Distributed Shared Memory Using Reflective Memory: The LAM System

Roger Denton
Encore Computer Corporation
Plantation, Florida

Theodore Johnson
Dept. of Computer and Information Science
University of Florida
Gainesville, Florida

Abstract

In the past, acceptance of Distributed Shared Memory (DSM) systems suffered because of poor performance. Performance has primarily been limited by the availability of an interconnect media whose properties are similar to those required by a DSM system. In an effort to compensate for inappropriate interfaces, significant research has been devoted to maximizing the utilization of the available interconnect.

A class of interconnect media is available that begins to address the inhibitors to DSM performance. These interconnect media have one essential property; the interface is mapped into the address space of the process participating in the DSM system.

This project focuses on the development of the LAM DSM system. LAM is a well-balanced hybrid DSM system implemented using Reflective Memory interconnect hardware with a software consistency policy and interface. A distinctive property of this system is that access time is directly proportional to the number of bytes accessed and unrelated to the size of the data structure being accessed. The system architecture, interface and results are described. Additionally, a case is made for the inherent advantage a memory-mapped interconnect has over an interconnect accessed through the operating system and network protocols.

1 Introduction

The nature of the problems being presented to computers are increasingly complex (e.g., the Grand Challenge problems). While single processor and multiprocessor machines can be expected to continue making incremental performance gains, distributed systems have the potential to introduce scalable increases in available performance. One of the fundamental components of a useful distributed computer system is a distributed interprocess communications mechanism.

To be useful, the communications mechanism must have the following properties:

- scalable
- high throughput
- low latency
- intuitive programming interface.

The goal of this project was to develop such a communications mechanism in the form of a Distributed Shared Memory (DSM) system.

In general, the communications mechanism in distributed systems is either a message passing (MP) system or a DSM system. Numerous studies have been done advocating MP systems as the mechanism of choice; a seemingly equal number of studies have been done resulting in proof that DSM systems are superior. By now it is clear that the choice of DSM or MP in the design of a given system depends on a number of factors peculiar to the system being designed.

In the past one primary factor in the choice between DSM and MP models was the underlying communication hardware [LeB92]. MP systems made few assumptions regarding the support hardware. In contrast, the performance of a DSM system is directly proportional to the performance of the underlying interconnect hardware. This factor has led to performance being listed as one of the primary advantages of MP over DSM systems. This also led to DSM systems being built on top of an MP interface/hardware.

Recently, the parameters associated with the premises regarding performance have evolved. Commercially available communications mechanisms routinely exceed 1Gb/s in speed (SCI [Gus92], ATM) and in addition, some hardware supports mapping the interconnect hardware into the address space of the process participating in the DSM system (e.g., Reflective

Memory[1]). These developments encourage a new generation of higher performance DSM systems.

This project focuses on the development of a DSM system meeting the previously mentioned criteria that utilizes a high-speed memory-mapped communications medium—Reflective Memory (RM).

2 Related Work

Over the last decade extensive research has been done in the area of DSM with a number of systems being built in both the research and commercial community. Below you will find information related to prior DSM implementations and observations on the trends found with attention drawn to key conclusions.

Previous research in the area of DSM has focused in two areas; the development of software systems utilizing readily available hardware, and hardware systems attempting to develop hardware specifically for solving a set of DSM issues. A grey area, hybrid systems, utilize both special purpose hardware and software to attack DSM issues. An extensive bibliography of DSM systems and related work can be found in [Esk96].

A number of significant software DSM systems have been developed. Amber [Cha89] took the approach of migrating the process to the data as well as the data to the process in order to reduce messaging. Clouds [Ram89] was one of the first systems to treat the shared data as an object. Linda [Ahu86] used compiler inserted DSM primitives to support parallelizing the application. Midway [Ber93] introduced entry consistency. Munin [Car91] implemented a variety of coherence policies that were selected by the user and enforced by Munin. Munin also developed facilities to support recovery in the face of system failure. Software DSM system research has tended to concentrate on the development of increasingly relaxed consistency schemes and attempts to increase effective throughput (e.g., multiple writers).

Hardware DSM systems are fewer as would be expected given the additional resources required to develop a hardware system. DASH [Len92] was one of the early hardware DSM systems, utilizing a distributed directory for cache coherence. Alewife [Aga95] supports coherent DSM and message passing interfaces. SCI [Gus92] offers a memory mapped interconnect interface and directory based cache coherence protocols. S3.mp [Now95] is attempting to produce a system capable of scaling from a few to several thousand processors.

