Performance Enhancements for Operating System Implementations of Distributed Shared Memory

Virginia Lo

CIS-TR-93-26
December 1993

Department of Computer and Information Science
University of Oregon

Performance Enhancements for Operating System Implementations of Distributed Shared Memory

Virginia Lo

Dept. of Computer Science

University of Oregon

Eugene, OR 97403

Tel: (503) 346-4408

FAX: (503) 346-5373

Email: lo@cs.uoregon.edu

October 26, 1993

Abstract

A distributed shared memory (DSM) system provides an implementation of the shared memory abstraction on a message-passing architecture which has no physically shared memory. This paper describes recent developments in the design of DSM systems in which the shared memory model is supported primarily through software modifications to existing virtual memory facilities in the operating system. These DSM systems are characterized by a larger unit of sharing, typically at the page level, and have been designed for loosely-coupled workstation networks as well as message-passing multiprocessor architectures. Our focus is on performance enhancements that have been developed for these DSM systems to reduce the memory latency incurred for coherence, data location, and data access in a system with physically disjoint memories. We describe enhancements designed to reduce (1) latency stemming from the design of coherence algorithms, (2) latency attributed to communication overhead generated by the network software and hardware, and (3) latency due to unnecessary message-passing due to large page sizes and *false sharing*.

Keywords: distributed shared memory, virtual shared memory, memory latency, memory coherence, coherence protocols.

Contents

1 Introduction

Distributed Shared Memory (DSM) provides a virtual address space shared among processes on loosely coupled processors (see Figure 1). The advantages offered by DSM include ease of programming and portability achieved through use of the shared memory programming paradigm, the reduced cost of distributed memory machines, and scalability resulting from the absence of hardware bottlenecks. DSM has been an active area of research since the early 1980's, although the foundations for this work in cache coherence and memory management have been extensively studied for many years. There have been three main approaches to implementing DSM (although some systems use a hybrid approach): (1) operating system and library implementations in which sharing and coherence are achieved through virtual memory management mechanisms, (2) hardware implementations which extend traditional caching techniques to scalable architectures, and (3) compiler implementations where shared accesses are automatically converted into synchronization and coherence primitives.

In this paper we focus on distributed shared memory systems supported primarily through software modifications to existing virtual memory management facilities. These DSM systems are characterized by a larger unit of sharing, typically at the page level, and are designed for loosely-coupled workstation networks and, more recently, message-passing multiprocessor architectures. The earliest of these software systems proved the feasibility of the DSM approach to scalable parallel computation through implementation and preliminary performance experiments. Spurred by the initial successes in both hardware and software DSM systems, yet keenly aware of the potential bottlenecks, DSM researchers have investigated a wide range of innovative techniques to enhance performance.

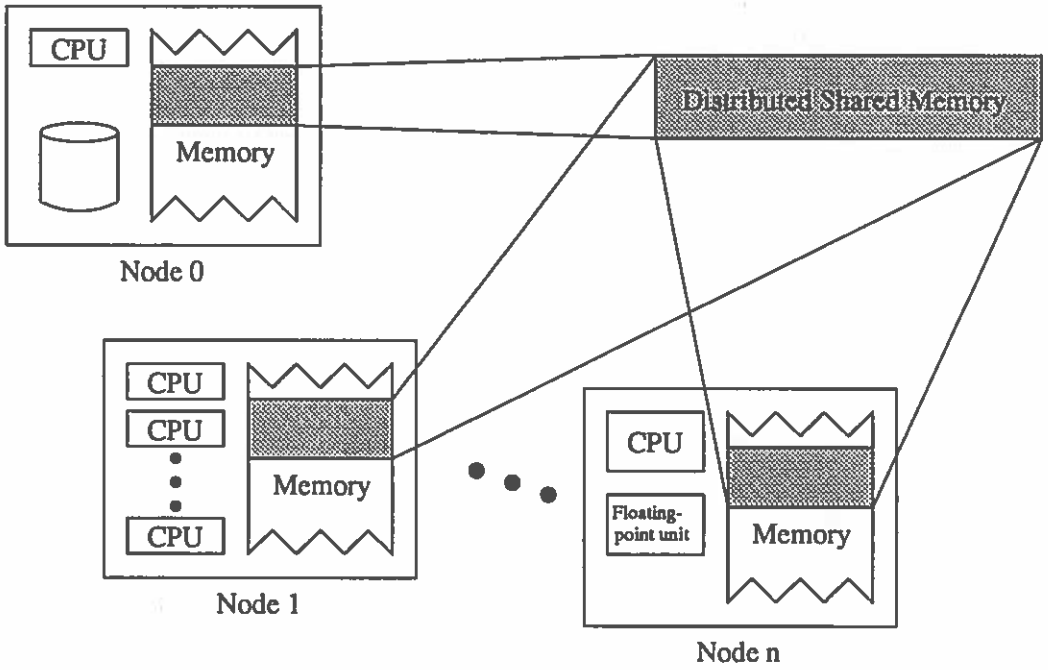


Figure 1: Distributed Shared Memory

These enhancements either directly or indirectly address the problem of **memory latency.** Memory latency is the delay incurred from the time a processor initiates access to shared data until the access is completed. Memory latency is inherent to DSM systems because of the potentially large overhead for data location and access in a system with physically disjoint memories. This overhead is more serious in software-based DSM systems since the overhead due to remote memory accesses may involve kernel calls, context switching, and buffering in addition to communication software and hardware latency.

The efforts of DSM researchers to reduce memory latency can be roughly divided into three categories:

- latency attributable to the choice of coherence semantics and the design of the coherence protocol for a given system. This overhead is reflected in the number and size of messages

needed for the protocol, as well as delays due to blocking, serialization, and bottlenecks.

- latency due to communication overhead generated by the network software and hardware.
- latency due to unnecessary message-passing because of large page sizes or because of dysfunctional behavior, particularly excessive message-passing and thrashing due to *false sharing.*

We describe performance enhancements that have been proposed by DSM researchers to address these sources of memory latency. The DSM systems and research projects we discuss include IVY, the first DSM system for workstation networks; Munin, a DSM system supporting type specific memory coherence and release consistency; Memo and related research on a relaxed coherence model known as lazy release consistency; Midway, a workstation-based DSM system supporting entry consistency; Mether and Clouds, two systems supporting user-defined coherence protocols for inconsistent memory; Galactica Net, a hardware-software DSM system that utilizes a hybrid approach to coherence in order to realize the best of a number of techniques; Mirage and DSVM6K, which use page locking to avoid thrashing; Mirage+, which uses data compression techniques to reduce communication overhead; Shiva and the memory server model for message-passing multicomputer systems; The proposed PLUS system and Tamir & Janakiraman's model which support two levels of granularity.

The performance results reported in this paper include both those based on simulation as well as those derived from empirical experiments with systems that have been fully implemented. Tables 1 and 2 summarize characteristics of the DSM systems discussed in this paper.

System Name	Current Implementation	Page Size	Coherence Semantics	Coherence Protocol	Performance Optimizations
Choices (1988-)	Sun SPARCstation 1 and 2, Ethernet (and other architectures)	8192 bytes	sequential	write-invalidate and lock-based (exclusive)	page locking with dynamic threshold, support for multiple protocols
Clouds (1987-)	Sun-3 workstations, Ethernet, Clouds OS + Ra Kernel	8192 bytes	sequential, user-defined	lock-based (exclusive and shared)	integration of synchronization with DSM, object-based approach
DSVM6K (1990-)	IBM RISC System/6000 workstations, optic fiber network, point to point interconnection, AIXv3	4096 bytes	sequential	write-invalidate	page locking, fast network protocols
Galactica Net (1990-)	Lynx multiprocessors, mesh/wormhole routing	8192 bytes	sequential, weak, release	hybrid update/invalidate	weaker consistency models, adaptive hybrid strategies, virtual sharing rings for update protocol
IVY (1984-86)	Apollo Domain workstations, Apollo token ring, Aegis OS	1024 bytes	sequential	write-invalidate	refinements to coherence algorithms: distributed managers, probable owner collapse, distributed copysets
Mach (1986-)	Hetrogeneous environment, Mach	multiple page sizes	sequential	write-invalidate	efficient distributed algorithm, page fault forwarding
Memo (TreadMarks) (1992-)	Sun SPARCstations, DECstation-5000/240, Ethernet, ATM LAN, Ultrix	4096 bytes	release	hybrid update/invalidate	eager and lazy release consistency, multiple concurrent writers protocol, efficient communication software

(continued on next page)

Table 1: Summary of Selected DSM Systems

The systems described in this paper are restricted to operating system implementations of DSM and thus do not include hardware DSM systems and language/compiler implementations of DSM. Also due to space and time limitations we do not discuss heterogeneous and fault tolerant DSM systems.

In section 2 we review some of the important issues regarding design and implementation of DSM systems. Section 3 discusses techniques to reduce the latency attributable to memory coherence algorithms. Section 4 discusses methods to reduce the communication overhead incurred by the underlying network software and hardware. Section 5 discusses innovations in the design

System Name	Current Implementation	Page Size	Coherence Semantics	Coherence Protocol	Performance Optimizations
Mether (1988-)	SPARCSstation ELC, Ethernet, SunOS 4.0, + Mether NFS	8192 bytes, 32 bytes	inconsistent	user-defined	inconsistent memory, short pages, demand and data driven updates
Midway (1991-)	DEC station 5000 workstations, Ethernet ATM LAN, Mach 3.0, some compiler support	4096 bytes	entry, release, processor	update	entry consistency, compiler support
Mirage (1987-89)	Vax 11/750, Ethernet, Locus distributed OS, Unix System V interface	512 bytes	sequential	write-invalidate	time-based coherence
MIRAGE+ (1990-)	IBM PS/2 Model 70 and 80, Ethernet, AIX and IBM TCF	4096 bytes	sequential	write-invalidate	time-base coherence, user locking, data compression
Munin (1989-)	Sun workstations, Ethernet, Stanford V kernel	objects	release	type specific protocol	release consistency, type specific coherency, multiple concurrent writers protocol, delayed update queue
PLUS (1988-)	Motorola 8800, Caltech mesh/wormhole routing, PLUS kernel	4096 bytes, 32 bit block	processor	non-demand update	page for sharing, block for coherence, specialized synchronization instructions
Shiva (1988-1990)	Intel iPSC/2 hypercube/circuit switching, NX/2 OS	4096 bytes	sequential	write-invalidate	page replacement algorithm

(continued from previous page)

Table 2: Summary of Selected DSM Systems

of DSM systems to minimize the impact of large page sizes and *false sharing*. We conclude with a summary of the lessons learned and achievements attained in performance during the past five years.

2 Review of DSM Design and Implementation Issues

This section gives a brief introduction to issues involved in the design and implementation of DSM systems. Much of the discussion in the later sections assumes a basic understanding of this

background material. Some of the information presented is condensed from an earlier article we wrote surveying hardware and software DSM systems [NL91]. The reader familiar with the area of distributed shared memory may wish to skip to the next section.

2.1 Granularity

Granularity refers to the size of the unit of sharing: byte, word, page, complex data structure. Most operating system implementations of DSM utilize the hardware page size as the unit of sharing, while hardware implementations typically utilize a unit whose size is closer to that of a cache line (16 to 64 bytes). Several systems use a hybrid approach which supports the page as the unit of sharing and a smaller block size as the unit of coherence. This paper focuses on page-based DSM systems, with page sizes ranging from 512 to 8192 byte pages.

A large page size is advantageous because the overhead of moving pages is amortized over a larger block of data. However, sharing pages among processes may cause contention, and the larger the page size, the greater the likelihood that more than one process will require access to that page. In this case, a smaller page size is better, as it reduces the possibility of thrashing. Thrashing occurs when a page is accessed by two or more processors in a manner which results in excessive movement of the page among the processors. This occurs when two processors are actively accessing the same variables and it also occurs when two or more unrelated variables are located in the same page – the page appears shared, even though the accessed variables are not. This phenomenon is known as *false sharing*. Another factor affecting the choice of page size is the necessity of keeping directory information about the pages in the system. The smaller the page size, the larger the directory must be to keep track of the pages.

In systems which structure the shared memory by data type, it is either structured as objects in distributed object oriented systems such as Choices [JC89], and Clouds [RAK89] [RYK91], or languages such as Emerald [JLHB88], the Shared Data Object Model and ORCA [BST89] [BKT92] [TKB92], Amber [CAL+89], and pC++ [LG91] [GL91]. In these systems, the grain size is variable as the sizes of objects and data types vary greatly, and as such, may be better matched to the application. However, object-based systems in which the data is attached to the object may still suffer from false sharing. This occurs when different parts of an object's data (e.g. the top and bottom half of an array) are accessed by distinct processors.

2.2 Coherence Semantics

The choice of coherence semantics defines the intended behavior of the distributed shared memory with respect to the load and store operations. The most intuitive semantics for memory coherence is *strict consistency* which means that a read operation returns the most recently written value. This type of coherence is achievable only in the case when there is a global notion of time that can provide a deterministic total ordering for all reads and writes. However, "most recently written" is an ambiguous concept in a distributed system where there is no global clock. For this reason, and to improve performance, DSM systems provide reduced forms of memory coherence.

Sequential consistency [Lam79] is the most common form of memory coherence in DSM systems. In a sequentially consistent system, the results of any execution is the same as if the operations of the individual processors were executed in some sequential order. The memory coherence protocols must ensure that all nodes view the same sequential interleaving of reads and

writes. Thus, the effect of each memory access must be globally *performed* before the next one is allowed to proceed. This can be achieved by serializing all read and writes through a central node; however, as discussed below, more efficient protocols can be used to support sequential consistency.

The notion of an access being *performed* was formally defined in [DSB86]. Intuitively, a write is *performed* when the value stored by the processor executing the write can be seen by all other processors. A read is *performed* when the value returned by the read cannot be affected by accesses made by all other processors. In this paper, we will use following terms interchangeably: performed, propagated, and updated.

We note that many researchers (including ourselves in [NL91]) use the term *strict consistency* for a distributed shared memory system, when they really should use the term *sequential consistency.* In order to be consistent with [Sch89], the term *strict consistency* should only be applied to systems in which there is no nondeterminism in the ordering of accesses.

