, so the barrier synchronization seems to have little effect on system performance while pagefaults and page invalidation have a great influence on the results. In this application, the number of barrier synchronizations is O(n) as they are executed only in outer loops, while the number of pagefaults is $O(n^3)$ because O(n) data accesses are required for each inner loop. In general, the number of pagefaults is expected to be relatively large when a great amount of data is processed. In

such cases, replicated shared memory is considered to be more suitable than shared virtual memory, especially for widely distributed computing because the penalty of pagefaults is quite large.

The worst case of the hybrid model (Model III) does not improve so much from the replicated shared memory model. This is because the time to copy all contents of one page from global memory to local memory is so large that it eliminates the merit of local memory's cache effect, which is coming from the difference of the access time between global memory and local memory. Therefore, if the copy time is shortened, for example, a burst transfer mode is used, performance must increase. The best way to achieve this is to use a DMA controller, which can hide the copy time by permitting to execute the following steps concurrently. In this case, the best performance for the hybrid model can be expected.

The best case of the hybrid model is also shown in Fig.3.4. It achieves excellent performance as the execution time is relatively short with short network length and remains almost constant even when L increases. With a short memory access time for shared memory and a small penalty for pagefaults, this model is preferable in most cases. We will discuss only the best case of the hybrid model in the following section.

Next, the amount of data transferred through the network during the execution of the program is shown in Fig.3.5. Here, shared virtual memory and replicated shared memory is compared when n and $nproc$ vary. From this figure, we can see the amount of transferred data of the replicated shared memory model is a little smaller than that of the shared virtual memory model. And the replicated shared memory model does not yield burst traffic, unlike the shared virtual memory model which must yield burst traffic when a pagefault occurs. According to these matters, replicated shared memory is preferable in terms of data transfer. Only objection is that the replicated shared memory model requires broadcast communication, but this is not so much different from point-to-point communication in the case of a ring-connected type network.

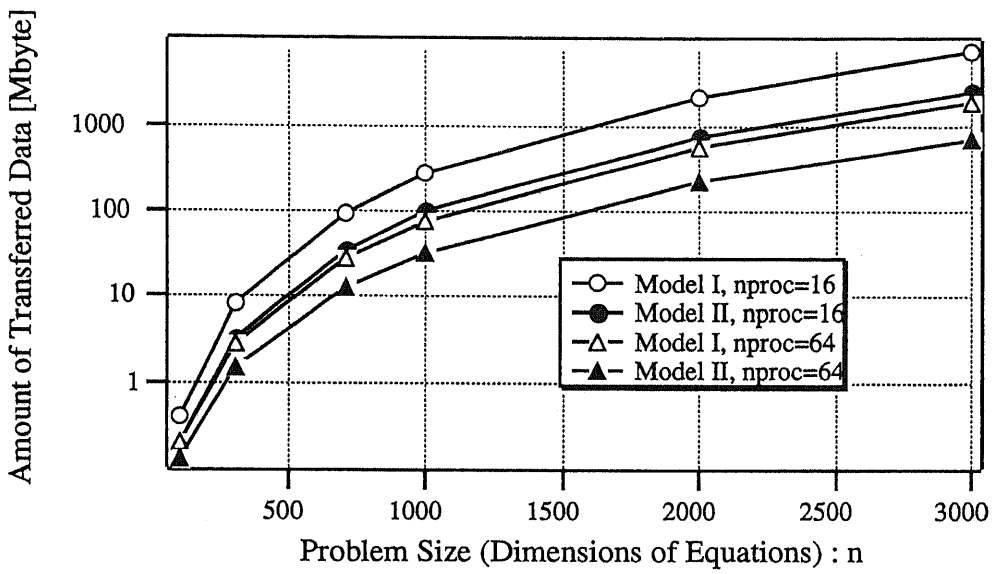


Figure 3.5: Amount of Transferred Data vs. Problem Size (The Gaussian Elimination Method)

3.4 Comparison of models with an evaluation program using semaphores

3.4.1 Evaluation program

We chose a traveling salesman problem as another evaluation program. This program works as follows: in a main loop, each process allocates itself a part of the problem by calling a scheduling function, then the minimum distance of the allocated part is calculated by calling recursive functions. Calling the scheduling function and updating shared variables with the results of each iteration must be executed exclusively, so semaphores must be used. Barrier synchronizations are not used because loop execution timing at each node does not have to be synchronized.

To achieve good performance, we should allocate a large portion of the problem

at each iteration. In this evaluation, however, we allocated a relatively small part at each iteration to highlight the differences among the memory models.

In this problem, the amount of data required in each critical section is small enough to be allocated in one page. As a result, a pagefault occurs only once in each critical section.

Of course it is not realistic to execute this program in a distributed computing environment again. This program, different from the Gaussian elimination method, uses frequent and high-cost synchronizations. For example, each loop contains almost 500 steps of exclusive executions while the steps of normal executions are 1500, when the problem size (the number of cities) n is 12. Thus this example should be assumed as a worst case analysis, in which execution is dominated by synchronization.

3.4.2 Evaluation results

Execution times for each model versus network length, L are shown in Fig.3.6. Here, the problem size (the number of cities), n is 12 and the number of nodes, *nproc* is 32. The performance of each model is similar to the case of Gaussian eliminations when network length is short. When network length becomes long, though the execution time of shared virtual memory increases just like the case of Gaussian eliminations, the execution time of replicated shared memory also increases, unlike the previous case. Since this tendency arises from exclusive executions, we discuss this further.

The semaphore waiting times for each model and the semaphore request time (round-trip delay time) are shown in Fig.3.7. This figure indicates the time to execute one semaphore operation.

According to this figure, semaphore waiting time, which is caused by a collision in a critical section, surpasses semaphore request delay in all models when network length is long. So semaphore request delay may be considered as having little effect on system performance. But actually, semaphore waiting time is influenced by

the semaphore request delay of other nodes, so system performance is decided by semaphore request delay after all. According to this result, it can be seen that the method used to process exclusive executions has a great influence on system performance as the network becomes larger.

Next, the amount of data transferred through the network during the execution of the program is shown in Fig.3.8. Here again, shared virtual memory and replicated shared memory is compared when n and $nproc$ vary. In this case, the amount of transferred data of the replicated shared memory model is far smaller than that of the shared virtual memory model. This is because the amount of data used in one iteration at each node in this problem is so small that to copy all contents of the page yield a lot of unnecessary data transfer in the shared virtual memory model. Thus replicated shared memory is considered to be preferable in terms of data transfer in this case also.

Finally, the execution time of each model versus the number of nodes, $nproc$ is shown in Fig.3.9. Here, the network length, L is fixed at 3000. Although the performance of all models increases up to around $nproc = 20$, thereafter they saturate, and even become worse, especially in the shared virtual memory model. This saturation is caused mainly because too small a part of the problem is allocated to each iteration, as mentioned in the previous subsection. In this case also, replicated shared memory and the hybrid model are better than shared virtual memory.

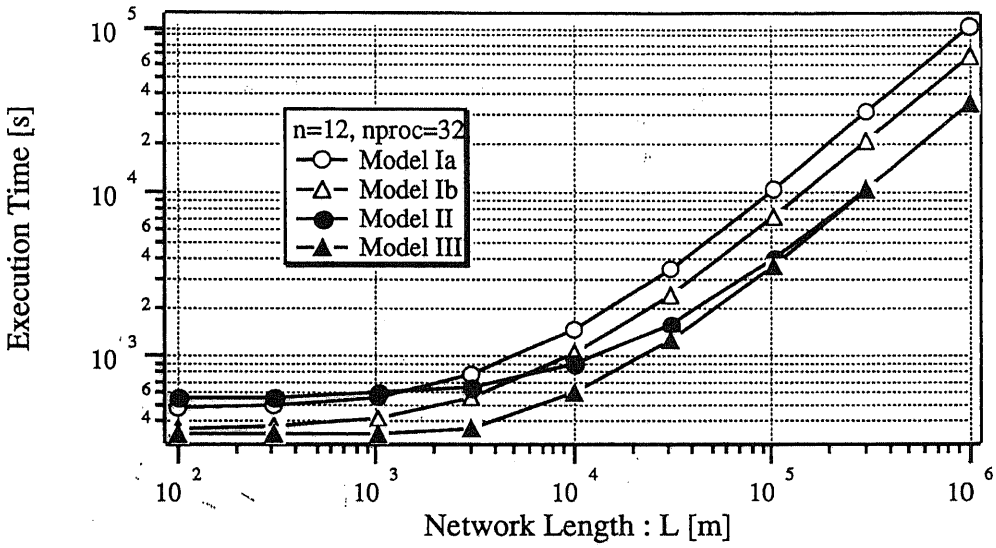


Figure 3.6: Execution Time of the Traveling Salesman Problem

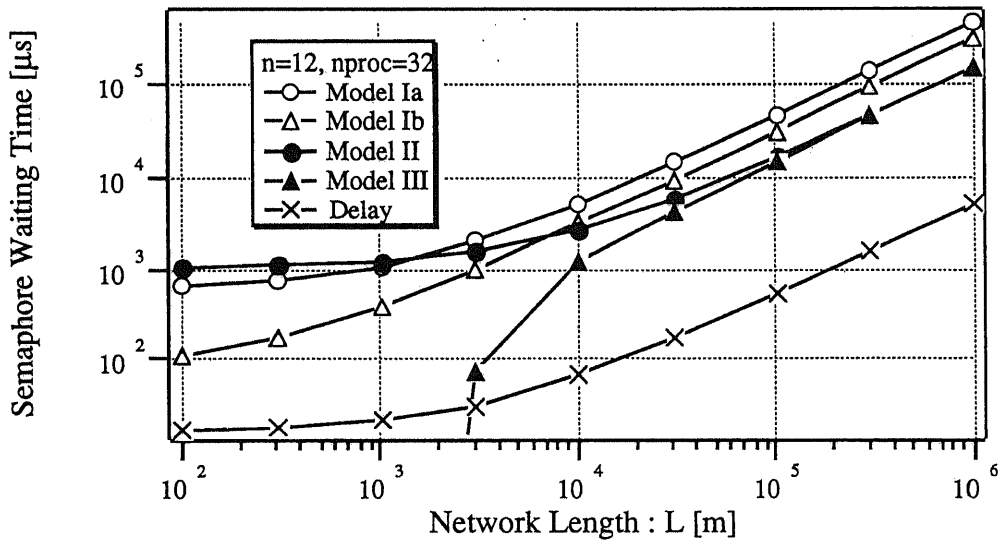


Figure 3.7: Waiting Time and Delay Time to Acquire Semaphore

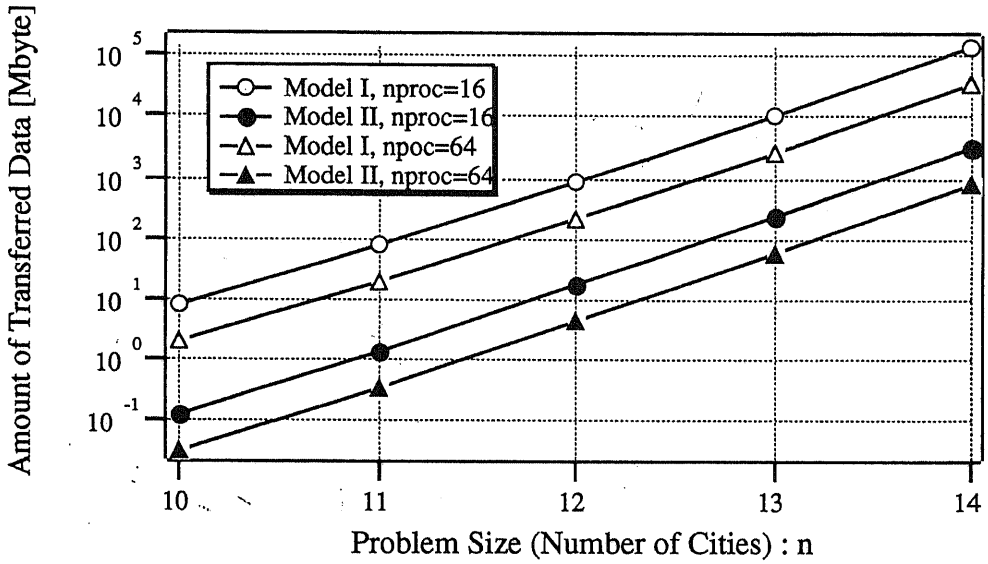


Figure 3.8: Amount of Transferred Data vs. Problem Size (The Traveling Salesman Problem)

