Moving Distributed Shared Memory to the Personal Computer:

The MIRAGE+ Experience

Brett D. Fleisch * Randall L. Hyde † Niels Christian Juul ‡

UCR-CS-93-6

*Department of Computer Science, University of California, Riverside. This research is sponsored by a Joint Study with IBM Corporation, NSF CDA09209405.

†Department of Computer Science, University of California, Riverside.

‡Department of Computer Science, University of California, Riverside. Also supported by the Danish Natural Science Research Council.

Abstract

This paper describes the evolution of a *distributed shared memory* (DSM) system, Mirage, from its original implementation on VAX computers to its current implementation on modern high-end personal computers. Mirage provides a form of shared memory that is network transparent in a loosely coupled environment. The system hides network boundaries for processes that are accessing shared memory and is upward compatible with the System V UNIX[1] interface.

This paper addresses the architectural dependencies in the design of the system and evaluates performance of the implementation. MIRAGE+ performance is similar to Mirage, but the communication bottleneck has become more severe because of the larger page size used in the implementation. We show how this problem can be resolved on conventional hardware at little additional expense by using compression techniques.

[1] UNIX is a Registered Trademark of AT&T.

Contents

1 Introduction

Loosely coupled distributed systems have relatively low bandwidth inter-site communication when compared to tightly coupled processors or uniprocessors. Achieving good performance presents a number of challenges to designers particularly when coordinated sharing is desired. Nevertheless, loosely coupled distributed systems and workstation environments provide the potential to scale to very large configurations and provide economies of scale that centralized systems do not.

In the past, operating system designers have exploited the similarity between network packets and messages in the design of loosely coupled distributed systems. Systems based on this approach [Accetta 86] often use message passing as the fundamental technique to structure the system and its components. Nonetheless, some researchers have observed that the message passing approach may not be well suited for multiprocessors and tightly coupled processors that have access to shared memory [Li 86, Li 90, Bisiani 90, Bennett 90, Ramachandran 88, Fleisch 89b]. Message passing interfaces require the programmer to use conceptually different primitives and organize the programmed code differently than shared memory interfaces used in single shared memory spaces. Further, communicating large or complex data structures may be difficult or inefficient using message passing. An alternate approach is to use distributed memory.

In this paper we will focus on *Distributed Shared Memory* (DSM) [Nitzberg 91, Fleisch 89b] as opposed to *Distributed Virtual Memory* (DVM) [Li 86, Li 90]. While there are similarities between the two, there are significant differences between DSM and DVM, that include: 1) a DVM system has less sharing than a DSM system, and 2) DVM is oriented towards separate distinct address spaces. A goal for both types of systems is that the memory of one site be accessible to another site through load and store instructions. This goal is increasingly important as CPU and memory speeds improve at a much faster rate than mechanical fixed disk I/O speeds. Further, since large address space machines will become common in the 1990s, transparent, globally-accessible distributed memory systems, that can be accessed from loosely coupled workstations, will be increasingly important. Both DSM and DVM require efficient storage and retrieval of pages between sites. Therefore, the base component costs for DSM and DVM are equally significant. However, because of the higher degree of sharing in DSM systems, DSM systems will be extremely sensitive to excessive page faults and poor locality from application-level programs.

The Mirage subsystem [Fleisch 89b, Fleisch 89a] is a DSM facility implemented entirely in the kernel of a UNIX-based operating system [Popek 81, Walker 83]. Mirage provides a network transparent shared memory subsystem for a loosely coupled environment. In this paper, we examine our recent work to adapt Mirage for modern high-end personal computers. We call the new system MIRAGE+.

1.1 Reasons for Porting to a New Platform

Our prior work [Fleisch 89b] focused on: extensions to operating system functionality to support DSM, performance measurement of synthetic applications which characterize DSM, examination of algorithms and protocols to support DSM, and performance optimizations to reduce page movement and associated overheads using a *time-based coherence* approach. Although experiences and results from Mirage were encouraging, there were a number of issues that remained to be examined; these issues motivated our port to a new research platform. First, the machine technology used to prototype Mirage (VAX 11/750s) was obsolete by the time the implementation was complete. Performance was not competitive with contemporary architectures in terms of raw speeds. Second, the system we developed our work on was an early version of the Locus operating system [Popek 81, Walker 83] that was compatible with the UNIX System V interface specifications. The system had been superseded and subsequent releases of the system did not function with the equipment we were using. This meant we were unable to obtain the benefits of a number of performance optimizations that new versions of the system possessed. Third, we had no "real" applications to test Mirage. This meant we had to resort to synthetic applications to characterize performance, which critics have argued, may not be representative of typical shared memory access. Fourth, the issue of the generality of the approach for a large scale system arose. In particular, our preliminary prototyping environment consisted of 3 VAX 11/750s connected to a 10 Mbit/sec Ethernet[Metcalfe 76]. Because of the limited number of VAXs available, some questions remained as to the extensibility of the algorithms and protocols to scale to a larger configuration system. Lastly, the issue of reliability needed to be addressed. Previously we characterized our system as having a "tight" degree of sharing among a small number of reliable, well-behaved, communicating sites. This may not be the case in a large scale, typical, distributed computing environment. In these environments it is common for network or site failures to occur. Mirage did not address the issue of communication failures nor site failures.

1.2 The New Environment

We have redesigned and reimplemented the Mirage system for a high-end personal computer environment. The computing environment consists of 9 IBM PS/2 Model 70s and 3 IBM PS/2 Model 80s located at the University of California, Riverside. The personal computers were upgraded from i386 to i486 CPUs. The model 70s were upgraded with IBM "Power Platforms" containing a i486/25MHz and the model 80s with i486DX2/33MHz CPUs with 8K cache delivered through a third-party vendor. Most of the model 70s have 16 Mbyte of main memory, plus a 160 Mbyte hard disk drive. The model 80s have 10Mbyte of main memory with two internal 70 Mbyte disks. The cluster has a total disk storage capacity just under 5 Gbyte.