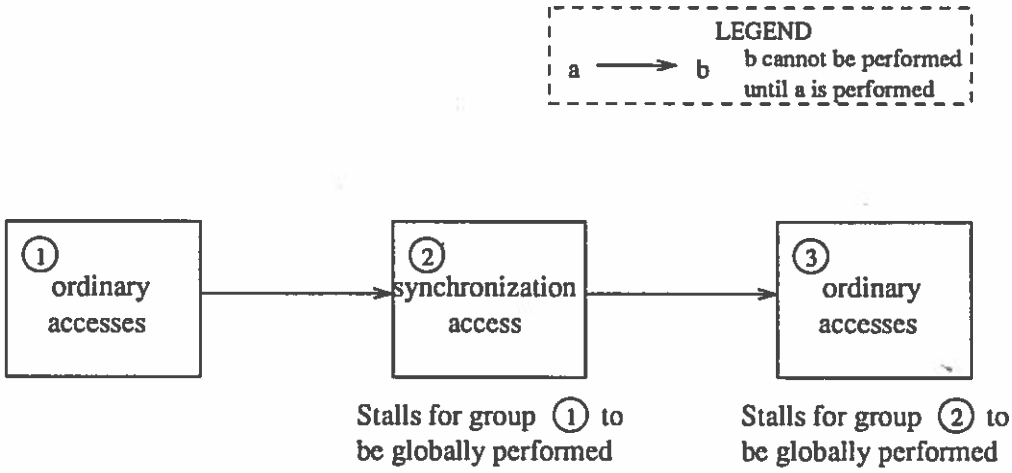
Processor consistency [Goo89] ensures that writes issued by a given processor are never seen out of order. However, the order in which the writes from two different nodes are observed by a third processor (or by themselves) need not be identical. This situation occurs when message-passing delays cause a local write to be seen earlier than a remote write and messages are guaranteed to be received in the order sent. Thus, given two nodes that write different values to a given variable, each node will first see its locally written value and then observe the value written by the other node. However, a sequence of writes issued by any one node will be seen in that order by all other nodes. The key performance gain achievable under processor consistency as compared to sequential consistency is that reads are allowed to bypass writes.

A number of weaker models of memory coherence have been proposed which give greater responsibility to the programmer or compiler for enforcement of coherence. All of these weaker models rely on the use of synchronization operations and associated synchronization variables to enforce consistency.

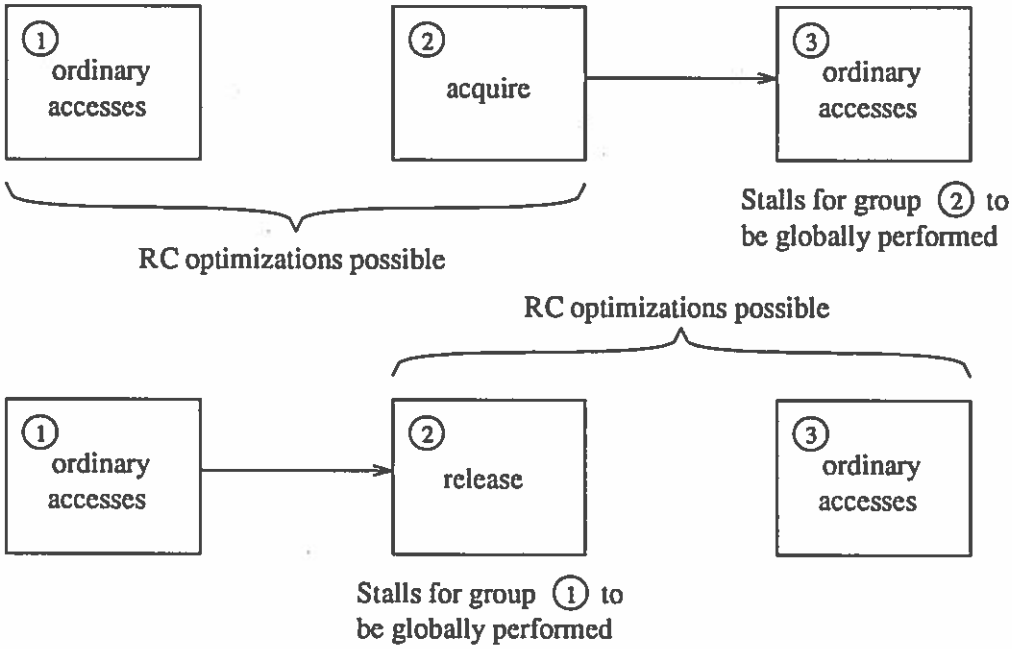
Weak consistency [DS90] distinguishes ordinary accesses (reads and writes) from special synchronization operations which when issued cause the system to become globally consistent. Specifically, in a system with *weak consistency*, when a synchronization operation is executed, the processor 'stalls' until all previous accesses are globally performed; in addition, accesses issued after the synchronization operation must wait for all previous synchronization accesses to be globally performed (see Figure 2(a)). Finally, the synchronization accesses themselves are guaranteed to be sequentially consistent. It is the responsiblity of the programmer to ensure consistency for shared data through correct use of the synchronization operations.

Under weak consistency, the only constraints on the ordering of the ordinary accesses are those specified above relative to the synchronization operations and the additional constraint that ordinary accesses must obey local program order. This allows ordinary accesses to be optimized for performance through pipelining or re-ordering, as long as they are guaranteed to complete before the next synchronization operation. The reduction in memory latency that results from weakening the consistency requirement can be seen by contrasting with sequential consistency in which each access must block until all previous accesses complete.

Release consistency [LLG+90a] is an extension of weak consistency which also distin-



(a) Weak Consistency



(b) Release Consistency

Figure 2: Weak Consistency *versus* Release Consistency

guishes regular accesses from synchronization accesses. Synchronization accesses are further divided into *acquire* and *release* operations. A system is release consistent if

1. before an ordinary access on a given processor is allowed to perform, all previous acquires on that processor must be performed;

2. before a release on a given processor is allowed to perform, all previous ordinary reads and writes on that processor must be performed, and

3. synchronization accesses are processor consistent.

This definition is illustrated in Figure 2(b). Acquire and release can be used to enforce mutually exclusive access to a set of shared variables with the sequence:

```
acquire(s); protected accesses; release(s);
```

Since the protected accesses stall until the acquire(s) is globally performed, the protected accesses are guaranteed to be guarded by the synchronization variable s. Similarly, since the release stalls until the protected accesses are globally performed, updates to the protected variables are also guarded by the synchronization variable s. Release consistency thus gives the user the responsibility to protect shared accesses through the use of (shared) synchronization variables.

Release consistency differs from weak consistency as follows: under weak consistency, there is only one type of synchronization operation. This single operation serves to stall the processor until all previous ordinary accesses are globally performed; in addition, the processor is stalled from executing any ordinary accesses that occur after the synchronization operation until the synchronization is globally performed (see Figure 2(a)). Under release consistency, these two

functions are split up with the former role assigned to the *release* operation and the latter role assigned to the *acquire* operation. As a result, the release operation only causes the processor to stall until the previous ordinary accesses are globally performed, but accesses issued after the *release* need not wait for the *release* to be globally performed. Similarly, acquire operations do not need to wait for previous ordinary accesses to be globally performed; only the ordinary accesses are required to wait for completion of prior acquires (see Figure 2(b)). Under release consistency the only constraints on the ordering of the ordinary accesses are those specified above and the constraint that ordinary accesses must obey local program order. In addition, to the optimizations that can be achieved through re-ordering and pipelining of the ordinary accesses, memory latency can be further reduced by overlapping ordinary accesses with the release and acquire operations subject to the above constraints.

Entry consistency [BZ91] is a refinement of release consistency that requires that the user associate a specific synchronization variable s with each protected item of shared data D(s). Thus, an access to data D(s) only needs to stall until the the previous acquire for the associated synchronization s is performed (whereas in release consistency, a shared access must stall for all previous acquires, including those guarding unrelated data accesses). Similarly, entry consistency only requires that accesses guarded by synchronization variable s be globally performed before the release is performed. Since entry consistency only places constraints on accesses to D(s) with respect to s, many optimizations of non-paired memory accesses are possible.

A detailed discussion of the access ordering restrictions associated with each of the memory coherence models defined above is given in [GGH91] and [BZ91]. Memory latency can be greatly reduced by exploiting the looser restrictions associated with the relaxed types of memory coher-

ence. In section 3.2, we describe the dramatic reductions in message count and message volume that can be achieved through careful design of the coherence algorithm under release consistency. The disadvantage of relaxing the coherence requirement is that existing programs which depend on a stronger form of coherence may not perform correctly if executed in a system which only supports a weaker form. This puts a heavier burden on the compiler and the programmer to insert synchronization primitives to ensure correctness.

2.3 Coherence Protocol

The coherence protocol consists of the data structures and algorithms used to implement a given coherence semantics. [SZ90] provide a taxonomy of existing coherence protocols.

Replication vs. no replication:

If the shared data is not replicated, then enforcing memory coherence is trivial. Requests are automatically serialized (in the order they occur) by the underlying network. A node which is handling shared data can merely perform each request as it is received, and this will ensure sequential consistency. Unfortunately, this serializing of data access causes a bottleneck, and defeats one of the major goals of DSM: parallelism. To increase parallelism, virtually all DSM systems allow data to be replicated. Thus, for example, multiple reads may be performed in parallel. However, replication complicates the coherence protocol. A centralized or distributed directory structure is needed to keep track of the location of pages.

Data location: migration vs. no migration:

To share data in a DSM system, a program must be able to find and retrieve the data it needs. If the data does not move around in the system, i.e., it only resides in a single static location, then locating it is easy. All processes can simply "know" where to obtain any piece of data. Some systems such as Linda [CG89] use a hash function to distribute the data statically. This has the advantages of being simple and fast, but may cause a bottleneck if the data is not distributed carefully (e.g. all the shared data ends up on a single node). An alternative is to allow the data to migrate freely throughout the system. This allows the data to be redistributed dynamically so that the data can be moved to where it is being used. However, locating the data then becomes more difficult. In this case, the simplest method of data location is to have a centralized server. The server can keep track of the locations of all shared data. The centralized method suffers from two drawbacks: the server serializes location queries which reduces parallelism, and the server may become heavily loaded which will slow the entire system down. Another method is to broadcast requests for data. Unfortunately, broadcasting does not scale well as all nodes must process a broadcast request, not just the nodes containing the data. The network latency of a broadcast may also be significant, requiring accesses to take extended periods of time to complete.

To avoid broadcasts and distribute the load more evenly, several systems use an owner based distributed scheme. This scheme is independent of data replication, but is seen mostly in systems which support both data migration and replication. In this scheme, each piece of data has an associated owner—a node which has the primary copy of the data. The owner changes as the data migrates throughout the system. When another node wishes to get a copy of the data, a request is sent to the owner. If the owner still has the data, it is returned. The owner,

however, may have given the data to some other node (the new owner). In this case, the old owner forwards the request to the new owner. The drawback with this scheme is that a request may be forwarded many times before reaching the current owner. This can be time consuming, and in some cases, more wasteful than broadcasting. In IVY, all of the nodes involved in forwarding a request (including the requester) are informed of the identity of the current owner. This "collapsing of pointer chains" helps reduce the forwarding overhead and delay.

Invalidate vs. update:

There are basically two types of protocols that handle replication: invalidation and update. Under sequential consistency, a write-invalidate protocol allows many copies of a read-only piece of data, but only one copy of a writable piece of data. When a write occurs, all copies of a piece of data except one are invalidated before the write can proceed. In a write-update scheme under sequential consistency, a write causes all copies of a piece of data to be updated. This is achieved either through broadcasting, or through cascading updates using a distributed directory that keeps track of the location of all copies. Systems supporting weaker consistency models also utilize invalidation or update protocols. However, multiple writable copies can exist because the page coherence is enforced through the synchronization operations rather than on each individual read or write access.

Traditional write-invalidate coherence protocols follow the following scenario: Each piece of data is tagged with a status which indicates whether the data is valid, whether it is read-only, whether it is shared, and whether it is writable. On a read, if the data is valid, it is returned immediately. If the data is not valid, a read request is sent (the location of a valid copy is

determined via one of the mechanisms described above), and a copy of the data is returned. If the data was writable on another node, this request will cause it to become read-only. The copy remains valid until an invalidate request is received. On a write, if the data is valid and writable, the request is satisfied immediately. Otherwise, an invalidate request is sent out along with a request for a copy of the data if the local copy is not valid. When the invalidate completes, the data is valid locally and writable, and the original write request may complete.

3 Reducing the Latency Due to Coherence Algorithms

The overhead due to coherence protocols is attributed to both the number of messages that must be sent in order to support memory coherence as well as the blocking and time delays incurred while executing the coherence algorithms. We divide systems that reduce memory latency through refinement of the coherence algorithms into three groups:

- **DSM systems that support sequential consistency.** These systems reduce latency through the careful design of data structures and algorithms for high efficiency and minimal message-passing.

- **DSM systems that relax the coherence requirements.** By supporting a weaker form of consistency, the number of update or invalidate messages can be vastly decreased. Furthermore, messages can be productively delayed, overlapped with useful computation, piggy-backed with other messages, and even omitted.

- **DSM systems that use hybrid coherence protocols.** These include both static hybrid schemes in which determination of the most suitable protocol is done before runtime,

and hybrid schemes which dynamically adjust the protocol based on monitoring of the runtime access patterns of the computation.

Below we describe examples of these types of systems and the performance improvements resulting from these efforts to reduce memory latency by addressing choice of memory coherence semantics and the design of the memory coherence protocol.

3.1 Systems that Support Sequential Coherence

The IVY DSM system was implemented on an Apollo DOMAIN network of workstations connected by a 12 megabit per second Apollo ring. This software implementation included changes to the Aegis operating system and the addition of memory management handlers to support DSM. The unit of sharing is the 1K byte page and IVY supports sequential coherence using a write-invalidate protocol. The designers of IVY proposed a series of algorithms ranging from a simple centralized manager scheme to several versions of a dynamic distributed manager scheme. We describe the message passing behavior of the MonitorCentralManager (MCM) and the DynamicDistributedCopySet (DDCS) algorithms [LH89] in detail in order to illustrate their impact on memory latency. Another highly efficient distributed algorithm is reported in [FBYR88].