Hybrid DSM systems are becoming more prevalent and are appealing because of the opportunity of performing operations that are common or time-critical in hardware and implementing the remaining logic in software—as we've all noticed, software is considerably easier to update than hardware. FLASH [Kus94], PLUS [Bis90] and SHRIMP [Blu94] implement DSM using a hybrid approach. LAM is also a hybrid system. LAM differs from the previously mentioned hybrid systems in that the RM hardware is considerably less complex. The reduced complexity of the hardware is not transferred directly to more complex software indicating an overall reduction in system complexity. Simpler hardware does not significantly impact performance as will be shown later in this paper.

A common thread through many of the previous DSM research projects was the search for improvements in the consistency model. In [Li89] it is stated that one of the most fundamental design decisions made in the development of a DSM system is the choice of a consistency policy. [Nit91] provides an intuitive definition of major consistency models.

2.1 DSM Performance Improvement

In [Kon95] a seemingly obvious conclusion is stated, DSM systems perform better with hardware assist. In this particular case they were working with the Cashmere system [Kon95] and a variety of network interfaces that allowed the shared memory to be mapped into the address space of the cooperating processors.

The authors of TreadMarks [Kel94] quantify the overhead incurred with interfaces that are not mapped into the local address space (e.g., ATM, Ethernet, ...). The TreadMarks group found that in their case the majority of the overhead is incurred in the operating system (in their case, Unix^{TM}) interprocess communication primitives and network protocols. Additionally, they provide measurements comparing the performance of TreadMarks using a 10 Mb/s Ethernet and a 100 Mb/s ATM LAN. Not surprisingly, a 10:1 performance increase was not seen; the average was closer to 2:1. Given that the operating system overhead would remain constant in the two cases, the implication is that the network protocols (UDP over Ethernet and AAL3/4 over the ATM LAN) are the primary bottleneck preventing the system performance from scaling with the performance of the underlying communication medium. [Ber93] corroborates these ratios with similar work done under their Midway DSM system.

[1]Reflective Memory is a trademark of Encore Computer Corporation

Clearly, significant improvement is possible in the area of scalability and effective use of available bandwidth. Since most (up to 78 percent [Blu94]) of the lost bandwidth is associated with the interface and associated protocols an efficient way to regain the lost bandwidth is through the design of a memory mapped interface. The RM/LAM combination provides an efficient method for taking advantage of this potential for performance improvement.

2.2 DSM Trends

While surveying previously sited DSM work a few trends were consistently noted.

- In-kernel implementations or fast user-level implementation are necessary for an efficient DSM implementation.
- Process synchronization must be kept to a minimum. Synchronization calls are expensive and will likely have secondary (local cache, TLB management, interrupt overhead) effects on the processors preempted to deal with synchronization. This requirement to minimize synchronization operations should influence the design of the DSM system as well as the ultimate application.
- Early design decisions dramatically impact the performance of a DSM implementation. Decisions related to hardware, software, hybrid implementation, coherence models, and granularity of the coherence model are critical.
- DSM implementations utilizing standard network interfaces and protocols will be limited by the performance of the protocol and interface.
- Hardware DSM implementations are faster than software and hybrids can be, on average, almost as fast as pure hardware implementation with significantly less complexity.

3 LAM Environment

Development and measurement of LAM was accomplished at Encore Computer Corporation using a four node Encore computer system. This multi-node system was configured as shown in Figure 1.

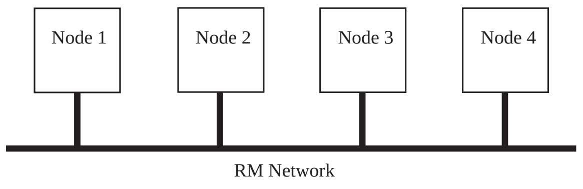

Figure 1 — System configuration

Each node contained 4 processors arranged in a symmetric, shared memory multiprocessor architecture. Each node is controlled by an autonomous operating system (Unix).

3.1 RM Hardware Overview

RM is a memory mapped interconnect media. Below are a few salient RM properties.