These systems are placed on a 10 Mbyte Ethernet[Metcalfe 76] and execute our modified AIX Version 1.2 which includes IBM's Transparent Computing Facility (TCF). The TCF subsystem, which was IBM's UNIX product that allowed a network of personal workstations to act as a transparent

single-system image cluster of machines, was developed by Locus Computing Corporation for IBM; TCF was the successor system to Locus[2].

There has been a significant improvement in technology for network interface adapters since our implementation began. In particular, network cards are faster, most of these cards have substantial onboard cache memories, and many of these cards handle back-to-back packet bursts better. Our version of AIX/TCF does not support the newer forms of network adapters that are on the PC market. Indeed, we are using obsolete boards so that we could get our system operating quickly. The consequences are that a *ping*-like operation between two i486/25MHz machines takes 5.4 msec.

1.3 Major Design Issues in Mirage+

The port of Mirage from a VAX implementation in Locus to a i386/486 based implementation in AIX/TCF was, at first, considered a straightforward task. However, in the process of moving between platforms, several difficulties have arisen. The differences in the hardware and software on which Mirage and MIRAGE+ have been based have presented a number of challenges. Among these, this paper focuses on the issue of how to handle the larger page size in MIRAGE+ effectively. Our experiences with the system have also made us aware of the need for fine grain scheduling and additional locking mechanisms, in addition to the implicit locking provided by Δ. Nonetheless, we present a convincing example that Δ is advantageous for DSM systems in Section 5.4. This paper also presents some of the lessons learned during the overall transition between platforms.

1.4 Completed Work

Currently MIRAGE+ is running on our hardware base. The implementation of MIRAGE+ seems to work well on this new platform. MIRAGE+ makes distributed shared memory pages transparently available to all sites as in the previous implementation of Mirage.

We have instrumented the operating system and the DSM system with optional timers and counters to instrument overall performance. These mechanisms are discussed further in Section 5.1. Further, we have added page compression, which presents a further step into our research in fast distributed shared memory systems.

1.5 Overview

The following section gives a brief introduction to the design, the philosophy, and the protocol for our DSM implementation (Section 2). The applications used to evaluate our system is presented in Section 3. We then discuss some of our problems designing and implementing the new system (Section 4). An evaluation of our performance is presented next (Section 5). The following section

[2]The TCF portion of the AIX product has recently been discontinued.

presents a unique way to remove the bottleneck that we encountered in the communication layer, including a preliminary implementation of a simple compression scheme. The measured performance impact on the DSM system is also presented (Section 6). Finally, we present our conclusions and outline our future work (Section 7).

2 Mirage+

The model of DSM used in MIRAGE+ is based upon Mirage. Here we present the model and its major concepts and terminology.

2.1 The Model

We have adopted a model of DSM based on our prior experiences with Mirage reported in [Fleisch 89b] that uses a paged segmentation scheme [Daley 68, Denning 70]. A DSM system consists of a shared address space that is accessed by *logical operations*. In our model, processes create shared memory by using a *segment;* the segment's size, name, and access protection are specified to create the memory. Processes locate and *attach* segments into their virtual memory address space by name. The attaching process can choose the exact virtual address range or elect to place the segment at a first-fit location in the address space. Once the named segment is attached, the shared memory can be used as any other locations of memory, the only difference being that changes to the underlying memory are also visible to the other local or remote processes that share the segment. When a process is finished with the segment it may be *detached*. The last detach of a segment destroys the data.

The shared portion of a process' address space (its segments) are partitioned into *data pages* which may be thought of as being replicated at the sites which have processes using the segment. The data pages are accessed using the same logical operations used to access the address space itself. The page holds the values associated with locations of the shared space. The logical operations that can be performed are *load*, which returns the value in a shared memory location, and *store* which changes a value in a shared memory location. Although each logical replica of a data page can be read or written independently, the user has the view that there is only one single copy of the data page in the network. The users views the entire shared address space as the union of all of the data pages present at all of the participating sharing sites. A *logical* operation issued at a site is implemented by executing a *physical operation* on one of the copies of the data pages associated with the location that is being accessed. We assume that when a logical *load* is issued, it is implemented by a physical load of one of the copies of the data pages associated with the referenced address. A logical *store*, on the other hand, results in an update to *all* replicas of the locations in the data pages associated with the referenced address.

Although the model we described seems conceptually straightforward there were some simplifica-

tions made in the implementation. In particular, when implementing a write to a page in our system, we designate one copy of the page as a writable copy of the page. We permit either the read-only replica of the page or the writable copy of the page to be present, but not both simultaneously. To accomplish this policy we invalidate all read-only replica of a page whenever a writable copy of the page is required in the network. This scheme permits an accurate adaptation of the model, described above, yet the programming of the implementation is considerably simplified.

2.2 Consistency Control and Coherence

Consistency control is an important issue in this model of shared memory. At the outset of the design we decided that it would be unacceptable for processes to read data that has become out-of-date or *stale*. In our model we provide *coherence* at the lowest system level. A coherent implementation is one in which a store to an address in a given segment is always visible by all subsequent load operations to the same address, independent of the machine location where the load occurs. Higher level synchronization primitives, such as semaphores or monitors [Brinch Hansen 73, Dijkstra 72, Hoare 74] may be used by applications that require stricter consistency. For example, the System V semaphore interface has been adopted for many of our applications (See Section 3).

Our past work on Mirage focused on a *time-based coherence approach* to DSM. The algorithm uses a clock mechanism to control when a site may be interrupted from its read/write processing to relinquish a page it is using. The clock mechanism grants the readers or the current writer a *time window* (Δ) which provides a guaranteed time period that the processes on a given site may 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 unused 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 controlling site (library) which attempts to invalidate the page on behalf of another requester. 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. In summary, this approach attempts to reduce thrashing, decrease the number of network invalidations, and minimize the amount of network activity required to provide coherence.

2.3 The Mirage+ Protocol

When processes are co-located and they share memory, the standard System V IPC implementation mechanisms augmented for AIX/TCF are used. However, when transparent distributed shared memory access is required, the MIRAGE+ protocol is used.

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 which performs the library's queueing discipline. 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 requests 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, the request is queued, and later when the request is processed the page is returned directly from the site which possessed it.

Another distinguished site in our model is the current *clock site*. 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. The clock site controls the decision-making for the time-based coherence mechanisms described in the previous section.

