Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

MIRAGE: A COHERENT DISTRIBUTED SHARED MEMORY DESIGN

Brett D. Fleisch
Gerald J. Popek

April 1989
CSD-890020

Mirage: A Coherent Distributed Shared Memory Design†

Brett D. Fleisch
Gerald J. Popek

University of California, Los Angeles

Abstract

Shared memory is an effective and efficient paradigm for interprocess communication. We are concerned with software that makes use of shared memory in a single site system and its extension to a multimachine environment. Here we describe the design of a distributed shared memory (DSM) system called Mirage developed at UCLA. Mirage provides a form of network transparency to make network boundaries invisible for shared memory and is upward compatible with an existing interface to shared memory. We present the rationale behind our design decisions and important details of the implementation. Mirage's basic performance is examined by component timings, a worst case application, and a "representative" application. Our experience is that the tuning parameter in our design can improve application throughput for some instances of page contention. In other cases, we find the effect of thrashing on overall system performance can be ameliorated using our tuning parameter.

1.0 Background

In this work we describe a protocol for a distributed shared memory (DSM) system and discuss the implementation of a prototype called Mirage. We have chosen to do a kernel implementation rather than an analytical or simulation model or library layer, for a variety of reasons. Some of these include the difficulty in otherwise accurately capturing: 1) the asynchronous behavior of computer systems, 2) the interactions between scheduling, interrupt processing, and the user program requests and 3) the host behavior and network loading from the use of the protocols, and the interaction of all these with applications. Although there has been some work in these areas, an actual implementation can provide a more substantial statement of the viability of DSM.

Developing a kernel prototype has its own difficulties. One of the most pertinent is in testing the prototype using realistic applications that exercise the system's functionality. It is also difficult to compare it fairly with previous mechanisms because of the many differences. Toy programs are often used to demonstrate the effectiveness of a system. However, this technique calls into question how effective these programs are at benchmarking the functionality in a realistic way. Standardized single machine benchmark programs may not be appropriate and we have no standard benchmarks with which to exercise shared memory.

To improve our prospects of obtaining useful applications for performance testing we built a distributed shared memory system that is upward compatible with the UNIX System V IPC defined interface[ATT86]. A comparison between single site and multisite performance of applications used in the research and industrial community should indicate how effective our proposed distributed shared memory model is in practice.

† This research was sponsored by DARPA Contract No. F29601-87-C-0072

2.0 Communications Approaches

A wide variety of approaches to handling the communication requirements of existing systems have emerged over the years. One of the most successful and widely used mechanisms is *message passing*. In message passing, data is exchanged by placing data in messages and transmitting the message from sender to receiver; synchronization is accomplished because the sender must have sent a message before the receiver can receive it. The approach is an elegant one, and perhaps because of the similarity between network packets and messages, has evolved as a powerful structural tool in the design and implementation of operating systems over the past three decades.

The message passing approach may not be as well suited for multiprocessors and tightly coupled processors that have access to shared memory. Using a message passing approach in a shared memory multiprocessor requires the programmer to use conceptually different primitives and organize the programmed code in a different way than in a shared memory system. Often message passing systems impose a significant performance penalty in data copying when passing a message locally. Further, passing complex data structures may be more convenient using shared memory than using message passing.

Message passing implementors have begun to explore memory management support for communication (Accent[RASH81], Mach[ACCE86]). Memory management is used to pass messages locally using sharing techniques (copy-on-write). However, at the other extreme, the exploration of memory management or shared memory in terms of network communications primitives is relatively new. Our work focuses on the latter case.

2.1 Past Work

In Appendix I we examine past work related to our distributed shared memory research. We begin by looking at the work of Kai Li[LI86]. We then examine the Agora work[BISI88, FORI87] and then a hardware approach to distributed shared memory presented by [RAMA88]. We conclude by reviewing Mach[ACCE86]. Readers interested in past work are directed to Appendix I.

2.2 The System V Interface Model

In this section we describe the System V Model of shared memory. Its selection as the basis of a DSM prototype is independent of the underlying protocol. In fact, one could develop other protocols for DSM and adopt the System V interface; conversely, one could use other interfaces for our protocol.

In Mirage, as in System V, processes access shared memory through the use of a *segment*. A segment stores shared memory data. Segments are not meant to store program text nor system state except as raw data. A process creates a shared segment by defining a segment's size, name, and access protection. Segment access protection works similarly to UNIX file access protection, but is limited to read and write permissions. The name provides a mechanism by which other processes can locate the segment. In particular, processes *attach* the segment into their virtual memory address space by name. The attaching process can choose the exact virtual address range. Alternately, the process may elect to place the segment at a first-fit location in the address space. Unlike other sharing models, processes can share locations at different virtual address ranges. Once attached, the shared memory can be used like any normal

locations of memory, the only difference being that changes to the underlying memory are also visible to the other processes that share the segment. When a process is finished with the segment it may be *detached*. The last detach of a segment destroys it.

3.0 Goals

The goals of this work are: 1) to present a protocol for distributed shared memory, 2) to describe the implementation of Mirage, 3) to measure the performance of Mirage applications. To accomplish 3) we examine component timings and an important worst case application. Further, we attempt to characterize average case performance with a "representative" case. This latter case provides an evaluation of our tuning parameter.

The following are features of the current implementation:

Transparent Access

The standard UNIX interface is preserved. Access to local memory uses the same interface as access to remote memory. The interface is used to create, locate, and destroy shared memory segments. Reads and writes to shared memory function the same in the local case as the remote case.

Preserved Semantics

The UNIX System V IPC semantics are preserved. Access to memory is made through the same system V IPC calls and uses the same architectural interface.

Binary Compatibility

Applications written for the System V IPC interface should not need to be recompiled. This is a consequence of System V interface compatibility.

Performance

Our goal is to minimize the overhead in distributed shared memory access and show effective performance of the protocols employed.

4.0 Environment

Our prototyping environment consists of 3 VAX 11/750s networked together using a 10 megabit Ethernet[METC76]. The VAXs run an early version of the Locus operating system[POPE81, WALK83] compatible with the UNIX System V interface specifications. We briefly highlight Locus and describe how that system led to the development of our model and the prototype.

4.1 Locus

The Locus operating system is a distributed version of UNIX that provides a superset of UNIX services. Support for the underlying network is almost entirely invisible to users and application programs. The system supports a high degree of *network transparency:* that is, it makes the network of machines appear to users and programs as a single computer, completely hiding machine boundaries during normal operation.

Locus provides a fully transparent file system and facilities for distributed processes. In a Locus network, which may consist of machines of various CPU types, both files and programs may be moved without effect on correct operation. Local and remote operations appear the same in Locus. Central to the design of the Locus architecture is the underlying distributed file system. The file system supports a number of high-reliability facilities, including a more robust facility than that of conventional UNIX systems, and support for interprocess communication. Process creation and migration are fully supported.

The Locus system has been operational for over seven years. Early on, the need for extending IPC mechanisms became apparent; work commenced in the summer of 1984 towards this goal. We began by upgrading the System V IPC model. By 1986, results were published describing new distributed interprocess communication facilities based on System V messages and semaphores that had been added to Locus[FLEI86]. Work on a distributed shared memory system began in late 1987. About 8 months later the design and implementation was complete; roughly 4 months were spent exploring the design and 4 months implementing. Subsequently, Mirage's performance was examined.

5.0 Coherence

At the outset of the design we decided that it would be unacceptable for processes to read stale data. Two approaches to memory consistency control were examined: coherence and user-level synchronization. In the next section problems with the latter approach are described. Like Li[LI86] we have chosen to maintain *coherence* at the lowest system level. A coherent implementation is one in which a write to an address in a given segment is always visible by all subsequent read operations to the same address, independent of the machine location on which the read takes place. Further, all writes to an address always preserve the latest value written. As in a single machine, it is the responsibility of user-level processes to synchronize writes from different processes.

To implement coherence, fixed size pages of the segment (the same pages as the hardware memory management) are used as the basis for all intersite consistency. Although only one site in a network will have a valid writable copy of a given page at any instant, there may be many sites simultaneously possessing readable copies of the page. In general, a given page will have either one site acting as writer or multiple sites acting as readers.