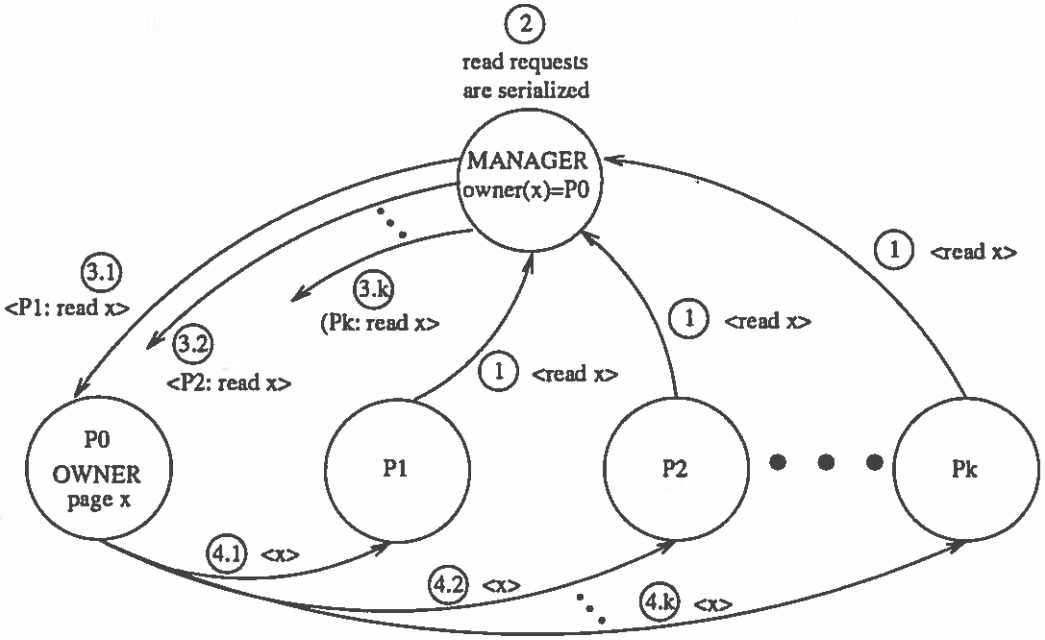
For the MCM algorithm, sequential consistency is enforced by having all read and write requests for a given page p serialized at the central manager. This is accomplished by locking a data structure called the Ptable which keeps track of the location, ownership, and access rights to each page. As seen in Figure 3(a), all read requests are sent to the central manager which forwards the requests to the owner node. The owner sends copies of the page to the requesting nodes. In the case of a write request, the central manager must also send invalidation messages

to all pre-existing copies of the page. A serious problem with this scheme is the high memory latency due to the bottleneck at the central manager node. The bottleneck occurs when there is a high number of page faults or in large distributed systems with many nodes. In addition, this algorithm incurs significant overhead for sending invalidation messages: if the hardware does not support broadcast, the messages must be sent serially.

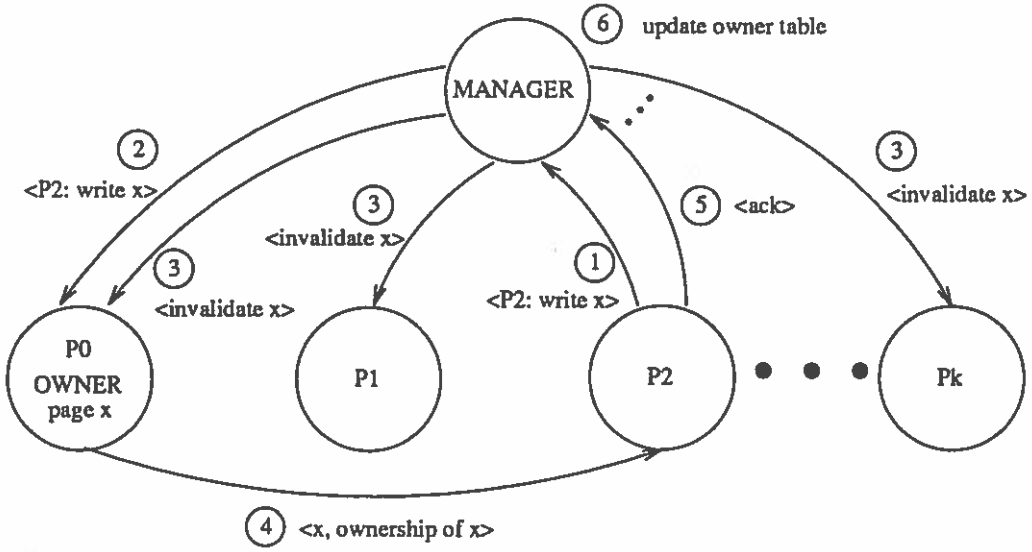
The DDCS algorithm is carefully designed to avoid these problems by distributing responsibility among several managers and by reducing the time to find the owner of a page and the time for sending invalidate messages. In this algorithm, the directory information for a given page p is structured as a bi-directional tree of processors rooted at the owner of p. Edges from the root to the leaves are defined by pointers in the copyset fields, while edges from the leaves to the root are defined by pointers in the probOwner fields (see Figure 4). Chains formed by the probOwner pointers are used to find the owner of a page and are then collapsed once the identity of the actual owner is known.

The DDCS algorithm incurs much less overhead than MCM ($log\ m$ messages, where m is the number of copies) in the case of write faults because invalidate messages are sent in parallel down the tree. The search for the page owner is reduced to $log\ m$, the depth of the tree, and is in practice further reduced because a read fault only needs to locate some processor (not necessarily the owner) that has a copy of the page. Note, that in practice m is small for many applications.

Figure 5 shows the dramatic decrease in forwarding messages that were actually sent by a parallel algorithm for 3D PDE using the improved centralized manager algorithm, the fixed distributed manager algorithm, and the dynamic distributed manager algorithm [LH89].



(a)



(b)

Figure 3: IVY Monitor Central Manager Algorithm: (a) read (b) write invalidate

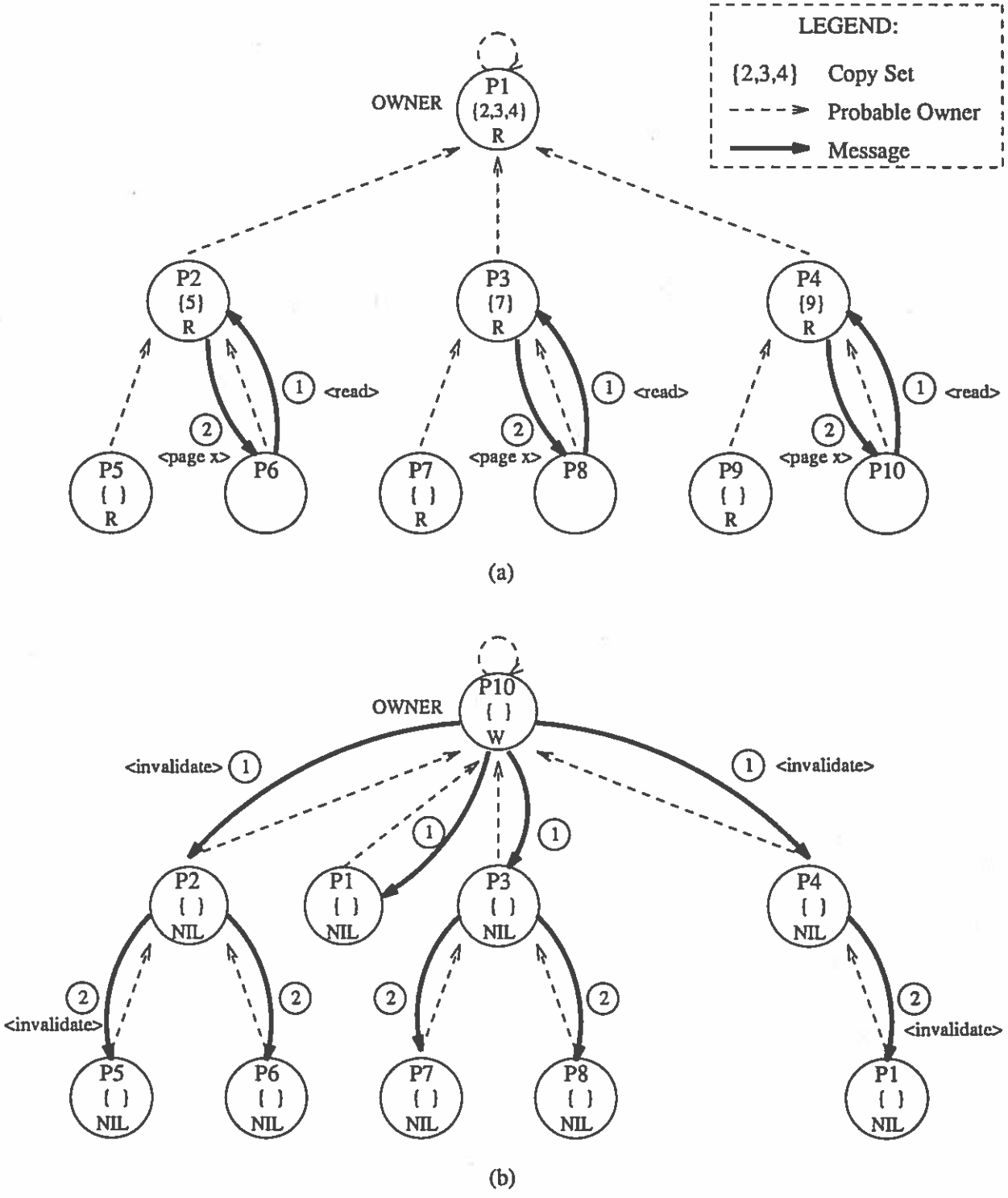


Figure 4: IVY Dynamic Distributed Copy Sets Algorithm: (a) read (b) propagation of invalidates

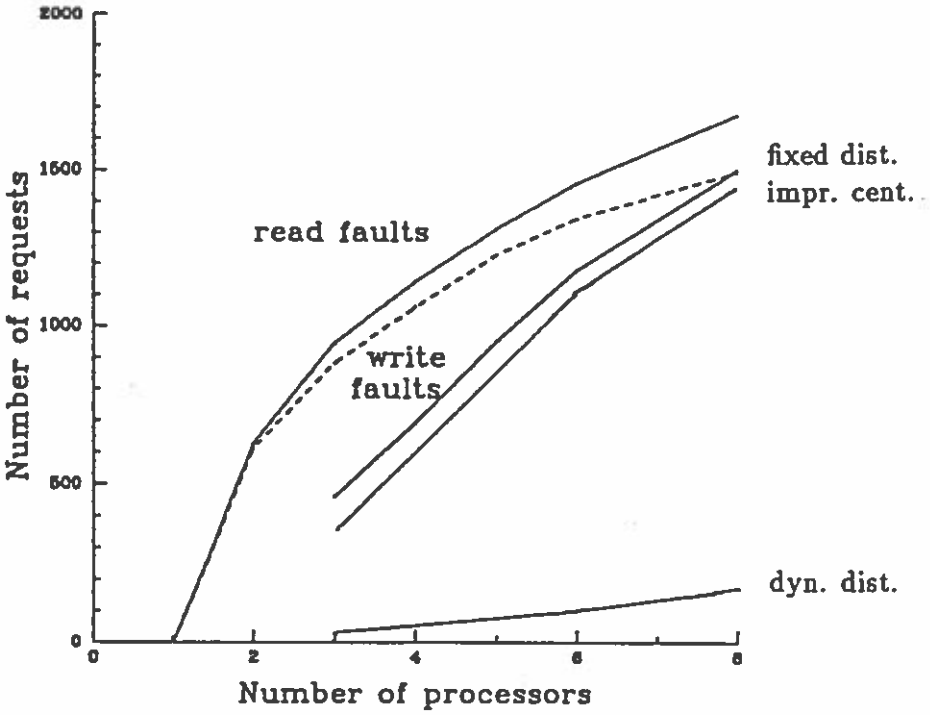


Figure 5: IVY Message Passing Overhead [LH89]

3.2 Systems that Support Relaxed Coherence

The technique of relaxing the memory consistency requirement has been shown to be an effective approach for reducing the number of messages and the amount of data transfer needed to support DSM. A number of software and hardware DSM systems support *release consistency* instead of sequential consistency, including Munin, Galactica Net, Memo, and Midway, as well as a number of systems not described in this paper. The Dash system, which is a hardware implementation of DSM, introduced the concept of release consistency [LLG+90a].

As described above, release consistency (RC) divides memory accesses into ordinary accesses and synchronization accesses, with the latter further divided into *acquire* and *release* operations. RC reduces memory latency by allowing the load and stores that occur outside the acquire-release brackets to be unconstrained. Thus, a processor that has updated a shared data item can proceed with the next instruction without waiting for the effect of the update to propagate through the system. The processor is only constrained to block as the result of the acquire and release synchronization operations. The tremendous reduction in memory latency achieved through use of relaxed consistency models, particularly release consistency, has been demonstrated by experiments across both hardware and software DSM systems, for a variety of coherence protocols, and for both distributed and scalable shared memory architectures [LLG+90a] [CBZ91] [WL92] [Car93] [DKCZ93] [ILP93]. Figure 6 shows the results of simulations modeling the performance of Dash under consistency models ranging from sequential consistency to release consistency. While these particular results were based on a hardware implementation of DSM, the same trend in improvements has been found for operating system implementations as discussed below.