3 Distributed Shared Memory Applications

The applications which we will use to exercise our system are: quicksort, shellsort, mergesort, Fast Fourier Transform, a summation program[3], matrix multiply, a shared memory storage allocator, a memory resident database, and a battleship simulation [Novak 91, Rawdon 91]. Furthermore, most of the applications have been instrumented so that they may perform timing and logging of memory access patterns.

3.1 Memory Resident Database

We are currently implementing a memory resident database based on System M [Salem 90]. Memory resident databases hold much promise when used in conjunction with Distributed Shared Memory. The original version of System M was operational under the Mach operating system and needed substantial modifications to operate in our environment. Activities to port system M were broken up into the following tasks: 1) build a threads package, 2) make the synchronization primitives

[3]A partitioned n-process factorial overflows, so instead we perform a summation.

Step	Action	Shared memory access
P1.	`Scan through the battle field searching for a new shot by our adversary.`	**Read** entire own field multiple times.
P2.	`Acknowledge the shot by writing 'hit' or 'miss' at the location of the adversary's shot in the battle field.`	**Write** once to one place on own field.
P3.	`Shoot into our adversary's battle field.`	**Write** once to one place on other field.
P4.	`Wait for our adversary to write a 'hit' or 'miss' value at the location of our shot.`	Multiple **reads** of one place in other field.

Figure 1: The memory access pattern of the central loop in Battleship Simulation

operational, and 3) make memory management compatible with our system.

In order to make System M operational we designed and implemented a lightweight threads package for AIX/TCF. The AIXthreads package [Green 90] replaces the threads package in System M that originally ran with Mach. To maintain compatibility with System M code that used threads calls, a translation layer was built to translate System M threads operations to AIXthreads operations. Furthermore, we have been able to modify the synchronization primitives used in System M to work under our AIX/TCF system using a compatibility layer to call the System V IPC semaphores.

3.2 Battleship Simulation

The battleship simulation is an extremely useful application to exercise DSM. The battleship simulation features two competitors. Each player acts as an adversary attempting to destroy one another by shooting into a battle field represented in shared memory. The shared battle fields consists of two arrays; each competitor must access both arrays. Our implementation has each competitor execute as a separate process with no explicit synchronization primitives in use between the two processes.

The simulation works as follows: each competitor scans the battlefield repeatedly until they find a shot made by the other party, which is then acknowledged by writing either 'hit' or 'miss' in the playing field. The competitor then shoot in the battle field and waits for the other party to acknowledge the shot. This process repeats until one player gets a sequence of shots that constitutes a "win". At that time the simulation concludes. Figure 1 outlines the algorithm of the basic loop for each of the two competitors.

4 Distributed Shared Memory Issues

A number of issues have arisen in the design and implementation of MIRAGE+ for our new environment. Although MIRAGE+ resembles Mirage, the system has been substantially redesigned for improved reliability (See [Juul 93]) and better performance.

Both new hardware and new software have required that we redesign MIRAGE+. Among the most important changes were the page size, segmentation using *vsegments*, the lack of multicast in the communication sub-system, and the granularity of system clock interrupts. The issues of locking are also addressed in this section.

4.1 A Larger Page Size

Specific hardware differences between Mirage and MIRAGE+ affect DSM design and performance. For example, in Mirage the page size was 512 bytes (DEC VAX 11/750) while MIRAGE+ has a page size (i386/486) of 4096 bytes. The factor of 8 difference in page sizes has significantly affected the latencies. We can expect that to assemble one i486 page will require the receipt of 4 network packets since an AIX/TCF network packet is limited to a buffer of 1 Kbyte. Furthermore, the Ethernet media can support no more than roughly 1500 bytes. Thus, MIRAGE+ will require 4 network interrupts to process one page rather than one interrupt as in Mirage.

When we first began work on MIRAGE+ it was not clear whether the increased page size would reduce the *number* of page faults. Nonetheless, we have observed the locality exhibited by the application determines overall performance. Reduction in the number of faults could potentially compensate for the increased latencies arising from processing four network messages for a MIRAGE+ page. However, reduction in number of faults may be mitigated because *false sharing* may be introduced with larger page sizes.

We believe the issue of *false sharing* will have a profound affect on performance. False sharing occurs when several shared data aggregates that are placed on the same page are referenced by different process groups. Generally these different groups will be located at different sites. False sharing affects performance because larger page sizes that store these data aggregates will exhibit additional movement to service reference requests by different process groups. By simply rearranging the shared data aggregates to be on different pages, the referencing processes will no longer fault the same page excessively.

We were particularly concerned with false sharing based on simulation results at Tulane using the larger page size. While experience in Mirage suggest 512 bytes was a good page size for DSM, a larger page size could potentially present serious performance problems to MIRAGE+. While a larger page size can be advantageous for I/O subsystem performance, localities in memory access tend to be small. Increasing localities in a DVM system may not cause problems, however increasing localities

in a DSM system can increase false sharing and lead to thrashing. We were extremely concerned that the 512 byte page size used in Mirage may give better performance than our current configuration on much faster equipment.

4.2 New Segmentation Support

A significant difference between Mirage and MIRAGE+ lies in the software that was used to support the implementation. MIRAGE+ supports a new underlying kernel implementation of shared memory that uses so-called *Vsegments*[4]. Vsegments and proc_vsegs provide a superior solution to handling shared memory in the UNIX kernel than the original implementation because the code attempts to be machine independent. Furthermore, Vsegments address a problem described in [Fleisch 89a] where invalidation of a shared page was difficult because each process' page tables had to be invalidated in order to invalidate a shared page. Invalidation was a source of inefficiency and difficult to implement efficiently in Mirage, particularly if a process had been swapped out. The software implementation of Vsegments implements a level of indirection in shared page access. To invalidate a shared page, one need only invalidate an indirect descriptor through which all processes access the shared page, rather than all entries that refer to the shared page. Microsoft's Windows NT operating system supports a *prototype page table entry* which provides similar functionality [Custer 93].

