A Comparative Study of Distributed Shared Memory System Design Issues*

Ajay Mohindra
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
ajay@watson.ibm.com

Umakishore Ramachandran
College of Computing
Georgia Tech
Atlanta, GA 30332-0280
rama@cc.gatech.edu

GIT-CC-94/35
August 1994

Abstract

In this research the various issues that arise in the design and implementation of distributed shared memory (DSM) systems are examined. This work has been motivated by two observations: distributed systems will continue to become popular, and will be increasingly used for solving large computational problems; and shared memory paradigm is attractive for programming large distributed systems because it offers a natural transition for a programmer from the world of uniprocessors. The goal of this work is to identify a set of system issues in applying the shared memory paradigm to a distributed system, and evaluate the effects of the ensuing design alternatives on the performance of DSM systems. The design alternatives have been evaluated in two steps. First, we undertake a detailed measurement-based study of a distributed shared memory implementation on the CLOUDS[1] distributed operating system towards understanding the system issues. Second, a simulation-based approach is used to evaluate the system issues. A new workload model that captures the salient features of parallel and distributed programs is developed and used to drive the simulator. The key results of the research are that the choice of the memory model and coherence protocol does not significantly influence the system performance for applications exhibiting high computation granularity and low state-sharing; weaker memory models become significant for large-scale DSM systems; the unit of coherence maintenance depends on a set of parameters including the overheads for servicing data requests as well as the speed of data transmission on the network; and the design of miscellaneous system services (such as synchronization and data servers) can play an important role in the performance of DSM systems.

1 Introduction

Technological advances in recent years have spurred a trend towards workstation-oriented computing environments. Each workstation has computing power comparable to the mini-mainframes of the past. Availability

*This work has been funded in part by NSF grants CCR-8619886 and MIP-9058430, and IBM Corporation grant S 919FU2W ND.

[1] CLOUDS is a distributed object-based operating system developed at Georgia Tech.

of powerful computers connected via local (wide) area network has sparked interest in the area of distributed computing systems. Current research is targeting its efforts in utilizing the available compute power in the network for solving large problems through cooperative computing.

To facilitate the programming of distributed systems, two basic paradigms exist: shared memory, and message-passing. These two paradigms have been used for interprocess communication and synchronization in multi-process computations. The duality between the two paradigms for structuring computations is well-known [LN79]. Nevertheless, shared memory has been an appealing paradigm from the point of programming ease even in distributed systems. It is no surprise that several researchers [LH89, RAK89, FP89, CBZ91] have proposed system architectures that provide the abstraction of shared memory in a physically non-shared (distributed) architecture. We refer to this abstraction as *Distributed Shared Memory* (DSM). Figure 1 shows the conceptual representation of a distributed shared memory system. In the system, a set of nodes (computers) are connected via an interconnection network, and do not physically share memory. The DSM mechanisms allow an application to access shared data not physically resident at that node. These mechanisms are usually provided as a software layer either integrated with or on top of the operating system.

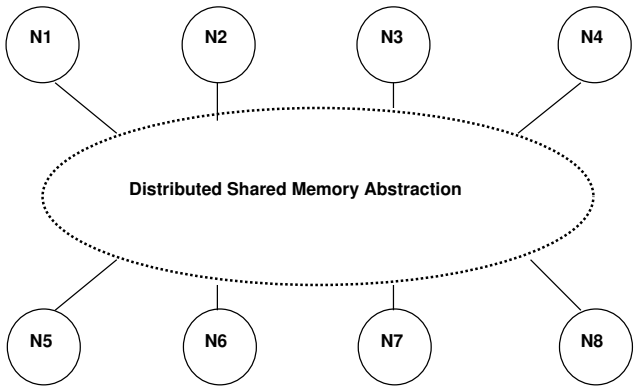


Figure 1: Distributed Shared Memory Abstraction

Another motivation for DSM arises from the structure of current distributed computing environments. A typical distributed computing environment consists of *compute servers*[2] and *data servers*[3] interconnected by a local area network. In such an environment, there are two tasks to be performed to execute a computation. The first task involves selecting a compute server, and the second task involves bringing the code and data from the data server to the selected compute server before executing the computation. The second task requires a remote paging facility. If sharing of data is coupled with this remote paging, it could be seen that DSM presents itself as a natural facility for combining the two.

In this work, we identify and evaluate the system issues (see Section 2) that need to be addressed for designing distributed shared memory systems. The issues relate to questions such as, whether to integrate DSM with virtual memory management, what type of memory model to provide, which protocol to use for maintaining coherence of shared data, and what kind of impact technology factors have on the DSM system performance. We evaluate these issues with respect to several proposed design alternatives. The evaluation is done in two steps. First, we take measurements of an existing implementation of DSM (see Section 4).

[2] Nodes where computation is performed.

[3] Nodes that serve as repositories for data.

Second, we use a simulation-based study to evaluate the interaction among the system parameters. In Section 5, we describe the design of a simulator that models a distributed shared memory system. The measurements are used to assign costs to the different system related activities of the simulator. A new workload model is developed and used to drive the simulator. The workload model captures the salient features of distributed and parallel programs. In Section 6, we discuss the results of the simulation study. The conclusions are presented in Section 7.

The key contribution of this research is that it enumerates the system issues and specifies the design parameters for addressing system issues for an efficient realization of a DSM system. The key results are:

- the choice of the memory model and coherence protocol does not significantly influence the system performance for applications exhibiting high computation granularity and low state-sharing;
- weaker memory models become significant for large-scale DSM systems;
- the unit of coherence maintenance depends on a set of parameters including the overheads for servicing data requests as well as the speed of data transmission on the network; and
- an efficient implementation of DSM requires a careful design of miscellaneous system services (such as synchronization and data servers).

2 Issues in the design of DSM systems

Several system issues need to be addressed in the design of a DSM system. The choice of solutions to these issues can significantly influence the overall system performance. In this section, we enumerate these issues and discuss the design alternatives available for addressing them.

2.1 Virtual Memory and DSM

In a DSM system, the remote memory accesses have to be reconciled with the virtual memory (VM) management at each node. The DSM and VM management at each node would have to cooperate to ensure that the semantics implemented by the DSM manager and the VM manager are not compromised. The normal VM chores such as page replacement, swapping, and flushing have to be done in consideration with the DSM algorithms. Similarly, in satisfying a remote memory request, the DSM would have to consult the VM manager to get a page frame, etc. Upon release of a page, the DSM has to instruct the VM manager to invalidate page table entries and take other related actions.

The effectiveness of the DSM paradigm depends crucially on how quickly a remote memory access request is serviced, and the computation is allowed to continue, which in turn depends on several factors:

- the speed at which the VM system detects that a memory access fault (i.e. a page-fault) or a pre-fetching request entails a remote access
- the software overhead involved in the DSM protocol (i.e. coherence mechanism) for servicing a remote memory access request
- the software overhead involved in the communication subsystem (i.e. the basic transport protocol) for effecting the inter-node message communication to service the request

- the speed of the communication medium (i.e. hardware).

2.2 Granularity

There are two dimensions to granularity: *computation granularity* and *data granularity*. The former deals with the amount of computation done between synchronization or communication points in a multi-process computation. The latter deals with the amount of shared information processed during this computation phase. While a computation to communication ratio of 100:1 may be reasonable in a tightly coupled shared memory system (such as the KSR-1) [Res91], this ratio is usually in the 1000:1 range for a DSM system.

In a uniprocessor memory hierarchy, the processor-to-cache transfer time is in the tens of nanoseconds, the cache-to-main memory transfer time is in the hundreds of nanoseconds, and the main memory-to-disk transfer time is in the order of milliseconds. Correspondingly, the granularity of transfer that makes sense are: byte or word between the processor and the cache, a block of several bytes between the main memory and the cache, and a page ranging from 512 bytes to a several kilobytes between the main memory and the disk. DSM systems add a new dimension to the memory hierarchy, namely remote memory access across the network. The choice of the network plays a big role in determining the latency. Nevertheless, independent of this choice, there is a fixed software overhead to be incurred depending on the choice of the data transfer protocol on the network. Moreover, such remote memory accesses need to be integrated somehow with the memory management at each node. This requirement often forces the granularity of access to be an integral multiple of the fundamental unit of memory management (usually a page). However, it is possible to reduce the network latency by transferring the page partially. The key point to note is that the data granularity has to be sufficiently high to make the DSM paradigm viable.

2.3 Memory Model and Choice of Protocol

In a uniprocessor, correctness of execution is ensured by preserving the order of memory references generated by a processor. Lamport [Lam79] has proposed *sequential consistency* (SC) as a memory model for ordering shared memory accesses to ensure correct multiprocessor execution. Essentially, sequential consistency ensures that the view of the memory is consistent at all times from all the processors. There have been several proposals for weakening the consistency requirements for shared accesses. Most of these proposals exploit explicit synchronization in the parallel program [GLL+90, LR91, KCZ92, AH93] to drive the consistency actions. *Release Consistency* (RC) [GLL+90] is the most well-known among these models, which distinguishes between two kinds of synchronization accesses, namely, *acquire* and *release*, establishing a consistent view of shared memory at the release point. A few others [AHJ91] use causality [Lam79], as a fundamental event ordering mechanism in distributed systems, to drive the consistency actions.

A related issue to the memory model supported by the DSM system is the choice of the coherence protocol used for consistency maintenance. Consistency maintenance of distributed shared memory is similar to cache coherence in shared memory multiprocessors with private caches. In shared memory multiprocessors with private caches, the caches are kept consistent using either a **write-invalidate** policy or a **write-update** policy [AB86]. In the former a writer acquires exclusive ownership of a cache block by invalidating all peer copies before performing the write, while in the latter concurrent writes to the same cache block are possible from several processors with the updates being sent to keep all the peer copies consistent. It is possible to use direct association of locks governing the access to shared cache blocks [LR90]. This would allow the data

associated with the lock being sent to the requester along with the granting of the lock. Upon release of a lock, the associated data is sent back (if modified) to the memory. We refer to such a protocol in which coherence maintenance and synchronization are intertwined as a **lock-based** policy.