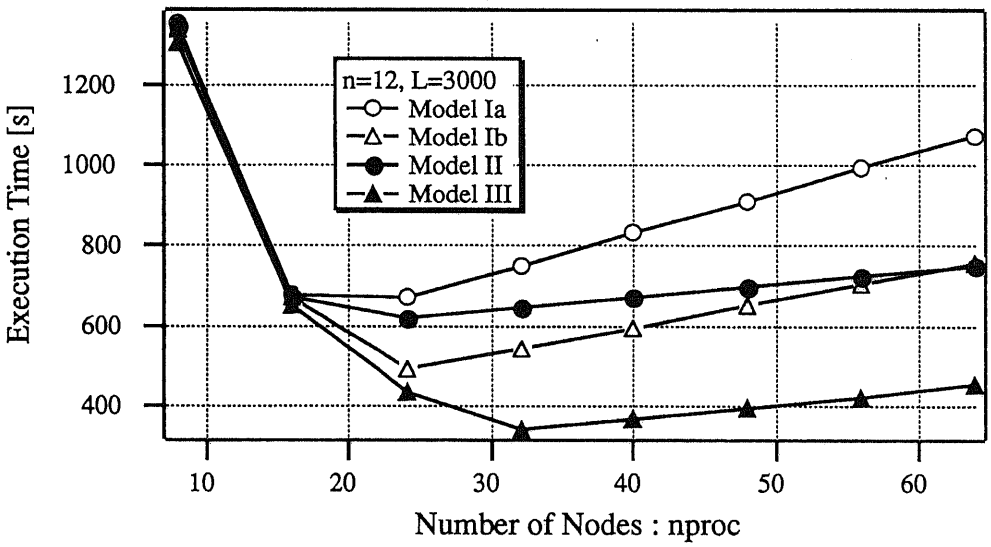


Figure 3.9: Execution Time vs. Number of Nodes (The Traveling Salesman Problem)

3.5 Summary of the evaluation

Three kinds of models, shared virtual memory, replicated shared memory, and their hybrid model are discussed in this chapter. Results of preliminary quantitative evaluations show the superiority of replicated shared memory against shared virtual memory when the length of the network is larger than 10^4-10^5 m, not only in the case that barrier synchronizations are used but also when semaphores are used. The hybrid model achieves the best performance whatever the network length is, so this model is suitable for the widest variety of cases. While barrier synchronizations have little effect on system performance, exclusive executions using semaphores have great impact on the performance, especially in widely distributed environments.

Chapter 4

Synchronization Mechanisms for Replicated Shared Memory

4.1 Synchronization using a semaphore controller

4.1.1 Why is a semaphore mechanism needed for replicated shared memory?

In replicated shared memory, data written to shared memory is first stored in a FIFO buffer, and sent to the network later. Thanks to this FIFO buffer, one can expect high performance because an originating node can resume its work without waiting for an acquisition of the network. Critical sections, on the other hand, bring about some problems. It is difficult for replicated shared memory to implement the synchronization methods using shared memory directly, such as Test&Set, due to the existence of the FIFO buffer. Thus some other methods, such as a semaphore mechanism, are required for replicated shared memory.

4.1.2 Realization of the semaphore controller

One method to realize a semaphore mechanism in replicated shared memory is using a semaphore controller. The semaphore controller is provided in the network, as is shown in Fig.4.1, and it manages synchronizations using global memory. With a semaphore request table having entries for each node and a semaphore acquisition node-ID area, this method works as follows:

Acquiring a semaphore

1. A node willing to execute a critical section checks the semaphore request table and makes sure the resource in need is available.
2. Then the node writes an request into the semaphore request table.
3. This request is sent to the network through a FIFO buffer, and is received by the semaphore controller.
4. If more than one node tries to acquire the same resource simultaneously, the semaphore controller chooses up to the number of available resources

and writes node-ID(s) into the semaphore acquisition node-ID area.

5. This approval of the request(s) is broadcast so that the requesting node(s) is informed whether the semaphore is acquired.

Releasing a semaphore

1. When execution of the critical section is finished, a release message is written into global memory by the node.
2. This message is sent to the network through the FIFO buffer so that the other nodes and the semaphore controller are informed that the semaphore was released.

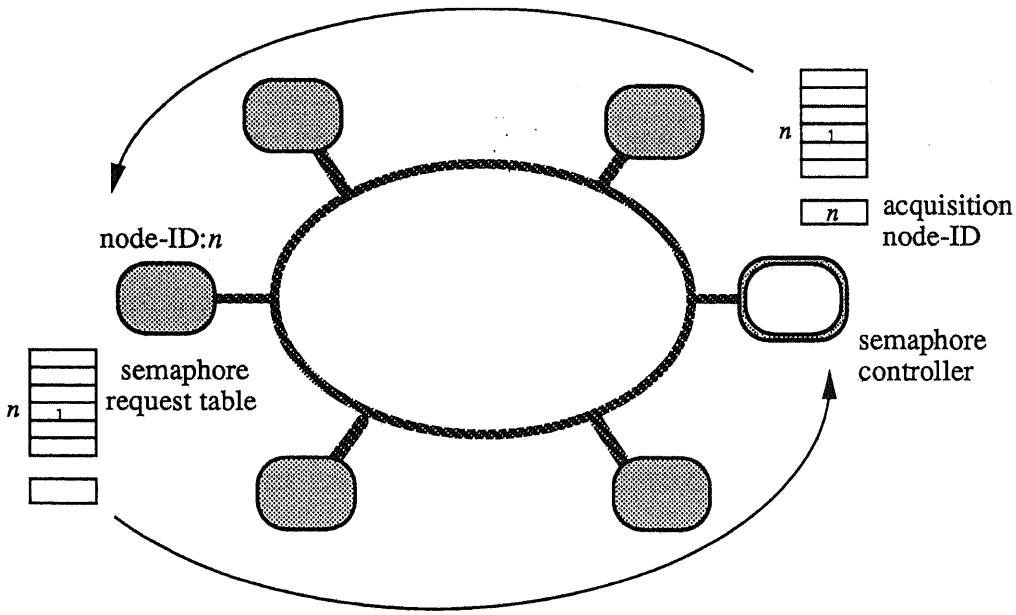


Figure 4.1: Basic Operations of a Semaphore Controller

In the case of failing to acquire a semaphore, a request must be resent when the semaphore is released by the other node. If the semaphore controller keeps all requests in its memory in FIFO-order, re-request is no longer necessary.

4.2 A distributed semaphore mechanism

4.2.1 Why is a distributed semaphore mechanism needed?

As is written in previous section, a semaphore mechanism can be realized by using a semaphore controller which manages synchronizations on global memory. A centralized management method like this, however, is not preferable for a large scale system because it may cause a bottleneck in semaphore executions.

In order to prevent this problem, one idea is to assign the semaphore management to some controller nodes. That is, some semaphore controllers are equipped on the network, and resources are divided and allocated to appropriate controllers. With this method, the system may scale well up to a fairly large scale system without causing the bottleneck problem in semaphore executions. However, this is an ad hoc method so that whole semaphore system must be re-constructed when the configuration of the system changes.

An alternative is to realize a distributed semaphore mechanism which has no centralized management organization like the semaphore controller. In order to realize this mechanism, a message packet sent to a network must reach all other nodes, and the originating node must be informed that it reached its destination. It is difficult to realize this mechanism in a general network topology, because all of the other nodes, in some cases, must each send back an acknowledgment to the originating node. In the case of a ring-connected type network, however, it is possible to realize an efficient distributed semaphore mechanism without any acknowledgment, by using its topological features.

4.2.2 A distributed semaphore mechanism using global memory

In a ring-connected network, a message packet sent to a network must have been received by all other nodes when the originating node receives its own packet re-

turned from the network. This characteristic is used as the basic idea of a distributed semaphore mechanism. A node willing to execute a critical section checks a semaphore request table on global memory, and writes an request into the table if the resource in need is available. Then the node checks the table again after its own request packet returns, and acquires a semaphore if no other requests exist on the table. This is described in Fig.4.2. If another node has sent a request for the same resource also while the first request packet was traveling, that node's packet must have passed this node so that the request must be found when the request table is checked later.

In the case that more than one node tries to acquire the same resource simultaneously, it must be decided which node's request succeeds and acquires the semaphore. This can be determined according to priority numbers. If one node's request priority is lower than any of the others', it suspends the request. The highest priority node makes sure that it succeeds in acquiring semaphore by confirming the suspension of the others' requests. Various schemes can be considered to decide the priority of requests. One simple method is to allocate a fixed priority number to each node. A more dynamic and impartial method is to allocate a random number to each semaphore request packet.

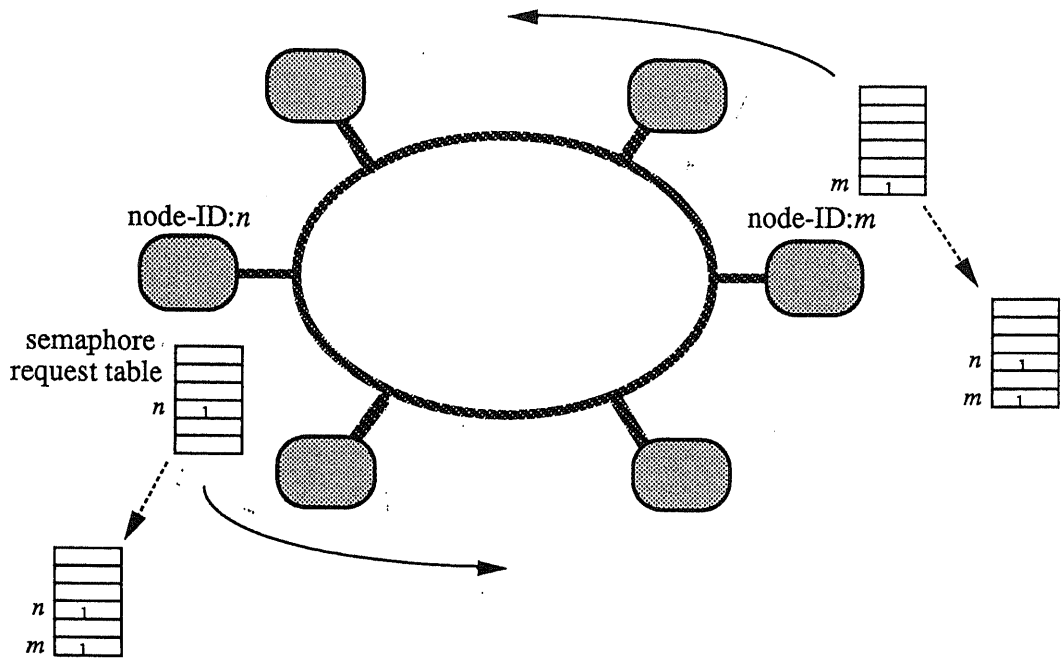


Figure 4.2: Basic Operations of a Distributed Semaphore Mechanism

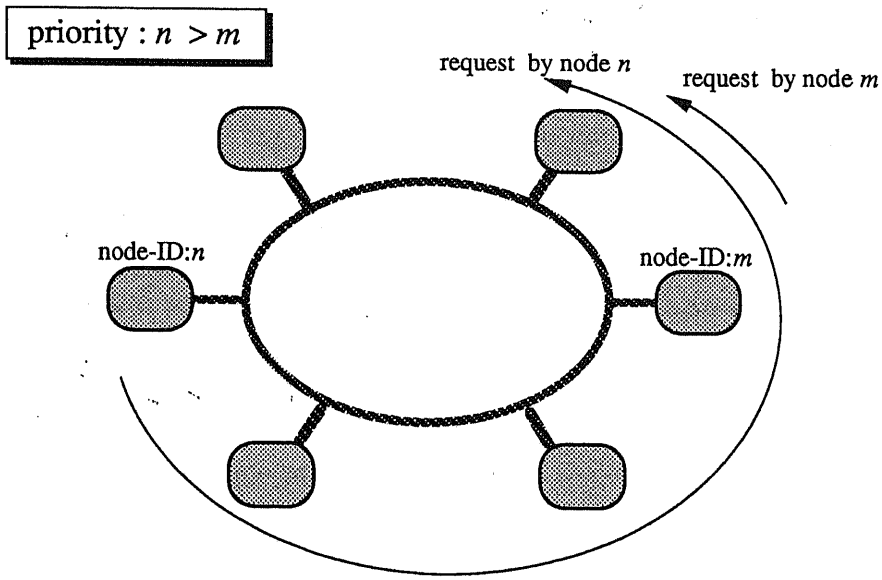
4.2.3 Critical timing cases

The distributed semaphore mechanism is realized with the method described in the previous subsection, but there are some critical timing cases in which a semaphore acquisition node cannot be decided properly. These cases occur when a node checks the request table at first and finds no other requests exist but another request packet passes the node before the node issues a request. These cases are shown in Fig.4.3.