- 145MB/s interconnect network (the RM network)
- 8 nodes per network (multiple networks per node are possible)
- 16 RM networks may be interconnected
- RM network distances supported:
  - 80 feet with copper cables
  - 240 feet with coaxial cable
  - 15000 feet with fiber optic cable
- 4KB RM window (page) size
- 64bit bus
- word reflection or block reflection
- processors not interrupted by RM activity.

Only memory writes are transmitted on the RM network. Writes are causally ordered on the RM network. Reads are not transmitted on the RM network; they are satisfied from the memory on the RM board. Reads and writes from RM space are not cached. Additionally, RM memory is not strictly consistent across nodes.

Each RM board has three ports to memory, one for the RM network, one for the I/O bus and one for the processor bus. Arbitration logic and speed matching buffers are used to manage the traffic between the busses. The RM board supports block transfer operations so that an I/O controller (e.g., SCSI) may transfer data directly to RM. Processor test and set instructions are not supported across RM since there is no inherent provision for freezing bus access and synchronizing all nodes on the RM network.

Control registers are primarily initialized at system boot time. A device driver is installed by the operating system at system boot time. This driver is primarily responsible for error handling so that processes utilizing RM do not need to be concerned with handling RM (board and network) errors.

RM windows provide a method to control mapping of the RM space; they have no other effect on the RM space. Window control allows determination of whether or not a given window is reflected and also allows mapping transmit and receive locations to different addresses. This ability to map transmit and receive to different addresses is a fundamental property used in the implementation of RM synchronization primitives. Additionally, a window may be mapped for peer to peer, multicast or broadcast transmission.

3.2 RM Software Environment

Operating system support for mapping RM pages into the address space of the process using LAM is required. This support is provided by versions of standard Unix system calls (shmget and shmat) that have been modified to support RM. Once RM is mapped into the process address space it is manipulated with memory access (e.g., load and store) instructions. RM pages that are mapped into the process virtual address space are protected (but not paged or replaced) by the same virtual memory support that protects local pages.

RM has been the interconnect media for a number of systems at Encore and recently, DEC [Gil96]. Projects employing RM have included a disk subsystem cache, a database distributed lock manager [Ald95] and a distributed, fault-tolerant filesystem [Vek95]. Obviously, each of these require coherent access to control and data space. In each case the problem was solved with an implementation designed specifically for the problem at hand. To date, there has been no general service implemented to support coherent shared memory across RM.

LAM was designed to fill this functional gap in RM services.

4 LAM Architecture

LAM is an implementation of coherent distributed shared memory. LAM was designed to be scalable and provide an intuitive interface to the programmer. Essentially, LAM must manage allocation of RM space and synchronize access to this shared space.

In this section the architectural properties of LAM are described.

4.1 Architectural Properties

In terms of the DSM classification system provided in [Pro95], LAM has the following architectural properties.

DSM implementation: hybrid with library routines. LAM uses RM hardware as the interconnect media. The library routines are linked into the application and allow the programmer to allocate and synchronize shared memory from RM space.

Shared data organization: data structure. LAM allows the programmer to allocate and share data structures of any size that fits within the available memory.

Granularity of coherence unit: data structure. In fact, LAM has no knowledge of the content or structure of the data; it only knows the size of the data structure, the structure is imposed by the application. Each allocated data structure is guaranteed to be consistent using LAM primitives. Maximum parallelism can be obtained by defining the size and content of the data structures to maximize data availability and minimize synchronization events.

DSM algorithm employed: multiple reader, multiple writer. Any cooperating process on any participating node may initiate a read or write operation at any time.

Responsibility for DSM management: distributed. Each node has access to all control information.

Consistency model: entry. LAM uses acquire and release primitives to achieve memory consistency. The shared data structure is guaranteed to be coherent and available for exclusive update after the acquire call to LAM has completed.

Coherence policy: write-update. RM is responsible for this portion of coherence management. As RM locations are updated locally they are staged for transmission (assuming they are shared locations) on the RM network to update the local RM space of the participating nodes.

4.2 Consistency Implementation

Fundamentally, although the RM network is causally ordered, strict consistency of memory at each processor is not guaranteed because of asynchronous, non-instantaneous communication. In LAM, mutual exclusion and entry consistency is implemented utilizing properties of the RM hardware.

A distributed lock is associated with each LAM structure. By convention, the process owning the lock owns the LAM structure.