Combination of the memory model with a particular coherence strategy gives rise to a unique memory system. In designing a page-oriented DSM system, however, not all combinations make feasible implementation sense. For example, it would be very expensive to track all writes to a page in such an implementation. Thus using a write-update policy for supporting the SC memory model may not make feasible sense. With a write-invalidate policy care has to be taken to avoid *false-sharing* wherein data that is not shared in a programmatic sense appears shared from the point of view of coherence maintenance. Similarly, if a write-invalidate policy is used in conjunction with the RC model, then it would limit the synchronization concurrency and negate any advantage that the RC model has over the SC model in weakening the memory consistency requirements. This is because of the false-sharing that is inherent in the write-invalidate policy exacerbated by the page-oriented nature of DSM. However, a simple minded write-update policy is infeasible to implement for the RC model both due to the difficulty in tracking writes as well as due to the enormous overhead in terms of message traffic that this might generate. Buffering the writes locally and propagating them to update peer copies prior to a release operation (as proposed in [LR91, CBZ91]) is a technique that will solve both of these problems in a DSM implementation.

In distributed systems, the number of messages is a measure of protocol performance. From this standpoint, the lock-based policy is expected to out-perform the other two, since coherence is maintained commensurate with the semantics of sharing in the computation. Moreover, since locking could be integrated with the data transfer, there is no need for any additional mechanisms for providing mutual exclusion for shared write accesses. In both write-update and write-invalidate policies there is a need to provide synchronization mechanisms on top of the coherence policy to assure mutual exclusion for multiple nodes requesting to write to the same page. However, lock-based policy has its drawbacks: In particular it does not have the generality of the other two policies. By decoupling memory coherence and synchronization, it is possible to devise synchronization mechanisms independent of the coherence policy. The lock-based policy requires explicit directives from the system software to know the semantics of sharing, while the other two do not require any such directives.

2.4 Synchronization

Another issue is the way interprocess synchronization is achieved in such systems. Extending the analogy of shared memory multiprocessors to DSM, it would seem that shared-memory style of synchronization would be expected in DSM systems as well. However, the granularity of accesses in DSM systems precludes using low-level primitives such as "Test-and-Set" on arbitrary memory locations. One possibility is to combine synchronization with sharing as has been suggested in some multiprocessor cache protocols [LR90]. Another possibility is to have an orthogonal set of primitives to achieve synchronization. This latter approach is attractive since there could be situations where there may be very little sharing of data but independent computation may have to synchronize with one another. For example, in compute-intensive applications, such as the embarrassingly parallel kernel [BBLS91] and matrix multiplication, interprocess synchronization is used only to indicate completion of computation. Some systems provide semaphore operations or lock operations in addition to the shared memory primitives.

2.5 Hardware Technology

There are two sources of overhead in a DSM system: the first is the communication overhead associated with the data transfer on the communication medium; and the second is the computational overhead associated with servicing remote memory requests. The choice of the communication medium (Ethernet, optical fiber, etc.) directly impacts the former, while the speed of the processor and any additional hardware support for DSM affects the latter.

In this section, we have enumerated the issues that need to be addressed efficiently in the design of distributed shared memory systems. These issues form the basis for the comparative study that is presented in the subsequent sections.

3 Related Work

Apollo Domain [LLD+83] is one of the earliest systems that implements a single level store (represented as a collection of shared objects) in a local area network of personal workstations and data servers. To assure the consistency of the replicated copies of an object a two-level approach is adopted. The lower level detects concurrency violations using a time-stamp based version number scheme for each object. The higher level provides an object locking mechanism.

Ivy [LH89] is a distributed shared memory system implemented on Apollo workstations interconnected by a token-ring network. It provides a shared virtual address space similar in concept to the Domain system with the difference that the granularity of access is a physical page in Ivy as opposed to an object in Domain. Ivy supports a SC memory model using a write-invalidate protocol. Mirage [FP89] is an extension of the Ivy memory system which allows a reader or a writer of a page to retain access to the page for a fixed duration of time regardless of pending requests. This is done to guarantee forward progress of the computation by reducing thrashing of heavily shared data pages.

CLOUDS [DLAR91] is a distributed operating based on passive objects. An object encapsulates data that can be manipulated only from within the object. To allow concurrent execution of more than one computation in the same object, shared-memory style synchronization primitives are provided by the operating system. The collection of objects in CLOUDS represents a distributed shared virtual space. Pages are the units of distribution. A lock-based protocol [RAK89] that unifies synchronization and data transfer is used for consistency maintenance of the distributed shared memory. In this protocol lock requests (exclusive and shared) result in the page associated with the lock being sent to the requester along with the granting of the lock. Upon release of a lock, the associated page is sent back (if modified) to the server. Reads or writes to shared data without explicit locking follow single-copy semantics that does not allow multiple-readers or writers. Thus this protocol provides a sequential consistent view of the shared memory at well-defined synchronization points. Ramachandran, et al. [RAK89] proposed this protocol which also supports a weaker form of read that allows multiple-readers to access shared data (without locking) without any guarantee of consistency.

The unique feature of the Munin [CBZ91] system is its ability to support multiple coherence protocols. The program variables are annotated with their expected access patterns, and the runtime system chooses the protocol best matched to the access pattern for each variable. The memory model supported by Munin is

	Domain	Ivy	CLOUDS	Mach	Agora	Memnet	Choices	Mether	Munin	DASH	KSR-1
Virtual Memory	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	Yes
Granularity	Page	Page	Page	Page	Data struct.	32-bytes	Page	Page	Data struct.	16-bytes	128-bytes
Memory Model	SC	SC	SC, UD	SC	UD	SC	SC	UD	RC, UD	RC	SC
Coherence Protocol	Version Number based	WI	LBS	WI	WU	WI	WI and LBX	WU	WI, WU, and LBX	WI, WU	WI
Synch.	Yes	No	Yes	No	No	No	Yes	No	No	Yes	Yes
Dedicated Hardware	No	No	No	No	No	Yes	No	No	No	Yes	Yes

Legend:

SC : Sequential Consistency
RC : Release Consistency
WI : Write Invalidate
LBX : Lock-based (exclusive)
UD : User-defined
WU : Write Update
LBS : Lock-based (exclusive and shared)

Table 1: Comparison of DSM systems

RC, and the write-update protocol provided by Munin buffers the writes to the shared pages and propagates them at a release point to the relevant peer processors.

In recent years, several systems have been proposed that implement the distributed shared memory abstraction in hardware. Two examples are the DASH multiprocessor [LLG+90], and the KSR-1 [Res91]. DASH uses a directory-based write-invalidate protocol to provide a release-consistent view of the shared memory which is physically distributed among the processing nodes. KSR-1, which uses a ring interconnect, provides a sequentially consistent view of the distributed shared memory with each node snooping on the network packets to take appropriate coherence actions.

Mach [RTY+87], Agora [BF88], Memnet [DSF88], Choices [SMC90] and Mether [MF90] are other DSM systems that have been proposed in the literature.

Though these systems have not been described here in complete detail, their features with respect to the issues outlined in Section 2 are summarized in Table 1. For more details, the reader is referred to [Moh93]. A similar survey can also be found in [NL91].

4 Distributed Shared Memory in CLOUDS: A Case Study

In this section, we summarize the measurements taken on the CLOUDS distributed shared memory. These measurements serve two purposes: First, it would help us better understand the interaction between various issues in a real environment; Second, they can be used for assigning costs to different components of the

simulator, which is used for the simulation study (see Section 5).

The unit of sharing in CLOUDS DSM is a segment. Associated with each segment is a node called the *owner* where the segment resides on stable storage. The DSMServer[4] at the owner node is responsible for maintaining the consistency of the segment. CLOUDS DSM uses a lock-based scheme to provide coherence of shared data. It supports two primitives for acquiring and releasing data: `get` and `discard`.

The request for a data segment is sent to the segment's DSMServer. If the requesting mode is compatible with the current mode for the segment, the DSMServer grants the request. Otherwise, the request is queued. Servicing a request may entail forwarding the request to the current keeper of the segment with instructions to immediately service the request. The DSMServers implement a First-Come-First-Served queue discipline for processing remote segment requests. The low-level communication protocol used in the CLOUDS operating system to support DSM is called `RaTP` [Wil89]. It provides reliable transfer of data between nodes.

4.1 Methodology

CLOUDS operating system is implemented on a configuration of Sun 3/60s connected by a 10Mbit/sec ETHERNET. The timing measurements are done using a microsecond timer [DM90]. Each call to read the timer has an overhead of 20 microseconds. The times reported in the next subsection are an average of number of such readings. A page refers to 8 Kbytes. We briefly report the performance results for three categories of experiments. More details about this study can be found in [AMMR92].

4.2 Performance Measurements

Table 2 summarizes the basic system times for the three categories of experiments. The network communication times shown in Table 2 are between two compute servers. A page transfer takes 12.3 milliseconds at the RaTP level as it breaks up an 8 Kbytes message into 6 packets. Note that Ethernet allows a maximum packet size of 1532 bytes [SDRC82].

The second category of experiments exercises the DSM subsystem. A `get` from a data server takes 15.5 milliseconds. Comparing the DSM and RaTP timings for a page transfer (rows 3 and 4), it can be seen that the DSM protocol has an overhead of 3.2 milliseconds. This overhead includes updating state information for the shared segment and coherence maintenance.

The third category of experiments deals with the servicing of page-faults. In the case of remote page-faults, there is no disk access involved (i.e. page is in memory at the remote server). The segment is currently with the data server that owns it. The DSMServer on a compute server requests a page from that segment while servicing a page-fault. The average time for servicing such a page-fault is 16.3 milliseconds. It should be noted that the VM overhead of installing a page, once a DSM `get` request completes, is only 0.800 milliseconds (difference between rows 4 and 5).

4.3 Analysis

Based on these measurements, Figure 2 shows the breakdown of the total time spent in each subsystem associated with servicing a DSM page-fault on CLOUDS. The total page-fault servicing time can be expressed as a sum of two types of costs: fixed cost and variable cost. The fixed cost consists of the overhead associated

[4] Process/thread that handles DSM related requests.