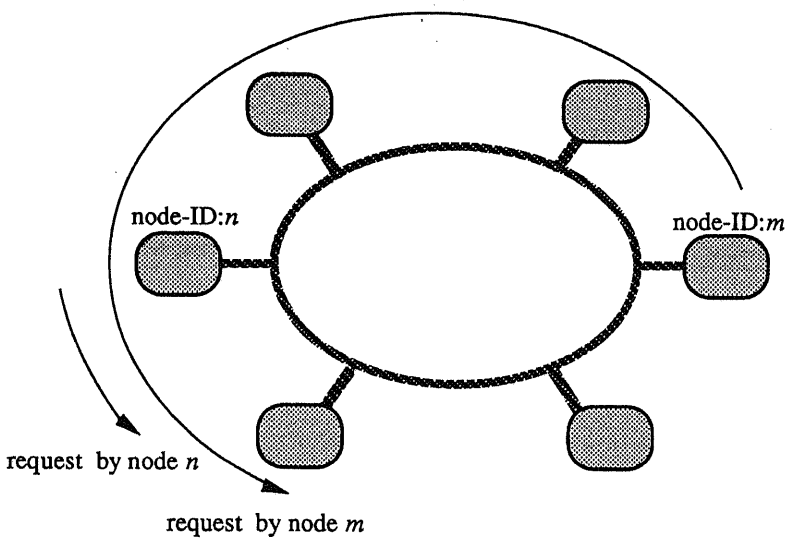
Fig.4.3 (a) indicates the case that a lower priority node issues a request after a higher priority node issues. In this case, the higher priority node may acquire the semaphore without noticing the other request. The lower priority node, however, must notice the other request when its own packet returns so that no problem occurs.

Fig.4.3 (b) indicates the another case in which a higher priority node issues a request after a lower priority node issues. In this case, the lower priority node may

acquire the semaphore without noticing the other request. As a result, the higher priority node is kept waiting for the suspension of the other request, which is in vain.



(a) case 1 : lower priority node issues a request later



(b) case 2 : higher priority node issues a request later

Figure 4.3: Critical Timing Cases

4.2.4 Solutions of the request collision problem

One solution to the problem described in previous subsection is to set a timeout function. That is to say, if a requesting node sees that another request is not suspended for a certain period of time, it suspends its own request even though it has a higher priority than the other request. In this method, however, it is difficult to decide an appropriate time to wait because the round trip time of a packet depends heavily upon the network scale and its traffic.

Then semaphore acquisition node-ID area is used to solve this problem. When a node succeeds in acquiring semaphore, it writes its node-ID into this area so that the message is sent to all other nodes. In ring-connected type network, it is difficult to know when a message packet arrives at other nodes, but no overtaking occurs. Therefore, if some nodes issue requests simultaneously, at most only one node may see the request table which includes only the request issued by itself. In other words, it will never happen that more than two nodes write into the semaphore acquisition node-ID area.

If a requesting node finds another request which has lower priority when it checks the request table, it can see on global memory whether the other node suspends the request or acquires semaphore, by using the semaphore acquisition node-ID area as well as the semaphore request table.

4.3 A study of memory coherency

4.3.1 Memory consistency model for replicated shared memory

In replicated shared memory, it is impossible to know exactly when a packet arrives at other nodes because a FIFO buffer exists. This means new data may be written to global memory while the previous data has not yet reached the other nodes. This is also true in the case of synchronization accesses such as semaphore acquisition

requests and release messages, so that release of the semaphore may be issued while other nodes have not yet received data written in a critical section.

Thanks to the FIFO buffer, however, normal and synchronization accesses to global memory are sequentialized so that release consistency[21] is maintained in this model.

The sufficient condition for release consistency shown in [21] is as follows:

Condition 4.3.1

(A) Before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed.

(B) Before a release access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed.

(C) Special accesses (acquire and release) are conform to processor consistency with respect to one another.

The processor consistency[22] mentioned above is another memory consistency model, and its conditions are stricter than that of the release consistency. Performing LOAD and STORE are defined as follows; remember that acquire is equivalent to LOAD, and release is equivalent to STORE:

Definition 4.3.1

(A) A LOAD by processor P_i is considered performed with respect to processor P_k at a point in time when the issuing of a STORE to the same address by processor P_k cannot affect the value returned by the LOAD.

(B) A STORE by processor P_i is considered performed with respect to processor P_k at a point in time when an issued LOAD to the same address by processor P_k returns the values defined by this STORE.

(C) An access is performed when it is performed with respect to all processors.

In replicated shared memory, it is impossible to know exactly when a STORE access is performed as described in Definition 4.3.1, due to the FIFO buffer. In [23],

however, it is shown that a STORE access can be defined in a different way and used in the conditions of a memory consistency model, if transactions in a network have no possibility of overtaking and the network nodes' buffers maintain a FIFO order. This definition is as follows:

Definition 4.3.2

A STORE access is considered performed with respect to a node when the node has buffered the STORE access for propagation to all other nodes in its subsystem.

The latter definition can be used for the condition of release consistency. It is possible for replicated shared memory to keep Condition 4.3.1, if Definition 4.3.2 is used as a STORE access. Thus replicated shared memory observes release consistency.

4.3.2 Possibility of synchronization access bypassing

In replicated shared memory, both normal data and synchronization accesses go through a FIFO buffer, whether the semaphore controller or the distributed semaphore mechanism is used. Therefore, if a processor accesses global memory heavily, STORE accesses may fill up the FIFO buffer so that synchronization accesses cannot be sent out to the network immediately, until all previous STORE accesses have been sent. In order to prevent this problem, it is possible for synchronization accesses to be sent to the network directly without going through the FIFO buffer. In this subsection, memory consistency in such a case is discussed.

If an acquire access is bypassed, there is the possibility that the acquire access overtakes the STORE accesses issued beforehand. However, this has nothing to do with the sufficient condition for release consistency described in Condition 4.3.1. Rather, this is preferable because taking less time to know another node's acquisition of the semaphore reduces the waste.

If a release access is bypassed, on the other hand, there is the possibility that the release access overtakes STORE accesses in a FIFO buffer. In this case, the

sufficient condition described in Condition 4.3.1 (B) is no longer satisfied. In fact, if a release access is bypassed, other nodes may receive it and begin to execute a new critical section before any STORE accesses in the previous critical section are received. Therefore, release consistency is violated.

In conclusion, if synchronization accesses are desired to travel faster, acquire accesses can be bypassed, but release accesses are obliged to go through the FIFO buffer.

4.4 Evaluation of the distributed semaphore mechanism

4.4.1 Implementation on SCRAMNet^{TM}

The distributed semaphore mechanism proposed in Section 4.2 is implemented on SCRAMNet^{TM}, an off-the-shelf replicated shared memory system. Five workstations (IRIS Crimson, SPARCstation IPX, SPARCstation 2, SPARCstation 1+, Sun-4/260) are connected by SCRAMNet^{TM} so as to configure a replicated shared memory system with five nodes.

An evaluation program, shown in Fig.4.4, is as follows:

Each node issues a request to acquire semaphore at first, and begins to perform synchronization accesses to global memory after it succeeds in acquiring a semaphore. When the synchronization accesses are finished, it releases the semaphore, and performs normal accesses to global and local memories. These executions are contained in one loop, and the evaluation program consists of repeating this loop. Both synchronization and normal accesses are write to memory.

The program is executed on five workstations, and evaluation values are measured on the SPARCstation 2 workstation. The request priority used in the case of a request collision is decided according to a random number allocated to each request. If the numbers of two requests are completely the same, node-IDs are used,

but such a case did not occur during the experiment.

As far as parameters are concerned, the number of synchronization accesses per loop is fixed, while the number of normal accesses per loop and the ratio of the number of local accesses to global accesses during normal accesses vary. Measured values are as follows: the synchronization execution time per loop, the ratio of semaphore request failure, the ratio of semaphore request failure in higher priority, and the ratio of the synchronization execution time to the total execution time. The synchronization execution time is calculated by subtracting the execution time of the program without synchronization from that of the complete program.

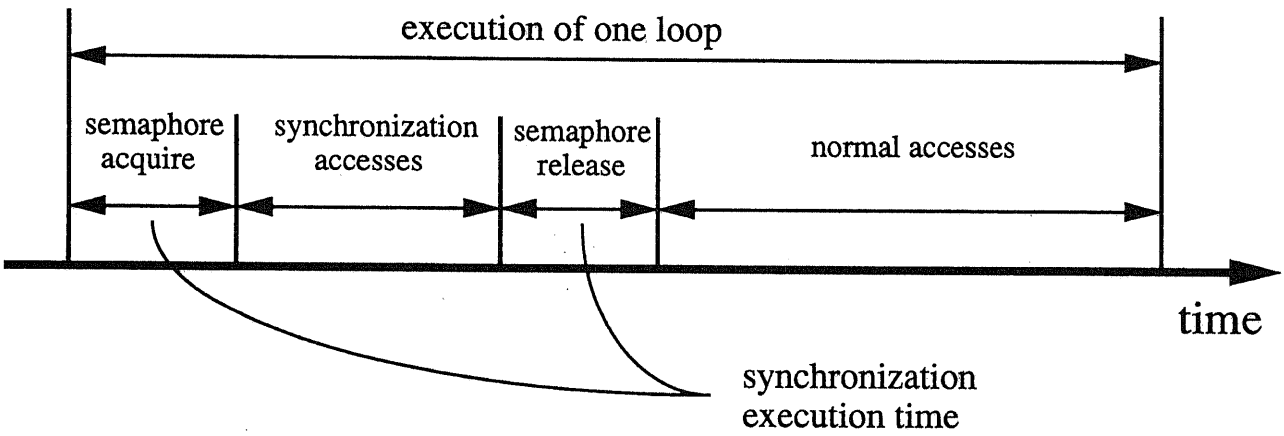


Figure 4.4: Evaluation Program Timing Chart

4.4.2 Evaluation results

The synchronization execution time per loop is shown in Fig.4.5, the ratio of semaphore request failure is shown in Fig.4.6, and the ratio of the synchronization execution time to the total execution time is shown in Fig.4.7.

In Fig.4.5, the synchronization execution time is almost constant when the number of normal accesses is large. This indicates that synchronization accesses are not performed frequently when the number of normal accesses is large, so that there is

little competition in synchronization and the time to acquire a semaphore is almost constant as a result. In this case, the synchronization execution time increases as the number of global accesses during normal accesses increases. This is because it takes a lot of time to exchange synchronization messages when the network is crowded as the number of global accesses becomes large.

When the number of normal accesses becomes small, on the other hand, the synchronization execution time increases dramatically. Since synchronization accesses are performed frequently in the case of the small number of normal accesses, semaphore acquisition requests should fail often and requesting nodes are compelled to wait for some time. This is proved in Fig.4.6, the ratio of semaphore request failure. When the number of normal accesses is small, the synchronization execution time increases as the number of local accesses during normal accesses increases. This is because the execution time of normal accesses themselves becomes shorter as the number of local accesses increases, so that synchronization accesses are performed more frequently and competition in synchronization intensifies as a result.

Fig.4.6 and Fig.4.7 indicate that if the number of normal accesses is large enough compared with the number of synchronization accesses, the ratio of semaphore request failure becomes small, and the ratio of the synchronization execution time to the total execution time becomes small also. In this experiment, when the number of normal accesses exceeds 500, which means the synchronization accesses are less than 2% of total memory accesses, both the latter two ratios become less than 10%. The ratio of semaphore request failure with higher priority is approximately 1% all the time, so it seems to have little effect on system performance.