There are many situations in which an implementation of shared memory coherence could perform poorly. A worst case scenario might be an alternating sequences of reads and writes to the same page issued by different sites. This situation requires that the page be written and then transferred to the reader for the last write to be visible. The page would be passed back and forth between sites in alternation. Our implementation attempts to handle such pathological cases of thrashing with tunable controls explained in Section 6.1.

5.1 Motivation for Coherence: Synchronization vs. Coherence

An alternate approach to consistency control requires that user programs provide all synchronization. For example, System V IPC provides semaphores that programmers could select to assure an up-to-date view of shared memory.

Consider the case of two different tasks, each with a critical section that mediates access to two shared data regions. Suppose these data regions have different virtual addresses *on the same page*. Assume that the programmer uses semaphores as the synchronization mechanism, that the two critical sections execute at different network sites, and that both critical sections use different semaphore variables. Strictly speaking, we should be able to interleave these critical sections, because they have different semaphore variables and access different shared data regions(see Figure 1). But, since data for the two critical sections is located on the same page, there is a consistency problem. The system must maintain the correctness of data on the page.

There are a number of approaches to maintaining page consistency. Serialization of the critical sections is not considered attractive in a distributed environment since the critical sections could execute at any site in the network and distributed serial scheduling may present difficulties. Also, the method inherently inhibits parallelism. Locking might be considered, but one must provide fine grain locks for noncontiguous locations as shown in Figure 1. Furthermore, it may be difficult to determine precisely which locations to lock because high-level programs generally do not make their mappings to physical locations visible. We do not find this approach attractive nor do we find any satisfactory substitute for coherence. Coherence gives a degree of flexibility and performance for user programs. User programs may employ higher level synchronization primitives as a layer on top of the low level mechanism. Applications that do not require synchronization need not be be burdened with their overhead, but will still be provided with the benefits of page consistency.

6.0 Distributed Shared Memory Protocol Terminology

We expect cooperating processes will use DSM. In Locus as UNIX, user processes are relatively heavyweight. Lightweight processes are used in the operating system to service network messages and provide efficient remote access.

Users organize shared memory data in *segments* which may be attached into the address space of a process with read-only or read-write protection. Segments attached read-only are useful in some applications when cross network sharing is required with the restriction that the data never be written. However, our discussion will, for the most part, focus on segments that have been attached read-write by at least some of the processes involved in sharing.

A segment is partitioned into a set of pages. Pages are the unit of distribution because of their fixed size and commonality with the underlying hardware. A Mirage process may read segment pages, write segment pages, or both. Processes at network sites act as readers or writers of a given page. Each process records whether or not its segment pages are present at the given site. In addition, the protection of each page is stored in accordance with the hardware architecture. In many architectures, as in ours, a page may be read-only or read-write. Colocated processes share the pages that are present using the standard System V IPC implementation mechanisms augmented for Locus.

Two Critical Sections Accessing a Shared Page

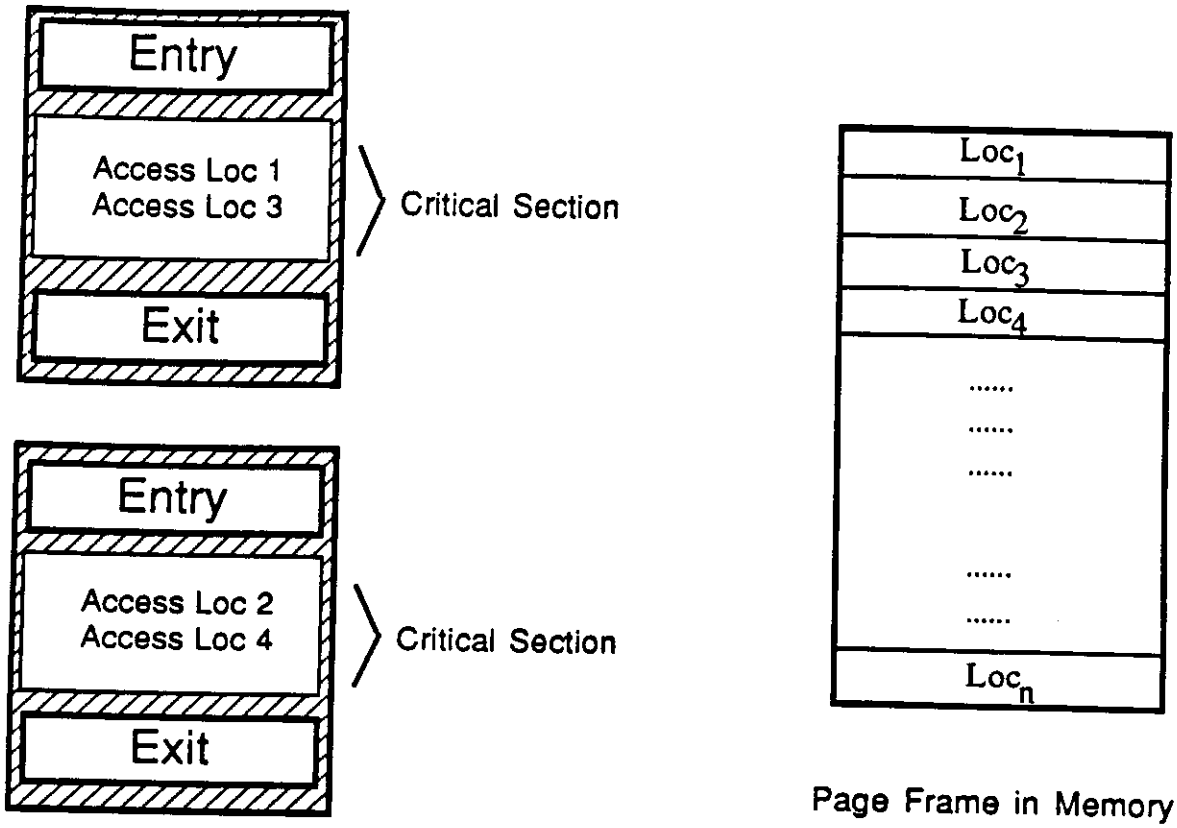


Figure 1

There is one distinguished site associated with each segment, called the *library site*. The library site is the controller for the pages of a given segment. Requests for pages are sent to the library site, queued, and sequentially processed. Depending on the configuration there may be several different sites used as library sites for the various segments created by user programs. This case is envisioned for a network of homogeneous processors of similar power. In an alternate configuration, one distinguished library site for all segments may be appropriate for a network where a fast backbone processor performs the library's queueing discipline. This processor may be a different type of CPU than the sites that manipulate the pages of the segment. In Mirage, the site that creates the segment is configured to be the library site for that segment.

The library site's primary function is to service the incoming request queue and record which sites are storing a given page. The library distinguishes writers from readers; there may only be one writable copy of a given page in the network at any one time. While there may be multiple read copies of the page in the net simultaneously, there may not be read copies at the same time as the write copy. All pages must be "checked out" through the library. To obtain a page, the requester sends a message to the library site and the requested page is returned directly from the site which is storing it.

Another distinguished site in our model is the current *clock site* for a page. The clock site is the site that has the most recent copy of a page. For example, if there is a writer for the page on the network, its site is always chosen as clock site. On the other hand, if there are a set of readers using the page simultaneously, one of the readers is selected and its site chosen as the page's clock site. We discuss the purpose of the clock site and why it is named this way in the next section.

6.1 Distributed Shared Memory Protocol Overview

Segments consist of pages that may be distributed throughout the network to sites that have had page faults and requested them. Locus interrupt handlers were modified to obtain the most recent copy of a given page from another site. When a page fault or protection fault occurs, the interrupt handler checks to see if the page is a shared memory page. If it is, the page is associated with a segment, the library located, and a network message sent to the library site queueing a request for the page. The network message indicates whether a read or write copy of the page is required.

All requests for DSM pages are queued at the library. Write requests are sequentially processed. Read requests for the same page are batched together and granted to all the readers at one time when the request is processed. During a user-level interrupt fault, the faulting process awaits the library's request processing by sleeping, the standard way UNIX tasks await the completion of an I/O operation.

A goal of this protocol is to ensure coherence. In order to maintain coherence, when a process writes to a page, all readable copies of the page must be *invalidated* before the write completes. Our invalidation unmaps and discards the page for all processes at all sites. If a writable copy was outstanding instead of multiple read copies, the page with stale data must be invalidated before the write to the new page completes. These operations are potentially expensive because of the number of sites that may be involved and the frequency with which these invalidations occur.