	Basic System performance	Time (in milliseconds)
1	Ethernet wire overhead (64 bytes, computed)	0.051
2	Ethernet wire overhead (8 Kbytes + headers, computed)	6.794
3	Transfer time at RaTP level (64-byte request one-way, 8 Kbytes other-way)	12.300
4	DSM get from a data server (no forwarding)	15.500
5	Page Fault Service	16.300

Table 2: Summary of basic system timings on CLOUDS

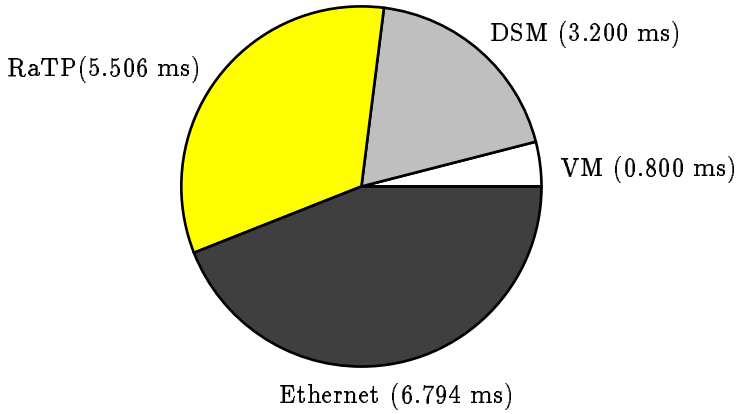


Figure 2: Cost associated with each subsystem in servicing a DSM page-fault. Total = 16.3 ms

with the VM subsystem and the cost of sending a data request to the data server while the variable cost consists of the cost of sending the data to the requester. The variable cost controls the latency of data as seen by an application process because the application process cannot start processing the data until the entire data page has been transferred. Ideally, in a DSM system, one would like to keep the *fixed cost per byte* (see equation 1) and *latency per byte* (see equation 2) low.

$$fixed\ cost\ per\ byte \quad = \quad \frac{VM\ overhead\ + data\ request\ cost}{PageSize} \tag{1}$$

$$latency\ per\ byte \quad = \quad (server\ proc.\ cost) * PageSize + \frac{PageSize}{Media\ bandwidth} \tag{2}$$

$$Total\ overhead\ per\ byte \quad = \quad fixed\ cost\ per\ byte + latency\ per\ byte \tag{3}$$

Equation 1 implies that systems that incur high VM overhead and high cost for sending a request can minimize *fixed cost per byte* by increasing the page-size. However, equation 2 dictates that the page-size should be kept small for keeping the *latency per byte* low. Ideally, one would like minimize the *total overhead per byte* as given in equation 3. We use these equations in Section 6 for deriving values for the page-size parameter for different system configurations.

Another point to note from Figure 2 is that the majority of the total time is spent in the communication subsystem (communication protocol and data transmission). This observation indicates that for an efficient

implementation of distributed shared memory, the cost of data transfer has to be reduced. Some techniques to bring this cost down is through using an improved communication protocol that has a relatively low overhead; using a faster communication medium to cut down the time spent on raw data transfer; data compression techniques for faster data transfer; and using additional hardware to improve processing overhead associated with the DSMServer.

These measurements are the first step in our evaluation process for understanding system issues in DSM design. The breakdown of costs associated with various components of distributed shared memory system are later used in the simulator that has been designed to evaluate the DSM system issues (Section 5).

5 Simulation Studies

Experimental study of different DSM implementations in the context of specific applications is one possible approach to evaluating the performance implications of some of the design issues. We did one such study for six different applications [AJM+93]. Unfortunately, it is difficult to glean insight regarding the impact of application characteristics and system parameters on the DSM design from such an experimental study. For instance, we would like to ask the question "What is the impact on performance when the degree of sharing in the application is changed?", or "What is the impact when the speed of the communication medium is changed?", etc. For this purpose, we use a simulation-based approach rather than an experimental approach because the latter approach places constraints on the study by limiting the choice of alternatives that can be studied. A simulation-based approach offers the flexibility to easily model different system configurations by tweaking the parameters of the simulator. The validity of the simulation results depends on the accuracy of costs assigned to different activities in the DSM system, and the accuracy of the workload used to drive the simulator. To this end we do two things: First, the costs are assigned to the different components of the simulator directly from the measurements of the CLOUDS implementation. Second, we validate the workload model, used to drive the simulator, against our experimental results described in [AJM+93].

5.1 Simulator

The simulator is written in CSIM [Sch86], a process oriented simulation language. The distributed system modeled by the simulator consists of a collection of nodes interconnected by a local area network. In the simulator, each node is modeled as a set of three CSIM processes: a compute engine, a DSM server, and a media server. The interconnection network is modeled as a CSIM facility. The compute engine models a processor with associated local memory. Shared references which are not currently encached in the local memory are communicated to the DSM server by the compute engine. The DSM server simulates the appropriate coherence protocol. The media server models the communication subsystem of a node. It differentiates between two types of messages: CONTROL and DATA. Each control message is 64 bytes long while the size of the data message is determined by the *page-size* parameter used in the simulation. The media server models the bandwidth characteristics of Ethernet and an optical fiber. It models the contention aspects of using a shared broadcast medium without modeling the collision and back-off aspects that are inherent in an Ethernet type of protocol. In addition to these three per node CSIM processes, a CSIM process serves as a centralized lock server. Figure 3 shows the conceptual picture of the simulator.

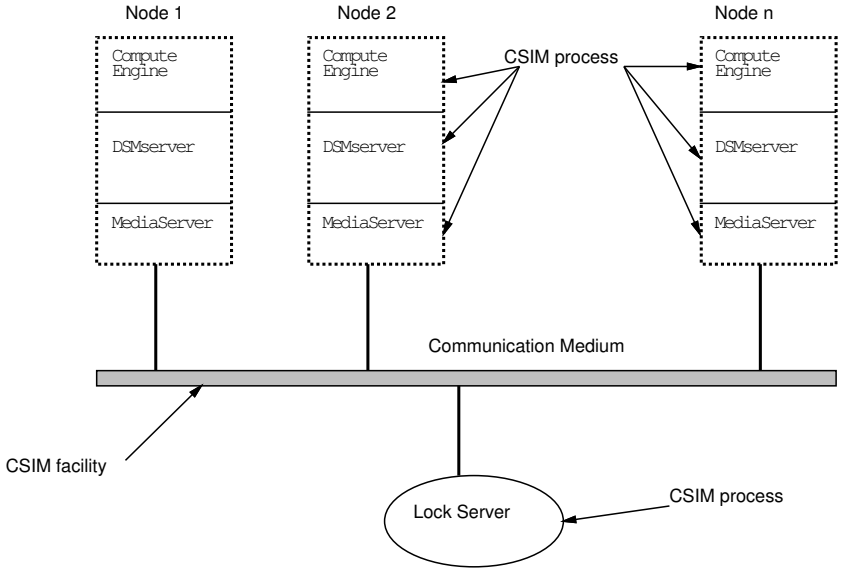


Figure 3: Conceptual picture of the simulator

5.2 The Workload Model

The workload model is a crucial component of any simulation study. Trace-driven, execution-driven, and probabilistic are the three methods of generating the workload for the simulation. Each has its merits and demerits. In terms of resource requirements (both space and time), and flexibility of varying the modeled application characteristics, probabilistic workload has some definite advantages over the other two. The main drawback in using such models is that the workload may not represent any real application. In our study, we develop a novel probabilitic workload model that captures the salient interactions that occur in parallel and distributed programs. Further, we show that by choosing the parameters of the model appropriately it is possible to replicate the results of some real applications which we have implemented on the CLOUDS DSM platform.

Archibald and Baer [AB86] have proposed a simple memory reference generator based on a probabilistic approach to evaluate cache coherence schemes in a shared memory multiprocessor. In their model, each processor generates a memory reference stream. A memory reference (read or write) could either be to private or shared blocks; locality of references to shared blocks is modeled by increasing the probability for accesses to recently used shared blocks. The interaction between the memory reference streams of the different processors is simulated for different coherence protocols. A synthetic reference generator is used by Kessler and Livny [KL89] to evaluate distributed shared memory algorithms, in which the main difference from Archibald and Baer's model is that the memory reference stream of each processor is a sequence of shared and private phases. During a private phase the accesses are strictly to private memory, while both shared and private memory may be referenced during a shared phase. Each phase is characterized by length, placement, locality, read to write ratio, and type (private or shared).

Synchronization is an important aspect of any parallel program design, and the memory reference streams of processors executing a parallel program will consist of synchronization accesses and normal read/write

accesses. The workload model, described in the next section, captures such synchronization aspects of a program, a feature absent in other probabilistic workload models.

5.2.1 Structure of the Workload Model

The workload described in this section models a class of applications that belong to the single-program-multiple-data (SPMD) style of programming. In a SPMD program, individual processors execute the same piece of code, albeit on possibly disjoint sets of data items. Processors synchronize with each other using semaphores, locks (shared or exclusive), or barriers. Semaphores and locks are used for protecting pieces of shared data, while barriers are typically used to indicate the end of a computation phase, or the computation itself.

A parallel program in our workload model is represented as a collection of tasks. The inter-relationship between these tasks is captured by a task dependency graph, that suggests a partial execution order for the tasks that constitute the parallel program. A task is ready for execution when all tasks that precede it in the dependency graph have been completed. A work queue is maintained that contains the set of tasks that are ready for execution. Tasks are inserted into this queue honoring the dependencies in the task graph. A processor accesses the work queue to acquire a task to be executed next. When the work queue becomes empty and all the tasks have terminated, the parallel program is said to have completed.

Each task is a memory reference stream of finite length (specified by a parameter) and is composed of a sequence of *compute* and *synchronization* phases. During a compute phase, the processor generates references (reads or writes) to private and shared data. A compute phase is characterized by the following parameters: the number of memory references, read to write ratio, probability for shared and private data accesses, and the degree of locality within the phase. The compute phase is similar to the shared phase as defined by Kessler and Livny [KL89]. A synchronization phase consists of read/write data accesses (both private and shared), with a percentage of the shared data accesses being done under the control of explicit synchronization. Thus, a compute phase corresponds to a phase in a SPMD program in which computation is performed, while a synchronization phase corresponds to a phase in which shared data is manipulated under the control of some synchronization variable. Figure 4 shows the composition of the the two phases within a task (the associated parameters are given in parentheses).

The degree of locality within a phase defines the spatial locality for references within a page. In addition to this, the workload model allows designating distinct and disjoint regions of the shared address space to each task; and there is a parameter, called `InterTaskRefProb`, that governs the fraction of shared references of a task that are directed to other tasks' as opposed to its own region. This feature of the workload model captures the SPMD style of programming, wherein individual processors primarily operate on distinct portions of shared data, with occasional references to other portions of shared data. To capture effects of false-sharing, we provide the `FalseSharingRefProb` parameter. Another parameter, called `SynchReferenceProb`, controls the percentage of accesses to shared regions that are performed under the control of explicit synchronization (shared or exclusive). This parameter models the number of critical sections in the SPMD program.

5.2.2 Domain Specific Models

By assigning proper values to the parameters shown in Figure 4, we can generate domain specific workload models. In our experiments, we used three different types of workload models representing three kinds of parallel programs.