The transition to adapt and understand Vsegments has taken more time to complete than originally anticipated. Indeed we were unaware of the changes in the system to support Vsegments when we approached our sponsor about incorporating MIRAGE+. One aspect that could have mitigated some of the delay was open operating system sources. An open platform would have prepared us for any differences between what we believed was in the operating system and the delivered version. As is often the case in computer systems, learning a new paradigm has had a steep learning curve.

4.3 Communication Between Kernels

The AIX/TCF kernel supports cluster communication that provides virtual circuits between cluster sites. The programmer is able to call a high level netsend() routine with a network packet. The low level communication subsystem is responsible for taking the programmer's packet and transmitting it to the destination site. The messages are sequenced, ordered, reliable, and possibly combined with other messages when sent to the destination. There is, however, no support for multicast or broadcast.

When the clock site needs to perform an invalidation of a number of participating readers, the relevant information must to be transmitted to all of the participants. AIX/TCF does not support broadcast messages, based on the philosophy that broadcasts fail to scale well in large scale environments and may depend on the media supporting such a broadcast mechanism. This philosophy,

[4]A detailed discussion of the underlying Vsegment implementation is beyond the scope of this paper.

however, is at serious odds with good performance for our DSM work. Instead, we have had to rely on emulating multicast by using iterative sends. This issue has been a source of performance problems over the past several years. When we move MIRAGE+ to a new environment we would expect this issue to disappear.

4.4 Clock Granularity

The granularity of the system clock influences the user process scheduling. In AIX/TCF processes are scheduled round-robin with a time slice of 20 msec. This permits scheduling at a frequency of 50Hz. On the other hand, DSM pages are locked for a minimum of at least Δ. In Mirage, Δ was equal to one or more time slices of 16.66 msec, whereas in MIRAGE+ Δ is a multiple of 20 msec. Both Δs are measured in wall-clock time. As pointed out in [Fleisch 89a], the way in which Δ is measured could have an important performance impact. Further, we have determined that the issue of DSM page use and process scheduling greatly impact one another.

In MIRAGE+ we correct a source of error that if Δ is measured in wall-clock time, the process requiring the page may never get to use the page on a heavily loaded site. This situation arises when Δ expires before the process is scheduled. There were two possibilities for correcting this problem. The first possibility is that we measure Δ in user process execution time. We did not adopt this solution because it could easily cause the system to deadlock. For example, consider a process P1 that holds page 1 and process P2 that holds page 2. If P2 faults page 1 and P1 faults page 2, the two sites are deadlocked since both require the resources of the other and both cannot accumulate user process execution time. Obviously, measuring Δ in user execution time would not be acceptable without some form of aging.

We solved the problem in MIRAGE+ by requiring that Δ be measured in wall-clock time and that the page be retained at the site until the process first accesses the requested page. This solution requires hardware support for an *accessed* and *dirty* bits in the page table entry. Specifically, a read-only page must have been accessed, whereas a writable page must be dirty before we permit Δ to expire.

Our version of time-based coherence uses a very general mechanism to delay an invalidation request that would preempt a page from a process. If Δ has not expired, the requesting process calls the kernel *timesleep()* call for the remaining Δ. If Δ is small, this period is often a fraction of a time slice. However, the system does not support a microsecond timesleep(). User processes and kernel processes are therefore required to wait for the next timer interrupt, which occurs every 20 msec. For this reason, we have not experimented with a Δ with a finer granularity than 20 msec.

The issue of not having a fine grain timesleep affects performance significantly. As an example of this, an application which performs a *yield* call greatly improves its performance on a single-site by forcing itself to be rescheduled instead of waiting until the end of a time-slice. Figure 3 shows

additional support for this conclusion.

4.5 Locking

In order to address the issue of excessive page movement at a higher level, *locking* has been added to MIRAGE+. One of the most significant reasons that we adopted locking was that we found that we needed to give the user better controls. For example, the user may require locking for System M database applications or to acquire a "working set" of pages. Allowing the user to set Δ to a large value is a poor idea since the library site may attempt to invalidate a page that has a large Δ value. Recall, the clock site has the unilateral authority to decide on the time a page is permitted to be invalidated. A page that was locked would improve the protocol. A useless attempt to invalidate a page could be eliminated since the library site would know that the page was locked.

The locking protocol works as follows: a system call has been augmented in the kernel to permit a programmer to lock a page. The *lock page* call sends a message to the library site requesting the page. The request is marked as being one that will locked-down the page at the requesting site when it is obtained. This request is easily implemented by using the normal message sent during a page fault with an additional "lock" field. The request is enqueued at the library site, as are normal remote page fault requests. When the library site processes the request for the page, and the page is sent to the requester, the library moves an entry which represents the page, to a *locked-down queue* until a corresponding *unlock request* is received from the lock holder. At that time, the library site removes the page from the locked-down queue and permits the page to be accessed by other requesters. The library site searches the locked-down queue before performing invalidations so that invalidation attempts for a locked-down page will not occur. Pending requests for the page will continue to be proceed after the page is unlocked.

5 Mirage+ Performance

We present our observations and experiences from using MIRAGE+. The performance figures presented in this section include an analysis of the time spent servicing a remote page fault, and measurements of the time it takes to execute an application that uses distributed shared memory intensely.

The performance analysis was made by instrumenting the kernel to accumulate information about code use and the amount of time spent in key portions of the implementation. A short overview of the instrumentation is given first.

5.1 Instrumentation for Performance Measurements

Kernel instrumentation gives us useful performance data for our analysis. First, we have instrumented the kernel with a micro-timing function that makes it possible to read a timestamp with the

Operation	User Site	Library Site	Transmission Time
Request send	1.0 msec		
Request received		0.1 msec	
Process request		1.4 msec	4.1 msec
Send page (part 1)		1.7 msec	
Receive page (part 1)	1.1 msec		
Send page (part 2)		1.9 msec	4.1 msec
Receive page (part 2)	1.1 msec		
Send page (part 3)		1.9 msec	4.1 msec
Receive page (part 3)	1.1 msec		
Send page (part 4)		1.9 msec	4.1 msec
Receive page (part 4)	1.2 msec		
Process restart	0.4 msec		
Transmission Time	8.2 msec	8.2 msec	16.4 msec
Total User Site	14.1 msec		
Total Library Site		17.1 msec	
Total	31.2 msec		

Table 1: The Component Cost of a Page Fault.
Library Site and User Site are both i486/25MHz machines.

granularity of 1μsec[5]. This function has been used to implement kernel timestamping. Timestamps are placed in a circularly accessed buffer inside the kernel that can be output when the kernel is executing. Second, counters have been added to critical routines to count function calls. Both the counters and the cyclic buffer may be returned with a system call. A utility program has been built, which formats the information returned by the system call, and prints it in a *gprof*-like format. Lastly, we have instrumented our applications to use kernel timestamping. Instead of printing the actual values of the counters when the kernel is executing, the values may be read at the beginning of the application and again at the end, and only the incremental values added to the counters during the application run are output.