Our method attempts to: 1) provide fairness for each site using the page in order to control thrashing that might otherwise ocurr, 2) decrease the number of times we perform network invalidations, and 3) minimize the amount of network activity required to provide coherence. To do so, we use a clock mechanism to control when a site may be interrupted from its read/write processing to relinquish pages it is using. The clock mechanism grants the readers or the current writer a *time window* (Δ) in which they are guaranteed to uninterruptably possess the page. Much like the traditional time slice used when allocating processes to a central processor, Δ is used to apportion time for the page (or the read page set) to the site(s). During the time window, processes may read or write the page; the page may also be idle during portions of Δ. The time window provides a control that allows fairness between processes requesting page access, the current process using the page, and the library which attempts to invalidate the page on behalf of another request. In a sense, Δ provides some degree of control over the *processor locality*, the number of references to a given page a processor will make before another processor is allowed to reference that page.

To invalidate a given page, the library site sends an invalidation message to the clock site. Recall the clock site is chosen to be one of the many readers or the current writer. When the clock site gets the invalidation message, it checks to see if the page's Δ has expired. If not, the clock site replies immediately with the amount of time the library must wait until the invalidation can be honored. The library waits until Δ expires and then re-requests the page's invalidation.

When an invalidation is accepted by the clock site, typically it: 1) invalidates the local page, 2) invalidates any other outstanding readers, if the page is a read-copy and 3) distributes the page to the new writer or any new readers. Table 1 below governs the specific actions of the clock site depending on the modes of the specific request. The clock site will either be a reader or a writer. The current state is indicated in the column "Current". The column "Incoming" indicates whether the incoming request is for readable copies of the page or a writable copy of the page. The "Clock Check" column indicates whether the clock check is required. The column "Invalidation" indicates whether it is necessary to invalidate the current copy of the page.

Current	Incoming	Clock Check	Invalidation
Readers	Readers	No	No
Readers	Writer	Yes	Yes, possible upgrade if new writer is in old read set
Writer	Readers	Yes	Downgrade writer to reader
Writer	Writer	Yes	Yes

Table 1: Page Operations for Read and Write Requests

The actual protocol contains two important optimizations:

1. When a reader is upgraded to a writer, a new copy of the page is not sent; a notification acknowledges the write request.

2. When write access is removed because readers require the page, the writer retains read access.

The first optimization reduces network traffic by not sending a page copy to a reader becoming a writer. The second optimization is based on the belief that the probability of subsequent read or write access by an invalidated writer is high; the resulting reduction in network traffic by downgrading the writer compensates for the increase in network traffic to invalidate that page if a remote write were to occur. If there is no remote write during the downgrade period, and the downgraded site requires subsequent write access, an upgrade can occur. In total, two advisory messages are sent rather than first invalidating the page and then later, when the process needs to write, transmitting the complete page to the site. Figure 2 shows an example of a remote page fault.

Table 1 shows there is only one case where the clock check can be ignored. This is the case when there are read-copies outstanding and an additional read request for the page is processed by the library. To assure proper invalidations, the current clock site must be fixed when additional readers require page access. Further, the clock site must be informed of the additional reader so that the proper sites are invalidated during the next invalidation phase.

6.2 Implementation Details

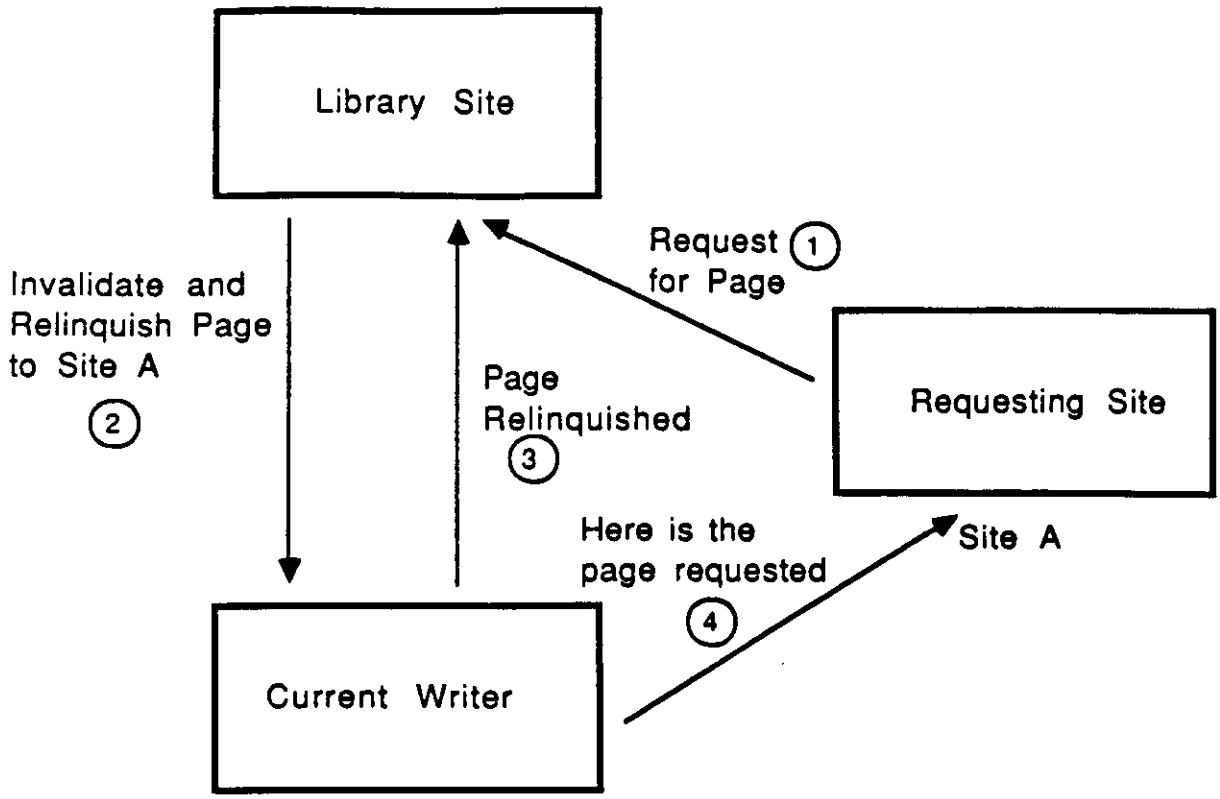
Our machine architecture deals with the segments using a *page table*. In the architecture we are using the page table is linearly structured. At a given site, for the processes sharing the same segments, and for the entries that pertain to those segment pages, each process' page table entries refer to the same resident page frames. Thus, when a process attaches a segment into its address space, a copy of a master shared segment's page table entries (PTEs) is conjoined with the current process's page table entries. The conjunction of these forms the process' virtual memory address space. Figure 3 shows two processes sharing common segments. Pages are 512 bytes in the current implementation of Mirage.

In the standard implementation of System V shared memory, segments are part of system space and are never swapped. In most machine architectures the *valid* bit indicates whether a page frame is resident or has been swapped out. For *standard* System V shared memory the PTEs are never marked *invalid*. Mirage needs to mark a page invalid to indicate that a page is not present at this network site. To do this, Mirage must locate all PTEs in *all* processes which share the page and mark them invalid - not just the current process's PTE. We discuss the design alternatives and the solution we use shortly.

Pages whose writable (or readable) copies are not present at a site are marked invalid for hardware interrupt processing to occur. We use an unused bit in the standard page table entry which indicates that an *auxiliary parallel page table* should be consulted when a page fault occurs. Table 2 shows an auxiliary parallel page table entry (auxpte). There is one shared copy of the complete table for each segment at each site. There are N entries in this table that correspond to the pages of the segment. Generally speaking, most architectures do not have sufficient space in the architecturally provided PTEs for the data stored in the auxpte.

Typed page fault detection is necessary for a reasonable implementation. The machine architecture must be able to distinguish between a read page-fault and a write page-fault. Even though some architectures can distinguish between these types of faults many operating systems never use this information in their memory management code. We have modified the interrupt service routine assembly code to examine the VAX hardware bit that indicates the fault type and have passed this data to the Locus interrupt service routine.