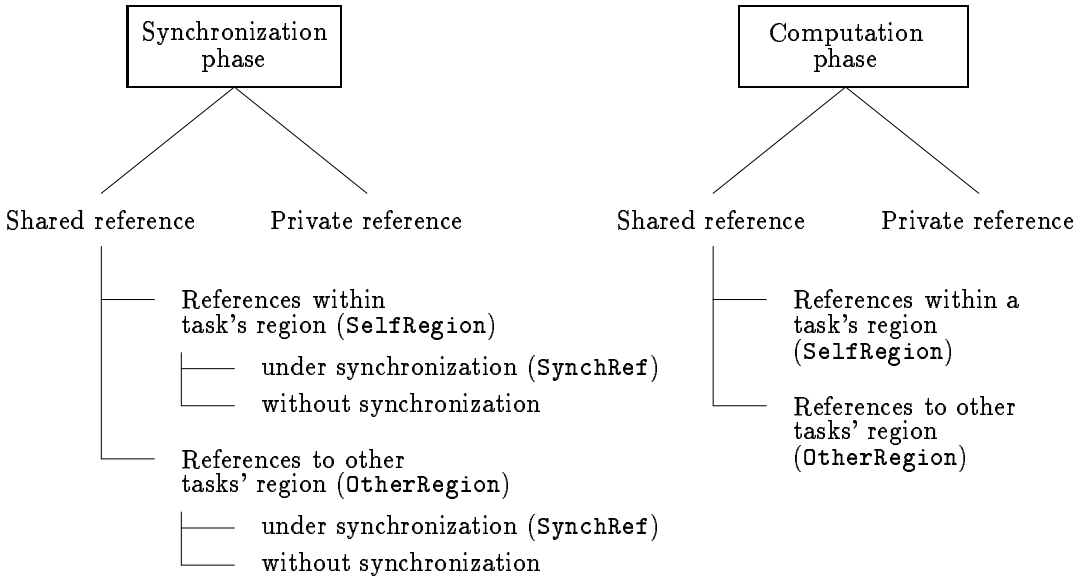


Figure 4: Reference parameters within a task

1. **Transaction model**: This workload model reflects a transaction processing system wherein all references to shared data are made under the control of either shared or exclusive locks. Thus, each task is composed of only synchronization phases, and all shared accesses are performed only under the control of explicit synchronization. (`SynchRef` = 1).

2. **Iterative Model**: Iterative algorithms such as linear equation solvers, have the characteristic that shared data is not modified except at well defined synchronization points (such as a barrier). Such a data access pattern would allow a task to access the shared data without acquiring any locks for the purposes of reading. The iterative model captures this characteristic by allowing some percentage of the shared references (only reads) to be directed to other tasks' regions. (In the computation phase, `OtherRegion` $\neq$ 0, 0.05 in our experiments).

3. **Asynchronous Model**: In this model, tasks that comprise a computation do not synchronize with one another explicitly. In terms of the workload model this feature would translate to tasks reading and writing to shared memory without explicit synchronization. However, an implementation of this model in a shared memory environment may involve the use of locks to govern access to mailboxes that may be used for asynchronous communication among the tasks. This workload model is similar to the data access patterns of asynchronous algorithms that rely on some other property such as convergence for correctness and termination [Bau78]. In terms of task parameters, some percentage of the shared references (both reads and writes) are directed to other tasks' regions. (In the computation phase, `OtherRegion` $\neq$ 0, 0.05 in our experiments).

Type of model	Parameter variable	Default values
Transaction Model	`PvtProb`	0.70
	`OtherRegion`	0.20
	`SynchRef`	1.00
	`InterTaskRefProb`	0.00
Iterative Model	`PvtProb`	0.70
	`OtherRegion`	0.20
	`SynchRef`	0.20
	`InterTaskRefProb`	0.05
	`ReadInterTaskRefProb`	1.00
Asynchronous Model	`PvtProb`	0.70
	`OtherRegion`	0.20
	`SynchRef`	0.20
	`InterTaskRefProb`	0.05
	`ReadInterTaskRefProb`	0.80

Table 3: List of parameters for domain specific workload models

Table 3 summarizes the default values for the parameters that define the three domain specific workload models. Table 4 shows the default values for the other parameters used in the simulator.

5.2.3 Validation of the Workload Model

We validate our workload model against the experimental results reported in a companion paper [AJM+93]. For completeness, we describe the two applications used in the validations here. The purpose of validation is to allow us to identify meaningful values for the parameters so that each workload (domain specific) corresponds to some "real" application. The performance of an application is measured from the experimental implementation. Using the program code as the basis, we determine the values for the key parameters of the workload model. We compare the simulation results obtained from the resulting workload model against the measurements on the real system. We show the results of this validation for Integer sort and SCAN.

Integer Sort

The integer sort kernel [BBLS91] is used in "particle-in-cell" applications. The problem statement for the integer sort benchmark requires that $\mathcal{N}$ keys be sorted in parallel. The keys are generated by a prescribed sequential key generation algorithm, and are stored contiguously in shared memory. The benchmark requires computing the rank for each key in the input sequence.

The application uses the bucket sort algorithm, partitioning the input key sequence among the available number of processors. Each processor maintains a local copy of the buckets and accumulates the bucket counts for its data partition without any communication with the other processors. The local buckets are merged into global buckets using a parallel-prefix sum algorithm. The final assignment of ranks is also done in parallel for the partition assigned to each processor. The implementation has been adapted from [RSRM93], has been shown to perform well on KSR-1, a tightly coupled shared memory machine. The program sorts $\mathcal{N} = 2^{20}$ keys into 2048 buckets.

Parameter variable	Parameter meaning	Default values
PvtProbInSynch	Probability of access to private data during a synch. phase	0.70
PvtProbInCompute	Probability of access to private data during a compute phase	0.70
SynchReferenceProb	Probability that the next phase is a synch. phase	0.20
AvgTaskLength	Average task length (number of references)	100,000
MaxLockRefRange	Granularity of synch. phase (number of references)	3000
MaxShdRefRange	Granularity of the compute phase (number of references)	3000
ReadLockProb	Probability of acquiring a read lock for a synch. phase	0.80
WriteLockProb	Probability of acquiring a write lock for a synch. phase	0.20
ReadProb	Probability that access to a memory location is a read	0.80
InASynchPhaseSynchRefProb	Probability that the shared reference is to locked data	0.50
LocalityProb	Probability that the next shared reference would be in the neighborhood of the current reference	0.80
LocalityDistribution	The function that specifies the probability distribution for shared data access	+/- 80 bytes
InterTaskRefProb	Probability that a task accesses shared data outside its domain	0.05
ReadInterTaskRefProb	Probability of a read access for inter-task references	0.80
FalseSharingRefProb	Models degree of false-sharing	0.00
SharedAddressSpace	Total size of the shared address space	1 Mbytes
NumberOfTasks	Total number of tasks in the parallel program	50
NumberOfNodes	Number of processors in the system	8
BlockSize	Amount of data transferred upon request for shared memory	8192
MediaSpeed	Speed of the network	10 Mbps

Table 4: List of parameters for the simulator along with the default values

SCAN

The SCAN benchmark [Ano85] specifies a sequential scan of a file, reading and updating records. Such scans are typical of end-of-day processing in on-line transaction processing systems. The benchmark requires that each record be locked, read, modified, updated, and unlocked. In the parallel implementation of the algorithm, the data is partitioned among available processors, and each processor performs a sequential scan of its portion of the database.

Table 5 show the results for the two applications for a memory system using SC memory model and a write-invalidate protocol. The values within parentheses are reported at 90% confidence level. As can be seen from the table, the simulation results agree quite well with the real results. Comparing the results using the `t-test` indicates no difference between the results obtained via the two techniques (The confidence intervals contain zero). The parameters for the workload models yielded by the validation are used for the simulation results of Section 5.4.

5.3 Parameters for the Simulation

We have designed a set of experiments to study the effects of the various design alternatives presented earlier. The approach we take is as follows: we use a set of compute nodes (3 MIPS CPU) connected by 10 Mbps Ethernet as the baseline system. We then designed our experiments to evaluate the effects of each issue on

Integer Sort for 2^{18} elements				
#	Measured		Simulated	
Proc.	Time	Conf. Interval	Time	Conf. Interval
1	6.51	(6.44, 6.57)	6.56	(6.56, 6.57)
2	6.83	(6.76, 6.89)	6.85	(6.73, 6.98)
3	8.41	(8.01, 8.82)	8.06	(7.93, 8.20)
4	13.02	(12.35, 13.69)	13.23	(12.85, 13.61)

SCAN benchmark for 10000 records				
No. of	Measured		Simulated	
Proc.	Time	Conf. Interval	Time	Conf. Interval
1	23.91	(23.90, 23.92)	23.18	(23.08, 23.21)
2	12.20	(12.16, 12.25)	11.59	(11.58, 11.60)
3	8.45	(8.28, 8.62)	8.80	(8.80, 8.81)
4	6.43	(6.36, 6.50)	6.47	(6.47, 6.48)

Table 5: Comparison of results obtained via simulation with actual measurements for the two benchmarks

Issues	Alternatives Studied
Data granularity (page size)	512, 1024, 2048, 4096, or 8192 bytes
Memory systems	SCinv, SCsynch, RCupdate
Communication medium	10 Mbps (Ethernet-like), or 1 Gbps (Fiber-like)
Processor Speeds	3 MIPS, 25 MIPS
Number of nodes	4, 8, 16

Table 6: List of alternatives evaluated using simulation

the performance of the overall system as compared to the baseline system. The system issues are evaluated with respect to three memory systems: first is based on a variant of the release consistency memory model and uses a write-update protocol for cache coherence, and we refer to it as *RCupdate*; the second is based on sequential consistency and uses a write-invalidate protocol for cache coherence, and we refer to it as *SCinv*; and the third is special in that it restores sequential consistency for only the data governed by the associated locks at well-defined lock release points using a lock-based protocol (see Section2), and we refer to it as *SCsynch*. As can be seen each is a combination of a particular memory model together with a specific coherence protocol chosen to implement it. As we pointed out earlier (see Section 2), not all combinations of memory models and coherence protocols make sense from an implementation standpoint and hence the choice of these three memory systems for this study.

All three memory systems are page-oriented. For the sake of fairness in comparison, all three memory systems are assumed to have the same unit of granularity for concurrency, namely, a logical *segment*. In the simulation a segment is fixed to be 8 KBytes, and is also the unit of coherence maintenance for the SCsynch memory system which combines synchronization with data coherence. A physical *page* is the unit of data transfer on the network for all three memory systems; page-size (data granularity) is specified as an input parameter and a logical segment is made up of (8KBytes/page-size) number of pages. A page is also the unit of coherence maintenance for the SCinv and RCupdate memory systems. We fixed the size of a logical segment to be 8 KBytes to match the physical page size used in the Clouds DSM implementation, since the costs associated with various components of distributed shared memory system have been assigned in the simulator from this implementation. It is assumed that the program level locks generated by the workload model map exactly to the unit of concurrency in the system, i.e., a logical segment. This assumption essentially ensures that none of the three memory systems will experience limited concurrency due to lock granularity and data transfer granularity mismatch. Table 6 summarizes the system parameters that are varied for this simulation study.