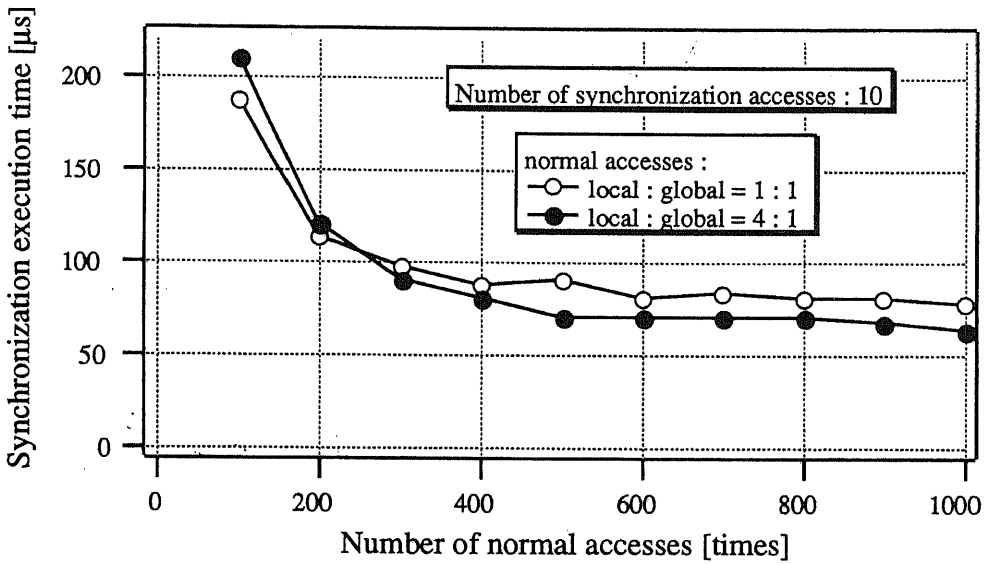


Figure 4.5: The Synchronization Execution Time

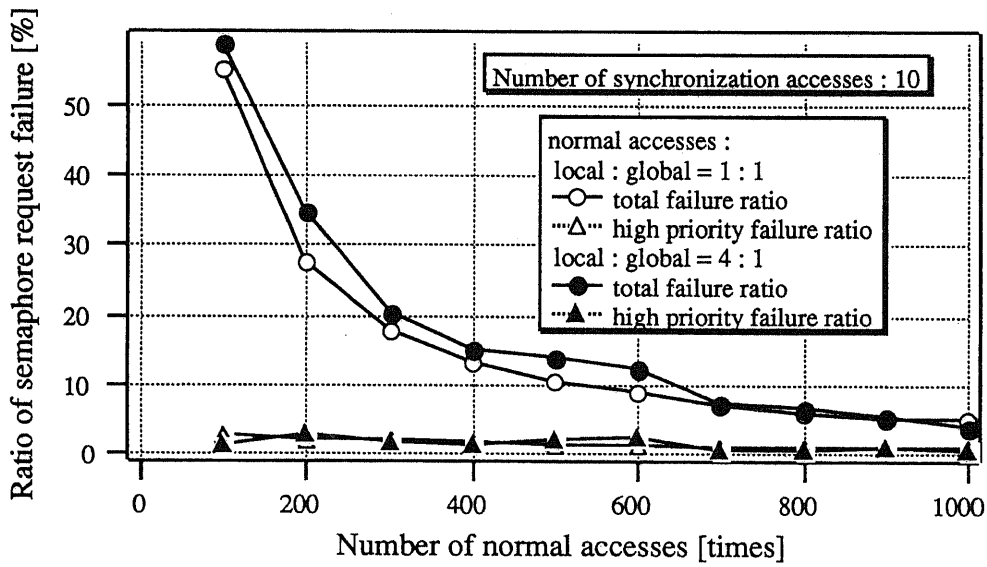


Figure 4.6: The Ratio of Semaphore Request Failure

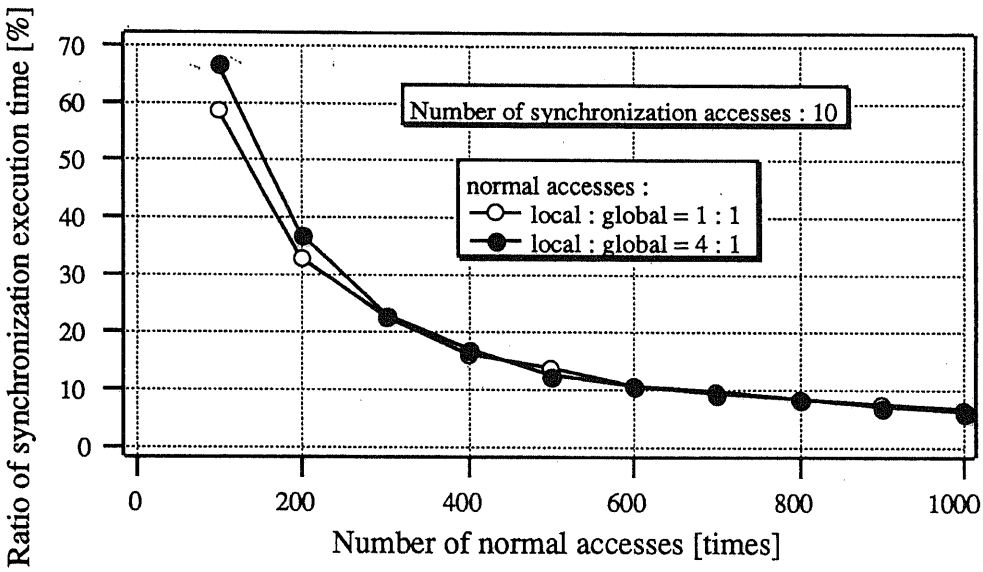


Figure 4.7: The Ratio of Synchronization Execution Time to the Total Execution Time

Chapter 5

A Proposal for a DSM Architecture suitable for a Widely Distributed Environment

5.1 An improved architecture

5.1.1 Problems of replicated shared memory

Replicated shared memory mentioned in the previous section is considered to be suitable for a widely distributed environment, because each node stores the whole contents of shared memory. As was mentioned, this mechanism is realized using a bus external to the computers, to which global memory is also connected. This method, however, is considered to have some problems.

First, access times to global memory on an external bus are much longer than those of local memory. As a result, system performance may decrease when variables on global memory are repeatedly accessed.

Another problem is the network. When replicated shared memory is constructed on an external bus, it is difficult to use a standard network interface with which most computer systems are equipped. Usually, replicated shared memory systems used a specialized network to connect nodes. In the case of a local environment, it may not matter even if an specialized network is used as well as a standard network such as Ethernet. However, this becomes a problem in the case of a wide area, since construction of a specialized network in a widely distributed environment is extremely troublesome.

5.1.2 An alternative method

In the rest of this dissertation, we call the memory which contains a copy of the entire contents of shared memory 'global memory'. Hence we use the term 'global memory' not only for the external memory of replicated shared memory, but also for the shared memory area which is allocated on internal machine memory.

In order to overcome the problems described in the previous subsection, we propose an alternative method to realize replicated shared memory. This is as follows (Figure 5.1):

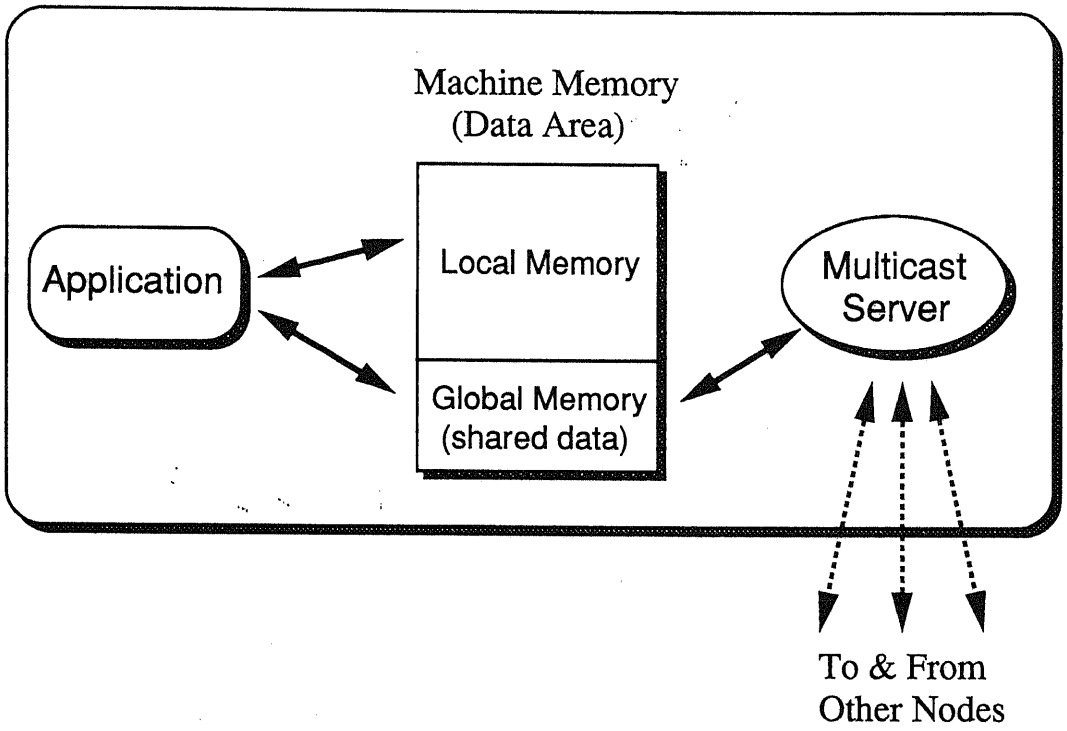


Figure 5.1: An overview of replicated shared memory using internal machine memory

Global memory, which means a shared memory area in this case, is allocated on internal machine memory (local memory) at each node. Each node has its own copy of the entire contents of shared memory in this global memory area. Write accesses to this area are broadcast to all other nodes by a multicast server so as to keep global memory consistent. Multicast servers can be realized in software, and a standard network can be used for this system. As opposed to the case of replicated shared memory using external memory, it is difficult to broadcast the global memory data every time it is written, so the contents of part of global memory are broadcast only after a series of executions have finished. This method can be realized using a semaphore mechanism, so that variables accessed in critical sections are broadcast only after a series of exclusive executions have finished.

Memory consistency in this model is looser than that of replicated shared memory using an external memory in which the global memory data is broadcast every time

it is written to. However, replicated shared memory using an external memory can only keep release consistency[27], which is also true for this model.

5.1.3 System environments required

Some problems have to be overcome in order to establish replicated shared memory using internal machine memory. One problem is software overhead. If multicast servers are realized in software, an application's execution must be suspended while global data is broadcast, if single-processor computers are used. This may be a huge overhead in some cases. Thanks to the extraordinary progress of computer system technology these days however, multiprocessors have become readily available. This is not only a trend for expensive specialized computers, but also even for workstations. Usually, only a couple of CPUs may be equipped in such computers, but that is enough for our purpose. If multi-CPU computers are used in this system, applications can continue their execution on one CPU, while broadcasting is performed by other CPUs. As a result, the application's software overhead for broadcasting is considered to be negligible compared with transmission delay, especially in a widely distributed environment. Multi-thread programming is advantageous in constructing this system, because multicast servers are supposed to share the contents of global memory with applications, which can be easily implemented using multi-thread programming.

A second problem of this system is network performance. If network bandwidth is not large enough, it is not realistic to broadcast all shared data which may not be used in some nodes. Fortunately, large bandwidth networks have become available both in LAN and WAN environments recently, thanks to the progress of network technology. For example, an ATM network has a very high throughput compared with traditional computer networks such as Ethernet, and it can be used seamlessly in LAN and WAN environments. Moreover, multi-casting is needed in order to realize replicated shared memory, which has a high cost in traditional networks,

but ATM networks are equipped with a multi-casting mechanism. Thus an ATM network is considered to be ideal for realizing our system proposed above.

5.2 A prototype implementation

5.2.1 Overview

The system configuration on a network is shown in Figure 5.2. Connecting all the nodes is a multi-casting network, while each node is also connected to a semaphore server.

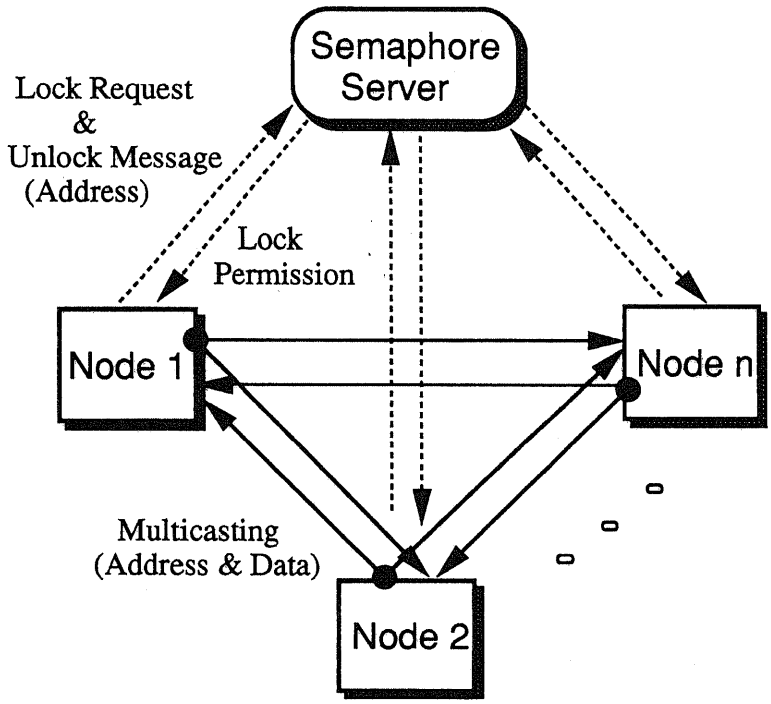


Figure 5.2: System configuration

Accesses to global memory, which contains data shared among nodes, are performed as follows: First, a node which desires to access global memory issues a lock request to the semaphore server, with arguments indicating an address (or address

range) of the global accesses. If some nodes make a request for the same region of global memory simultaneously, the semaphore server chooses one of them and sends back permission for the lock appropriately. The node which acquires the lock can access the region of global memory freely.