As previously mentioned, RM allows mapping transmit (write) and receive (read) operations to different addresses. Additionally, the RM network arbitration logic guarantees that RM network traffic is causally ordered with respect to the nodes attached. This allows for an efficient implementation of distributed locks as described below.

LAM allocates an area of RM for locks. This area is configured to have the receive/transmit locations at different addresses. A node bids for an apparently available lock by setting a word (a processor test-and-set instruction first acquires a node local guard lock to achieve efficient nodal mutual exclusion) associated with its node id in the lock structure. When this transmission appears in the receive window the lock word is checked for competing bids. RM ordering ensures that all bids that were present at the time this process made a bid are visible at the time this check is made. If no other nodes have bid for the lock then the bidding processor has acquired the lock and may enter the mutual exclusion region. If other nodes have bid for the lock then the competing nodes enter a prioritized retry algorithm (each node clears its bid and retries in a few microseconds; the number of microseconds is based on the node identification number and in a four node system will be less than 5 microseconds) until one node successfully acquires the lock. RM locks are released by clearing the RM bid word and then the processor local guard lock.

LAM uses such locks (transparent to the programmer) as part of the acquire and release code. Since writes are causally ordered on the RM network, if a node is able to obtain a lock then it must also be true that writes to the shared data structure preceding the acquisition of the lock must have been stored in the local copy of the shared data structure. Again, due to write ordering on the RM network, once the lock is acquired this also guarantees that the associated LAM shared data structure is globally consistent. In [Bir87] a similar scheme using causal updates to guarantee consistency is discussed.

4.3 Internal Structure

LAM manages three chunks of RM space; data, control, and lock space.

Lock pages contain the node local and distributed locks used to maintain shared data structure consistency. One lock is required for each LAM shared data structure.

Control pages contain information related to allocating, locating and updating LAM shared data structures. The process allocating a shared structure provides an integer tag uniquely identifying the shared data structure. This tag is used by other processes to attach to the shared structure. One control entry is required for each LAM shared data structure.

Data pages contain the shared data. LAM has no knowledge regarding the structure or content of the shared data. LAM structures may cross page (and window) boundaries and no particular byte alignment is required. LAM simply manages the allocation and deallocation of the data pages.

Since the lock and control pages must be shared across nodes they reside in RM space. RM spinlocks protect these LAM control areas to maintain consistency.

4.4 External Interfaces

One of the design goals was to produce an intuitive, programmer friendly interface. LAM has an interface that is certainly simple and hopefully intuitive. Each entry point is listed below:

- lam_init() — initialize the LAM system. Called once by each participating process.
- lam_alloc() — allocate a LAM shared data structure.
- lam_acquire() — obtain exclusive access to a LAM shared data structure.
- lam_release() — relinquish exclusive access to a LAM shared data structure.
- lam_free() — remove a LAM shared data structure from the global pool.
- lam_retire() — withdraw this process from the LAM system.

5 Performance Model

The most common sequence of operations within a program using LAM will be a call to lam_acquire(), followed by some sequence of data access, followed by a call to lam_release(). This sequence may be described with the following relationship.

$$\begin{aligned} T_{access} &= T_{LAM} + T_{RM} * Bytes_{accessed} \\ T_{LAM} &= T_{lam\_acquire()} + T_{lam\_release()} \end{aligned}$$

Where T_{access} is the total time required to acquire, manipulate, and release the data. T_{LAM} is the time associated with LAM overhead. T_{RM} is the time required to read or write a RM location (see Table 1). $T_{lam\_acquire()}$ and $T_{lam\_release()}$ can be further dissected into the relationships given below.

$$\begin{aligned} T_{lam\_acquire()} &= T_{local\_acquire} + D_{RM} * \\ &\quad (2T_{RMread} + T_{RMwrite}) \\ T_{lam\_release()} &= T_{local\_release} + D_{RM} * T_{RMwrite} \end{aligned}$$

Where $T_{local\_acquire}$ and $T_{local\_release}$ is the time required to acquire or release a node local lock. T_{RMread} and $T_{RMwrite}$ is the time required to read or write a RM location. D_{RM} is the RM coefficient of delay as described in the following paragraph.