The experiments have been conducted for the three workload models described in Section 5.2.2. An application is modeled as a 4-level deep task dependence graph, with 16 tasks at each level, yielding a total of 64 tasks. A task on level $i+1$ is not executed until all tasks at level i have been completed. Each task

generates 100,000 references. The lengths of the compute and the synchronization phases are specified as input parameters. The shared address space is 1 Mbytes divided into 128 logical segments of 8 Kbytes each.

In all our experiments, we fix the following parameters to be unchanged: 70% private data references, 80% reads, and 20% of shared references performed under explicit synchronization in the iterative and asynchronous workload models. We use completion time as the metric for comparison.

5.4 Simulation Results and Discussion

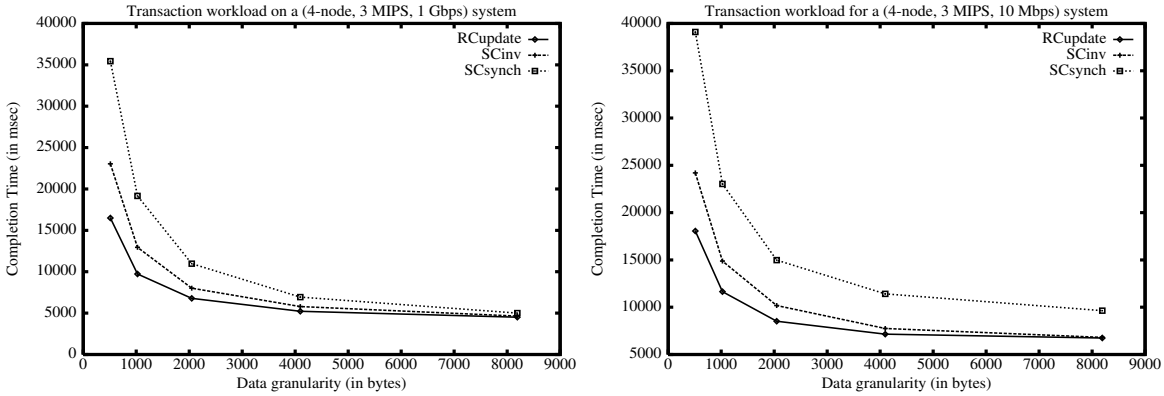
We discuss here only the results for the effects of granularity of data transfer, and the choice of the three memory systems with respect to the three workload models. The results for the impact of the hardware technology on performance can be found in [Moh93].

5.4.1 Transaction Model

One would expect that larger data granularity would reduce the number of messages in the system as fewer data requests are generated, and would increase spatial locality. However, larger data granularity also increases the potential for contention of shared data due to false sharing, thereby degrading system performance. Figures 5a, 6a and 7a show the performance for a 4-, 8-, and 16-node system connected via a 1Gbps communication medium. In the transaction workload model (see Figure 5a), we observe that the performance improves as the data granularity is increased for all three memory systems. False sharing is not an issue for this workload since all shared data references are performed under the control of a lock and since we assume lock granularity is a segment.

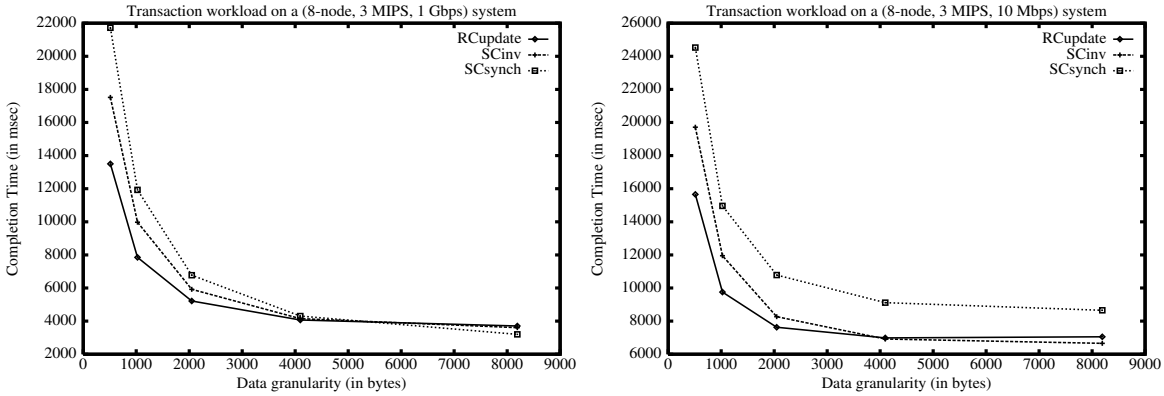
The SCsynch memory system is expected to incur a lesser number of messages on synchronized data accesses since it combines data transfer with synchronization. However, in this memory system pages associated with the lock are always shipped to the requester along with the granting of the lock irrespective of whether the requester has a valid copy or not. As can be seen in Figure 5a, SCsynch performs poorly at low data granularity compared to the other two. The reason is because at low data granularity more number of messages are required to bring in the entire segment associated with the lock. The SCinv and the RCupdate memory systems may not have to incur this message overhead if the data is valid at the requester. However at higher data granularity the SCsynch memory system performs better since the number of messages per lock request reduces significantly. Overall the RCupdate memory system performs better than either the SCinv or SCsynch memory systems (see Figure 5a), although at large data granularity, the difference between the SCinv and the RCupdate memory systems is statistically insignificant. In the RCupdate memory system, only the updates are sent to the server at synchronization points, and further it does not incur the overhead of invalidation messages. It is interesting to note for larger systems (8 and 16 nodes) SCsynch performs much better than the other two memory systems for large data granularity (see Figures 6a and 7a). For the SCinv memory system, the probability of the data associated with a lock being valid decreases due to the increased concurrent activity over the same number of shared segments. Similarly, for the RCupdate memory system the updates are sent to the current set of potential readers and all of them may not actually use it in the future. On the other hand, the SCsynch memory system incurs exactly the minimum number of messages required to get the lock and data. As we increase the number of nodes in the system, the number of messages becomes an important factor (due to contention for the communication medium) in determining the system performance.

In contrast, if one considers a system with Ethernet as the communication medium then the results for the transaction workload model are completely different. Figures 5b, 6b, and 7b show the results for the transaction workload model on a 4-node, 8-node, and 16-node system connected via a 10Mbps Ethernet. Unlike the earlier results, the SCsynch memory system performs considerably worse than either the RCupdate or the SCinv memory systems. The cost associated with the data transmission upon lock grant becomes dominant with the slower Ethernet medium, resulting in poor performance for the SCsynch memory system.



(a) On fiber (b) On Ethernet

Figure 5: Transaction workload model's performance on 4 nodes

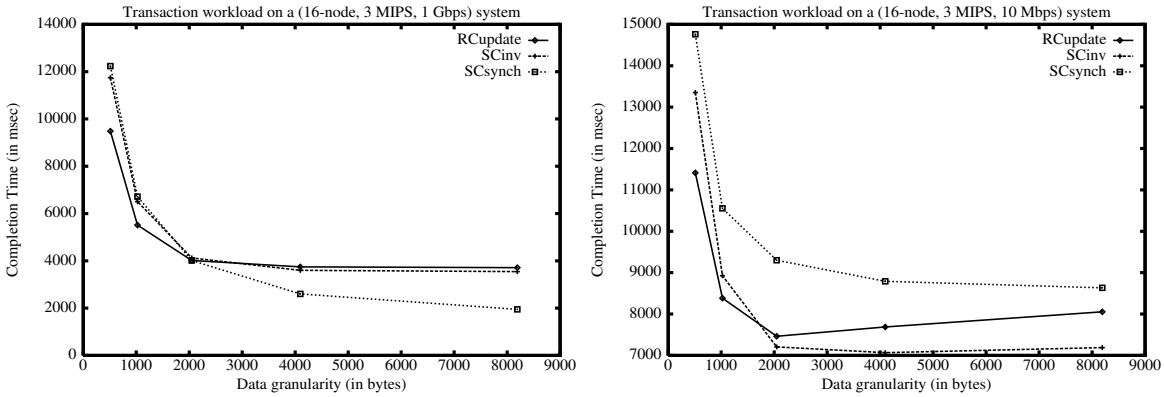


(a) On fiber (b) On Ethernet

Figure 6: Transaction workload model's performance on 8 nodes

5.4.2 Iterative Model

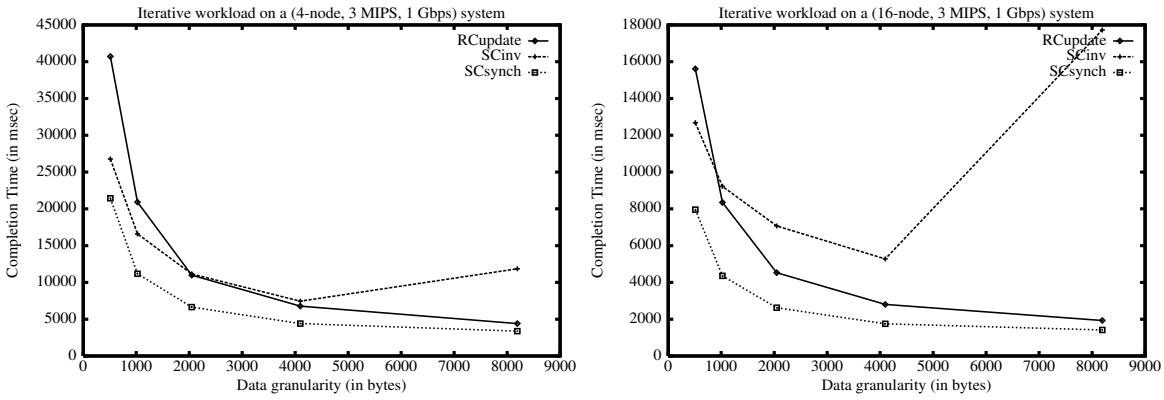
Recall that the iterative workload model (see Section 5.2.2) allows a task to access shared data for reading without explicitly acquiring read-locks. For this model, increasing data transfer granularity improves system



(a) On fiber (b) On Ethernet

Figure 7: Transaction workload model's performance on 16 nodes

performance for the SCsynch and the RCupdate memory systems (see Figure 8a). However, for the SCinv memory system, the performance benefit due to the reduced number of messages (at larger data granularity) is offset by an increase in false sharing, thus resulting in system performance degradation. Since read-shared copies are invalidated upon a write, the cost of re-reading a new valid copy increases with increasing data granularity for a given sharing pattern. The problem becomes even more acute when more nodes are added to the system, as now it is more likely that read-shared data pages may become invalid (see Figure 8b). Since false sharing is not an issue with either the SCsynch or the RCupdate memory systems, we do not see a similar performance degradation with either of these memory systems.