After the accesses are finished, the node issues an unlock instruction. The unlock instruction has the same arguments as the lock request, indicating an address (range) for the global accesses. The data contents of this region of global memory are first broadcast to the other nodes. Then the unlock message is sent to the semaphore server so that this region is released and becomes available to other nodes. Once the unlock instruction is issued by an application, the unlock procedure is performed by a multicast server, which is realized by another thread working on another CPU. Thus the application resumes its work immediately after the unlock instruction is issued, since the application does not have to receive any results from this procedure.

5.2.2 Prototype architecture

We have implemented a prototype of the system described in the previous section. This experimental system is realized on multi-CPU SPARCstations [1] with the SunOS 5.x operating system (which supports multi-thread programming), and they are connected by Ethernet [2]. TLI (Transport Level Interface) is used for network programming. All of the system functions are realized by library function calls, so that applications only have to issue some function calls such as initialization and synchronization. For advanced programming, an explicit multi-casting function is also provided.

The prototype architecture is shown in Figure 5.3. There are two kinds of servers in each node. One is a send server, which reads global data and broadcasts it to other nodes, and the other is a receive server, which receives broadcast data from

[1] Not all nodes are multi-CPU in our current implementation.

[2] We are now planning to implement this system on an ATM-LAN environment.

other nodes and writes it to global memory. Pairs of address and data are broadcast as a byte stream. Sending the absolute value of a global address, however, has no meaning because global memory may be mapped to a different area of machine memory depending on each node. Hence relative addresses, which are calculated from an offset value of the global area, are sent out, and re-translated to absolute addresses at the receiver.

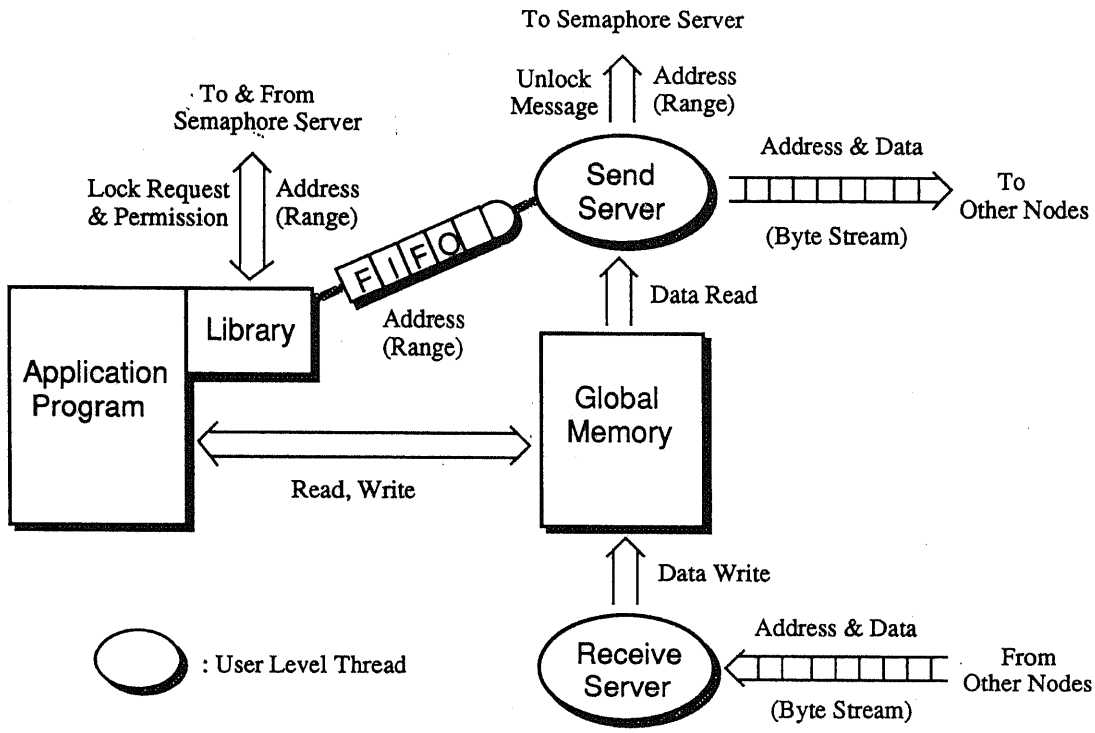


Figure 5.3: Architecture of prototype implemented on SPARCstations

A FIFO queue is provided between libraries and the send server. The libraries simply put an address (range) of access data into the FIFO. Thanks to the FIFO, applications and libraries will not be blocked even if the data cannot be broadcast immediately for reasons such as the send server being busy. When the FIFO becomes full, the library thread sleeps until it becomes available. The FIFO length can be easily changed.

Each node has a connection with the semaphore server, through which lock requests and permission for locks are transmitted. Unlock messages, by contrast, cannot be sent through this connection. They are put into the FIFO queue and sent out by the send server just the same way as normal data is broadcast, in order to keep global memory consistent[27].

5.2.3 Semaphore server

In this implementation, the semaphore server can be placed on any node. This is because the same network interface can be used whether the semaphore server exists within the same node or on the other node.

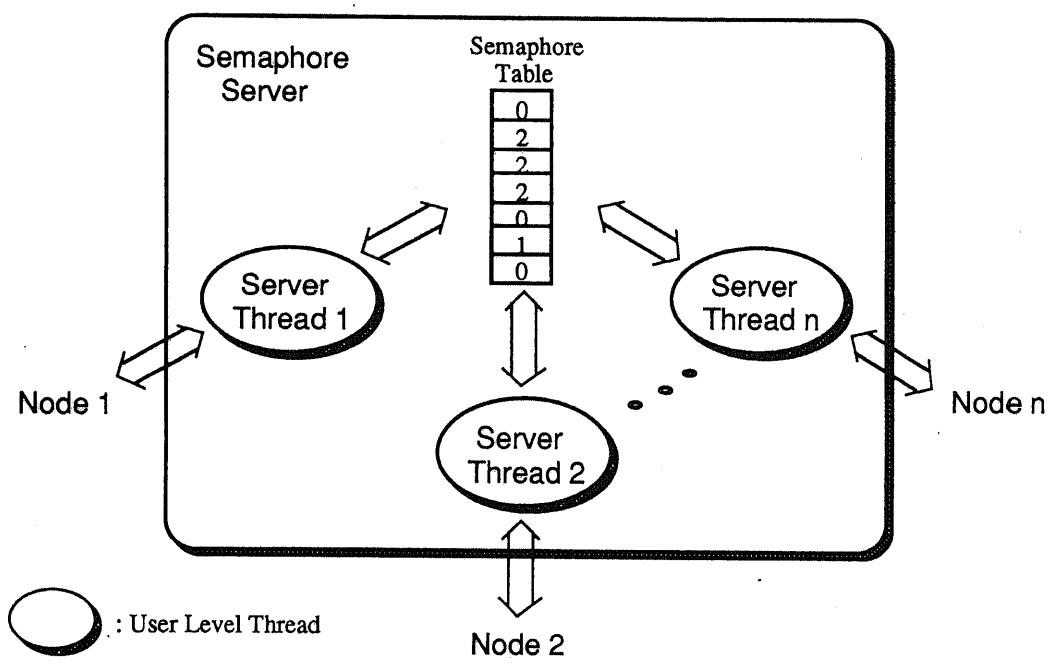


Figure 5.4: The semaphore server

The structure of the semaphore server is shown in Figure 5.4. The semaphore server is composed of a semaphore table and server threads, each corresponding to a node. Entries in the semaphore table correspond to relative addresses of global

memory, and they indicate whether or not the address is locked by any node. The semaphore table can be accessed by multiple server threads concurrently.

When a server thread receives a lock request, it tries to acquire a *mutex_lock* in order to access an area of the semaphore table which corresponds to the requested address (range). The *mutex_lock* is provided for each entry of the semaphore table respectively, so it is possible to access the table concurrently if the requested address (range) is different. After the server thread acquires the *mutex_lock*, it looks up the entry of the table, and then sets a flag on the table and returns a successful result to its node if the entry is available. If the entry is locked by another node, the server thread sleeps with a *cond_wait* call and waits until the entry is unlocked.

When a server thread receives an unlock message, it accesses the semaphore table in just the same way as the case of lock requests. In this case, the server thread removes an entry flag, and wakes any thread waiting for entry to acquire a lock by a *cond_signal* call. The thread which receives a *cond_signal* call locks the entry of the table at this point, and sends permission of the lock back to its node.

Usually, semaphores return the request result immediately after looking up a table whatever the result is. In this semaphore mechanism however, the result is not sent back when the request fails. If a failure message is sent back, the requesting node must transmit the same request to the semaphore server again. Such a method is not considered to be suitable for a system communicating over a network, because the network may sometimes be flooded with request and result messages. Above all, such a mechanism should not be used in a widely distributed environment, because high latency causes a decrease in performance.

5.2.4 Library functions

Library functions provided by this prototype system are as follows:

void *shm_init(size) /* shared memory initialization */

int size; /* shared memory size */

shm_init sets up the system environment. First, connections are established between the semaphore server and each node, and then multi-casting connections are formed among all nodes. Next, a global memory area, whose size is designated by the argument, is allocated and the first address of the area is returned to the caller. Server threads such as send server and receive server are also created and dispatched in this routine.

int shm_finish() /* wait until FIFO queue becomes empty */

In most cases, explicit termination function is not necessary because connections and allocated memory are automatically released by exiting programs. **shm_finish** waits until all entries in a FIFO queue are broadcast, which may be useful in some cases.

int shm_lock(addr1, addr2, step)
/* request to acquire a lock of shared memory */
int *addr1; /* start address of exclusive execution */
int *addr2; /* end address of exclusive execution */
int step; /* address interval of exclusive execution */

shm_lock requests to acquire locks for a global memory area used in exclusive executions. An address (range) of the memory area is designated by arguments. When only one data address needs to be locked, the first argument is used to indicate its address. Multiple data can also be designated by using the second and third arguments, not only for the case that the data are consecutive but also in the case that the data exists at regular intervals. If the data are consecutive, the first argument indicates the start address of a memory area and the second argument indicates the end address of the area used in exclusive executions, and the third

argument is not used. If the data exists at regular intervals, the first and second arguments are used in the same way, and the third argument is used to indicate the interval value. This implementation envisages a situation such as one row or column of a matrix being entirely locked.

```
int shm_unlock(addr1, addr2, step)
/* release the lock and multicast the area contents */
int *addr1;     /* start address of exclusive execution */
int *addr2;     /* end address of exclusive execution */
int step;     /* address interval of exclusive execution */
```

shm_unlock releases locks of a global memory area acquired by a *shm_lock* request. Before sending the unlock message to the semaphore server, contents of memory area locked so far are broadcast by the multicast server. Sending the unlock message is also executed by the multicast server, so this routine returns immediately after the request is put into a FIFO queue. Arguments are used in the same way as the case of *shm_lock*.

```
int shm_sync(addr1, addr2, step)     /* multicast the area contents */
int *addr1;     /* start address of exclusive execution */
int *addr2;     /* end address of exclusive execution */
int step;     /* address interval of exclusive execution */
```

shm_sync executes explicit multi-casting. In some applications, acquiring a lock before accessing shared memory is not necessary, if no conflict occurs. Since communication with the semaphore server is performed over a network, it is desirable to avoid unnecessary locks. Hence accessing global memory without acquiring locks is also allowed under the programmer's responsibility. In this case, the programmer should call *shm_sync* after a series of executions accessing global memory so that the latest data values are broadcast. Arguments are used the same way as the case

of *shm_lock* and *shm_unlock*. Actually, *shm_sync* is called by *shm_unlock*.