5.2 The Cost of a Page Fault

Table 1 gives the component cost of a remote page fault. These figures are measured on two i486/25MHz machines, where a site requests a page checked-in at the library site[6]. All other processing was moved to isolate these figures. The measurements were obtained by determining the delay

[5]Thanks to Noah Mendelson of IBM for this code.

[6]Checked-in means that library site is also the clock site for that page.

Operation Time	Mirage Time	MIRAGE+ Time
Request send	2.5 msec	1.0 msec
Receive page		4.5 msec
Transmission costs	10.7 msec	8.2 msec
General overhead		0.4 msec
Total user side	13.2 msec	14.1 msec
Request received	1.5 msec	0.1 msec
Process request	2.0 msec	1.4 msec
Send page		7.4 msec
Transmission costs	10.7 msec	8.2 msec
Total library site	14.2 msec	17.1 msec
Total costs	27.5 msec	31.2 msec

Table 2: Comparison of Component Cost of a Page Fault between Mirage and MIRAGE+. Mirage was measured on VAX 11/750 with 512 byte page size, and MIRAGE+ was measured on i486/25MHz machines with 4K page size.

on the user site when a message is sent, until the next message arrives, and subtracting the library site processing time during this period. Each of the network transmission figures shown above is 4.1 msec. This transmission time covers twice the delay between a message being sent from one site and received at another. Consequently, the delay introduced by our underlying network communication layer is approximately 2 msec per message.

We are currently incorporating low-level performance enhancements in MIRAGE+ that will improve these results. Our protocols currently block waiting for high level acknowledgments for each message. By sending the four messages asynchronously, instead of waiting for the four individual acknowledgments for each message, we expect to reduce the transmission total from 16.4 msec to 4.1 msec. This improvement will have a significant impact on the total cost to fault a page across the network. In our case, the cost is expected to be 18.8 msec instead of the 31.2 msec currently measured.

We have compared the results of Table 1 with the previous published results for Mirage [Fleisch 89b] running on a set of 3 VAX 11/750s. The comparison is shown as Table 2. The results compare favorably given that MIRAGE+ has a 4K page size.

5.3 Running the Battleship Simulation

The two processes making up the *Battleship Simulation*, as described in Section 3.2, exercise their shared memory page intensively. We did not strive to provide an efficient implementation of the simulation, but instead we looked for a worst-case example where our distributed shared memory

Competitor on site 1			Competitor on site 2	
Action	Page status		Page status	Action
P3. Write a shot	W	×		•
P4. Loop read shot ack.	W	×		•
		→		
P4. Loop read shot ack. (again)	R	R		P1. Loop read to find new shot
•	×	W		P2. Ack. shot
•	×	W		P3. Write a shot
•	×	W		P4. Loop read shot ack.
		←		
P4. Loop read shot ack. (again)	R	R		P4. Loop read shot ack. (again)
P1. Loop read to find new shot	R	R		P4. Loop read shot ack. (again)
P2. Ack. shot	W	×		•
P3. Write a shot	W	×		•
P4. Loop read shot ack.	W	×		•
		→		
P4. Loop read shot ack. (again)	R	R		P4. Loop read shot ack. (again)

Legend:
- • program blocked from getting page as the other site has the page.
- ← page transfer in arrow direction.
- R Site has page in (R)ead mode.
- W Site has page in (W)rite mode.
- × Site has no access to page.

Figure 2: The observed shared memory access pattern of the Battleship Simulation

would be exercised to the extreme.

The access pattern of our implementation of the Battleship simulation follows the algorithm shown in Figure 1. We have implemented the simulation with one shared page for both competitors. The shared page holds the two arrays representing the battle fields. The observed access pattern for this case of *false sharing* between the two competitors are shown in Figure 2.

Table 3 shows the simulation performance based on whether the simulation runs on one or two sites, and whether competitors use a system call *yield* to voluntarily give up the CPU before the time slice is exhausted, thus permitting another process to execute. When running on one site, the yield call is important as the two processes would otherwise waste the remainder of their time slice waiting.

The total run of the simulation contains 788 iterations, where each iteration has a shared memory access pattern as illustrated in Figure 2. Thus the number of iterations per second reflects the average number of times each site can perform *a read fault*, including *a page transfer*, and *a write fault*, including *an invalidation message* transfer.

	One site		Two different sites	
Using *yield*	3.16 sec	(262.7)	94.88 sec	(8.4)
Without *yield*	460.78 sec	(1.7)	94.64 sec	(8.4)

Table 3: Performance of the Battleship Simulation
Figures show total run-time and in parenthesis the number of iterations per second.

5.4 The Use of Delta and Importance of Locking

Measurements in Mirage showed that time-based coherence can be useful in the design of DSM systems. A significant result from the Mirage work was that performance for some applications could be better with a large value Δ. Since that time, critics have continued to suggest that time-based coherence can be eliminated without impeding DSM performance.

In the battleship simulation, using Δ, which provides implicit locking, prevents the page from being downgraded (from write to read-only access) prematurely. Without locking, the writable page would have been downgraded when obtained in step P2, and two or more messages would be needed to upgrade it to be writable later. In short, Δ ensures that both P2 and P3 are done sequentially. Without the Δ window, the page would be downgraded between step P2 and P3, and sent to another site. Thus with our Δ, the battleship simulation is expected to execute in almost the half the run-time than one that does not perform the locking function. In fact, Δ prevents considerable thrashing which would otherwise occur.