(a) 4 nodes on fiber (b) 16 nodes on fiber

Figure 8: Iterative workload model's performance

Since both the SCsynch and the RCupdate memory systems allow the copies of shared data to remain inconsistent between synchronization points, these two are expected to perform better than SCinv for the

iterative workload model. Figures 8a and 8b confirm this hypothesis. However it is surprising that the RCupdate memory system does not do as well as SCsynch. In the RCupdate memory system, updates for all modified pages are sent at the end of each synchronization epoch. This set of pages could potentially include ones that are unrelated to this particular epoch. As a result this memory system could incur more overhead than entirely called for in the iterative workload model. The SCsynch memory system by associating locks with segments does not have to incur this unnecessary overhead. This effect is more apparent at low data granularities (small page sizes). In fact, as can be seen even SCinv performs better than the RCupdate memory system at sufficiently small data granularity since the need for unnecessary updates in the latter over-shadows the ill-effect of false-sharing in the former. At higher data granularities, the distinction between SCsynch and RCupdate is less.

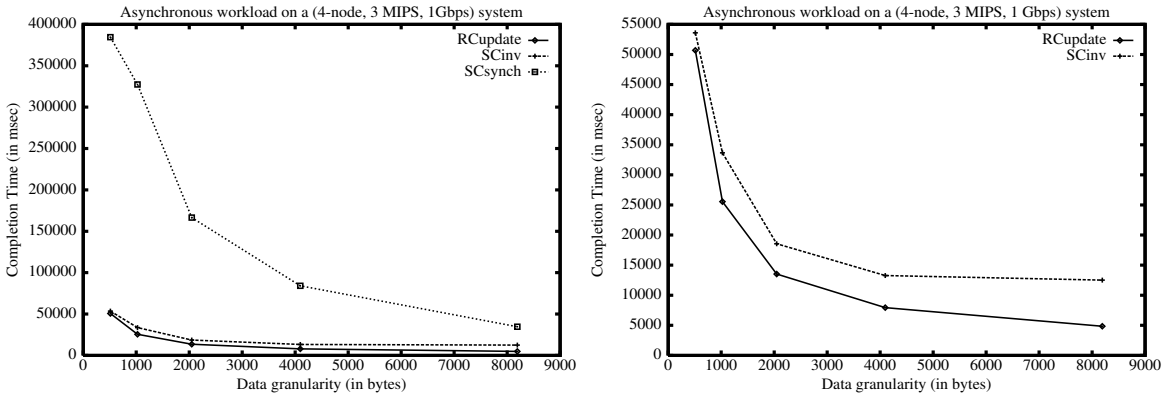
The results for the iterative workload model do not change if the communication medium is replaced by a 10Mbps Ethernet because only 20% of the data accesses are made under the control of a lock. Hence, the performance degradation as a result of shipping data with the lock is not very significant for the SCsynch memory system.

5.4.3 Asynchronous Model

In this model, unsynchronized write-sharing of data is allowed. Further the domain of write-shared data is the entire shared data space. Thus the model itself has a high built-in overhead (as compared to the iterative model) for both the SCinv and the RCupdate memory systems. In the former, invalidations may have to be sent to all the nodes while in the latter updates may have to be sent to all the nodes. This is evident by comparing absolute completion times for the same amount of total work (in terms of number of memory references) for the two workload models (see Figures 8a and 9b). As can be seen from Figure 9b increasing the data granularity helps both the memory systems. The positive effect of reducing the number of messages at larger data granularities seems to dominate the negative effect of false-sharing for the SCinv memory system. The SCsynch memory system (owing to its assumption that computations obey a synchronization model) is basically incompatible with this asynchronous workload model. Owing to the SCsynch memory system allowing exactly one-copy of a segment (regardless of the data granularity) for such asynchronous accesses it performs consistently worse than the other two for all data granularities (see Figure 9a). However due to lesser number of messages at larger granularities the performance of the SCsynch memory system approaches that of the other two.

The results for the asynchronous workload model do not change if the communication medium is replaced by a 10Mbps Ethernet because only 20% of the data accesses are made under the control of a lock. Hence, the performance degradation due to shipping of data with the lock is not very significant for the SCsynch memory system.

We conducted several experiments to determine the effects of new technology on the overall performance. We briefly summarize the findings here. When the communication medium in the baseline system is replaced with a faster medium, the impact of communication speed on the performance becomes more significant as the data granularity is increased because at low data granularity the access to the medium is the primary source bottleneck (due to large number of messages). In contrast, when the processor in the baseline system is replaced with a faster processor, the impact of the processor speed on the performance is more significant at low data granularity. This is because for low data granularity, more number of data requests are generated,



(a) Comparing all three memory systems

(b) Comparing RCupdate and SCinv

Figure 9: Asynchronous workload model's performance on 4 nodes on fiber

thereby increasing the computation requirements associated with page-fault handling and DSM related state maintenance; as a result, the processor speed has a significant impact on the performance than for large data granularity.

6 Discussion

We started out this research with the goal of evaluating the system issues in the design of distributed shared memory systems. We first identified a set of system issues along with the possible design alternatives available for addressing them. We re-visit these issues in the light of the simulation results presented in the previous section and present our observations.

6.1 Virtual Memory and DSM

There are two ways in which the distributed shared memory abstraction can be provided in a system: One, integrate the distributed shared memory mechanisms with the operating system; Second, provide the abstraction as a set of library functions accessible from the user-level. We call the first approach as the integrated-approach to DSM, and the second approach as the library-approach to DSM. The implementation of DSM considered in our study uses the integrated-approach. The advantage of using this approach is that the overheads associated with servicing DSM page-faults are very low, as all DSM related processing is done inside the operating system. In CLOUDS, the integrated-approach incurs an overhead of approximately 800 μsec per page-fault. As a result, the overall performance of DSM is very good. On the downside, the integrated-approach is quite inflexible as any minor change to the distributed shared memory system requires modifications to the operating system. The library approach to DSM, on the other hand, is quite flexible to deal with, as only the library needs to be modified. However, it would perform quite poorly due to the overheads associated with context-switching, crossing user-to-kernel address boundaries, etc. As

DSM deals with physical pages as units of data, a system designer implementing the library-approach would also have to modify the operating system to provide hooks for manipulating data pages (such as installing and invalidating) from the user-level. Some operating systems, such as MACH, do provide such hooks (via *external pagers*), thereby simplifying the implementation of the library-approach. Table 7 summarizes the advantages and disadvantages of the two approaches.

	Approach	
	Integrated	Library
1	low overheads, $O(\mu sec)$	high overheads, $O(msec)$
2	inflexible	flexible
3	transparent to the user	provide hooks in the operating system for installing, invalidating pages

Table 7: Integrated vs Library: Comparison of the two approaches

6.2 Granularity

There are two aspects to the issue of granularity: computation granularity, and data granularity. As mentioned earlier, computation granularity is the amount of computation a process has to do between synchronization and communication points in a multi-process computation, while data granularity deals with the amount of shared information processed during a computation phase.

• *Effects of computation granularity:* In distributed systems connected via a local area network, network latencies are high. Therefore, any problem that has to be solved in a distributed environment (through cooperative computing) should have sufficiently high computation granularity to justify the added communication costs. The goal is to have a high *CGRatio* in equation 4.

$$CGRatio = \frac{Time\ spent\ in\ the\ computation}{Time\ spent\ in\ requesting\ data\ for\ the\ computation} \quad (4)$$

Figure 10 shows the plot for a curve with $CGRatio = 1$. In order to achieve good speedups, the *CGRatio* for an application should fall in the shaded region for a given DSM implementation ($CGRatio > 1$). The vertical lines on the chart indicate the minimum time that is spent in transferring a unit of data between two nodes in a particular DSM implementation. For example, in CLOUDS, at least 16 msec is spent in transferring data between two nodes. This is because during each transfer a minimum of 8-Kbytes is transferred. Values for other systems differ depending on the size of the unit of data transfer, speed of the communication medium, and other overheads associated with the transfer. To achieve good speedups on a particular implementation, the *CGRatio* for an application should fall in the shaded region to the right of the vertical line for that system. Table 8 classifies the systems surveyed in Section 3, based on the relative grain of computation needed to achieve good performance. A system designer can calculate the computation requirements for a DSM design by matching the minimum communication time for the system with those shown in the chart (see Figure 10).

• *Effects of data granularity*: The issue of data granularity can be related to the amount of data exchanged between nodes at the end of a computation phase because it is this data that will be processed in the next computation phase. On page-based systems, regardless of the amount of sharing, the amount of data

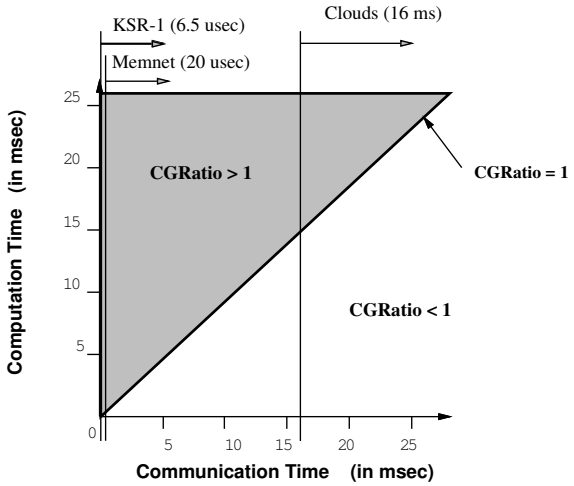


Figure 10: Computation to communication ratio requirements

Computation Granularity		
Large	Medium	Small
Domain, Ivy, CLOUDS, Mach, Agora, Choices, Mether, Munin	Memnet	DASH, KSR-1

Table 8: Computation granularity requirements

exchanged between nodes is usually a multiple of the physical page-size of the underlying architecture. Problems arise when applications that exhibit very small data granularity are run on systems that support very large physical pages (8 Kbytes). If the shared data is stored in contiguous memory locations then most data can be stored in few physical pages. This strategy often gives rise to the problem of false-sharing wherein disjoint pieces of shared data, operated upon by distinct processors, reside on the same physical page. As a result, the system performance degrades as the common physical page thrashes between different processors. The problem further exacerbates as more nodes are used for solving the problem. One way to reduce the problem of false-sharing is by partitioning the shared data structures on to disjoint physical pages. For systems with a large physical page-size, such partitioning of data can result in significant wastage of the virtual address space. Such wastage can be reduced if the distributed shared memory system is implemented on architectures which support a smaller physical page-size.