Remote Page Fault



If Site A requires a writeable copy, the current writer is invalidated. If Site A requires a readable copy, the current writer is downgraded to be a reader.

Figure 2

Two Processes Sharing Segments

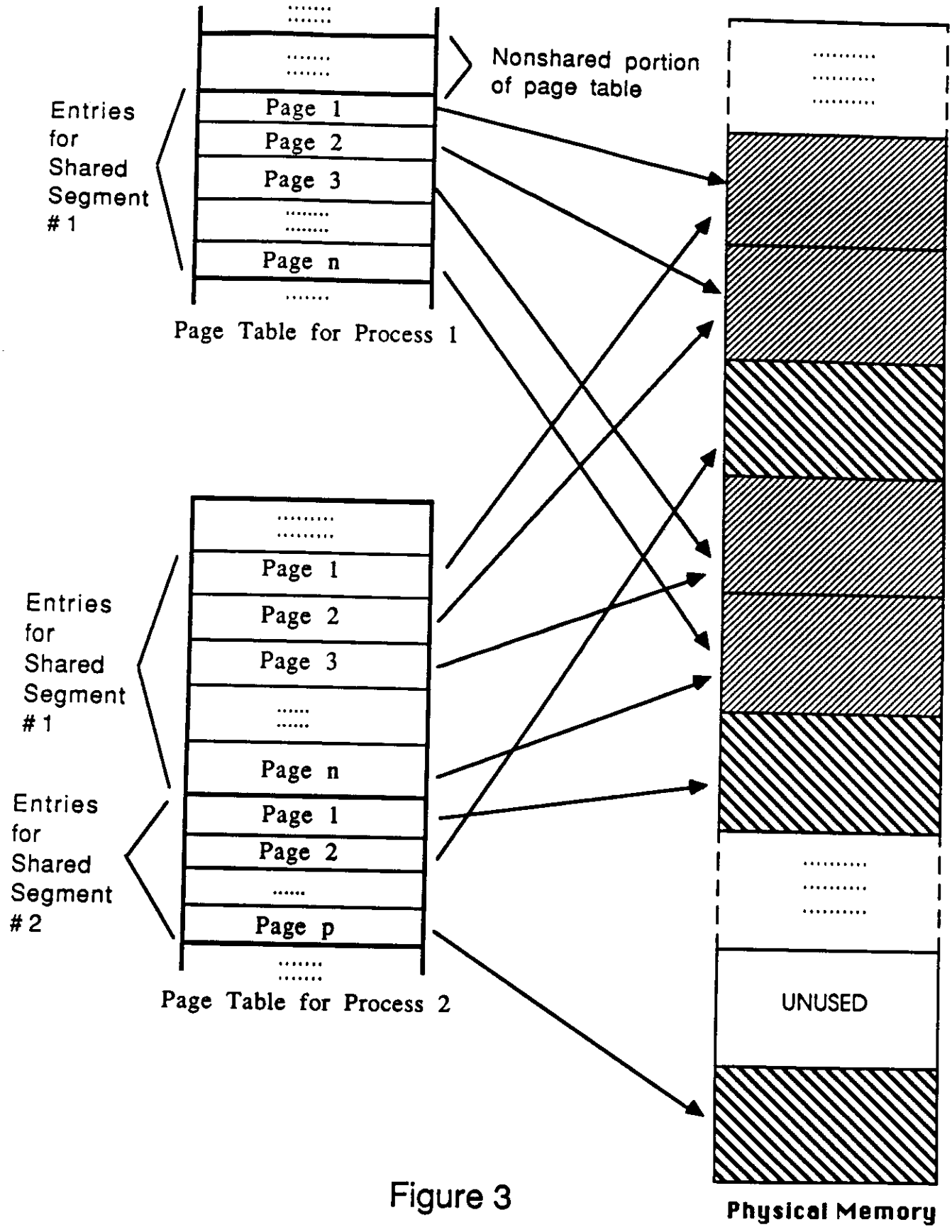


Figure 3

Contents	Comment
reader mask	list of sites using this page
writer	current writer site
window ticks	number of ticks allocated for this page
install time	installation time for this page at this site

Table 2: Contents of an auxpte entry

We encountered an implementation problem when marking a page invalid at a given site. The difficulty arises in consistency between the master version of the PTE table and each of the corresponding PTEs associated with processes that have the page mapped. Although most UNIX implementations describe each page of physical memory with a table that provides the page's state, location, and number of referring processes[BACH86], we observed that system space used no such table and so this mechanism would not assist our implementation. When an incoming network message invalidates a page, the master version of the PTE table is updated by the network server process. In addition, however, it is necessary to invalidate the page in all processes which map the page.

There are two broad categories of consistency control design alternatives: *active* methods and *lazy* methods. In an active method once there is a change in the master PTE, all processes that map the page are immediately notified and updated. The active class of methods is rejected because it would be expensive and difficult to implement in a UNIX environment. Further, because the page is mapped, there is no guarantee (especially if the process executes at low priority) the process would be scheduled before the entry or other entries change again. Even with a mechanism to postpone the invalidation until the process is scheduled (a "summary" mechanism), it would be necessary to queue requests.

Another class of methods are termed *lazy* methods. This class is the one we selected to use. Whenever a process is scheduled, we determine if it is using shared memory. If it is, before the context of the new process is resumed, the appropriate master PTE entry is copied into the new process' map. For simplicity in the prototype, we remap *all* the shared memory pages of the process using a simple for-loop rather than detecting which specific ones have changed. Every time a shared memory process is scheduled, the system must remap its pages. The cost of this mapping is not a fixed cost; it is a function of the size of the segments being mapped. The measured cost of mapping one 512 byte page ranges from 106-125 microseconds; the largest segment allowed in our intersection of memory configurations for the various VAXs is 128K. We observe that Xenix System V shared memory systems use a similar remapping strategy and that processes that do not use shared memory pay no penalty.

7.0 Performance

In order to measure the effectiveness of Mirage we examine component timings, a worst case, and a "representative" application. These component timings provide a breakdown of the cost of each operation. The worst case application is useful because it provides an analysis of what can be expected at the extreme ranges of the performance spectrum. The representative case illustrates the effect of the time window, Δ.

There are other approaches to measuring performance. Test programs could be constructed that cause many fewer faults per second and show better performance than ours. However, we question how much information we would derive from examining their performance. It is heavily-used applications that are representative of the performance users will likely encounter using DSM. We would like to obtain these applications from outside sources.

Another measurement approach (taken by Li[LI86]) is to characterize the system using a suite of synthetic test programs such as matrix multiply, dot product, traveling salesman, etc. However, these programs may not be representative of the actual everyday performance these systems exhibit. Some of the programs are data size or data input sensitive in their fault rates. For example, the size of the matrix in matrix multiplication could significantly affect the page fault rate.

Lastly, perhaps a more theoretical approach to assessing performance is to examine access characteristics of shared memory by comparing it to similar shared data spaces. For example, there are strong similarities between Mirage's shared data segments and shared text segments. Because of these similarities there may be some previous performance studies that can be applied to our work. Arnold[ARN86] has observed in shared C libraries that some parts of the library execute frequently, while other parts execute hardly at all. The C libraries are reported to exhibit random execution patterns because of the many kinds of applications that use its routines. However, libraries dedicated to a single purpose (akin to our segments), such as database or graphics service, are reported to have much different dynamic behavior than the C library. Taking this information into account, it would be hard to generalize access patterns to Mirage segments without carefully analyzing the programs which access segments and the types of segments in use. Mirage's general purpose nature makes it difficult to capitalize on these analyses.

7.1 Component Costs

We examined Mirage's performance by instrumenting the implementation to read the microsecond clock before and after kernel operations. The measured performance of a short network message (no buffer) sent round trip between two sites is 12.9 ms. This message is sent through the protocol layers and in and out of the network interface cards. When sending a network message with a 1024 byte buffer and receiving a short response message, an average of 21.5 msecs elapsed time was measured.