We have also tested a version of the battleship simulation with explicit locking around the two writes at P2 and P3. The performance results show the same performance with explicit locking. Thus, we believe that the implicit locking, using the Δ time window, works effectively. By implicitly locking the page to a site for Δ, we become less vulnerable to programming errors, such as omitted unlocking. Nonetheless, our techniques require system tuning or application tuning to be effective.

The granularity of the system clock interrupt has also affected our performance. The granularity of the system clock interrupt available to wake up sleeping processes in our version the AIX system is 20 msec. A process performing a *timesleep(n)* sleeps until n timer interrupts have occurred. Thus, we have not been able to perform experiments with Δ less than 20 msec in our current implementation. As the amount of work done by the battleship simulation is fairly small, and it is in fact communication intensive, a better tuned Δ would have shown improved performance.

5.5 Performance Implications

The main lesson learned from the experiments are that the page size supported by the underlying hardware and software layers has a significant impact on overall application performance. Significant performance problems have arisen from the costs associated with sending a larger page and the

problems that arise from false sharing. In the next section, we examine additional performance improvements suitable for DSM page sizes that require multiple network messages.

6 The Communication Bottleneck

6.1 Communication Costs

While DSM systems may exhibit good temporal and spatial locality patterns, processor locality can be rather poor since DSM applications on different processors often access the same page concurrently. MIRAGE+'s Δ improves system performance by serializing processor write access, which reduces thrashing caused by poor processor locality. Even with Δ, however, poor processor locality can lead to poor performance.

Communications overhead provides a significant contribution to the overall cost of using DSM. Several possible ways to reduce this overhead include:

- Increase the speed of the communication medium. With networks approaching and even exceeding 100 Mbit/sec, this technique may hold promise in the future.
- Reduce the page size. Smaller page sizes (e.g., 512 bytes or 1K) would reduce the number of network messages MIRAGE+ must transmit.
- Use a separate network for DSM. In our experimental environment, there are very few collisions on the Ethernet. A heavily loaded network would have an impact on DSM performance[7].
- Compress the data before transmitting it over the network and decompress it at the destination.

Note that all these techniques are complementary. Indeed, a system could use all four techniques to achieve better performance. The first three generally depend upon special hardware.[8] The fourth possibility, compression, is more flexible, easier to instrument, and well-suited for experimentation since it can be implemented in software. We explore this latter technique in this section.

6.2 Compression Techniques and Problems

There are many compression algorithms available or published in the literature [Bell 89b, Raita 87, Ziv 77, Williams 90, Bell 89a, Storer 88, Nelson 91]. We originally believed that we would be able to select from these and incorporate it in MIRAGE+ and obtain improved performance. However,

[7] Conversely, DSM would also affect the network performance.

[8] Most CPUs provide a fixed page size. Others offer a limited number of fixed pages sizes (usually two or three different sizes). Even on CPUs which offer multiple page sizes, a DSM system such as MIRAGE+ must coexist with an operating system which may places its own constraints on the page size.

Input Block Size	Compression Ratio
1K	68%
2K	63%
4K	59%
8K	55%
16K	53%
32K	51%

Table 4: Expected Input/Output Compression Ratios
The file *progc* from the Calgary Compression Corpus [Bell 89b].
Table by Burrows, Jerian, Lampson, and Mann [Burrows 92].

the issue turned out to be much more complex than we originally imagined. There are constraints which limit the suitability of the algorithms for DSM systems. These constraints include compression/decompression time, lossless vs. lossy compression, and discrete compression ratios.

6.3 Compression and Decompression Time

The primary constraint in the selection of a compression algorithm is that the data compression, decompression, and transmission be faster than the transmission of the uncompressed data[9]. Table 4 shows results obtained from Burrows, Jerian, Lampson, and Mann [Burrows 92] study. Burrows, Jerian, Lampson, and Mann report on a variant of the Lempel-Ziv(LZW) algorithm which requires a compression/decompression time of approximately 17 msec for a 16 K block Extrapolation suggests that this algorithm would require just over 4 msec for 4K blocks on a DECStatation 5000/200.

We examined the time to obtain a page from a remote site in MIRAGE+ and found that approximately 31 msec elapsed to fault a remote page (See Table 1). Since the anticipated compression ratio from Table 4 is only 68%, this suggests that the compression algorithms studied by Burrows, Jerian, Lampson, and Mann [Burrows 92] might be of limited utility in MIRAGE+. A 68% compression ratio would reduce data transmission time by at most 25% in MIRAGE+[10]. This implies the compression/decompression time must be less that 8 msec for there to be a net gain. At first glance, the numbers above would suggest that a LZW-variant would suffice. However, results in Table 4 were produced on a much faster platform than the 486/25 machines on which MIRAGE+ operates. Therefore, it is unlikely that the algorithm used by Burrows, Jerian, Lampson, and Mann would produce a substantial performance improvement in MIRAGE+.

We surveyed a number of additional compression techniques for MIRAGE+. For example, Burrows, Jerian, Lampson, and Mann outline simple hardware which would dramatically speed up compression.

[9]Of course, if the network traffic is high, compressing the data may still be advantageous since it reduces net traffic. This would allow MIRAGE+ to scale somewhat beyond the limitations of a CSMA bus-based network like Ethernet.

[10]Assuming zero compression/decompression time.

Bunton and Borriello [Bunton 92] also describe compression/decompression algorithms which are relatively easy to implement in hardware and support variants of the LZW algorithm running between 10 Mbyte/sec and 20 Mbyte/sec. Finally, Compression Technologies manufactures a device which performs data compression in real time and connects directly to the communication link[11]. Using a faster CPU or adding one of these hardware compression devices may improve performance.

6.4 Lossless vs. Lossy Compression

Another important, and often overlooked constraint, is that the compression/decompression algorithms be lossless in MIRAGE+[12]. Even in an environment where lossy compression is acceptable (e.g., video compression), a lossy compression scheme may not be ideal since repeated compression/decompression[13] might ruin the underlying data. For example, if on each retransmission only one bit is lost, after several thousand transmissions, the data would sufficiently approach randomness so as to be useless.