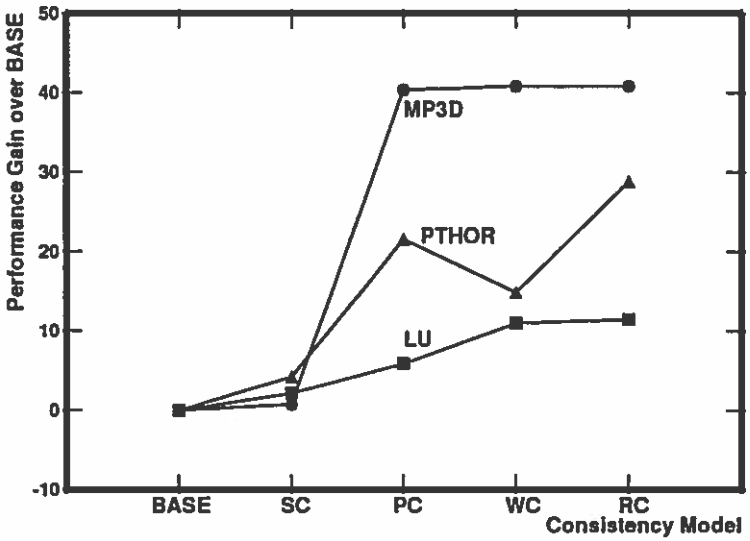


Figure 6: Improvements Achieved Through Relaxed Consistency models [GGH91]

In the following sections, we describe two software DSM systems that support release consistency: Munin and Memo. In addition, we describe several systems that further relax the memory coherence requirements. The Midway system supports entry consistency, while the Mether and Clouds DSM systems support inconsistent memory and user-level consistency protocols. Mether provides a number of specialized memory management operations that can be used to tailor the consistency requirements to specific applications. The Clouds object-based DSM system achieves performance gains by integrating support for inconsistent memory with synchronization operations.

3.2.1 Eager and Lazy Release Consistency

Research involving the Munin and Memo DSM systems has focused on the use of release consistency to reduce memory latency. Earlier research with Munin focused on support of type-specific

coherence protocols, with relaxed consistency supported for certain types of shared data. More recently, the focus has been carefully designed implementations of release consistency (RC) and the extension of RC to an optimization called lazy release consistency (LRC).

According to the definition of RC, before execution of a release operation can be performed, all previous remote accesses must be propagated to all copies of the affected pages. Under Dash's implementation of RC, remote accesses are propagated as they occur, resulting in a pipelining effect over time. Under Munin's implementation of RC, known as eager release consistency (ERC), these writes are held up until the release is executed, at which time they are propagated to other processors holding copies of the page. The savings in message-passing overhead realized by Munin's ERC approach are twofold: the changes for a given destination can be merged at the sender and they can be piggybacked onto the lock-release message (see Figure 7(a)). If an update protocol is used, the amount of data transferred is further reduced by only sending the changed portions of the page rather than the full page by doing a *diff* on the copies of the page. In the case of a write-invalidate protocol, invalidate messages must be sent to all copies.

The lazy release consistency (LRC) protocol is designed to yield further savings in message-passing overhead. LRC delays the propagation of writes until the next **relevant** acquire is issued. At that time only those writes that "precede" the acquire will be propagated. Informally, a write "precedes" an acquire if it occurs before any release such that there is a chain of release-acquire operations on the same lock, ending with the current acquire. The potential savings realized through this protocol is illustrated in Figure 7(b). In the ERC situation, each release results in propagation of the results of a write to all copies of the page. Under LRC, the message-passing traffic is reduced by delaying the propagation to the time of the acquire so that only the

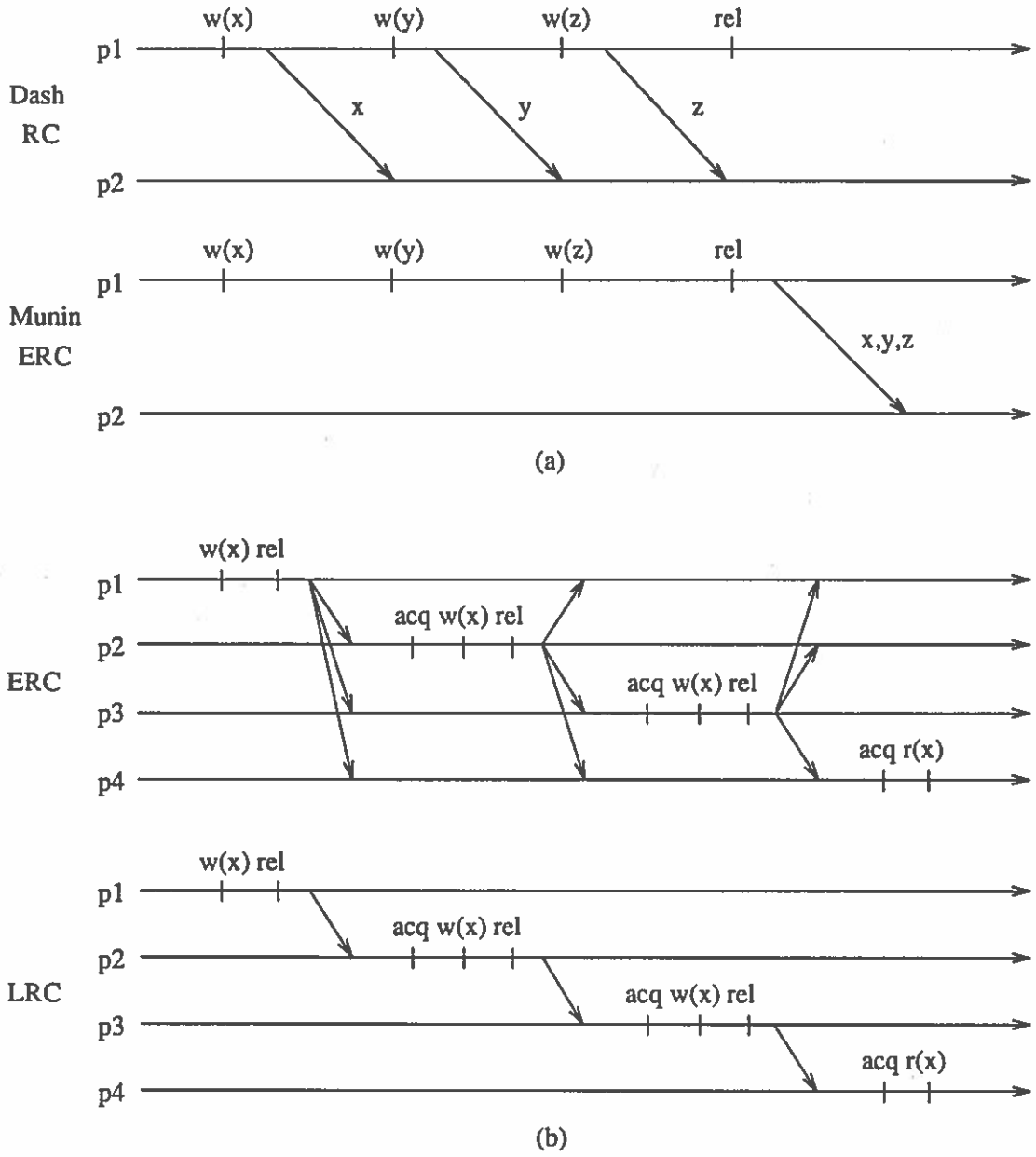


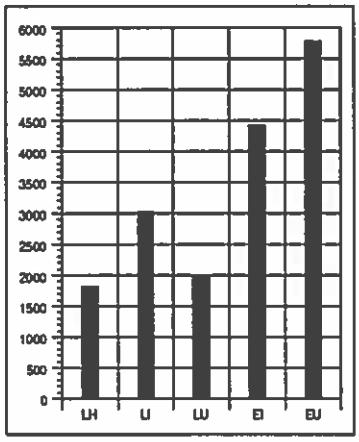
Figure 7: (a) DASH (Eager) Release Consistency *versus* Munin Eager Release Consistency (b) Eager Release Consistency *versus* Lazy Release Consistency [KCZ92]

relevant writes need to be propagated. At the time of acquire, *write notices* are sent by the releaser for those pages that need to be changed, along with the actual changes.

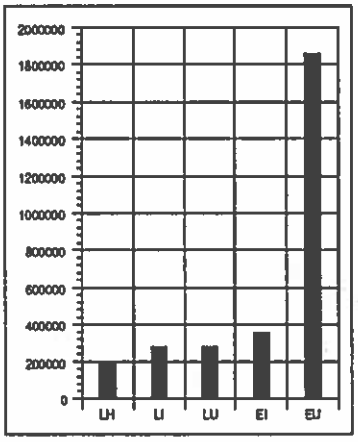
The method that is used to keep track of the "precedes" relationship in order to determine which stores need to be pulled by a given acquire is based on the *happened-before-1* partial order scheme of Adve and Hill [AH90]. A vector timestamp mechanism is used to implement this scheme [KCZ92].

Figure 8 shows the significant improvement in the number of messages and amount of data transferred under Munin's ERC implementation versus the LRC implementation for three parallel computations: traveling salesman, water, and cholesky. These results were obtained through simulations assuming a page size of 4096 bytes, infinite local memory, 40MHz RISC processors with 64 Kbyte direct-mapped local caches, and a communication network consisting of a 100 Mbit/sec crossbar switch. The coherence protocols tested included the four possibilities obtained by combining eager and lazy release consistency with write-invalidate and update protocols, yielding (EI, EU, LI, and LU). A fifth protocol called lazy hybrid (LH) combines invalidate and update methods (for details see [DKCZ93]).

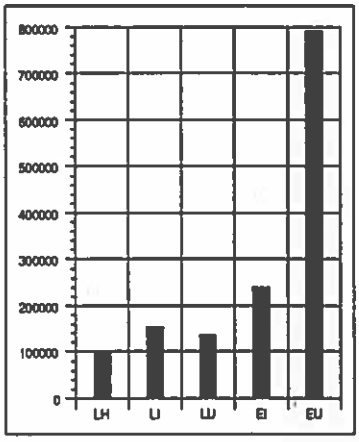
Independent experiments on the performance of lazy release consistency for shared memory architectures [PL93] and for the Dash system [LLG+90b] confirm the tremendous performance gains that can be realized. At the same time, it is important to note that the degree of savings in message passing overhead realized through these release consistency protocols depends on several other important factors. These include the characteristics of the parallel computation (granularity/synchronization frequency, degree of sharing, and access patterns); and architectural characteristics particularly page size and networking software and hardware.



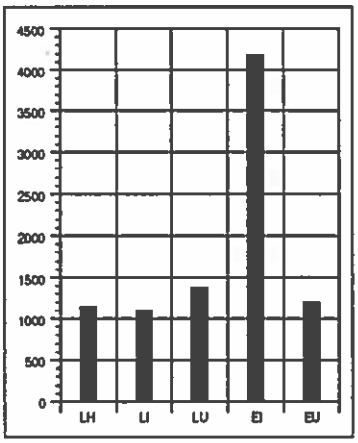
(a) Message Count in TSP



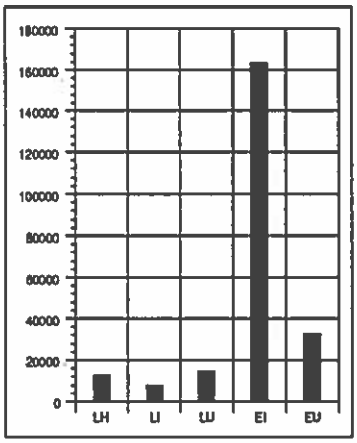
(c) Message Count in Water



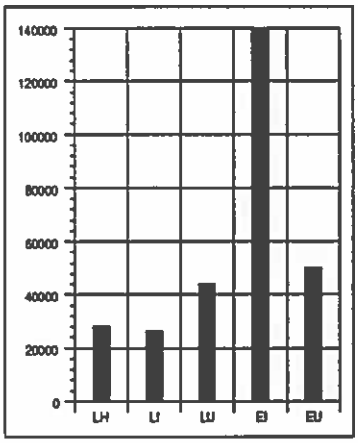
(e) Messasge Count in Cholesky



(b) Data (Kbytes) Transmitted in TSP



(d) Data (Kbytes) Transmitted in Water



(f) Data (Kbytes) Transmitted in Cholesky

Figure 8: Latency Reduction Due to Eager versus Lazy Release Consistency [DKCZ93]

3.2.2 Entry Consistency

The notion of entry consistency was introduced by Bershad and implemented in the Midway distributed shared memory system [BZ91] [BZS93]. Midway is currently running on DECstation 5000 workstations with both ethernet and an ATM network. Entry consistency is supported through modifications to the C compiler and to the underlying Mach 3.0 operating system. Midway also supports processor consistency and release consistency; however, for purposes of this survey, we will focus on entry consistency and operating system support for entry consistency.

Under entry consistency, data accesses are paired with synchronization accesses through language level declarations. A processor never enters a critical section that has been protected with acquire(s) and release(s) until all updates to data guarded by the synchronization variable s have been performed. Entry consistency requires fewer messages than under release consistency since only those updates to data protected by the synchronization variable need to be sent. The vast reduction in number of messages and number of bytes transferred for entry consistency versus release consistency can be seen by examining the last two rows of Table 3. Midway further reduces latency by transmitting updates only if they are more recent than the acquiring processor's locally cached values. This is accomplished using Lamport's happened-before relation to partially order the updates. A further optimization under entry consistency delays updates until the next acquire(s) operation. As a result, all updates occur outside the critical section. Finally, entry consistency distinguishes between exclusive and non-exclusive accesses to the synchronization variable s in order to support read-replication of s.

	# Proc.	Elapsed (secs)	Speedup	Input Transfer (secs)	Compute Results (secs)	Output Transfer (secs)	Data Transferred (mbytes)	# Msgs
MM-ec: slow-ether	1	282	1	0	0	0	0	0
	2	148.4	1.90	12.92	132.4	3.05	2.14	24
	4	84.9	3.32	12.08	66.3	6.51	4.81	72
	6	65.0	4.34	13.72	44.4	6.92	7.13	120
	8	58.6	4.81	17.56	33.2	7.87	9.36	168
MM-ec: fast-ether	1	164	1	0	0	0	0	0
	2	92.8	1.77	8.70	81.4	2.66	2.14	24
	4	53.3	3.08	8.50	40.6	4.17	4.81	72
MM-ec: fast-ATM	1	164	1	0	0	0	0	0
	2	83.5	1.96	1.53	81.7	0.31	2.14	24
	4	43.3	3.79	1.86	40.9	0.50	4.81	72
MM-rc: fast-ATM	1	164	1	0	0	0	0	0
	2	86.8	1.89	3.14	82.0	1.61	2.17	1802
	4	48.4	3.39	6.06	41.1	1.28	4.97	5106

Table 3: Midway Performance [BZS93]

3.2.3 Inconsistent Memory

Mether is a DSM system under development at the Maryland Supercomputing Research Center [MF90] [MP93]. Mether supports inconsistent memory, leaving the responsiblity for enforcing consistency to user-defined protocols. The developers of Mether believe that significant performance gains can be achieved by allowing the user to tailor the coherence protocols to the needs of each specific application program. The current configuration of Mether consists of a cluster of Sun SPARCStations connected with Ethernet and running SunOS 4.0. DSM under Mether is implemented through modifications to the NSF file system. Long-range plans for Mether include implementation on a wide variety of hardware platforms. These include the Mether NFS implementation running on workstations with higher bandwidth communication networks as well as a full hardware implementation of DSM.