Table 3 depicts a breakdown of the amount of time required to obtain a checked-in page from the library site. The items marked with an (*) indicate directly measured values. The other items were extrapolated from the measured cost of receiving similar messages. For example, transmitting and receiving a 1024 byte message one-way in the prototype can be extrapolated

Operation Time	Total Time (msec)	Time(msec)
Using Site Read Request*		2.5
Read Request output transmission elapsed		3.2
Page input reception elapsed		7.5
TOTAL	13.2	
Server process time for request*		1.5
Read request input reception elapsed		3.2
Page output transmission elapsed		7.5
Processing Time*		2
TOTAL	14.2	
TOTAL ELAPSED TIME*	27.5	

Table 3: Component Breakdown: Time to obtain an in-memory page remotely
(*) indicates directly measured values

from 21.5 msecs to take roughly 15 msecs.†

There are two significant caveats in the implementation that affect performance. First, an invalidation which is not honored must be resent later. Because of the overhead in sending and receiving this (short) invalidation message, if there is less than 12.9 msecs remaining in Δ, the invalidation should be honored (or delayed and then honored) rather than requiring the requester repeat the invalidation later. So, in Mirage it may require two attempts to invalidate a page since the clock site replies with the amount of time to wait before the library should retry. Unfortunately, the current implementation does not support the queued invalidation optimization. Second, invalidations are processed sequentially rather than using a broadcast or multicast. Locus supports point-to-point communication. The Locus programmer uses network messages to communicate between sites, while the Locus system at the lowest of levels, maintains a form of virtual circuit between sites to sequence network messages and maintain topology. Although providing multicasting would be a short project in some workstation environments, the effort to integrate such a facility would be substantial in our system. Considering the few sites in our experimental network, the investment did not seem warranted.

7.2 Worst Case Description and Analysis

In order to exercise Mirage, a worst case application was constructed. This application consists of 2 processes that execute at different sites. First, an adjacent pair of memory locations on the same page is chosen. Process 1 writes a value into the first location and waits for process 2 to write its value into the next. Process 2 waits for Process 1 to write and then writes its value into the second location. We then choose another adjacent pair of memory locations and repeat these operations. Figure 4 shows a C version of the application.

† Note that to handle page requests, server processes at both requesting and servicing sites allocate a PTE, map in the data to be sent or received from the appropriate frame in system space, copy it to or from the network message, and then unmap and deallocate the PTE.

```
main()
{
    ..... /* initialization code omitted for brevity */

        /* ping pong test */
        pint = (int *) addr2; /* start of shared memory */

        for (i=0;i<NUMTRIALS;i++) {
                *pint++ = CHECKVAL;
                while ((*pint) != CHECKVAL+1) yield();
                pint++;
        }
        *pint++ = ENDVAL;

    ..... /* termination code omitted for brevity */

}
```

Code at Site 1

```
main()
{
    ..... /* initialization code omitted for brevity */

        /* ping pong test */
        pint = (int *) addr2; /* start of shared memory */

        for (;;) {
                if ((*pint) == ENDVAL) break;
                if ((*pint) == CHECKVAL) {
                        *pint++;
                        *pint++ = CHECKVAL+1;
                }
                yield();
        }


    ..... /* termination code omitted for brevity */
}
```

Code at Site 2

Figure 4

This application (or its N-site version) is a worst case application for Mirage. For each read or write to the specific locations, page faults occur which transfer the entire page between sites. The ratio between accesses to shared memory and the amount of system operation to support those accesses is very low. Notice that while spatial locality is high in the application, it executes in a configuration that causes significant system overhead. Such a configuration has poor processor locality because each processor retains the page for an exceedingly short duration. This program is an example of a worst case for a network virtual memory system and in that way is analogous to an application executing on a single site that is thrashing.

Figure 5 shows the modes of the page during the various steps of the program. Read copies are necessary for page faults that occur during program statements such as "if (*pint == CHECKVAL)." Figure 6 shows a timeline of the exact messages sent and received in the protocol. By running this program we can observe the system overhead because there are few application (user-level) operations between required system events. The program provides us with an intense exercise of the protocol at full speed of the host memory, processor, and network interface cards. This experiment factors into the measurements the effects of process scheduling and other operating system services.

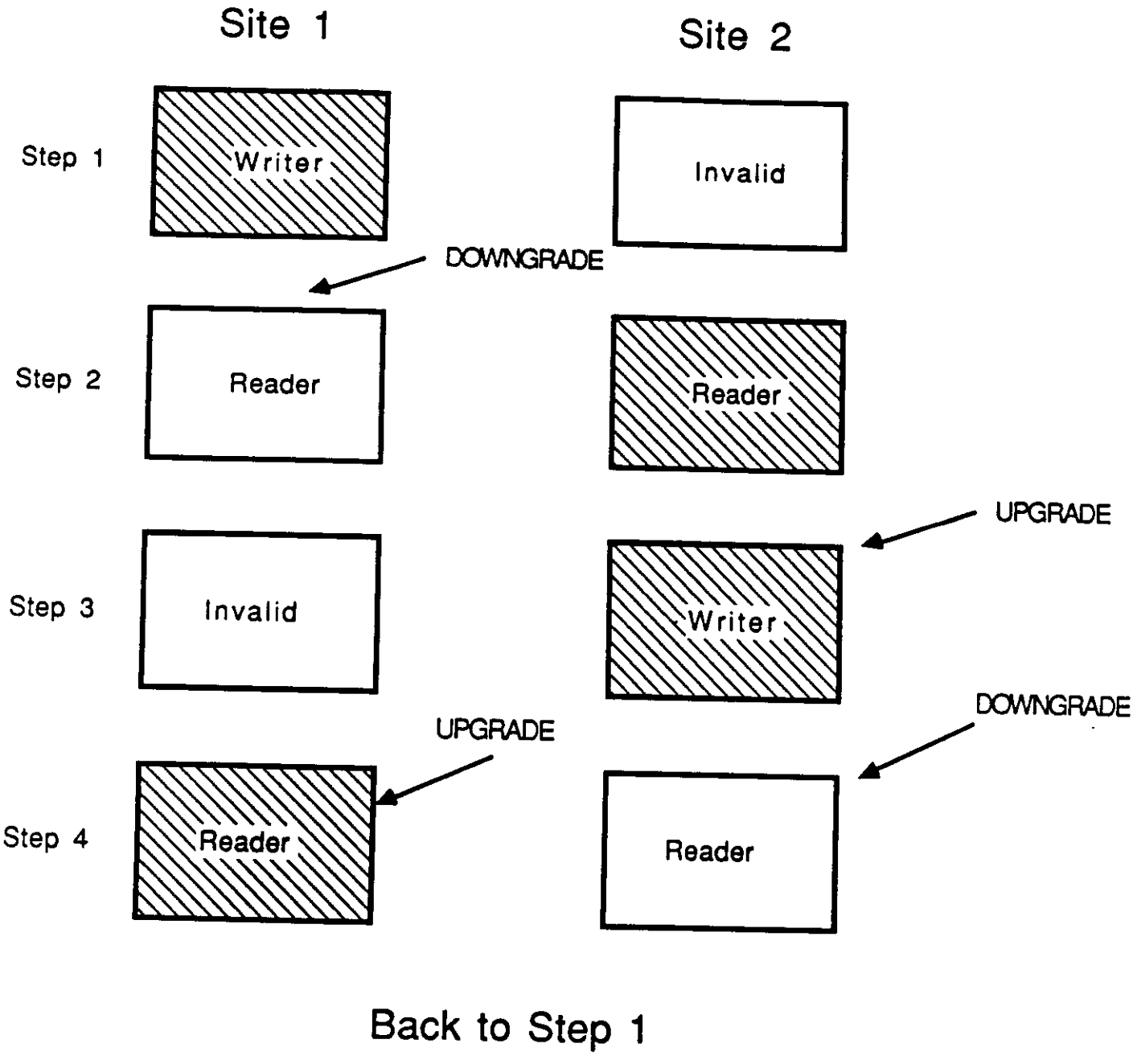
To gauge our experiments, we executed the application on a single site. However, our initial measured throughput was surprisingly only 5 cycles/second†. We reinspected the program (a version of Figure 4) for problems and observed that once a process did its write, it remained in a while-loop waiting for the other process to execute its write. The while-loop continually read a shared variable waiting for it to change. Obviously what was happening was that the entire remaining portion of the process's scheduling quantum was being wasted busy waiting. To solve the low throughput problem a new system call *yield()*. was added to our experimental version of Locus. The call was inserted in the application during all loops that inspect shared variables. We remeasured the performance of the application locally and obtained 166 cycles/second or a factor of 35 speedup.