6.5 Discrete Compression Ratios

To improve network communication performance, MIRAGE+ needs to reduce the number of transmitted messages. AIX/TCF requires nearly the same amount of time to transmit a short network message as it does to transmit a 1K message. This is because the system has been optimized to pass messages with buffers by reference, as much as possible, to the low-level driver. The overhead of using the code in terms of packaging a message, using the send code, and copying the message to the driver routine level, is the dominant cost in sending a message. Whether the message has a 1K buffer or not does not significantly affect the cost of kernel communication. Therefore, reducing the size of transmitted messages does not provide a big performance improvement. However, reducing the number of messages that MIRAGE+ must transmit has the potential of greatly enhancing system performance. In virtual memory systems, this is analogous to reducing the service time of a page fault being much less important than reducing the number of page faults.

Since the number of messages, rather than the size of the messages, seems to be the most significant factor in the transmission time, a compression scheme which works best with MIRAGE+ is one which produces compression ratios of less than 75%, less than 50%, or less than 25%. This is because the AIX/TCF communication subsystem transmits MIRAGE+'s 4K page as four separate 1K network transmissions. If the compression algorithm cannot achieve at least 75%, compression will not provide a performance improvement since the compression/decompression overhead is greater than the transmission savings. This *quantization* of the compression benefit may seem to be a problem. After

[11] Based on information received via email.

[12] MIRAGE+ provides a guarantee that data placed in memory will be stored and retrieved reliably.

[13] Which would occur as pages "ping-pong" between sites.

all, it would be nice to reduce the transmission times in proportion to the actual compression ratio. However, this quantization to 1K message sizes presents the opportunity to trade off compression performance (speed vs. space) to achieve a well balanced algorithm.

6.6 Other Issues

Another important issue is that the compression algorithm work well when compressing small blocks of data. Most compression algorithms work best with large blocks of data and, conversely, perform poorly with small blocks of data. For example, Table 4 contains compression ratios for a block of textual data. Since MIRAGE+ uses 4K blocks, one would expect to see only a 59% compression ratio (i.e., a savings of one network message) using this algorithm operating on textual data.

In our work we plan to rely on the assumption that the results of Burrows et al. may not apply to (binary) memory pages equally well. In support of this assumption, Bell, Witten, and Cleary [Bell 89b] contrast several different compression algorithms and their relative performance with respect to differing types of data. Their research indicates that typical compression schemes will typically save at least one message for many types of data, not just text. Fortunately, although typical binary data may not compress as well as text, transmitting a page which achieves only a 75% compression ratio doesn't take substantially more time than a page which achieves a 59% compression ratio. We believe this 16% margin would probably be sufficient to cover the differences encountered between a typical memory page and a text page. So even if we are unable to achieve comparable results, we assume the impact on transmission time will be minimal, as long as the compression ratio does not climb above 75%. That implies that MIRAGE+ could save transmitting one network message for each page with most types of data.

6.7 Current Implementation

Despite the well-know performance of LZW-class compressor performance [Bell 89b, Raita 87, Ziv 77, Williams 90, Bell 89a, Storer 88, Nelson 91], there are drawbacks which convinced us that the LZW may not be suitable for our current implementation. LZW-class algorithms typically maintain a dictionary which must be present in non-swapped memory[14]. To produce reasonable results, the dictionary should be maintained between transmissions to save having to reconstruct it for each transmission. Finally, although high performance LZW implementations exist, they are not the fastest algorithms and the added compression LZW provides may not be significant considering the quantization issues described in Section 6.5.

As a test case, we chose to use run-length encoding (RLE). We made this choice because many of our benchmark programs exhibit large runs of the same data (e.g., sparse matrices), RLE is relatively fast, and RLE is very easy to implement.

[14] If the dictionary were swapped to disk, the performance benefit of compression would certainly be lost.

	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 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 5: Compression of page transmission during Battleship Simulation
The number of packets and pages are given for each of the two participating sites.

Our RLE compression attempts to find runs of double words (32 bits) in memory and replaces such runs with a two-byte or six-byte sequence denoting the run. Two byte runs consist of a byte count between two and 255 and a data byte. If the first byte is zero, then the next byte is a length and the next four bytes specify the data to replicate. A 4K block (1K double words) of zeros compresses down to about 26 bytes using our scheme[15]. Although the granularity of this compression algorithm is rather coarse (e.g., a "pure" RLE compressor should be able to encode 4096 bytes of zeros into three bytes or so), this RLE algorithm works reasonably well for typical values found in MIRAGE+ shared memory and it allows the processor to work mostly with double word aligned values which produces the fastest compression/decompression rates in our system.

On our systems, the current RLE compression algorithm requires about 400 μsec to compress a 4K block of zeros, and roughly two milliseconds to process a block of random data which it cannot compress. The decompression algorithm runs at approximately the same speed as the compression algorithm. The worst case occurs when the compression algorithm can only reduce the page by 75%[16]. In this situation, the compression/decompression operation requires about 4 msec and MIRAGE+ saves approximately 8 msec because it need not transmit one of the messages that has a 1K buffer.

Our experiments indicate that a software-based compression/decompression scheme improves the performance of MIRAGE+. The relative performance figures of the battleship simulation running with and without compression are shown in Table 5. Note that although the number of RLE messages transmitted is roughly half the non-compressed number, the overall performance is enhanced 34%. The difference is due to the time required to compress and decompress the pages. Compression and decompression improves throughput but increases overheads.

[15]Four six-byte entries encoding 4*255*4=4080 bytes and a single two-byte entry encoding the remaining 16 bytes.

[16]When the compression algorithm discovers that the compression savings will not be at least 75%, it immediately aborts and MIRAGE+ transmits uncompressed data. The time spent compressing the data up to the 75+% point is lost, but at least the receiving site will not have to waste time decompressing the data.

7 Conclusions

This paper has addressed a number of issues in the design and implementation of a DSM system in our new hardware and operating system environment; the paper relies heavily on our actual experiences with DSM. We have shown the complexity of the problems in moving memory management between environments and presented solutions that have work well for MIRAGE+.