Mether maintains one consistent copy of a given page along with multiple inconsistent copies.

User-defined coherence protocols are constructed by utilizing the following primitive page operations supported by Mether. The first three operations relate directly to memory coherence, and the remaining operations contribute to reductions in memory latency through mechanisms other than relaxed memory coherence.

- **Network refresh** is a sender-initiated type of user-driven page propagation in which an application can deliver the latest copy of the writable page to all processors who hold read-only copies of that page.

- **User purge** is a receiver-initiated type of page propagation in which an application can purge a read-only data. The next access causes the most recent copy to be transmitted over the network.

- **Request consistent copy** is another receiver-initiated action in which an application can request a consistent copy of a given page. At that time all processors holding copies of the page wil receive an up-to-date copy.

- **Short and long pages** (32 bytes and 8192 bytes, respectively) are supported by Mether. The advantages of short pages are discussed later in section 5.1.2.

- **Latency tolerance** provides a special address space for data items that may take a long period to fetch. Special pre-fetch operations can be invoked to fetch these pages before they are actually used.

- **Event-driven memory synchronization (EDMS)** provides a mechanism which allows a processor to pause in the middle of a memory read until some action is taken by another processor.

EDMS attempts to reduce latency by integrating memory coherence with synchronization operations.

Below we show the results of experiments performed with Mether which highlight the reduced latency achieved through its support of inconsistent memory. Table 4 shows the performance of three user-defined coherence protocols running a simple distributed program to count up to 1024. The protocols support (a) consistent memory with long pages, (b) consistent memory with short pages, and (c) inconsistent memory with long pages. From (b) and (c), it is clear that inconsistent memory involves much less latency than consistent memory. It is important to note that the user-defined protocol whose performance data are given in (c) was developed after experimentation with several alternative protocols for inconsistent memory. The effect of using short pages is illustrated by comparing columns (a) and (b) in the table.

Operation	Cost		
	(a) Consistent Long Page	(b) Consistent Short Page	(c) Inconsistent Long Page
Wallclock Time	128 seconds	68 seconds	57 seconds
User Time	10 seconds	3 seconds	.7 seconds
Sys Time	30 seconds	17 seconds	6 seconds
Network Load	66 kbytes/sec	approx 2.2 kbytes/sec	.5 kbytes/sec
Context Switches	4 per addition	4 per addition	5 per addition
Average Latency	120 ms	68 ms	20 ms

Table 4: Mether User-defined Coherence Protocols [MF90]

Clouds is an object-based distributed operating system which was developed at the Georgia Institute of Technology. The current implementation is constructed on Sun 3 workstations connected by ethernet. The distributed shared memory subsystem of Clouds is currently implemented in software as a server that interacts with the virtual memory management subsystem [RAYK89] [AMM90] [RYK91]. Long range plans for Clouds include a hardware version with the

DSM controller as a co-processor.

Clouds' distributed shared memory controller (DSMC) recognizes both data movement and synchronization primitives. These primitives provide efficient support for both sequential and inconsistent memory. Because the DSMC responds to both types of primitives, a number of optimizations are possible. For example, requests for remote data can be combined with the associated lock request. In addition, Clouds' object-level lock and unlock operations are translated into get and discard primitives. This allows further integrated optimizations on locks and associated data objects. The disadvantage of combining synchronization and data access is the inability to separate coherence policies from synchronization. The primitives recognized by the DSMC are:

- get(segment,mode), where the mode can be read-only, read-write, weak-read or none. Read-only mode provides non-exclusive access but guarantees that the segment will not change until it is explicitly discarded. Read-write mode provides exclusive access with a guarantee that the segment will be retained until explicitly discarded. If only the first two modes are used, the system will be sequentially consistent. Weak-read mode signifies non-exclusive access with no guarantees on whether the segment will change or not. None mode signifies non-exclusive access with no guarantees on whether the segment will be retained or not. If the last two modes are used, memory is inconsistent and it is the responsiblity of the user to ensure consistency.

- discard(segment). The use of explicit discard operations enables the elimination of unnecessary invalidation messages under a write-invalidate scheme.

- P(segment,semaphore), used for synchronization within objects.
- V(segment,semaphore), used for synchronization within objects.

Other optimizations implemented in Clouds include the following

- Reuse of data associated with discarded segments. Version numbers associated with segments allow an old copy to be used if the version numbers match, thus eliminating unnecessary data transfer.
- Recognition of queued requests for the same page. While one request for a given page is being processed, it is possible the same processor may issue subsequent requests for the same page which are then queued at the owner. Under certain conditions, e.g. no pending read-write requests, the new requests can be satisfied from the first copy without data transfer.
- Pre-allocation of memory buffers while initiating a remote request.

3.3 Systems that Use Hybrid Coherence Protocols

The use of hybrid approaches is based on experimental evidence showing that no one protocol performs consistently across application types (granularity and patterns of data access) and architectural parameters (page size, networking software and hardware, scheduling support). In this section we describe the reduction in latency to be gained by tailoring the coherence protocol to application-specific data access patterns. Munin provides system support for tailoring the coherence protocol to one of eight types of shared data, while Galactica Net's operating system dynamically chooses between write-invalidate and update protocols based on runtime behavior.

Galactica Net gives the user a choice of three models of consistency: sequential, release, and weak; while Midway supports processor, release, and entry consistency. Mether and Clouds support inconsistent memory, relying on the user to develop application-specific coherence protocols. Clearly, there are tradeoffs between the gains to be realized by greater flexibility versus the overhead of selecting or developing an appropriate protocol.

3.3.1 Munin Type-specific Coherence

The Munin DSM system [CBZ91] [Car93] was implemented on Sun workstations through modifications to the Stanford V kernel. The Munin system supports type-specific coherence by tailoring the choice of coherence protocol to user and compiler-generated information about the access characteristics of shared data. A formal model for type-specific coherence is proposed by Leong and Agrawal in [LA92]. Under Munin, shared program variables are labeled using one of seven sharing annotations. The Munin runtime system then adjusts the consistency protocol based on the annotation. In addition, Munin provides a number of library routines that allow the user to further refine the relationship between the data and the coherence protocol at runtime.

The annotations currently supported by Munin include:

- Read-only. Objects that are only read after initialization. The coherence protocol supported for read-only objects is replication on demand.

- Migratory. Objects for which a single thread performs multiple accesses including one or more writes before another thread accesses it. The coherence protocol supported for migratory objects is to move the object to the thread upon demand, giving it both read and write access, and to invalidate the old copy.

- **Write-shared.** Objects that are concurrently written by multiple threads without synchronization because the programmer knows that updates do not modify the same words in the object. The coherence protocol supported for write-shared objects allows replication and concurrent writing to the disjoint portions of the object. Updates are delayed and when sent, only the *diff* portions of the shared object are actually transferred.

- **Producer-Consumer.** Objects that are written by one thread and read by one or more threads. The producer-consumer consistency protocol replicates the object and uses update instead of invalidation. Release consistency allows the updates to be delayed until the release is issued, at which point they can be combined into a single message.

- **Reduction.** Objects that are used in global reduction operations such as min or max. Reduction objects are implemented using a fixed owner protocol. The object is replicated on reads, but all writes are sent to the owner who updates the other nodes.

- **Result.** Result objects are accessed in alternating phases. In one phase, multiple threads modify the object; in the next phase, a single thread accesses the object. The protocol supported for result objects allows multiple concurrent reads and writes for the first phase. For the second phase, updates are sent only to the single thread needing exclusive access.

- **Conventional.** All unannotated objects are classified as conventional objects. These use replicated readers, single owner-writer using a write-invalidate protocol.

Munin's preliminary experiments with type-specific coherence protocols show that significant performance gains can be realized but that they are dependent on the characteristics of the parallel application. For example, the execution times for Matrix Multiply shows only around

4% improvement when result and read-only objects were declared. However, for Successive Over Relaxation (SOR), the execution time was reduced by more than 50% when the producer-consumer annotation was utilized.

3.3.2 Galactica Net Update and Invalidate Protocols

The Galactica Net DSM Architecture [WL92] currently consists of a collection of Lynx computers, each containing four Motorola 88110 processors with 8 Kbyte local caches, 256 Kbyte secondary caches, and 256 Mbyte local RAM. While Galactica Net is not purely an operating system implementation of DSM, we have included it in this paper because it supports sharing and coherence at the page level (8192 byte page). The Lynx multiprocessors are connected utilizing specialized hardware which provides a point to point mesh interconnection network.

The Galactica Net supports both update and invalidate coherence protocols in order to adapt to the shared memory reference patterns exhibited at runtime. Their approach is supported by studies that indicate that as the size of the unit of sharing increases update-based protocols exhibit better performance, while as the size of the unit of sharing decreases, invalidate-based protocols become more desirable. In addition, their hybrid protocol scheme is used to support three types of memory coherence: sequential, weak, and release.

Galactica Net's update protocol is implemented through the use of *virtual sharing rings*. A virtual sharing ring links together the multiple copies of a page that has been determined to be an update mode page. When a processor attempts to write to an update mode page, the Galactica Net Interface Module (GIM) intercepts the message on the local bus and then propagates it around the virtual ring from node to node until the message returns to the initiating node.

Each node on the virtual ring updates its local copy. Pages that have not been allocated as update mode pages are treated as invalidate mode pages. These are handled through standard replicate-on-read, write-invalidate protocols.

The decision of whether to classify a page as update mode or invalidate mode is made dynamically by the operating system. Update counters in the GIM mapping table are used to keep track of the number of updates received for a given page. The counter is incremented with each update and reset whenever the page is referenced locally. If the counter exceeds a threshold value, the page is dropped from the update ring for that page and the local copy of the page is invalidated. Table 5 shows the number of pages and amount of data transferred under the two protocols supporting sequential consistency. The update protocol requires fewer pages by an order of magnitude and significantly fewer bytes of data to be transferred. However a pure update protocol was not chosen for Galactica Net because of evidence that only a small percentage of the pages requires sharing and because other methods for reducing memory latency (such as a smaller unit of coherence and intelligent data placement) are well suited to write invalidate protocols.

Comparison of Update and Invalidate Bandwidth Requirements

	Update			Invalidate		
Program	Updates	Pg Xfers	Total bytes	Invalidates	Pg Xfers	Data bytes
ecas	9,157,557	21	121,650,624	1,591,245	1,590,563	144,990,208
hart	870,166	14	11,766,659	30,103	29,914	245,055,488
locus	802,550	28	11,127,588	34,474	34,011	278,618,112
mp3d	4,458,265	0	62,172,640	374,146	370,981	3,039,076,352
mstracer	1,225,495	107	17,255,504	55,518	55,140	451,706,880
notoy	947,937	31	12,935,661	37,138	36,859	301,948,928
piletracer	1,459,347	81	20,147,284	53,121	52,558	430,555,136
qsort	8,271,048	471	116,724,384	1,077,504	1,076,808	231,276,544

Table 5: Galactica Net messages [WL92]

The designers of Galactica Net show that the update protocol provided by the virtual sharing ring can efficiently support sequential consistency as well as weak and release consistency. The key issue is careful design of the update scheme to avoid race conditions. For example, they describe a naive update protocol in which any process issuing an update waits until the update propagates around the ring. If updates to the same variable are initiated from two different locations on the virtual ring at the same time, it is possible for these two updates to be seen in different orders based on the location of the readers relative to the two writers, a clear violation of sequential consistency. To avoid this problem, Galactica Net uses a two-phase update protocol in which each update propagates around the ring twice. During the first pass, data values are updated but flagged as *pending*. Any processor that tries to reference a flagged value blocks until the pending flag is cleared. During the second pass of the update protocol, all the pending flags are cleared. This ensures sequential consistency for each virtual ring and across multiple rings. The two-phase protocol is not needed for weak consistency or release consistency. In these cases, it is only necessary to ensure that all previous memory references are completed before a synchronization access is issued.

The performance of Galactica Net for sequential, weak, and release consistency using the virtual sharing ring implementation was tested through simulation using a combination of Stanford's Tango simulation package and the Galactica Net Simulation software, GNetSim. Their results support those of other researchers which exhibit dramatic reductions in execution time for weaker consistency models. In particular, the Galactica Net experiments show 87% to 98% of memory latency costs hidden through the use of release consistency for the programs used in their test suite.

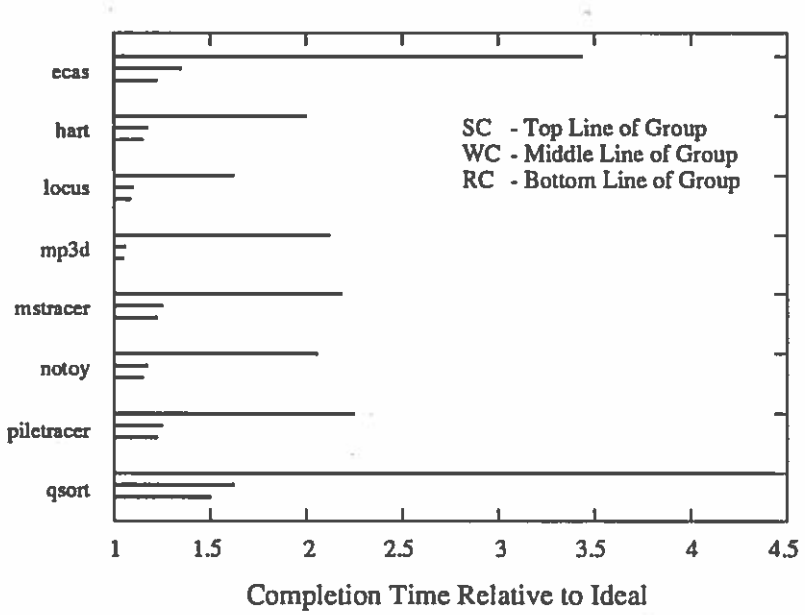


Figure 9: Latency Reduction due to Release Consistency for Galactica Net [WL92]

4 Latency due to Network Communication Overhead

In this section we describe recent efforts to reduce the communication overhead incurred by DSM systems at the networking level, both hardware and software.