With 2 sites, 9 messages are sent for one cycle of the application. Three of these message are large responses (1024 bytes of data); the other 6 are short messages. Based on the component timings, the raw communications component should be 84 msec, excluding interrupt processing CPU time. We add 12.5ms for the 5 interrupts that request remote pages. We add 9ms for the 6 input interrupts to install, invalidate, or upgrade the page. Additionally, two faults are generated locally and serviced by a library colocated with the requester. We add 3ms to service these two faults. Our total is 109 ms. This total corresponds to roughly 9 cycles/second in the distributed case. We can do no better than this bound unless interrupts could be serviced more rapidly or our message passing component times were improved. Further, these calculations assume that user-level processes are synchronized and ready for immediate execution after network messages change the underlying segment state. Scheduling overhead generally does not permit such user-level response.

Lastly, we experimented with the test&set instruction on the VAX. Despite the application programmer's reported preference for this instruction over other queueing interlocked instructions[AGAR88], this instruction uses busy waiting and did not perform well. In particular, we found its performance potentially low in the remote fault case. After a locking writer sets the bit to enter a critical section, the testing reader obtains the page remotely. When the locking writer completes, it faults on write to clear the lock bit and exit the critical section. If the locking writer requires use of the page for data access while the region is locked, the tester

† A cycle consists of a write by Process 1 and by Process 2 and subsequent operations performed until Process 1 is about to write again.

Two Site Worst Case Application

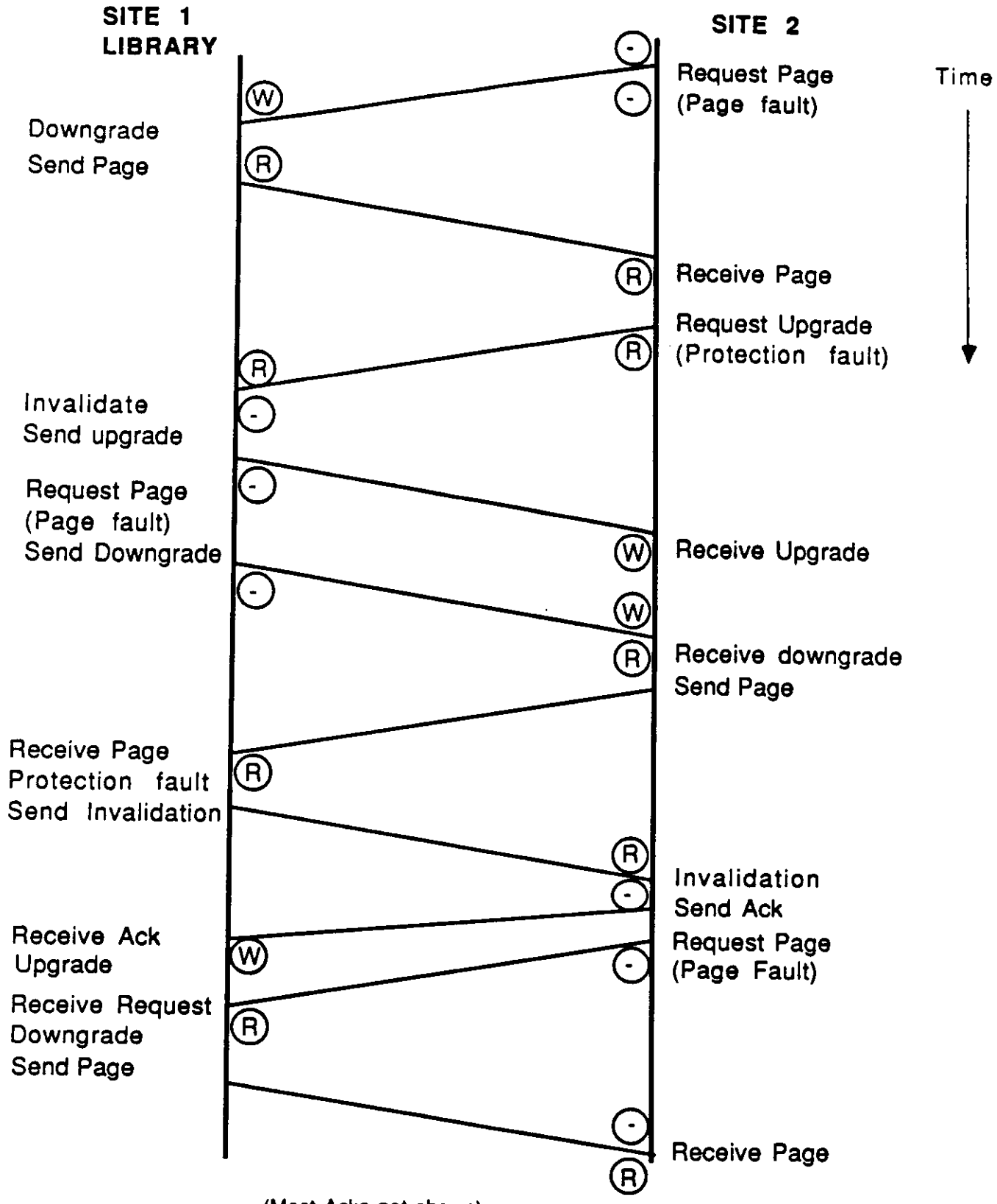


Back to Step 1

(Shaded Box indicates active site)

Figure 5

Sequence of Message Events for Worst Case Application



(Most Acks not shown)

Figure 6

and the writer thrash the page; the use of $\Delta > 0$ can be helpful to the writer in this situation. In summary, we recommend that the test&set instruction not be used because of its performance.

7.3 Worst Case Measurements

We measured the performance of the application varying Δ during each of the experiments. Our experiments were run on otherwise idle processors when there was little network traffic. Figure 7 shows the throughput for the application as a function of Δ. One curve depicted in Figure 7 is for a version of the application that uses yield() and one curve is for a version without yield(). Notice that at Δ=2 there is nearly a 50% improvement in throughput using the yield() instruction remotely. Observe that the intersection of the two curves (Δ=6) is the system's scheduling quantum.

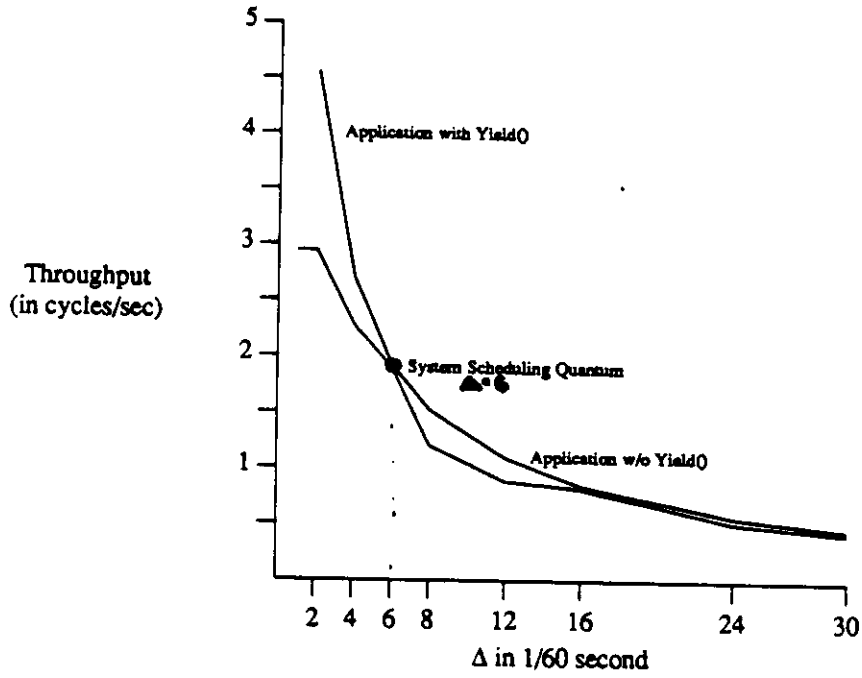


Figure 7: Worst Case Application with two remote processes

We observed that with Δ = 2, 2.75 sleeps of 33 msecs were added into each cycle for the yield() version of the program. This additional time adjusts the 109 ms figure to total 200 ms per cycle. Thus, 5 cycles/second is the maximum rate we could expect. Our observed performance is 90% of the calculated maximum at Δ=2. At Δ=0 we would expect roughly 8 cycles/second.