7.1 Results

Without compression and using synchronous communication, the time to fault a page across the network is 31.2 msec. The expected reduction in performance due to the larger page size and potential for false sharing is ameliorated by the faster CPU speeds and compression. With simple compression we have shown a 34% speed-up and a 50% reduction in the amount of network traffic for our worst case application, the battleship simulation.

7.2 Reflections

In monolithic operating systems kernels, as our version of UNIX, code reuse appears throughout the kernel. Often a verbatim copy of old code is reused with particular changes. This leaves nearly identical variant throughout the kernel. The maintenance and modification of these variants becomes increasingly troublesome as time passes and patches are added.

Thus, we believe that a more modular or object-oriented design would decrease the size of the kernel and ease maintenance. Such design strategies cannot, however, be rigorously followed, as specialized version of modules are sometimes necessary due for efficiency. Our effort would have been significantly improved had the kernel been structured with reuse in mind and written in an object-oriented programming language like C++.

7.3 Future Work

Currently MIRAGE+ is running on our PS/2 hardware base. We plan to spend the next several months exercising additional MIRAGE+ applications and reporting the results.

The MIRAGE+ compression succeeded insofar as it demonstrated the feasibility of compression to improve the performance of a DSM system. We assume the run length encoding algorithms in MIRAGE+ compression is not the best choice for the job. We plan to examine different compression algorithms with a representative set of applications and compare the various performance benefits of each compression technique. Work is currently underway to develop special compression algorithms which complement MIRAGE+ memory access patterns.

MIRAGE+ reliability is another issue that we are currently addressing. Our goal is to have a working version of MIRAGE+ that is tolerant of single site failures using techniques that could enhance performance *over DSM systems that do not have reliability mechanisms* within a couple of months.

7.4 Final Remarks

We believe that the lack of success for many DSM implementations is due, in part, to race conditions, which are hard to debug. Our approach, that uses time based coherence, has helped us to expose race conditions in the debugging phase. For the ultimate success of our approach, we intend to improve performance to be comparable with the efficiency of message-passing systems.

References

[Accetta 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *Summer USENIX Conference*, USENIX, June 1986.

[Bell 89a] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[Bell 89b] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–589, December 1989.

[Bennett 90] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 1990 Conf. Principles and Practice of Parallel Programming*, pages 168–176, New York, NY, 1990. ACM Press.

[Bisiani 90] Roberto Bisiani and Mosur Ravishankar. *PLUS: A Distributed Shared-Memory System.* Technical Report, 1990.

[Brinch Hansen 73] Per Brinch Hansen. *Operating Systems Principles.* Prentice-Hall, Englewood Cliffs, New Jersey, July 1973.

[Bunton 92] S. Bunton and G. Borriello. Practical dictionary managment for hardware data compression. *Communications of the ACM*, 35(1):95–104, January 1992.

[Burrows 92] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Proceedings of ASPLOS-V*, pages 2–9, October 1992.

[Custer 93] Helen Custer. *Inside windows NT.* Microsoft PRESS, Redmond, Washington, 1993.

[Daley 68] R. C. Daley and J. B. Dennis. Virtual memory processes and sharing in multics. *Communications of the ACM*, 11(5):306–311, May 1968.

[Denning 70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[Dijkstra 72] E. W. Dijkstra. *Hierarchical Ordering of Sequential Processes.* Academic Press, New York, 1972.

[Fleisch 89a] Brett D. Fleisch. *Distributed Shared Memory in a Loosely Coupled Environment.* PhD thesis, Computer Science Department, University of California, Los Angeles, 1989.

[Fleisch 89b] Brett D. Fleisch and Gerald J. Popek. Mirage: a coherent distributed shared memory design. In *Proceedings Twelfth ACM Symposium on Operating Systems Principles*, pages 211–223, Litchfield Park, AZ, 1989.

[Green 90] Tammy A. Green and Darin W. Goodstein. *AIXthreads.* Technical Report, Department of Computer Science, Tulane University, New Orleans, Luisiana, 1990. Unpublished Student Paper.

[Hoare 74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):545–57, October 1974.

[Juul 93] Niels Christian Juul. *Research Discussions on Reliability of Distributed Shared Memory.* Mirage+ Research Note 2, Department of Computer Science, University of California, Riverside, CA, June 1993. Unpublished research notes.

[Li 86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings 5th ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, Canada, August 1986.

[Li 90] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *IEEE Computer*, 23(5):54–64, May 1990.

[Metcalfe 76] R.M. Metcalfe and D.R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–403, 1976.

[Nelson 91] M. Nelson. *The Data Compression Book.* M & T Books, Redwood City, Ca, 1991.

[Nitzberg 91] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[Novak 91] Shannon Novak, Jennifer Lamb, Tammy Green, Lucy Weldon, and Brett Fleisch. Performance evalutation for a loosely coupled parallel processing environment. In *IEEE Proceedings of the SOUTHEASTCON '91*, pages 201–204, IEEE-CS, IEEE, Williamsburgh, VA, April 1991.

[Popek 81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: a network transparent, high reliability distributed system. In *Proceedings 8th ACM Symposium on Operating System Principles*, Pacific Grove, CA, December 1981.

[Raita 87] T. Raita and J. Teuhola. Predictive text compression by hashing. In *Proceedings of the 10th Annual ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 223–233, New Orleans, June 1987.

[Ramachandran 88] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. *Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory.* Technical Report GIT-ICS-88/23, Atlanta, GA, June 1988.

[Rawdon 91] Michael Rawdon, Harvey Delery, Robert Driskill, Edward Blakes, and Brett Fleisch. Evaluation tools for distributed shared memory. In *IEEE Proceedings of the SOUTHEASTCON '91*, pages 198–200, IEEE-CS, IEEE, Williamsburgh, VA, April 1991.

[Salem 90] Kenneth Salem and Hector Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[Storer 88] J. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, Maryland, 1988.

[Walker 83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings Ninth ACM Symposium on Operating System Principles*, pages 49–70, Bretton Woods, NH, October 1983.

[Williams 90] R. Williams. *Adaptive Data Compression.* Kluwer Books, Norwell Ma, 1990.

[Ziv 77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.