Hardware: Several researchers have shown that the bandwidth of existing 10-megabit per second provided by ethernet quickly becomes a bottleneck for DSM systems. A number of systems have experimented with the use of faster communications hardware such as ATM networks. This new technology not only supports higher bandwidths in the 100-megabit per second range, but ATM networks also provide lower levels of message contention because messages collide only when bound for the same destination. Experiments at Rice University indicate that point to point networks with 100-megabit capacities provide good performance for large and medium grained DSM applications. Table 6 shows the higher speedups achieved for two applications as a function of communication bandwidth. For this experiment, the lazy hybrid coherence protocol

was used and the number of processors was fixed at 16. Experiments with Midway also confirm the performance gains achieved through the use of ATM networks (see Table 3).

	Jacobi	Water
10 Mbit Ethernet w/ Coll.	2.5	0.7
10 Mbit Ethernet w/o Coll.	4.1	1.3
10 Mbit ATM	10.1	4.0
100 Mbit ATM	13.7	8.3
1 GBit ATM	13.8	8.8

Table 6: Effect of Communication Bandwidth on Speedup [DKCZ93]

In addition, a number of DSM implementations have been designed for message-passing multicomputer systems. These systems take advantage of the reductions in network latency possible on systems with point to point connections and flow control technologies such as circuit switching and wormhole routing. In the following subsections we describe the Shiva and memory server model implementations of DSM for the iPSC/860; and the Galactica Net DSM systems which is constructed using communications hardware providing a two-dimensional mesh with wormhole routing.

We reiterate that our focus in this paper is on software DSM systems running on off-the-shelf hardware. Thus, we do not discuss hardware DSM systems such as Dash, KSR, Alewife, in which the design of the communication hardware is an active component of the research.

Software: Research conducted at Rice University on a distributed shared memory system running on standard workstation and operating systems demonstrates that the software communication overhead is the dominant source of latency. In their experiments with the Memo DSM system [KDCZ93], memory management costs (both user and kernel levels) were small and wire time was negligible compared to the Unix communication overhead. Under ERC, Unix commu-

nication overhead ranged from 20% to 36%, while under LRC, Unix communication overhead ranged from 30% to 55% of the total communication overhead.

4.1 DSM using multicomputer communications hardware

4.1.1 Shiva and the Memory Server Model

Shiva was a direct successor to IVY that was implemented for the Intel iPSC/2 message-passing multicomputer [LS89]. More recently, an implementation of DSM for the Intel iPSC/860 illustrates its natural evolution to this environment using a memory server model to provide transparent distributed caching across the multicomputer nodes [ILP93]. Both of these systems take advantage of high speed communication networks available for multicomputer systems which currently are capable of bandwidths up to several orders of magnitude beyond raw ethernet speeds as well as beyond typical transfer speeds between memory and disk. An additional advantage to the use of interconnection networks for DSM systems is the lack of contention due to ethernet style collisions, although link contention remains a performance issue for further study. While these systems achieve much of the reduction in memory latency through the communications hardware, we have included them because they represent operating system implementations of DSM based on the page level of granularity.

Shiva provides a single, flat, shared virtual memory address space. The size of the virtual address space equals the sum of the capacities of the individual physical memories, minus the space used by the Shiva kernel. Like IVY, Shiva supports sequential consistency using invalidation techniques. Initially, the fixed distributed manager strategy was adopted in which pages are statically assigned to managers using a simple hashing function. This earlier system served

as proof by existence for the support of DSM on message-passing multiprocessors.

Under the memory server model [ILP93], a variety of virtual memory management services, including DSM, are provided under a unified client-server model. This model distinguishes nodes as either computation nodes (clients) or memory server nodes. Memory server nodes act as caching devices at a level between the local physical memory and the disk. Thus, when a page fault occurs on a node, the page may be brought in from a memory server node rather than the disk. Similarly, the page replacement algorithm may write a page to a memory server node rather than the disk. The savings in memory access time is based on the fast interprocessor data transfer rates provided by multicomputer communications hardware, currently three orders of magnitude faster than movement between disk and a node.

One of the design issues to be addressed in the use of the memory server model involves the function of each node: exclusively computation, exclusively memory server, or both. In addition, the assignment of function can be done statically or dynamically by the operating system or under user control. Finally, the page replacement algorithm can be carefully designed to reduce the latency incurred for a page fault. Under Shiva pages are classified as no-access, read-copy, read-owner, or writable. The page replacement algorithm selects a page to be replaced based on these classifications as a secondary factor with criteria such as LRU used as the primary factor. For example, page frames classified as no-access should be reclaimed with higher priority than all others because the processor will page fault regardless of its presence in physical memory. Next in priority are read-copy pages because there is another copy of the page in the system. Third are read-owner pages because purging of this type of page does not require the page to be written to disk.

Preliminary experiments were performed with a suite of application programs running on an iPSC/860 hypercube multicomputer which provides a network bandwidth of about 2.8 Mbytes per second. Depending on the application, the reductions in execution time under the memory server model ranged from 10% to improvement by a factor of three. Programs with large data structures and low reference locality showed the greatest improvements under this model. Increasing the bandwidth of the underlying communications hardware to the capacities provided by Intel's Paragon (200 Mbytes per second) improved program execution time by an order of magnitude.

4.1.2 Galactica Net

Another system that is designed to utilize the high bandwidth offered by multicomputer communications hardware is the Galactica Net described earlier. Each node of the Galactica Net DSM system includes a Mesh Router Chip (MRC) similar to the Cal Tech router chip which implements a two-dimensional mesh with wormhole routing. Each link in the mesh is capable of transmission speeds of about 100 Mbytes per second in each direction, yielding a total throughput of about 500 Mbytes per second. Simulation experiments show that the bandwidth requirements of Galactica Net's hybrid coherence protocol are easily satisfied by this hardware. These results are compatible with experimental results from the Memo DSM system which tested the performance on a 100 Mbyte per second point-to-point ATM LAN.

4.2 Data Compression to Reduce Communication Overhead

Software communication overhead can be reduced in any system which reduces the number of messages generated by the coherence protocol, as described in Section 3. Another approach to reducing the software communication overhead is being tested by the Mirage+ DSM system which exploits data compression to reduce the number of packets needed to transmit a page of data.

Mirage+ [FHJ93] is a DSM system for high-end personal computers. It is a successor to the earlier Mirage [FP89] system. Mirage+ is currently implemented on nine IBM PS/2 Model 70s and three IBM PS/2 Model 80s connected by a 10 MB ethernet. The CPUs have been upgraded to i486s, and each node has either a 10 MB or 16 MB main memory disk capacity of 140 to 160 Mbytes. Mirage+ is built on an AIX/TCF software platform, a modification to IBM's Transparent Computing Facility which allows a network of PCs to function as a transparent single-system cluster of machines.

The size of a page for the IBM PS/2 workstations is 4096 bytes. Transmission of a page over the network requires four network packets due to AIX/TCF's buffer size of 1K and the Ethernet limitations of about 1500 bytes per packets. The designers of Mirage+ were thus faced with potentially very large delays due to network latency.

To address the problem of network latency, Mirage+ uses data compression techniques to reduce the number of packets needed to deliver one page. By measuring the elapsed time for a page fault without compression, it was determined that the compression/decompression time must be less than 8 msec in order for there to be a net gain. Some of the criteria used in the selection of an effective data compression algorithm included

- effective performance on binary data (many compression algorithms are designed or tested only for textual data);
- effective performance on data in the 4K byte range (many compression algorithms have diminishing performance as the size of the data block decreases);
- quantized compression rates of 75%, 50%, and 25% in order to reduce the number of packets needed per page; in particular, the algorithm must achieve at least 75% compression;
- the need for a lossless compression algorithm since the overhead of re-transmission would cancel the reduction in latency achieved by compression;
- ease of implementation.

The preliminary choice made by the Mirage+ team was a run-length encoding (RLE) algorithm which fulfilled the above criteria and had the additional advantage that it was suitable for data with repeated runs such as that found in sparse matrix applications. The RLE algorithm compresses a 4K block of zeroes into 26 bytes in about 400 μsec and takes about 2 msec to handle a 4K block of data that it cannot compress. Decompression incurs approximately the same overhead. The worst case performance occurs when only 75% compressions can be achieved; in this situation, the overhead of compression/decompression is about 4 msec but the reduction in one network packet yields an overall savings of about 8 msec. Finally, if the compression algorithm determines that it cannot achieve at least 75% compressions, it immediately aborts and sends the whole uncompressed page. Preliminary performance results for a battleship simulation program running on two nodes are shown in Table 7. This experiment shows a 50% reduction in number of packets transmitted yielding an overall 34% reduction in total runtime. Continuing

work in this area will investigate the performance of additional compressions algorithms over a wide range of parallel applications. Work is also underway to design compression algorithms tailored to the needs of memory access patterns exhibited by Mirage+.

	No compression	RLE compression	Speed-up
Total run time	94.64 sec	62.92 sec	30.72 sec
Iterations per sec	8.4	12.5	4.1
Total iterations	789	789	
Total number of pages exchanged	789 + 789	789 + 789	
Total number of page packets	3156 + 3156	1509 + 1509	
Page compressed into one packet	0 + 0	87 + 88	
Page compressed into two packets	0 + 0	684 + 682	
Page compressed into three packets	0 + 0	18 + 19	
Pages sent uncompressed	789 + 789	0 + 0	

Table 7: Compression of Pages for Battleship Simulation [FHJ93]

5 Latency Due to Page-sized Granularity and False Sharing

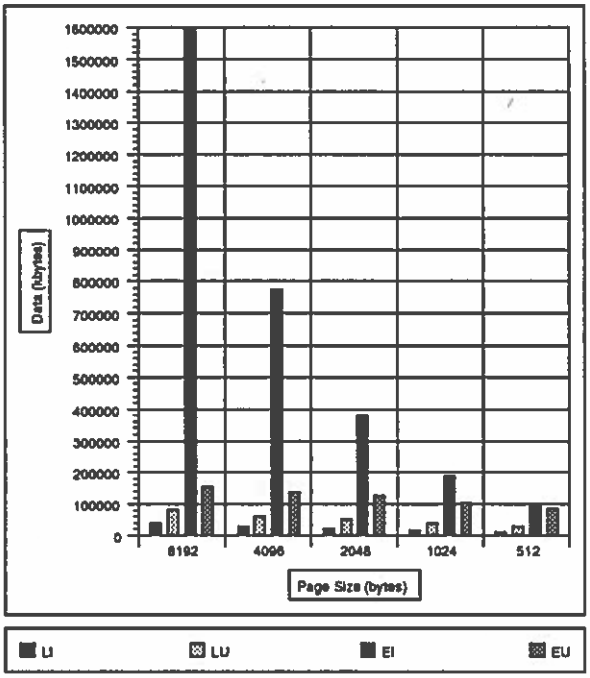
Because most of the operating system implementations of DSM are based on page-level sharing, it is clear that often more data is transferred than is necessary for remote accesses to small amounts of shared data. In addition, page-sized granularity can result in unnecessary data movement due to false sharing when two or more processors access unrelated pieces of data which happen to be located in the same page. In the worst case, it can lead to thrashing behavior in which exorbitant amounts of time are spent moving pages among nodes with little useful computation occurring. In this section we describe several approaches that address the issue of latency due to page-sized granularity: reducing the size of the unit of coherence, page-locking, and multiple-writers protocols. An approach not discussed in this paper is to support sharing and/or coherence for user-defined objects, rather than for a fixed-size system-dependent unit of granularity such as

the page, block, or cache line. Object-based DSM reduce latency by allowing for arbitrary-sized objects that are tailored for the specific distributed application, thus reducing or eliminating thrashing and false sharing. Clouds and Choices are two operating system implementations that take an object-oriented approach to DSM. A number of compiler-based DSM systems also are object-oriented.

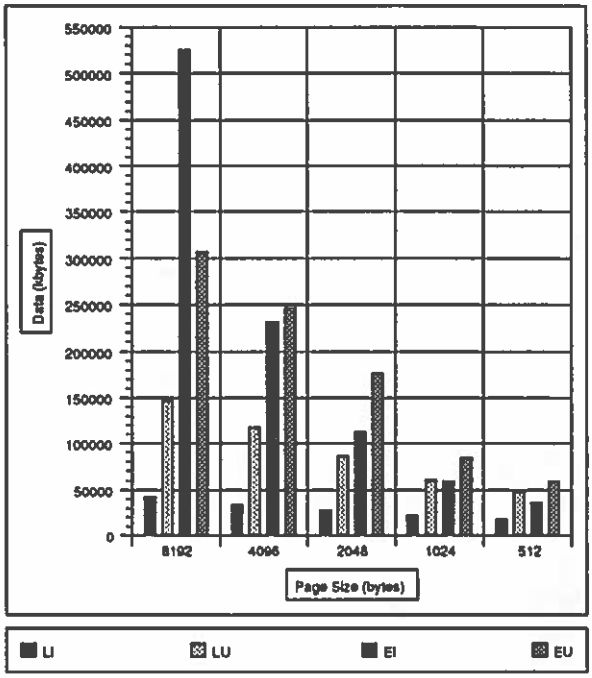
5.1 Reducing the Size of the Unit of Coherence

Several DSM systems address the memory latency problem by supporting two levels of granularity: sharing is supported at a large page level while coherence is supported at a smaller level of granularity called a "block." This approach reduces both the number and size of the messages transferred by the coherence algorithm and, in addition, reduces the occurrence of excessive data movement due to false sharing. The effect of page size on performance is clearly illustrated in experiments which show the decrease in message-passing overhead as page size decreases. This phenomenon occurs independently of the coherence protocol for many, but not all, parallel applications (see Figure 10).