Another factor that affects the value for the page-size is the *total overhead per byte* associated with fetching a data-page. Recall, in Section 4 we computed the value for *total overhead per byte* as the sum of the *fixed cost per byte* and *latency per byte* (see equations 1, 2, and 3).

$$\begin{aligned} Total\ overhead\ per\ byte &= \frac{VM\ overhead\ + data\ request\ cost}{PageSize} \\ &+ (server\ proc.\ cost) * PageSize + \frac{PageSize}{Media\ bandwidth} \end{aligned}$$

Using the values for different components of the distributed shared memory system, one can compute the effect of increase in page-size on the *total overhead per byte* for a particular system. Figure 11 shows

the expected *overhead per byte* for the CLOUDS implementation of DSM using a 10 Mbps Ethernet. In the plot, we assume that the VM overhead is 0.800 msec, cost of sending a data request is 3 msec, and server processing cost is 0.200 msec/Kbyte of data. As can be seen from the figure, the minimum occurs somewhere between 1 - 2 Kbytes. For page-size values larger than 2 Kbytes, the *latency per byte* dominates the *total overhead per byte* while for values less than 1 Kbytes, the *fixed cost per byte* dominates the *total overhead per byte*. Table 9 lists values for the page-size parameter for different values of the VM overhead, server processing overhead, and data transmission cost. The values listed in Table 9 indicate the optimum value of page-size; and are obtained by differentiating the *total overhead per byte* with respect to the page-size parameter and solving for page-size (see equation 5).

$$PageSize = \sqrt{\frac{Media\ bandwidth(VM\ overhead + data\ request\ cost)}{(Media\ bandwidth) * (Server\ proc.\ cost) + 1024}} \quad (5)$$

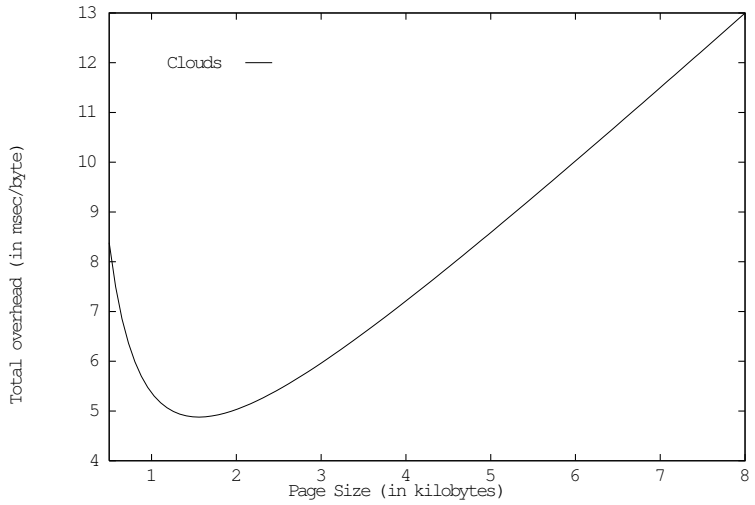


Figure 11: Total overhead per byte for DSM on CLOUDS

	VM ovhd	Data req cost	Server proc.	Media Speed	Page-size
1	0.80 msec	3.00 msec	0.20 msec/K	10 Mbps	1 - 2 Kbytes
2	10.0 msec	3.00 msec	0.20 msec/K	10 Mbps	3 - 4 Kbytes
3	0.80 msec	1.00 msec	0.20 msec/K	1 Gbps	2 - 3 Kbytes
4	10.0 msec	1.00 msec	0.20 msec/K	1 Gbps	7 - 8 Kbytes
5	0.005 msec	0.02 msec	0.05 msec/K	8 Gbps	0.7 Kbytes

Table 9: Optimal value of page-size for different system configurations

Table 9 indicates that a single value of the page-size parameter is not appropriate for all types of DSM system designs. The value should be decided based on the other design decisions, such as approach to DSM, expected server processing overheads, and cost of data transmission. For example, a page-size of 1 - 2

Kbytes is appropriate for a software implementation of DSM using the integrated-approach and Ethernet-like communication medium. Systems such as CLOUDS that have similar characteristics but are implemented on architectures with 8-Kbyte page-size pay a high penalty for *latency per byte*. On the other hand, systems that provide hardware support for DSM (indicated by small VM and server processing overheads), and faster communication medium can utilize smaller page-sizes (see entry 5 in Table 9). KSR-1 is an example of such a system that uses a 128-byte sub-page as the unit of data transfer and coherence maintenance. As the value for the page-size is usually tied to the system architecture used for the implementation, a system designer should carefully analyze his design decisions before selecting the architecture for implementation of DSM.

6.3 Memory Model and Coherence Protocol

The memory model presented to the programmer by the DSM system and the choice of protocol used for maintaining coherence of shared data are closely related. We have analyzed through simulation three memory system, namely, SCinv, SCsynch, and RCupdate. It should be understood that the discussion is with respect to these three memory systems each of which is a combination of a particular memory model together with a specific coherence protocol chosen to implement it.

Application	Rank
High *CGRatios*	(1) RCupdate, SCinv
Medium *CGRatios*	(1) RCupdate (2) SCinv
Small *CGRatios*	(1) SCinv (2) RCupdate
Transaction Workload	(1) SCsynch (2) RCupdate (3) SCinv
Iterative Workload	(1) SCsynch (2) RCupdate (3) SCinv
Asynchronous Workload	(1) RCupdate (2) SCinv (3) SCsynch

Table 10: Ranking of the three memory systems

Table 10 ranks the performance of the three memory systems for the three workload models that we studied. Interestingly, for applications that exhibited high *CGRatios*, the choice of memory system does not make a significant difference on the performance of the application. The main reason is that the application's communication and synchronization requirements are very low such that it does not matter which memory system is used. For medium-grained applications, the RCupdate memory system performs well because it supports concurrent writes to heavily shared data pages (which shuttle back and forth due to the write-invalidate protocol). The SCinv memory system performs poorly because it pays a high overhead for

maintaining consistency of heavily shared data pages. For small-grained applications, the RCupdate memory system performs poorly compared to SCinv because the former incurs high overheads at synchronization points. These overheads negate any gains of using a weaker memory model for the RCupdate memory system. For our simulation studies, we considered a wide range of workload models, and weaker memory systems perform well for configurations with large number of processors (SCsynch for iterative, RCupdate for asynchronous; see Section 5.4). The SCinv memory system did not perform well due to the increase in overhead for maintaining coherence of data in large configurations.

Even though the unit of coherence is different in SCsynch from SCinv and RCupdate, the results presented in this study will not significantly change for the transactional and the iterative workloads because none of these workloads do extensive asynchronous shared writes to a segment. For the asynchronous workload, however, the difference in unit of coherence gives a slight disadvantage to the SCsynch memory system by reducing the degree of concurrency for asynchronous shared writes to distinct pages of a segment. Nevertheless, even with the same unit of coherence for the three memory systems, the qualitative results for the asynchronous workload will not change since the source of the problem for the SCsynch memory system (for the asynchronous workload) is the one-copy semantics for unsynchronized access to a segment.

Other factors that can influence the choice of the memory system are the ease of programming, and system scalability.

Programming Ease

It is easier to design and reason about distributed programs developed using the SCinv memory system. However, since most parallel applications are developed with explicit synchronization the programming effort for the RCupdate memory system is expected to be no more difficult. The SCsync memory system requires explicit association of shared data with the lock variables that govern their access and hence is expected to require more programming effort than the other two though not significantly more.

Since in a distributed system an update-based protocol with the SC memory model does not make sense, an invalidation-based protocol is the only choice. In this case, to achieve good performance, the programmer (or the compiler) has to do a good job of data placement to avoid false-sharing. As mentioned earlier, performance degradation due to false-sharing magnifies in systems with large page-sizes since it results in limiting synchronization concurrency. Although, for RC memory model it is conceivable that either invalidation- or update-style protocol may be used, an update-based scheme is better suited in a distributed setting since it enhances synchronization concurrency in the presence of false sharing. As in the case of SC, if an invalidation-based protocol is used for RC then false-sharing will need to be addressed similarly.

Our simulation results (which are also corroborated in our experimental studies reported in [AJM+93]) show that there is no clear choice of a memory system that performs well across a variety of applications. A logical question then is whether a memory system that allows multiple coherence protocols to co-exist *a la* Munin [CBZ91] is the right approach to realizing efficient DSM systems. Though from the performance standpoint the answer is 'yes', such a system places a substantial burden on the programmer to specify the access patterns for the shared variables so that the correct protocol may be chosen by the system. It appears that from the point of view of minimizing the programming effort and application portability, a memory system that implements a well-defined memory model to the programmer is the right approach. Any optimization at the lower level (such as multiple protocols) should be transparent to the programmer and be done in the compiler, runtime, or the operating system.

System Scalability

By system scalability, we mean how many nodes can be added to a DSM implementation without incurring significant performance degradation. One measure of system scalability is the number of messages required for maintaining coherence of shared data. Table 11 shows the number of messages generated in the three memory systems (with and without multi-cast). If no multi-casting is used then one can see that both the RCupdate and SCinv memory systems can potentially generate a number of messages proportional to the number of nodes participating in the computation ($r \rightarrow \mathcal{N}$). On the other hand, the SCsynch memory system is insensitive to the number of nodes participating in the computation. However, in both the SCsynch and RCupdate memory systems the number of messages increases as the degree of sharing is increased (number of messages is a function of the degree of coherence, c).

Memory System	Number Of Messages	
	Without multi-cast	With multi-cast (r=1)
RCupdate	$\mathcal{S}$(5 + 2rw) + 2$\mathcal{P}$(1-h)	$\mathcal{S}$(5 + 2w) + 2$\mathcal{P}$(1-h)
SCinv	$\mathcal{S}$(5 + 2rwc + c(1-w)) + $\mathcal{P}$(1-h)(2+ c(w(5 + 2r) + 1))	$\mathcal{S}$(5 + 2wc + c(1-w)) + $\mathcal{P}$(1-h)(2+ c(7w + 1))
SCsynch	3$\mathcal{S}$ + $\mathcal{P}$(1-h)(2 + c)	3$\mathcal{S}$ + $\mathcal{P}$(1-h)(2 + c)

$\mathcal{S}$ Number of synchronization phases
M Amount of memory operated by a processor during a computation phase
w Probability that an access is a write operation
h Hit ratio
c Probability that an access read/write will cause coherence messages to be sent to other nodes
$\mathcal{N}$ Number of nodes participating in the computation
r Number of nodes involved in receiving coherence messages. $r \leq \mathcal{N}$
$\mathcal{G}$ Unit of data transfer
$\mathcal{P}$ Number of messages needed to bring in $\mathcal{M}$ bytes of memory. $\mathcal{P} = \frac{\mathcal{M}}{\mathcal{G}}$

Table 11: Number of messages generated in the three memory systems

Table 12 rates the scalability of the three memory systems based on different parameters values, assuming no multi-cast. We analyze each of the four cases below.