5.2.5 Example program

Example skeleton program is shown in this subsection. Although an address value returned by *shm_init* call may be different from node to node, application programmer does not have to be conscious of it.

```
#include "cast.h"

main()
{
    int *shm_ptr, *ptr;

    /* head address of shared memory is returned */
    shm_ptr = (int *)shm_init(MEMSIZE);

    ptr = shm_ptr;
    shm_lock(ptr, NULL, 0);
    /* execution with data of the designated area (*ptr) */
    shm_unlock(ptr, NULL, 0);

    shm_lock(ptr, ptr+10, 0);
    /* execution with data of the designated area (*ptr, *ptr++, ...) */
    shm_unlock(ptr, ptr+10, 0);

    shm_lock(ptr, ptr+100, 10);
    /* execution with data of the designated area (*ptr, *ptr+10, ...) */
    shm_unlock(ptr, ptr+100, 10);

    /* execution with no-lock is also possible */
    *ptr = 1;
    shm_sync(ptr, NULL, 0);

    shm_finish();
    exit(0);
}
```

Chapter 6

Quantitative Evaluation of DSM using Hardware Memory System

6.1 An evaluation method to compare proposed models

6.1.1 An outline of the evaluation method

Although the main purpose of DSM which constructs a distributed computing system in a network environment is not to improve performance, DSM models should be evaluated in some way. However, it is difficult to evaluate DSM models by software simulation, especially because in the case of a widely distributed environment the network has a great influence on performance. A distributed computing system is a complicated combination of computer nodes and a network system which is hard to describe by a simple event-driven or trace-driven simulation. For example, the number of accesses to shared data requiring communication among nodes may affect the performance of a network, which in turn influences system performance at each node.

Hence we had applied an analyzing traces method for an evaluation of DSM models[28]. In this method, some DSM models in a widely distributed environment are evaluated concerning both executions of jobs at each node and effects of a network. This method, however, is considered to be a little deterministic, while real distributed computing systems may somehow behave depending on probabilities. After all, some kinds of simulation, but not simple software simulation, are desirable for this purpose.

Therefore, we have decided to use SCRAMNet™ for an evaluation of DSM models. This choice is reasonable because SCRAMNet™ itself is based on a DSM model, and other DSM models can also be easily simulated on this network system. Since SCRAMNet™ is realized in hardware, little overhead exists when other DSM models are simulated on this system, which is suitable for a comparison. Moreover, this method is much more realistic than software simulation, because the computers are connected by a network and distributed computing jobs can be realistically

executed. Although it is difficult to realize a real widely distributed environment, we have simulated transmission delay using SCRAMNet™ so that such an environment is virtually realized. This will be explained later. Before details of this experiment is described, an overview of SCRAMNet™ is mentioned in next subsection.

6.1.2 An overview and functions of SCRAMNet™

SCRAMNet™ is ring-connected type network using 150Mbit/sec dual optical fiber. It can connect up to 256 nodes per single network ring, and maximum node separation is 300m. Available per-node bandwidth is 2.4Mbytes/sec for normal mode, and 6.5 Mbyte/sec for burst mode (explained later). Node latency is 250-800ns/node. Shared memory size is 8Kbyte-2Mbyte.

SCRAMNet™ supports various kinds of workstations and PCs. That is, it can connect to various kinds of external buses of computers, such as VMEbus and MULTIBUS.

Some of functions and mechanisms provided by SCRAMNet™ are as follows.

Error detection

When each node receives its own packets, it performs a bit-by-bit comparison of the received message with the transmitted message. If an error is found there, the message is re-transmitted.

Error detection on a receiving node is handled with parity bit checking. If there are any errors in a message packet, the message is taken off the network. The originating node will re-transmit the message when it does not receive the packet after the time-out period.

These error detection mechanisms are performed automatically by hardwares, so that no software overhead exists.

Burst mode transfer

SCRAMNet™ is based on the register insertion network protocol so that there can be many messages on the network at the same time. In normal mode, however, there may be only one message from each node on the network at any given time. That is to say, no new message will be transmitted by a node until its previous message has traveled the network ring and returned. Bit-by-bit error checking, mentioned in previous subsection, is performed.

In burst mode, on the other hand, it is possible for each node to transmit messages onto the network ring continuously without waiting for any message to return. If only one node is transmitting data while the other nodes on the ring are listening, then the transmitting node could achieve the 6.5Mbyte/sec bandwidth of the network if the host CPU could offer the data at that rate. When more than one node is transmitting in burst mode, then the effective output per node is the 6.5Mbyte/sec divided by the number of transmitting nodes. In this case, error detection still takes place on each node but the originating node will not re-transmit any message.

Interrupt operations

SCRAMNet™ has an interrupt mechanism which allows each processor to receive an interrupt from and to transmit to any other processors on the network. Interrupts can be generated under these two conditions: 1) data writes from the network to shared memory; and 2) a SCRAMNet™ network error detected on the local node. The first condition can be accomplished by two methods: 1) any data writes to any locations of shared memory from the network (forced mode); and 2) data writes to selected shared memory locations from the network (selected mode).

Each SCRAMNet™ node has an interrupt FIFO as well as the transmit and receive FIFOs. When a message to cause an interrupt comes into a node, the contents of the message is written into shared memory, the address of the message is put into the interrupt FIFO, and then an interrupt signal is sent to the host processor. The

processor knows the location of arriving data which cause the interrupt according to the contents of the interrupt FIFO. If additional network data interrupts occur before the processor is able to service the interrupt, those shared memory locations are updated and the addresses are added to the interrupt FIFO. However, no additional interrupt signals are sent to the processor until interrupts are re-armed. If an interrupt occurs before enabling interrupts, the interrupt will occur when interrupts are re-armed.

Control registers

Each SCRAMNet™ node has two types of control registers. One of them is Control/Status Register (CSR), and the other one is Auxiliary Control RAM (ACR).

The CSRs can control many kinds of SCRAMNet™ modes, such as normal/burst transmit, interrupt enable/disable, ACR enable/disable, and so on. Some kinds of errors, such as FIFO full or carrier missing, may be notified through the CSRs. The interrupt addresses which is kept by the interrupt FIFO can also be get through the CSRs.

The ACR provides a method of interrupt control when a particular SCRAMNet™ shared memory address is written into. In other words, the ACR is used for the selected mode of interrupt. One ACR byte is associated with every four-byte word of shared memory. When the ACR is enabled, the least significant byte of every shared memory four-byte address responds as the ACR byte. The ACR byte value will control interrupt actions taken whenever any byte of the shared memory four-byte word is written into.

6.1.3 Simulating a widely distributed environment using SCRAMNet™

In the case of realizing a distributed computing system in a widely distributed environment, delay is a significant factor to be considered. Delay consists of two

elements; transmission delay and node latency. Node latency, which includes protocol processing time at end nodes and routing time at relay nodes, may be shortened by realizing effective protocols and utilizing suitable hardware. Transmission delay however, cannot be shortened, which is significant especially in a widely distributed environment. Thus transmission delay, rather than node latency, should be focussed on as far as a distributed computing system in a widely distributed environment is concerned.

We have simulated transmission delay using SCRAMNet™ so that a widely distributed environment is realized virtually. One of the nodes connected by SCRAMNet™, which is called a delay node, handles transmission delay. When the delay node receives data transferred through the network, it sends out the data after a specified transmission delay time. As a result, the data arrives at its destination as if it were sent from a distant node.

In the case of SCRAMNet™, however, data packets received from the network are automatically sent out to the next node, so that it is impossible for any node to intercept packets transferred over the network. In order to overcome this problem, two data spaces are provided for SCRAMNet™ shared memory: one is for writing and the other is for reading. The data written to one space is copied to the other space by the delay node. That is, each node writes data on the address space for writing and other nodes read the data from the address space for reading. Since SCRAMNet™ is using a ring-connected network, the delay node may not exist between a sender and a receiver. But this is not a serious problem because actual network length is relatively short (we used 600m optical fiber) so that an extra round trip will not take so much time.

As the access time to SCRAMNet™'s global memory is quite large and a lot of data may be exist in the data space, it may take quite a long time to know arrival of data, if busy waiting method is used at the delay node. This may not be negligible in some cases. Then two measures are used in order to overcome this problem. First, the interrupt mechanism provided by SCRAMNet™ mentioned in

previous subsection is used. When the delay node's SCRAMNet™ receives data, SCRAMNet™ sends an user defined signal to the node's CPU, which in tern calls a signal handler so that the delay operation is performed. And second, unnecessary delay operation is avoided. Usually, a flag, which indicates that the sender's execution is finished, is sent at the end of a series of data transfer. It is only these flags that have to be delayed at the delay node. This is realized using selected interrupt mode described in previous subsection. In short, only flags which indicate a finish of sender's execution are registered at the delay node to cause an interrupt so that the delay operation will work. The mechanism of the delay node is shown in Figure 6.1.

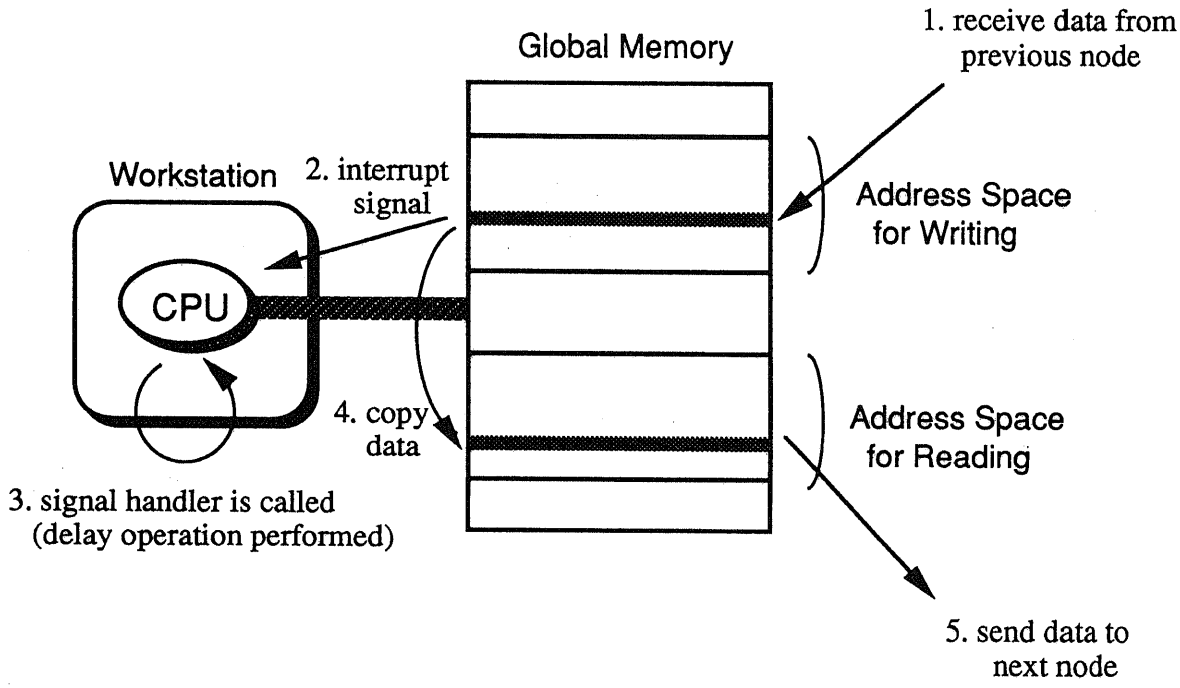


Figure 6.1: Mechanism of delay node

6.2 Models of distributed computing evaluated in this paper

In this section, some memory models evaluated in this paper are shown at first. They are shared virtual memory type and replicated shared memory type. How to simulate these models using SCRAMNet™ is shown. Then applications of distributed computing are generalized and implemented on these systems.

6.2.1 Memory models

Shared virtual memory

In general, shared virtual memory is hard to implement on commercially available computers because it requires modification of the operating system of the host machine, which is usually very difficult or impossible. Hence we decided to realize 'pseudo' shared virtual memory. In this method, page management is performed at the application program level, not at the operating system level as is in shared virtual memory. That is to say, the application sends a request for the page by itself when a pagefault occurs, and the execution resumes after the contents of the page are arrived. Of course this method is not realistic for actual systems, but this is enough for comparing the performance of shared virtual memory with replicated shared memory.

In order to compare the performance accurately, SCRAMNet™ is used for communication among nodes. In this case, global memory of SCRAMNet™ is not used for the execution itself, but used for sending and receiving messages and data. When the contents of a requested page arrive at a node, they are copied to local memory at first, then execution resumes using local memory only. A requested node, on the other hand, copies contents from local memory to global memory when a request message arrives.

Since shared virtual memory has studied actively until now, a lot of techniques

are proposed to improve the performance. For example, Munin[10] employs various data types and coherence protocols including write-update protocol associated to each variable, which can be designated by the application programmer. Such kinds of techniques seems to be effective for routine calculations whose data flow can be predicted in advance. In the case of distributed computing, however, unexpected operation may sometimes be executed, as is mentioned in Chapter 1. Thus these techniques are not considered to work effectively in distributed computing, so we adopt the simple write-invalidate model as a representative of shared virtual memory.