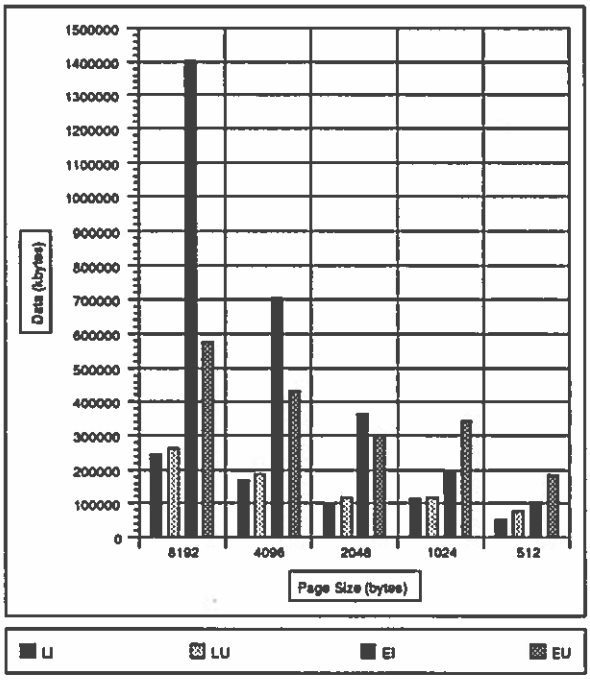
A key issue in the design of these systems is the choice of directory data structures for space-efficient and time-efficient mapping of both pages and blocks. As an illustrative examples, we describe the hierarchical directory structure proposed by Tamir and Janakiram which uses a write-invalidate coherence protocol. We also briefly discuss several systems that support two levels of granularity: the PLUS system of Bisiani and Forin which uses a write-update coherence protocol, Mether, which supports small and large pages, and Mirage which uses arbitrary sized segments for sharing and the page for coherence.



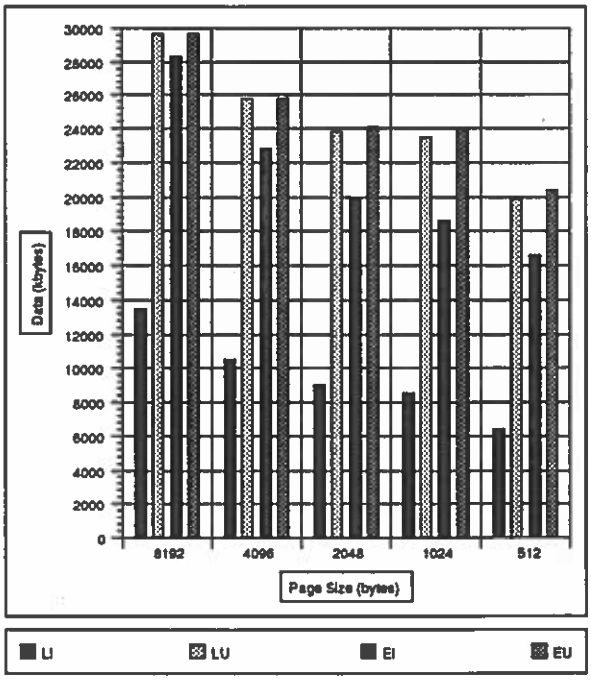
(a) Locus



(b) Cholesky



(c) MP3D



(d) Water

Figure 10: Data Transmitted versus Page Size [KCZ92]

5.1.1 Hierarchichal Directory Structure

Tamir and Janakiram propose a two-level directory structure that can be integrated into the memory management system to support page level sharing and block level coherence. Their directories are carefully designed to minimize the amount of space needed to store the directory structures (less than 1%) and have been tested through simulation to achieve low data miss ratios and low network latency. In Tamir and Janakiraman's scheme, pages are distinguished as either private or shared and can dynamically change between these two states. Shared pages are further divided into smaller blocks and managed using the two-level directory scheme described below.

- Present Pages Table (PPT). The PPT is an inverted page map table that keeps track of the virtual page contained in each physical frame of local memory. For each frame, the PPT contains the virtual page number, a bit indicating whether the page is private or shared, and (if shared) a pointer to the second level table entry. A hash table provides a fast associative lookup in the PPT.

- Shared Page Table (SPT). The SPT keeps track of shared pages. Each entry contains fields indicating the owner of the page, the access rights of each *block* in the page, and a pointer to the copy directory for the page (if owned by this node).

- Copy Directory. The copy directory keeps track of the owners of each block belonging to this page. If a given block is non-exclusively owned by this node, the copy directory entry also lists all nodes that have read-only copies of that block.

The role of these directories in the memory mapping process is illustrated in Figure 11.

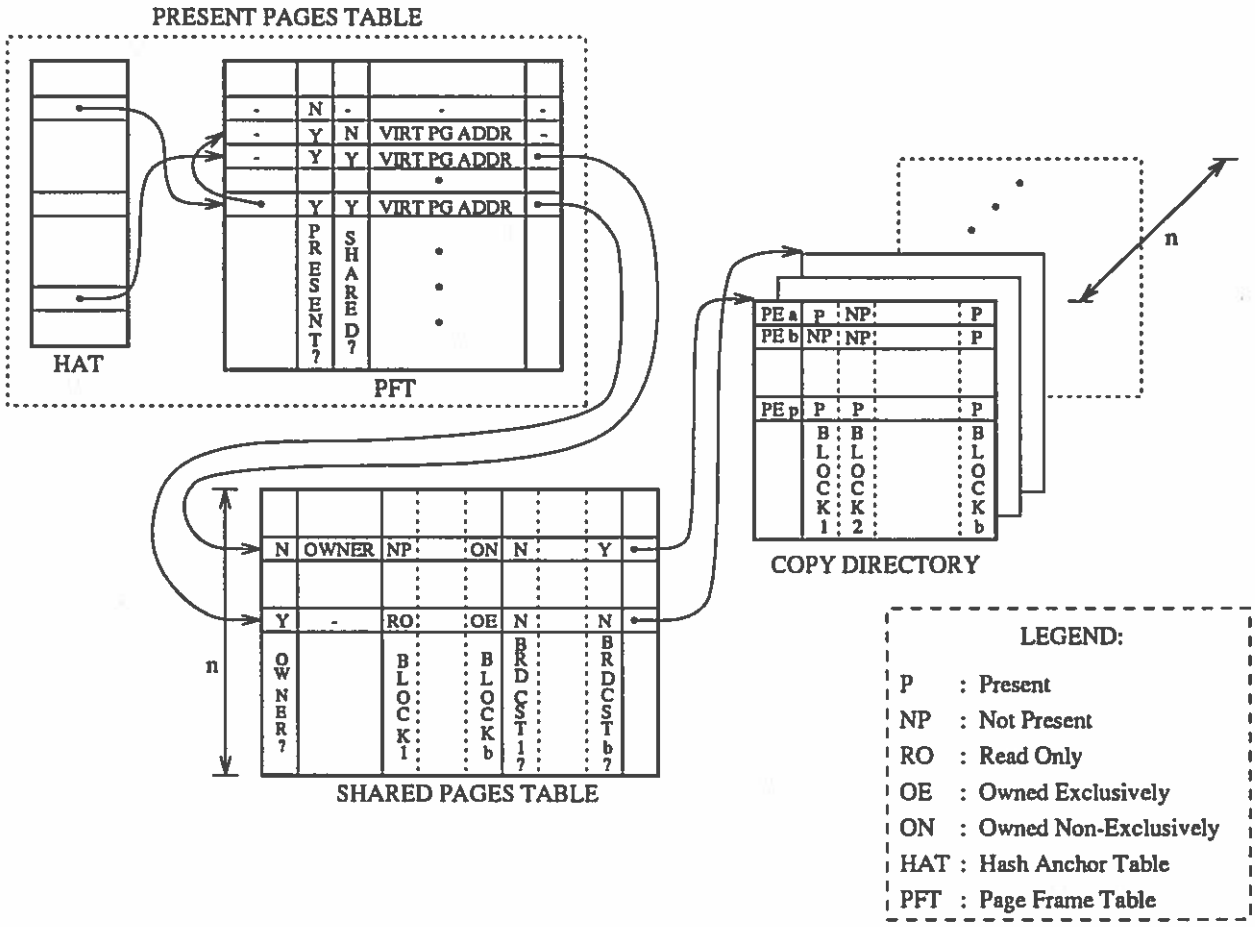


Figure 11: Two-level Directory Scheme [TJ92]

The effectiveness of this directory scheme with respect to space efficiency is based on the observation that most parallel computations share only a small percentage of their address space (less than 15%). This fact ensures that the number of SPT entries remains small. Simulation tests indicate that an SPT that is about 10% the size of the PPT will be sufficient. Since the PPT is an inverted page map table, its size is proportional to the size of the system's physical address space rather than that of the virtual address space. The size of the copy directory is dependent on the number of shared blocks and on the number of nodes sharing a given block. The former number is limited by the same factor that affects the size of the SPT - the 15% cap on percentage of address space being shared at any time. In order to limit the size of the copy directory when one block is shared by many nodes, a fixed limit is placed on the number of read-only copies of a block that are allowed. When a new read request is received that would overflow the copy directory, an *eviction* policy selects a victim from the existing directory and replaces it with the new owner. Experiments indicate that about eight copy directory entries are sufficient to support reasonable performance.

Tamar and Janakiraman have also carefully tuned the design of their scheme to reduce latency. The choice of block size is shown through simulation to be the key factor affecting performance. Both miss ratios and network traffic overhead were shown to rise dramatically as the block size was varied from 128 bytes to 4096 bytes. The effect of SPT size and copy directory size were shown to be important but less critical. The miss ratio and the network overhead reached acceptable levels given a minimum number of around 32 entries in both cases. Additional sources of efficiency include careful design of the copy directory to reduce search time during execution of the write invalidate protocol, and merging of invalidate messages for blocks

contained in the same page when an SPT entry must be released during an eviction.

5.1.2 PLUS and Mether

PLUS [BR90] is a software/hardware implementation of DSM that supports granularity at both the page level and the block level. The unit of replication is a 4K byte page, while the unit of memory access and coherence is a 32 bit word. PLUS uses a write-update protocol in which writes are propagated to all copies of the data. A distributed data structure similar to Galactica Net's virtual sharing rings is used to propagate updates to pages. This data structure consists of a master copy of the page and copies of the page linked to the master through a copy-list. The page map tables in PLUS are only partially replicated across the processors. If a local page map table does not contain the needed entry, an exception handler consults a centralized table and then updates the local tables. PLUS supports weak consistency through the use of a specialized *fence* synchronization operation. In addition, PLUS provides a number of specialized hardware instructions which affect all copies of a page needing to be updated. These instructions are called *delayed operations* because they allow an operation that must propagate through the copy-list to be overlapped with regular computation, thereby significantly reducing the latency. PLUS is currently implemented with Motorola 88000 processors and an interconnection network using the Caltech mesh router.

The Mether DSM systems, described earlier in Section 3.2 supports two page sizes: a 32 byte small page and an 8K byte large page. Columns (a) and (b) of Table 4 show the improved

performance realized through use of the smaller page. Recall that under Mether, user-level protocols are responsible for making the decision about whether to access small rather than large pages.

5.2 Page Locking to Avoid Thrashing

Mirage+ and its predecessor, Mirage, use a technique they call *time-based coherence* to avoid thrashing. Their coherence protocol, which supports sequential consistency, uses a clock mechanism to allow a node to hold on to a page regardless of the demand for that page from other nodes. A global system parameter Δ determines how long a node may hold onto a page without relinquishing it. During that time window, the node may read or write the page or may not access the page at all. This technique is designed to reduce thrashing, decrease the number of network invalidations, and minimize the amount of message passing needed to provide coherence. The idea for page locking was first proposed by Li in [Li86] and implemented in both Mirage and DSVM6K [BB93].

Implementation of time-based coherence in Mirage and Mirage+ is integrated into the coherence protocol which uses replicated-read and write-invalidate. A distinguished node called the library node serves as a central manager for all the pages contained in a given memory segment. All read and write requests for pages in that segment are sent to the library node where they are processed serially. The clock site for a given page is another special node, that holds the most recent copy of a page. If the page is writable, there can only be one copy so the clock node is the holder of that copy. If there are multiple read copies of the page, one of the read nodes is selected as the clock site. The library node handles read requests by sending a copy of the page

to the requester. For write requests, the library node sends an invalidation message to the clock site which checks to see if the time window Δ has elapsed for that page. If not, it informs the library site about the amount of time remaining, and the library site then waits and reissues the invalidate message. When the clock site accepts the invalidate message, it sends invalidations to all the other copies and sends the page to the requester.

For both Mirage and Mirage+, Δ is measured in wall clock time. The decision to use wall clock time led to a problem in Mirage in which the time window Δ could expire before the process was actually scheduled. This problem was corrected in Mirage+ by allowing a process to retain a page until either the time window Δ elapses or until the process actually accesses the page, whichever comes later. The *timesleep()* kernel call is used to delay processes waiting for Δ to expire. In both systems, the granularity of the *timesleep()* call was greater than the interval of the timer interrupt. As a result, the time window Δ was set to be a multiple of the frequency of the system interrupt clock. For Mirage, this was a multiple of 16.66 msec; for Mirage+, Δ is a multiple of 20 msec. Under DSVM6K, the minimum value for Δ is 500 μsec, and the user can increase the page residency time up to 20 msec. in increments of 500 μsec.