One reason for the performance degradation is that one site acts as user and library site. In order to perform these two functions, the site must context switch to perform some of the library functions and later resume the user process which is reading and writing. At the other extreme, if a third site were performing only library functions there would be a fixed cost

involved in transmitting and receiving messages from a remote library. The tradeoff between context switches locally vs. remote communications costs to the library in our prototype greatly favors colocating the library and the requester. For reasonable process loads, our component costs for remote communication far exceed the system's rescheduling speed.

Lastly, we observed that the effect of an application that is thrashing on overall system performance can be ameliorated by adjusting Δ. By increasing Δ, although application throughput is reduced, system performance is improved for other processes.

8.0 Time Window Evaluation

We constructed a simulated application that has a higher degree of processor locality than our worst case example. This application is intended to be more representative of an average case. Our purpose is to evaluate the utility of Δ rather than to measure performance per se. The application consists of two process that execute for-loops that decrement separate values in shared memory on the same page. The loops execute for a fixed period of time† until the decremented values reach zero. Each time a for-loop is executed the termination condition is tested. Thus, the for-loops exhibit read faults and write faults.

Figure 8 depicts throughput as a function of Δ. The curve has two distinct portions. One side, Δ<600, we call the "contention" side. The other side, Δ>600 we call the "retention" side. The low throughput on the "contention" side when Δ<120 is because of page conflicts between the processes that are reading and writing. When Δ=600 a maximum of 115,000 read-write instructions/second are achieved. When Δ>600 throughput is decreased because one of the processes retains the page for longer than it needs. Notice the decrease in throughput is more gradual in slope than the "contention" side when Δ<120. Also note that the range between (120<=Δ<=600) exhibits relatively good performance. Retention is an artifact of a protocol which uses a time window, but contention is a general problem for most network virtual memory management systems.

Mirage currently uses Δs that are uniform for a particular segment. Uniform Δs are not intrinsic to the design nor the implementation. The auxpte data structure contains the per-page Δ values and the implementation could be easily modified to use different values to tune system performance and page access. As one example, consider hot spot pages. These pages may exhibit behavior similar to our worst case application. There are two useful approaches we considered to organize hot spots. In one approach, hot spots are separated from the remainder of the segment data. A uniform Δ for each segment is a possibility in this organization. In another approach all data is in one segment, including the hot spots. In this organization, per-page Δs may be useful.

Our data from Figure 8 suggests general strategies for selecting and tuning Δ values. It is best for overall system performance to err by selecting a value for Δ on the "retention" side of the throughput curve rather than the "contention" side. Although throughput will be reduced for the application depending on the degree of error from the optimal selection, the falloff on this side of the curve is gradual. Also, increased sleep time for the particular application provides additional processor cycles to other applications. Therefore, overall system performance will be better than a choice on the "retention" side of the throughput curve. On the other hand, it is best for application throughput to err by selecting a value for Δ on the "contention" side. In our example, there are a wide range of values that give high throughput before the rapid falloff at

† In this example loops execute for 10 seconds. This amount of time is used for easier presentation.

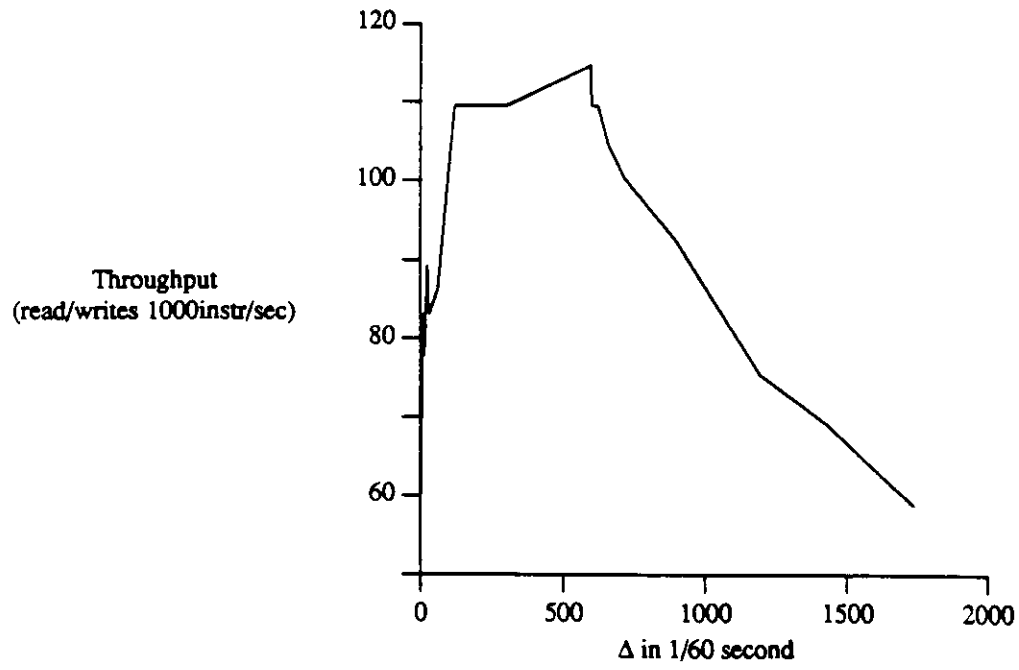


Figure 8: Two Conflicting Read-Writers

$\Delta<120$, in this example. However, a choice on the "contention" side would more severely affect overall system performance.

Lastly, the system itself could assist by increasing or decreasing page Δs dynamically. When the library sends an invalidation to the clock site, the page's Δ value can be changed before it is forwarded to the target site and installed. We are evaluating some alternatives for this dynamic tuning. Currently, the Mirage routine which performs this function is disabled.

9.0 Measuring Time and References

Mirage's performance is affected by how Δ is measured. Recall that Δ provides the amount of time a given process retains a shared memory page for read or write access. In Mirage Δ is measured using real-time. However, site loads can influence a real-time measure because heavy loads influence scheduling latencies. The load would decrease the effective Δ.

The time window Δ could be measured using user-process time. The problem with this approach is there may be many processes sharing the page on one site. It would be necessary to sum the individual process's page usage to accurately calculate when Δ has expired. Because of process loads, if one site executes considerably slower than the other sites, user time will not provide fair time allocations for processes using the page at other sites. Of course, one may be able to factor the site's load into the user time, but it may be of limited value because knowing the exact load does not adequately describe how many processes in the scheduling queue will reference the page. If few processes in the scheduling queue access the page versus many processes in the scheduling queue requiring access to the page, a different function may be required.

Lastly, Mirage provides a facility for logging all page requests at the library site. Each log entry contains the memory location, a timestamp, and the process identifier of the requester. We envision that a user-level process could analyze these reference strings as the basis for an automatic process migration facility or for later reference string analysis. Note, however, that reference strings from sites with valid page copies are not recorded.

10.0 Conclusions

From our preliminary results we approach DSM with cautious optimism. Mirage's component costs in accessing a shared page are no worse than average disk access latencies. However, in a network with a larger number of sites sharing pages than ours, invalidations may become expensive. Mirage's performance can be sensitive to simple application-level programming constructs. For example, loops that wait for shared variables to change should make the yield() call so that the remainder of the process's scheduling quantum is not wasted. Additionally, we found that the use of test&set can degrade performance substantially if the process in the locked region writes to the particular page of the lock while a remote test&set reader is testing. However, our design parameters are meant to ameliorate some of these difficulties by providing latitude in tuning page access. The time window Δ is Mirage's primary mechanism for doing so.

Mirage's performance for applications with poor processor locality suggest that Δ be small or equal to zero for such cases. However, for our synthetic application which exhibits substantially improved processor locality, throughput is best optimized with a larger Δ. Furthermore, this synthetic application showed the page contention portion of the throughput curve was worse in terms of performance than page retention side of the curve. The effect of an application that is thrashing on overall system performance can be ameliorated using a $\Delta > 0$ at the cost of reduced application throughput.