Replicated shared memory using external memory

This model is a part of SCRAMNet™ itself. An execution of a program at each node includes both local accesses and global accesses. In other words, global memory, which is on the external memory board in this case, will be read and/or written during execution. As is already mentioned, access time to global memory is much longer than that of local memory, but all data necessary for the execution must exist in global memory of each node so that no pagefault will occur. Performance of this model is considered to be heavily depending on the ratio of local and global accesses in the execution. The performance is also influenced by the ratio of read and write accesses to global memory, because only write access yields a message packet, which in tern influences network traffic so that latency may increase.

Replicated shared memory using internal machine memory

This is a 'pseudo' realization of the model proposed in Chapter 5. Global memory is allocated on internal machine memory in this case. Different from the case of replicated shared memory using external memory, data written to this global memory is not broadcast immediately. After a series of executions is finished, data which was used in this job is first broadcast, then a signal is sent to the next node, which indicates the end of executions at this node. Broadcasting is realized by copying from internal machine memory to SCRAMNet™'s external memory. SCRAMNet™ is

also used for communication among nodes, for the sake of the comparison.

Differing from the model proposed in Chapter 5, broadcasting is not performed automatically by other threads in this case. Modified data is copied from internal machine memory to SCRAMNet™'s external memory by the application itself. Also at receiving nodes, data sent from the previous node is first copied to internal machine memory by the application, then the actual execution begins. As a result, copy time cannot be hidden in this evaluation model, although one of the advantages of the model proposed in Chapter 5 is that broadcasting and receiving data are performed automatically by other threads. That is to say, the performance may be estimated to be a little lower in this evaluation model.

6.2.2 Application model of distributed computing

In order to evaluate the memory models mentioned in the previous subsection, we generalized the types of distributed computing application and implemented them on workstations. Unfortunately, there is no well-recognized benchmark program for distributed computing. Therefore, we chose a typical distributed computing application. Since this is only a simulation, the program does not accomplish any meaningful task.

First, an originating node hands a job to a following node with some data. The node which received the job begins its execution with the data, and hands this job to another node when the execution is finished. In this way, each job is executed at a number of nodes in succession. Unlike parallel processing, executions are not performed in parallel, but some jobs are running concurrently because the originating node yields multiple jobs and sends them out one after another. This model is considered to simulate the typical case of distributed computing applications.

The number of nodes at which each job is executed is decided in advance, but the next node in each case is dynamically chosen at random. The interval time at which jobs are sent out from the originating node is based on exponential distribution.

After the job has been executed at the decided number of nodes, it returns to the originating node again.

6.3 Evaluation results

6.3.1 Experimental conditions

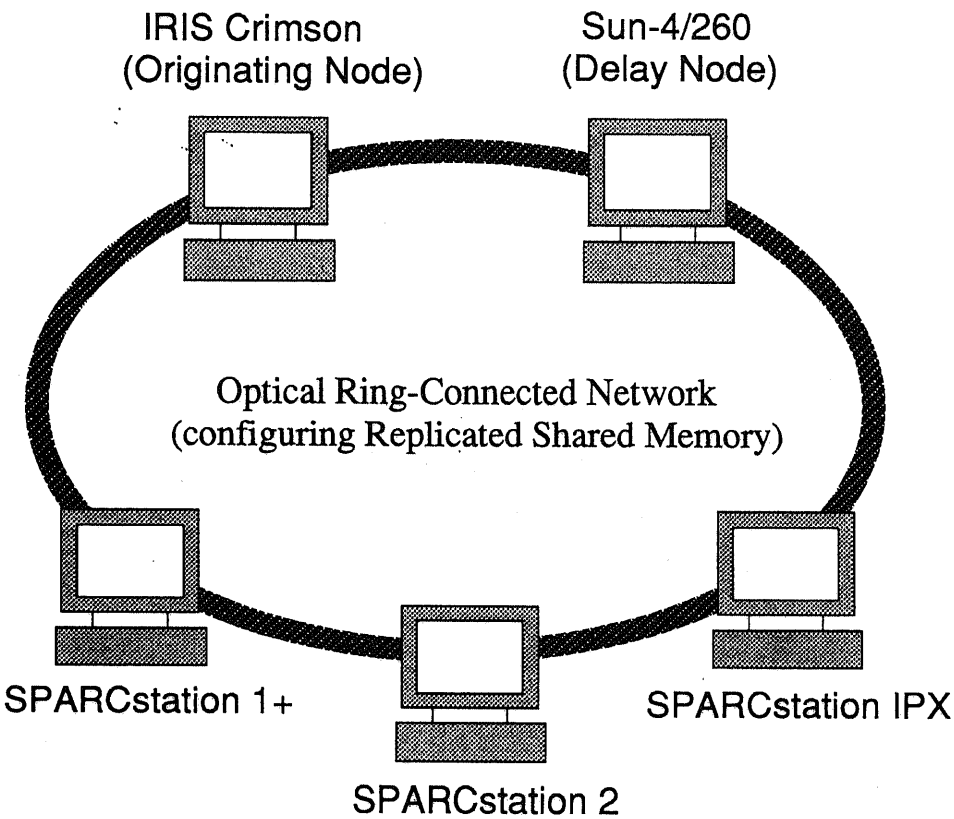


Figure 6.2: Environment of experimental system

We have constructed an experimental system simulating distributed computing in a widely distributed environment, using five workstations (IRIS Crimson, SPARCstation IPX, SPARCstation 2, SPARCstation 1+, and Sun-4/260) and SCRAMNet™. This is shown in Figure 6.2. The IRIS Crimson is used as the originating node, while the Sun-4/260 is used as the delay node, so the number of execution nodes including

the originating node is four. SCRAMNet™ is operated in normal mode.

The data size for each job is 32bytes. Global accesses are performed with this amount of data in replicated shared memory, and this amount of data is transferred among nodes in shared virtual memory. That is to say, the amount of data sent when a pagefault occurs in shared virtual memory is only 32bytes. Instead, pagefaults will always occur at the start of execution at each node, which may not be actually realistic.

6.3.2 Discussion of the results

The execution times for each model versus network length L are shown in Figure 6.3. Here, the number of jobs sent out by the originating node is 30, and the number of nodes at which each job is executed is 1000. The processing at each node consists of 3000 steps.

In the case of shared virtual memory and replicated shared memory using internal machine memory, all executions are performed on internal machine memory. In the case of replicated shared memory using external memory, on the other hand, some of them are accesses to external global memory. The ratio of local and global accesses is 1 : 9, 1 : 1, and 9 : 1, as is indicated in the figure. The ratio of read and write accesses to global memory is also 1 : 9, 1 : 1, and 9 : 1.

Even as the network length varies, the execution time for replicated shared memory remains almost the same. By contrast, the execution time for shared virtual memory increases as the network length becomes long. According to this result, the execution time should be greatly influenced by pagefaults in shared virtual memory. This is because the on-demand data transfer time is enormous in a widely distributed environment.

In the case of replicated shared memory using external memory, the performance is greatly influenced by the ratio of local and global accesses, and the ratio of read and write accesses to global memory. The execution time for replicated shared memory

with many global accesses is longer than that of shared virtual memory when the network length is short. This arises from the difference of access times to external memory and machine memory. On the other hand, the execution time for replicated shared memory with a small number of global accesses indicates a good performance. Thus replicated shared memory using external memory is considered to be suitable for a widely distributed environment compared to shared virtual memory, although its performance is greatly influenced by the ratio of local and global accesses, and the ratio of read and write accesses to global memory.

Replicated shared memory using internal machine memory achieves the best performance in this evaluation. It even surpasses the best case of replicated shared memory using external memory. This result is significant because the real performance may be estimated a little lower in this evaluation model, as is mentioned in previous section. According to the result, replicated shared memory using internal machine memory is considered to be suitable for widest variety of cases.

In order to investigate influences of a network, we carried out another experiment with different parameters. The number of nodes each job is executed is changed to 3000 and the number of steps executed at each node is changed to 1000. In this case, influences of a network must be stronger compared with the previous case, since the number of communication among nodes becomes large. The execution times for each model versus network length L are shown in Figure 6.4.

In this figure, the execution time for shared virtual memory increases dramatically as the network length becomes long. And it begins to increase where the network length is about 100km, which is shorter than that of the previous case. By contrast, replicated shared memory changes very little compared with the previous case. Therefore, it can be said that replicated shared memory seems to be got little effect from communication among nodes, while shared virtual memory is influenced so much.

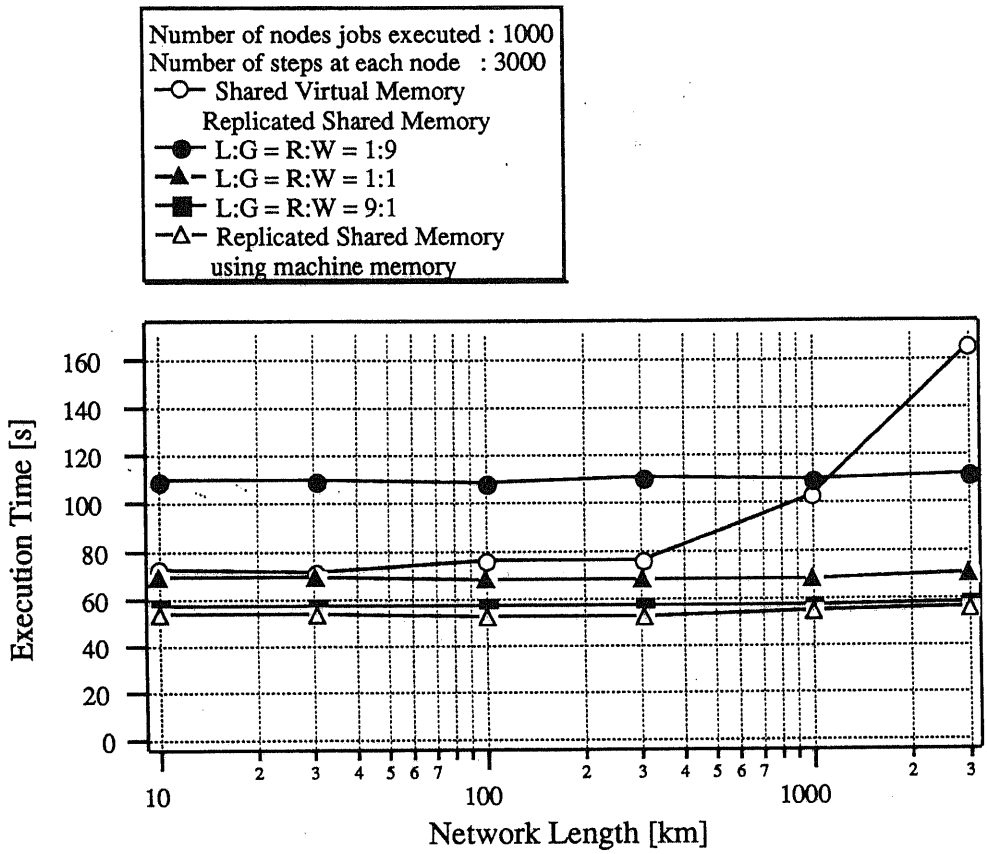


Figure 6.3: Execution time vs. Network length (Number of nodes each job is executed : 1000, Number of steps executed at each node : 3000)

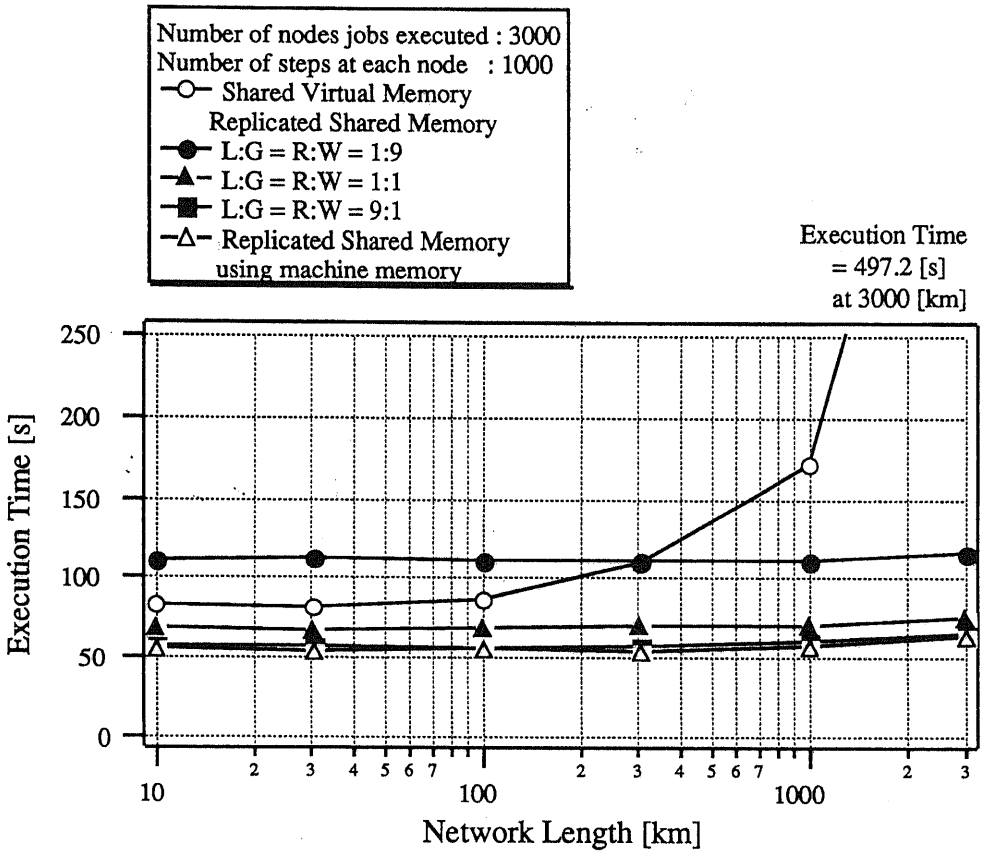


Figure 6.4: Execution time vs. Network length (Number of nodes each job is executed : 3000, Number of steps executed at each node : 1000)

6.4 Summary of the evaluation

Some distributed shared memory models in a widely distributed environment were discussed and evaluated in this chapter. First, an evaluation method for comparing models have been explained. SCRAMNet™, whose mechanism is based on replicated shared memory itself, was used for this comparison. Next, models of distributed computing, which include memory models and application models, have been addressed. How to simulate these models on SCRAMNet™ has been also shown. Results of this evaluations showed the superiority of replicated shared memory against shared virtual memory when the length of the network is large. While replicated shared memory using external memory is influenced by the ratio of local and global accesses, replicated shared memory using internal machine memory, which is proposed in this paper, is suitable for widest variety of cases.