1. If an application does not require any coherence to be enforced ($c = 0$) then the SCsynch memory system will generate a fewer number of messages because it combines data transfer with synchronization. One example of such an application is an implementation of TSP that allows the nodes to use their local copies of the best tour-value. Only when a processor needs to update the global best tour-value, it does so under the control of a lock. This application does not need any coherence activity to be performed during computation of the best tour-value. The other two memory systems will generate equal number of messages, albeit more than SCsynch, because separate messages are needed for acquiring/releasing locks during the computation.

2. For applications that access data under the control of a synchronization, the SCsynch memory system generates fewer number of messages than the other two memory systems because it combines data

	Condition	Order
1	No coherence needed, $c = 0$	(1) SCsynch (2) SCinv, RCupdate[a]
2	No computation phase, $\mathcal{M} = 0$ $\Rightarrow \mathcal{P} = 0$	(1) SCsynch (2) RCupdate (3) SCinv
3	$r \rightarrow \mathcal{N}$	(1) SCsynch (2) RCupdate (3) SCinv
4	Number of synchronization phases tend to 0, $\mathcal{S} \rightarrow 0$	(1) RCupdate (2) SCsynch (3) SCinv

[a]Provided the reader turns off receipt of updates

Table 12: Scalability of the three memory systems without multi-cast

access with synchronization. The RCupdate memory system generates fewer messages than SCinv because the former supports concurrent writers to the same physical page while the latter does not. The SCAN benchmark is one example of such an application.

3. If the number of nodes for which memory consistency needs to be enforced reaches $\mathcal{N}$ then the number of messages generated for the SCinv memory system increases more rapidly than for the RCupdate memory system because the former enforces memory consistency during the synchronization and computation phase while the latter enforces memory consistency only at the end of the synchronization phases. The SCsynch system scales better than the other two because the number of messages is independent of the number of nodes participating in the computation.

4. If an application has very few synchronization phases then the benefits of the SCsynch memory system in combining data access and synchronization become negligible. As a result, the RCupdate memory system scales better than the other two because it does not generate messages to enforce memory coherence during computation phases.

A system designer can analyze the target set of applications that will run on the DSM system to see which type of applications will be more often used. The designer should then select the memory system accordingly by comparing the number of messages using Table 11.

6.4 Synchronization

We discuss the issue of providing synchronization with DSM under a broader category of *miscellaneous system services*. Simulation studies (see Section 5) have been performed under the assumption that miscellaneous system services (such as acquiring/releasing locks and barriers) incur negligible cost; therefore the results of the studies do not show significant effect of these services on the performance.

In a companion paper [AJM+93] we report on experimental studies of DSM systems that we carried out on CLOUDS in which we observed that these services play a key role in determining the overall performance of the application. Most applications that we considered in that study belong to the class of SPMD programs with approximately equal amounts of computation being performed at each node. As a result, the processors have a tendency to reach a synchronization point in the program at about the same time, causing bursts of synchronization activity. Such bursts of activity caused the (central) synchronization server to become overloaded, resulting in severe performance degradation especially for large number of processors. Similar performance degradation due to the data server becoming a bottleneck was observed for the RCupdate memory system. At a release point, each processor identifies the modified shared data and sends it to the data server. As all processors reach the synchronization point at approximately the same time, the data server became the bottleneck at synchronization points. The performance deteriorated further as more nodes are added to the system.

One approach to alleviating the synchronization overhead is to reduce the number of messages by combining data transfer with synchronization as is done in the SCsynch memory system. This point is supported by the simulation studies for the transaction workload model (see Section 5.4.1). Using distributed servers for providing miscellaneous services is another approach to reducing these overheads.

6.5 Summary

Emerging trends in processor and communication technology (faster RISC-based processors, fiber-optic and ATM networks) indicate that newer technology might alleviate the performance related problems of large-scale loosely coupled DSM systems. Faster processors would reduce DSM related processing overheads, while better communication technology will reduce data latency. To improve the scalability of the systems, it is imperative that the number of messages generated for coherence maintenance does not increase disproportionately as more nodes are added to the system. In this sense, both SCsynch and RCupdate memory systems have an edge over SCinv in that they require fewer messages for coherence maintenance. Further, RCupdate memory system does not exhibit the ill-effects of false-sharing (provided receipt of unnecessary updates is turned-off). Therefore, this memory system may be the best candidate for architectures with large physical page-sizes.

7 Concluding Remarks

The paper starts with the premise that distributed shared memory is a viable programming paradigm for programming large distributed systems. Based on this premise, we have investigated several issues that arise in the design of such systems, and tried to answer the question whether we can identify a set of issues, along with the design parameters, that define an efficient implementation of distributed shared memory systems.

First, we have identified a set of system issues that form the core of a distributed shared memory system design. These issues include integration of distributed shared memory with virtual memory management, granularity of computation and data, choice of memory model, choice of the coherence protocol, and technology factors. We have also identified a set of possible design alternatives that are available for addressing each of these issues.

Second, we have analyzed the performance of an implementation of distributed shared memory on the CLOUDS distributed operating system. The study has provided us with timing measurements associated

with individual components of the DSM subsystem, which are later used to assign costs to the different components of the simulator.

We studied the issues using a simulation model. To drive the simulator, we designed a workload model that captures the salient features of programming parallel and distributed systems. The simulator is used to analyze system performance with respect to data granularity, memory models and coherence protocols, effect of communications media, and any additional hardware support. Some of the key results of the study indicate that the choice of coherence protocol does not matter for applications that exhibit high computation granularity and low state sharing; weaker memory models and update style protocols become important in large distributed shared memory systems; the unit of data granularity (page-size) depends on the overhead associated with servicing data requests and cost of data transmission; and miscellaneous system services, such as the synchronization server, and the data server, play a significant role in influencing the performance of an application.

References

[AB86] J. Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273 – 298, Nov 1986.

[AH93] S. Adve and M. D. Hill. A unified formalization of four shared memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[AHJ91] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed memory. In *11th International Conference on Distributed Computing Systems*, pages 274–281, 1991.

[AJM+93] R. Ananthanarayanan, R. John, A Mohindra, M. Ahamad, and U. Ramachandran. An evaluation of state sharing techniques in distributed operating systems. Git-cc-93-73, Georgia Institute of Technology, 1993.

[AMMR92] R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, and Umakishore Ramachandran. Experiences in integrating distributed shared memory with virtual memory management. *Operating Systems Review*, 26(3):4–26, 1992.

[Ano85] Anonymous. A measure of transaction processing power. *Datamation*, pages 112–118, April 1985.

[Bau78] G. M. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, 25(2):226–244, April 1978.

[BBLS91] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, Moffet Field CA 94035, August 1991.

[BF88] Roberto Bisiani and Alessandro Forin. Multilingual parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *The 13th ACM Symposium on Operating Systems Principles*, Oct 1991.

[DLAR91] Partha Dasgupta, Richard LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The CLOUDS distributed operating system. *IEEE Computer*, April 1991.

[DM90] Peter B. Danzig and Stephen Melvin. High resolution timing with low resultion clocks and A microsecond resolution timer for Sun workstations. *Operating Systems Review*, 24(1):23–26, 1990.

[DSF88] Gary S. Delp, Adarshpal S. Sethi, and David J. Farber. An analysis of Memnet: An experiment in high-speed shared-memory local networking. In *Computer Communication Review*, volume 18, pages 165–174, Stanford, California, August 1988. ACM SIGCOMM.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. *Operating Systems Review*, 23(5):211–223, Dec 1989.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessey. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *The 17th Annual International Symposium on Computer Architecture*, May 1990.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KL89] R. E. Kessler and Miron Livny. An analysis of distributed shared memory algorithms. In *9th Intl Conference on Distributed Computing Systems*, pages 498–505, 1989.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transaction on Computers*, C-28(9):690–691, Sep 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transaction on Computer Systems*, 7(4):321–359, November 1989.

[LLD+83] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A Hamilton, David L Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–857, November 1983.

[LLG+90] D Lenoski, J. Laudon, K Gharachorloo, A Gupta, and J Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[LN79] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, April 1979.

[LR90] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *Proc. 17th Int'l. Symp. on Computer Architecture*, pages 27–37, May 1990.

[LR91] Joonwon Lee and Umakishore Ramachandran. Locks, directories, and weak coherence - a recipe for scalable shared memory multiprocessors. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[MF90] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *10th International Conference on Distributed Computing Systems*, pages 468–475, May 1990.

[Moh93] Ajay Mohindra. *Issues in the design of distributed shared memory systems.* PhD thesis, College of Computing, Georgia Tech, Atlanta, GA 30332-0280, June 1993.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, pages 52–60, August 1991.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *18th International Conference on Parallel Processing*, pages 160–169, Aug 1989.

[Res91] Kendall Square Research. *KSR1 Principles of Operations*, 1991.

[RSRM93] Umakishore Ramachandran, Gautam Shah, S. Ravikumar, and Jeyakumar Muthukumarasamy. Scalability study of the KSR-1. In *International Conference on Parallel Processing*, 1993. Also available as a technical report GIT-CC 93/03 from Georgia Institute of Technology.

[RTY+87] Richard Rashid, Avadis Tevanian, Micheal Young, David Golub, Robert Baron, David Black, Willian Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[Sch86] H. D. Schwetman. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, December 1986.

[SDRC82] John F. Shoch, Yogen K. Dalal, David D. Redell, and Ronald C. Crane. Evolution of the ETHERNET local computer network. *IEEE Computer*, pages 1–27, Aug 1982.

[SMC90] A. Sane, K. MacGregor, and Roy H. Campbell. Distributed virtual memory consistency protocols: Design and performance. In *Second IEEE Workshop on Experimental Distributed System s*, pages 91–96, Oct 1990.

[Wil89] Christopher J. Wilkenloh. RaTP: A transaction support protocol for *Ra*. Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, 1989.