Determination of the best value of Δ depends on many factors including the locality and page utilization of the application and the system scheduling quantum. Experiments with the Mirage system show that increasing the value of Δ for an application that is thrashing improves overall system performance although it decreases the performance of the application. In addition, it was found that allowing processes to relinquish the CPU using a *yield* kernel call resulted in 50% improvement in throughput for a fixed value of Δ. Finally, limited experiments with applications that exhibit high locality show that the relationship of throughput to Δ is characterized by a

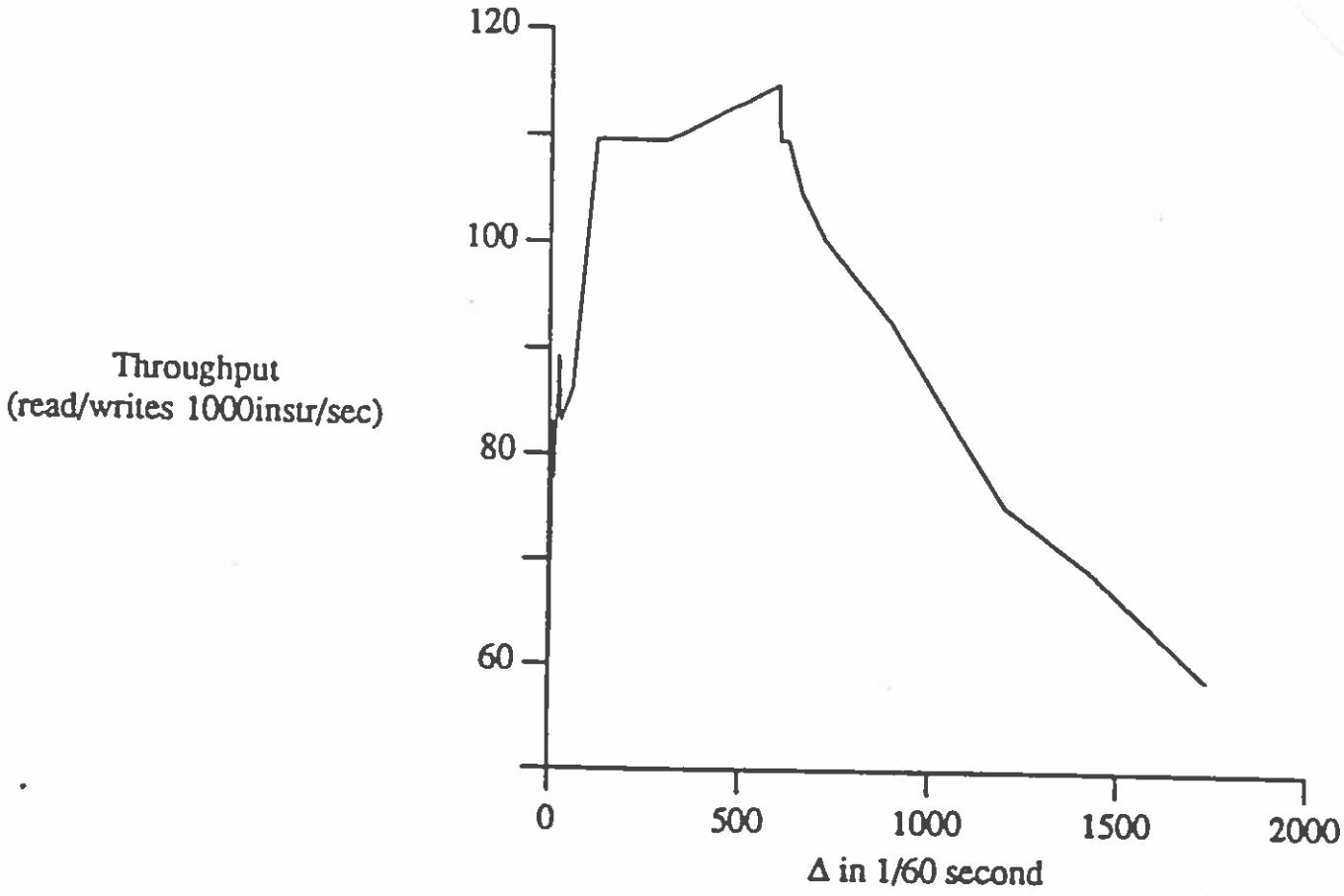


Figure 12: Mirage Page Locking [FP89]

threshold effect. For values of Δ below the threshold value, application throughput declines rapidly, while for values of Δ above the threshold value, throughput improves rapidly (see Figure 12.) This relationship reinforces the intuition that the locality of the application requires that the page be retained for a reasonable window of time at a given node.

In addition to providing page locking through the time window mechanism, Mirage+ also supports user level locking of pages. This is achieved using a *lock page* kernel call which causes the library site to put the page in a *locked-down queue* until it receives a *unlock request* from the

user.

5.3 Multiple-writers Protocol

As described earlier, the Munin and Memo systems support a multiple-writers protocol in which more than one processor can write to different parts of the same page without generating message traffic to move the page among the processors. This is accomplished as follows: at the beginning of certain execution *intervals* a shared page is write-protected. When the first write protection violation occurs, the DSM software saves a twin of the original page and removes the write protection from the user copy. At the end of the interval, the twin and the modified user copy are compared and the changes are detected using a *diff*. Under ERC, only the changes are transmitted at the time of a release. Under LRC, only those updates causally related (by the "precedes" relationship for a given synchronization variable) are transmitted. In the latter case, if multiple writers are involved in false sharing, it is unlikely that the updates are causally related, and the updates may not need to be communicated at all.

6 Conclusions

It is clear from the research reported above that large strides have been made in reducing memory latency for distributed shared memory systems in the short period from the late 1980's to the present. We have categorized the sources of latency reduction as

- latency stemming from the choice of coherence semantics and the design of the coherence algorithms,

- latency attributed to communications overhead generated by the network software and hardware,
- latency due to unnecessary message-passing overhead due to large page sizes and false sharing.

These results should be viewed as preliminary because they are based primarily on simulations and because they are based on experiments with a limited number of parallel applications. In addition, it is important to remember that the effect and magnitude of all performance improvements is highly dependent on characteristics of the parallel application. By carefully studying the data presented in this paper and the original research papers, it can be seen that some optimizations are more effective on one application but not another. Application characteristics that affect performance include granularity, communication patterns, and inherent parallelism.

Below is a sampling of the magnitude of some of the performance enhancements reported in this paper. These should not be interpreted as general principles nor as conclusive evidence, but as examples of the types of gains that are realizable in current DSM systems.

- **Relaxed coherence requirements:** Reductions in number of messages for LRC over ERC of 9:1 were reported in simulations at Rice University [DKCZ93]; and reductions in number of messages for entry consistency over release consistency of 84:1 were observed in empirical experiments with Midway [BZS93]. Reductions in completion time of release consistency over sequential consistency as great as 3:1 were reported in simulations performed for the Galactica Net system [WL92]. It is important to note that the huge reductions in numbers of messages does not yield correspondingly large computation speedups. For

example, the 9:1 message reduction ratio mentioned above yielded only about a 1:3 improvement in speedup. This is due to the fact that under weaker consistency models, the percentage of a computation's execution time that is involved in consistency operations is not very large.

Conclusive evidence from numerous sources, based on simulation and empirical experiments, and for both software and hardware implementations of DSM show that it is unnecessary to support sequential consistency in all situations. However, support for weaker consistency models must be automated to the greatest extent possible in order to avoid burdening the programmer and to avoid programmer errors. We note that there are very few compilers in existence to date that provide the needed support.

- **Smaller page sizes:** DSM system support for smaller granularity also yields significant performance gains at the expense of greater overhead to keep track of multiple units. Reduction in message volume of 16:1 under ERC for Munin were achieved for certain applications [KCZ92]. Reduction in run time of 2:1 was reported when a 32 byte pages was supported along with the large 8192 byte page, under sequential consistency for Mether [MF90]. It is unclear at this time whether application independent approaches to granularity (e.g. Mether and Plus) are more or less effective than object-based sharing (e.g., Clouds, Choices, Midway) and segment-level sharing (e.g. Mirage).

- **Communications software and hardware:** The use of advanced communications software and hardware has also been a source of improved performance for DSM systems. It is generally accepted that ethernet speeds are insufficient to support DSM for a wide

variety of parallel applications. For example, Munin simulations report 5:1 improvements in speedup by simply switching to ATM technology over ethernet at the same bandwidth [DKCZ93]. However, improvements due to ATM hardware technology are shown to level off quickly, and experimental evidence points to the greater significance of software communication overhead. Successful implementations of DSM for commercial and experimental message-passing multicomputers based on high speed interconnection networks extend the use of the distributed shared memory paradigm to a new arena.

- **Other sources of performance enhancements:** Further reductions in message-passing overhead through data compression, data reduction (such as in the multiple-writers protocol), and fine-tuning of coherence protocols are also important methods for reducing memory latency. Large reductions in memory latency are achievable when comparing simple centralized coherence protocols with refined distributed protocols such as those reported in [LH89] (see Figure 5). DSM efforts to prevent false sharing through mechanisms such as page-locking are most crucial for SC implementations. Relaxed consistency, object oriented approaches, and smaller granularities may make false sharing less of a problem However, thrashing will continue to be a problem particularly under systems using co-scheduling since all the processes that share memory may be active simultaneously.

Future areas for enhancement of operating system DSM systems include careful data placement in which shared data is located with the processors likely to require heavy access to that data; refinement of scheduling policies in two ways: design changes to meet the needs of distributed shared memory systems, and runtime tailoring of the scheduling of processes to be sensitive to the access patterns of shared data. Finally, the use of both adaptive and application-

specific routing algorithms may yield improvements due to reduced message contention.

7 Acknowledgments

This work was supported in part by NSF grants CCR-8808532, MIP-9108528, and the NSF Research Experiences for Undergraduates Program. We appreciate the comments from the researchers who provided useful comments on the draft and who verified information about their systems. Thanks also to Kurt Windisch who helped in the preparation of the figures.

References

[AH90] S. Adve and M. Hill. Weak Ordering: A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer ARchitecture*, pages 2–14, 1990.

[AMM90] R. Ananthanarayanan, S. Menon, and A. Mohindra. Experiences in Integrating Distributed Shared Memory with Virtual Memory Management. Technical report, College of Computing, Georgia Institute of Technology, 1990.

[BB93] M. L. Blount and M. Butrico. DSVM6K: Distributed Virtual Shared Memory on the Risc System/6000. In *Proceedings of the 1993 IEEE Spring COMPCON*, pages 491–499, 1993.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[BR90] R. Bisiani and M. Ravishankar. PLUS: a distributed shared-memory system. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 115–124, May 1990.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tannenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), September 1989.

[BZ91] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical report, Carnegie-Mellon University, September 1991.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 1993 IEEE Spring COMPCON*, pages 528–537, 1993.

[CAL+89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceeding of the ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[Car93] J.B. Carter. *Efficient Distributed Shared Memory Using Multi-Protocol Release Consistency.* PhD thesis, Rice University, April 1993.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings 13th Symposium on Operating Systems Principles*, pages 152–164, 1991.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[DKCZ93] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *20th Annual International Symposium on Computer Architecture*, pages 144–155, 1993.

[DS90] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceedings 13th International Symposium on Computer Architecture*, pages 434–442, 1986.

[FBYR88] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, Implementation, and Performance Evaluation of Distributed Shared Memory Server for Mach. Technical report, Carnegie-Mellon University, 1988.

[FHJ93] B. D. Fleisch, R. L. Hyde, and N. C. Juul. Moving Distributed Shared Memory to the Personal Computer: The Mirage Experience. Technical report, College of Engineering, University of California Riverside, 1993.

[FP89] B. D. Fleisch and G. J. Popek. Mirage: a coherent distributed shared memory design. In *Proceeding of the ACM Symposium on Operating System Principles*, pages 211–223, December 1989.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.

[GL91] D. Gannon and J. K. Lee. Object Oriented Parallelism: pC++ Ideas and Experiments. In *Proceedings 1991 Japan Society for Parallel Processing*, pages 13–23, 1991.

[Goo89] J. R. Goodman. Cache consistency and sequential consistency. Technical Report Technical Report no. 61, IEEE Computer Society, SCI Committee, March 1989.

[ILP93] L. Iftode, K. Li, and K. Petersen. Memory Servers for Multicomputers. In *Proceedings of the 1993 IEEE Spring COMPCON*, pages 538–547, February 1993.

[JC89] G. M. Johnston and R. H. Campbell. An object-oriented implementation of distributed virtual memory. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 39–57, October 1989.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KDCZ93] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Memo: Distributed shared memory on standard workstations and operating systems. Technical Report RICE COMP TR93-206, Rice University, June 1993.

[LA92] H. V. Leong and D. Agrawal. Type-specific Coherence Protocols for Distributed Shared Memory. In *Proceedings 12th International Conference on Distributed Computing Systems*, pages 434–440, 1992.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[LG91] J. K. Lee and D. Gannon. Object Oriented Parallel Programming. In *Proceedings Supercomputing '91*, pages 273–282, 1991.

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

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, 1986.

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

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

[LS89] K. Li and R. Schaefer. Shared virtual memory for a hypercube multiprocessor. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, pages 371–378. Golden Gate Enterprises, March 1989.

[MF90] R. G. Minnich and D. J. Farber. Reducting Host Load, Network Load, and Latency in a Distributed Shared Memory. In *Proceedings International Conference on Distributed Computing Systems*, pages 468–475, 1990.

[MP93] R. G. Minnich and D. V. Pryor. Mether: Supporting the Shared Memory Model on Computing Clusters. In *Proceedings of the 1993 IEEE Spring COMPCON*, pages 558–567, 1993.

[NL91] B. Nitzberg and V. M. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[PL93] K. Petersen and K. Li. Cache coherence for shared memory multiprocessors based on virtual memory support. In *Proceedings 7th International Parallel Processing Symposium*, pages 49–55, 1993.

[RAK89] U. Ramachandran, M. Ahamad, and M.Y.A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *International Conference on Parallel Processing*, pages 160–169, August 1989.

[RAYK89] U. Ramachandran, M. Ahamad, M. Yousef, and A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *1989 International Conference on Parallel Processing*, volume II, pages 160–169, 1989.

[RYK91] U. Ramachandran, M. Yousef, and A. Khalidi. An implementation of distributed shared memory. *Software - Practice and Experience*, 21(5):443–464, May 1991.

[Sch89] C. E. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors.* PhD thesis, University of Southern California, May 1989. Tech Report No. CENG 89-19.

[SZ90] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990.

[TJ92] Y. Tamir and G. Janakiraman. Hierarchical Coherency Management for Shared Virtual Memory Multicomputers. *Journal of Parallel and Distributed Computing*, 15:408–419, 1992.

[TKB92] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.

[WL92] A. W. Wilson and R. P. LaRowe. Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture. *Journal of Parallel and Distributed Computing*, 15:351–367, 1992.

List of Figures

List of Tables