Lastly, implementing Mirage with a more modern machine architecture, faster CPU, better Ethernet interfaces, and with a more recent version of Locus would improve performance substantially. Since memory and processor speeds are rapidly improving, our fixed costs will decrease significantly. These aspects make distributed shared memory a much better performing and more attractive facility.

11.0 References

[ACCE86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., Mach: A New Kernel Foundation for UNIX Developement, *Proceedings USENIX 1986 Summer Conference*, Atlanta, Georgia, 1986.

[AGAR88] Agarwal, A., Gupta, A., Memory-Reference Characteristics of Multiprocessor Applications under Mach, *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Fe, New Mexico, May 24-27, 1988, pp. 215-225.

[ARNO86] Arnold, J. Q., Shared Libraries on UNIX System V, *Proceedings USENIX 1986 Summer Conference*, Atlanta, Georgia, 1986, pp. 395-404.

[ATT86] AT&T. *System V Interface Definition, Issue 2*, Customer Information Center, P.O. Box 19901, Indianapolis, IN, 1986.

[BACH86] Bach, M. A., The Design of the UNIX Operating System, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[BALL76] Ball, J. E., Feldman, J., Low, J. R., Rashid, R., and Rovner, P., RIG, Rochester's Intelligent Gateway: System Overview, *IEEE Transaction on Software Engineering*, Vol. SE-2, No. 4, December, 1976, pp. 321-328.

[BISI87] Bisiani, R., Forin, A., Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems, *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, Oct 5-8, 1987, pp. 21-30.

[FLEI86] Fleisch, B. D., Distributed System V IPC in LOCUS: A Design and Implementation Retrospective, *Proceedings ACM SIGCOMM 86 Symposium on Communications Architectures and Protocols*, Stowe, Vermont, August 5-7, 1986, pp. 386-396.

[FORI87] Forin, A., Bisiani, R., Correrini, F., Parallel Processing with Agora, Technical Report CMU-CS-87-183, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, December 1987.

[FORI89] Forin, A., Barrera, J., Sanzi, R., The Shared Memory Server, *Proceedings 1989 Winter USENIX Technical Conference*, San Diego, CA, Jan-Feb, 1989, pp. 229-244.

[LEAC83] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., Stumpf, B. L., The Architecture of An Integrated Local Network, *IEEE Journal on Selected Areas in Communications*, Volume SAC-1, No. 5, November, 1983, pp. 842-857.

[LANT82] Lantz, K. A., Gradischnig, K. D., Feldman, J.A., Rashid, R. F., Rochester's Intelligent Gateway, *Computer*, October, 1981, pp. 54-68.

[LI86] Li, K., Hudak, P., Memory Coherence in Shared Virtual Memory Systems, *Proceedings 5th ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, Canada, August, 1986.

[METC76] Metcalfe, R. M., Boggs, D. R., Ethernet: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM*, July, 1976, Vol. 19, No. 7, pp. 395-403.

[POPE81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G., LOCUS: A Network Transparent, High Reliability Distributed System, *Proceedings of the Eighth Symposium on Operating System Principles*, Published as SIGOPS Operating Systems Review, Vol. 15, No. 5, December, 1981, pp. 169-177.

[RAMA88] Ramachandran, U., Ahamad, M., Khalidi, M., Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory, Technical Report GIT-ICS-88/23, June 1988.

[RASH81] Rashid, R.F., Robertson, G.R., Accent: A Communication Oriented Operating System, *Proceedings of the Eighth Symposium on Operating System Principles*, Published as SIGOPS Operating Systems Review, Vol. 15, No. 5, December, 1981, pp. 64-75.

[WALK83] Walker, B., Popek, G., English, R., Kline, C., Thiel, G., The LOCUS Distributed Operating System, *Proceedings of the Ninth Symposium on Operating System Principles*, Published as SIGOPS Operating Systems Review, Vol. 17, No. 5, October, 1983.

Appendix I: Past Work†

Kai Li

Kai Li[LI86] experimented with a shared virtual memory system on a loosely-coupled multiprocessor, the Apollo Domain system[LEAC83]. Shared data is paged between processors, some of which have copies of the virtual address space pages. The model assumes ownership of pages can vary from processor to processor either statically or dynamically. This work concentrates on consistency problems and theoretical performance based on experimentation with centralized and distributed managers to locate the page owner. The last writer to a page becomes the new owner. Unless the local processor owns the page a managing site must be inquired before a write can occur.

Li's system supports a property called *coherence*. A coherent implementation is one in which a write to an address within the shared memory is always visible by all subsequent read operations to the same address from any site. This property provides the guarantee that stale data will not be read. We believe this property is essential for the correct operation of any distributed shared memory system and feel it's exploration is one of the most significant aspects of Li's work.

One problem is there are no hard performance figures because his prototype was built outside of the kernel. The implementation was approximately 4700 lines of application-level code. While results favor the use of distributed shared memory on loosely-coupled systems, a distributed implementation would have to be built in the kernel of the underlying operating system. Further, Li's work provided measurements of numeric applications only; no non-numeric computations were measured.

Agora

The Agora system[BISI88, FORI87] supports operating system and programming language functionality for parallel programs. Of interest is the operating system functionality which implements a form of shared memory for heterogeneous distributed computers. This sharing is done at the level of objects or data structures. A data structure is stored in the memory of the machine that creates it and is replicated in all the other machines that read it. These *cache* copies are always the copies read. A master copy is stored on one machine; all writes are forwarded to this master copy. A special agent notifies other machines when the master copy is changed, but no guarantee is made as to the recency of data read. Cache copies may become out-of-date and thus Agora does not assume coherence of the underlying objects.

Agora is the first implemented mechanism for sharing memory across machines in a heterogeneous computing environment. Generally, this is difficult to do because of incompatible byte orderings or alignment requirements of different processors. There is little hope of taking a low level (unstructured) shared memory system and expecting it to work across heterogeneous machines. The only way we could expect this to work is to build higher level representations whose semantics can be understood so that data representations can be translated. This would be expensive.

---

† This section may be omitted in the final draft of the conference paper.

The expense of data translation has been examined by other researchers. For example, there has been quite a bit of heterogeneity support in the Locus system. Early message systems such as RIG[BALL76, LANT82] and more contemporary message systems such as Mach[ACCE86] support different data types across a number of heterogeneous processors. These systems require a commonality in data representation or data translation capability between the heterogeneous sites. However, these data translations would be expensive to impose on a DSM system such as ours. For example, in one measured version of the Locus system, just the byte flipping of a read network message adds about 14% to the processing time at both the requesting site and the server site over the cost of processing a homogeneous read network message†. Performance could be even worse depending on the complexity of the flipping and the size of the data to be flipped.

Although it would be expensive to implement heterogeity support in Mirage, our main point of disagreement with the Agora approach is that without the ability to make a positive statement about the recency of data being used, there may be little benefit for many applications. User level synchronization is an unsatisfactory substitute for coherence.

Distributed Shared Memory Hardware Controller

In this work [RAMA88], the authors propose a distributed shared memory controller that provides efficient access and consistency maintenance of distributed shared memory. There is no low level coherence protocol; user programs are expected to utilize process synchronization primitives for consistency maintenance. The environment for the distributed shared memory controller is the *Clouds* distributed operating system. This system is an object-oriented operating system that supports synchronization within objects, and atomicity of computation.

The described hardware device has not been implemented but simulations have been performed to illustrate its effectiveness. The authors claim that their controller is effective relative to its object oriented environment. However, the object-oriented approach may not to be the best way to structure our low level UNIX system which is not object oriented. Nor does it seem reasonable to support objects on the particular hardware we are using because of architectural inefficiencies.

Mach

Mach[ACCE86, FORI89] now supports a shared memory server. Memory objects are managed either by the kernel or by user programs through a message interface. Sharing of memory is provided between tasks running on the same machine or across machines. An external memory paging task handles the paging duties and is responsible for the memory object. Mach attempts to deal with multiple page sizes and some aspects of heterogeneity. Coherence is supported. A substantial difference in philosophy between our work and Mach's pertains to overall system structure. In Mach, memory is managed by processes outside of the kernel including an external pager and network server processes. In our system shared memory management is handled by the kernel in the system nucleus. Communication is direct between kernels and no external networking processes are used. Since there are fewer user level context switches than in Mach with its external system processes, our scheme should perform favorably.

† A read response message (120 bytes) which returns an inode was used for this measurement.