Chapter 7

Conclusions

High throughput private networks will soon be realized in a widely distributed environment in which only public networks have been constructed thus far. Special functional computers distributed in a wide area, such as graphic workstations, supercomputers, database machines, and data I/O computers, are desirable to connect with each other in these networks so as to realize functionally distributed computing.

Distributed shared memory (DSM) is an attractive option for realizing functionally distributed computing in a widely distributed environment, because of its simplicity and flexibility in software programming. However, up till now, DSM has only been studied in a local environment. In a widely distributed environment, latency of communication greatly affects system performance, even if a large bandwidth network is used.

In this dissertation, some DSM models in a widely distributed environment have been discussed and evaluated. In Chapter 3, existing DSM models were analytically compared. They were shared virtual memory, replicated shared memory, and their hybrid model. Results from preliminary quantitative evaluations show the superiority of replicated shared memory against shared virtual memory when the length of the network is larger than 10^4-10^5 m, not only in the case that barrier synchronizations are used but also when semaphores are used. The hybrid model achieved the best performance whatever the network length is, so this model is suitable for the widest variety of cases.

Next, memory coherent problems of replicated shared memory have been discussed in Chapter 4. Some semaphore mechanisms suitable for replicated shared memory were proposed and evaluated. Conditions for replicated shared memory to keep release consistency was also clarified.

According to the results of Chapter 3, replicated shared memory seems to be suitable for a widely distributed environment. Replicated shared memory, however, has some problems. In order to overcome these problems, an innovated method to realize replicated shared memory in a wide area has been proposed in Chapter 5. In this method, internal machine memory was used as global memory area in stead of

external memory. Multi-thread programming executed on multiprocessor and ATM network is used to realize this mechanism.

In Chapter 6, distributed computing systems using DSM models were evaluated. SCRAMNet™, whose operation is based on replicated shared memory itself, has been used for this comparison. Results from this evaluation show the superiority of the replicated shared memory over shared virtual memory when the length of network is large. While replicated shared memory using external memory is influenced by the ratio of local and global accesses, replicated shared memory using internal machine memory is suitable for the widest variety of cases.

The replicated shared memory model is considered to be suitable particularly for applications which may require real-time operation in a widely distributed environment. This is because other latency hiding techniques such as context switching are not effective for real-time operation. On this point, replicated shared memory is expected to succeed in hiding such network latency and achieving good results.

Acknowledgements

This research was conducted during the author's Ph.D. program under the supervision of Associate Professor Hitoshi Aida, at the Graduate School of the Department of Electronic Engineering at The University of Tokyo.

The author would like to express his deepest gratitude to Professor Tadao Saito and Associate Professor Hitoshi Aida for their valuable guidance, helpful suggestions, and encouragement throughout this research.

The author also would like to acknowledge to Mr. Tadahiro Tomiyama and Mr. Shingo Chiba, the staff of the laboratory for their help and cooperation.

The author also would like to thank Mr. Ryuuichi Ohno and all other his fellow students for their cooperation and discussion during this program.

Last but not least, the author thanks all his family and friends for their warmest encouragement and support.

Bibliography

[1] "Gigabit Network Testbeds", IEEE Computer, pp.77-80, September 1990.

[2] D. D. Clark *et al*, "An Overview of the AURORA Gigabit Testbed", In Proc. of INFOCOM '92, 4D.1, pp.569-581, May 1992.

[3] H. T. Kung *et al*, "Network-Based Multicomputers: An Emerging Parallel Architecture", In Proc. of Supercomputing '91, pp.664-673, November 1991.

[4] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", In Proc. of the International Conference on Parallel Processing, pp.94-101, August 1988.

[5] M. C. Tam, J. M. Smith, and D. J. Farber, "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems", ACM Operating Systems Review, Vol.24, No.3, pp.40-67, July 1990.

[6] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", IEEE Computer, pp.52-60, August 1991.

[7] M. C. Tam and D. Farber, "CapNet - An Approach to Ultra High Speed Network", In Proc. of International Communication Conference '90, 323.1, pp.955-961, 1990.

[8] P. Steenkiste, "Analyzing Communication Latency using the Nectar Communication Processor", In Proc. of SIGCOMM '92, pp.199-209, August 1992.

[9] R. Bisiani and M. Ravishankar, "PLUS: A Distributed Shared-Memory System", In Proc. of the 17th International Symposium on Computer Architecture, pp.115-124, May 1990.

[10] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin", In Proc. of the 13th ACM Symposium on Operating Systems Principles, pp.152-164, October 1991.

[11] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Adaptive software cache management for distributed shared memory architectures", In Proc. of the 17th International Symposium on Computer Architecture, pp.125-134, May 1990.

[12] W. Weber and Anoop Gupta, "Analysis of cache invalidation patterns in multiprocessors", In Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Systems, pp.243-256, April 1989.

[13] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Relase Consistency for Software Distributed Shared Memory", In Proc. of the 19th International Symposium on Computer Architecture, pp.13-21, May 1992.

[14] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel, "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology", In Proc. of the 20th International Symposium on Computer Architecture, pp.144-155, May 1993.

[15] G. S. Delp, D. J. Farber, R. G. Minnich, J. M. Smith, and M. Tam, "Memory As A Network Abstraction", IEEE Network Magazine, pp.34-41, July 1991.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cashe Coherence Protocol for the Dash Multiprocessor", In Proc. of the 17th International Symposium on Computer Architecture, pp.148-159, May 1990.

[17] D. H. D. Warren and S. Haridi, "Data Diffusion Machine - A Scalable Shared Virtual Memory Multiprocessor", In Proc. of the International Conference on Fifth Generation Computer Systems 1988, pp.943- 952, 1988.

[18] "SCRAMNet™ Network REFERENCE MANUAL", SYSTRAN Corporation.

[19] L. A. Barroso and M. Dubois, "Cache Coherence on a Slotted Ring", In Proc. of International Conference on Parallel Processing, Vol.I, pp.230-237, August 1991.

[20] L. A. Barroso and M. Dubois, "The Performance of Cache-Coherent Ring-based Multiprocessors", In Proc. of the 20th International Symposium on Computer Architecture, pp.268-277, May 1993.

[21] K. Gharachorloo et. al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", In Proc. of 17th International Symposium on Computer Architecture, pp.15-26, May 1990.

[22] J. R. Goodman, "Cache Consistency and Sequential Consistency", Technical Report no.61, SCI Committee, March 1989.

[23] A. Landin, E. Hagersten and S. Haridi, "Race-free Interconnection Networks and Multiprocessor Consistency", In Proc. of 18th International Symposium on Computer Architecture, pp.106-115, May 1991.

[24] C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors", In Proc. of the 14th International Symposium on Computer Architecture, pp.234-243, June 1987.

Publication List

Transactions Papers

[25] T.Saito, H.Aida, M.Oguchi, and M.Kuroiwa, "Implementation and Evaluation of a Shared Memory Using Broadcast Memory with FIFO", Transactions of the Institute of Electronics, Information and Communication Engineers, D-I, Vol.J75-D-I, No.12, pp.1125-1131, December 1992 (Japanese).

[26] M.Oguchi, H.Aida, and T.Saito, "An Evaluation of Distributed Shared Memory in a Widely Distributed Environment", IEICE Transactions on Information and Systems (submitted).

International Conferences

[27] M.Oguchi, H.Aida, and T.Saito, "A Study of Synchronization Mechanisms for Ring-Connected type Replicated Shared Memory", In Proc. of the IFIP Conference on South East Asia Communications '94, pp.424-438, October 1994.

[28] M.Oguchi, H.Aida, and T.Saito, "A Study of Distributed Shared Memory in a Widely Distributed Environment", In Proc. of the IPSJ/IEEE Ninth International Conference on Information Networking, pp.267-272, December 1994.

[29] M.Oguchi, H.Aida, and T.Saito, "A Proposal for a DSM architecture suitable for a Widely Distributed Environment and its Evaluation", Fourth IEEE International Symposium on High Performance Distributed Computing (submitted).

Conventions

[30] M.Oguchi, H.Aida, T.Saito, and M.Okada, "A Study on Circuit Cost of Broadcast Network to Interconnect Distributed Terminals", In Proc. of the 1991 IEICE Spring Conference, B-633, March 1991 (Japanese).

[31] M.Oguchi, M.Kuroiwa, H.Aida, and T.Saito, "An Experimental System of Broadcast Memory with FIFO", In Proc. of the 1991 IEICE Autumn Conference, D-64, September 1991 (Japanese).

[32] M.Oguchi, H.Aida, and T.Saito, "Connecting Different Systems using Broadcast Memory with FIFO", In Proc. of the 1992 IEICE Spring Conference, D-124, March 1992 (Japanese).

[33] M.Oguchi, H.Aida, and T.Saito, "Study of Synchronization Method for Replicated Shared Memory", In Proc. of the 45th Conference of IPSJ, 6-109, October 1992 (Japanese).

[34] M.Oguchi, H.Aida, and T.Saito, "Implementation of Synchronization Mechanism for Ring-Connected type Replicated Shared Memory", In Proc. of the 1993 IEICE Spring Conference, D-144, March 1993 (Japanese).

[35] M.Oguchi, H.Aida, and T.Saito, "An Evaluation of Distributed Synchronization Mechanism for Replicated Shared Memory", In Proc. of the 1993 IEICE Autumn Conference, D-75, September 1993 (Japanese).

[36] M.Oguchi, H.Aida, and T.Saito, "A Study of Shared Memory System in Widely Distributed Environment", In Proc. of the 1994 IEICE Spring Conference, D-136, March 1994 (Japanese).

[37] M.Oguchi, H.Aida, and T.Saito, "An Evaluation of Distributed Shared Memory in a Widely Distributed Environment using a Replicated Memory System", In Proc. of the 49th Conference of IPSJ, 1-279, September 1994 (Japanese).

[38] M.Oguchi, H.Aida, and T.Saito, "Replicated Shared Memory Architecture suitable for a Widely Distributed Environment", In Proc. of the 1995 IEICE Spring Conference, March 1995 (Japanese).

Workshop Papers

[39] M.Oguchi, H.Aida, and T.Saito, "Memory Coherency in Replicated Shared Memories", In Proc. of the IPSJ(Information Processing Society of Japan) Workshop on Distributed Processing Systems, 58-23, November 1992 (Japanese).

[40] M.Oguchi, H.Aida, and T.Saito, "A Study of Distributed Shared Memory in Widely Distributed Environment", In Proc. of the IPSJ Workshop on Distributed Processing Systems, 64-24, March 1994 (Japanese).

[41] M.Oguchi, H.Aida, and T.Saito, "Implementation and Evaluation of Synchronization Mechanism for Ring-Connected type Replicated Shared Memory", In Proc. of the IPSJ Workshop on Distributed Processing Systems, 66-24, July 1994 (Japanese).

[42] M.Oguchi, H.Aida, and T.Saito, "A DSM architecture suitable for a Widely Distributed Environment", Second IEEE International Workshop on Services in Distributed and Networked Environments (submitted).