Note that an equivalent set of relationships for a DSM utilizing a non-memory mapped interconnect would also contain terms for operating system service calls and protocol execution. Each of these terms would be relatively expensive in terms of time. LAM makes no system calls after system initialization and the protocol is limited to library calls to lam_acquire() and lam_release(). LAM/RM impose no software overhead for propagating writes to other nodes.

Also, note these relationships imply that the time required to access one byte in a small data structure will be equivalent to the time required to access one byte in a much larger data structure.

The time required to complete each RM operation is primarily dependent on the available capacity of the RM network. As the latency through the RM network increases T_{access} will increase proportionately. With respect to this performance model, the RM delay related to increased latency (traffic) on the RM network may be expressed as a delay coefficient, D_{RM}. The value of D_{RM} will range from 1 (when RM latency is at its minimum) to a larger value as latency increases.

In an attempt to measure D_{RM}, a program was written to provide a configurable background load on the RM network. Measurements of latency, throughput and LAM overhead were taken with various background loads.

Three of the four nodes in the system (the fourth was used to run the timing programs) were dedicated to flood the RM network—each node running 20 copies (60 copies total) of the load generation program. The 12 processors in the three nodes were 100 percent utilized during the run. This load did not significantly affect (less than 10 percent) the RM network latency in any of the measurements. This implies that in this 4 node configuration the RM coefficient of delay (D_{RM}) was equal to 1.

6 Results

Measurements of LAM fundamental quantities (latency and overhead) are provided followed by results related to scaling and speedup.

6.1 Latency

A relatively simple program was developed to measure the rate at which RM reads and writes were processed by the system. Recall that RM reads and writes are not cached by the processor. This was primarily a measurement of the RM system since LAM simply provided the memory allocator. These measurements were taken using one processor on one node. The memory accesses were timed by sequentially accessing each word in a contiguous array of 64K bytes. Table 1 shows the amount of time required to perform the indicated operation on one word (32 bits, 4 bytes).

Table 1 — Memory latency measurements

Operation Type	Time (usec)
RM writes	0.20
RM reads	0.37

6.2 Overhead

In this section timings are provided for the fundamental LAM primitives. (T_{LAM}). Also, a significant optimization to the LAM primitives is discussed.

A program was written to time the relative cost of LAM operations as the size of the LAM shared data structure varied. This was accomplished by timing multiple calls to lam_acquire() and lam_release() then reporting the average cost. Table 2 provides timing information on the original as well as the optimized version of the LAM primitives (T_{LAM}).

Table 2 — LAM primitive timings

	Original time	Optimized time
T_{LAM}	43.0usec	14.7usec

The initial timings of LAM primitives indicated significant opportunity for optimization. As previously described, lam_acquire() is essentially a mutual exclusion entry point and lam_release() is a mutual exclusion release point additionally, each contains a component of time related to the number of instructions required to execute the LAM code and the associated mutual exclusion entry or exit. With an understanding of the underlying media and the algorithms employed, there was reason to believe that the acquire and release times could be significantly decreased. The source code was streamlined and flattened and the performance improvements in Table 2 were measured. Please note that unless specified otherwise all results in this document were recorded using the optimized primitives.

The value of this optimization is apparent in Figure 2 — access to less than 1024 bytes of a LAM structure requires significantly less time. Figure 2 shows the amount of time required to acquire an uncontested LAM shared data structure, access a given number of bytes in the structure, and then release the structure. Results are shown for the original unoptimized primitives, the optimized primitives and predicted results derived from a linear regression model. The linear regression model was calculated to be:

$$T_{access} = 14.7usec + 0.20usec * Bytes_{written}.$$

Note that the predicted results are well aligned with the measured values for the optimized primitives.

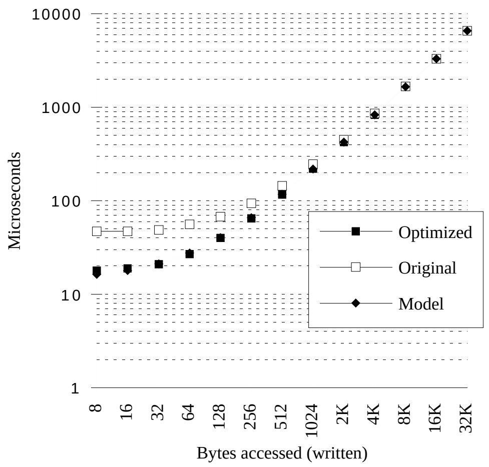

Figure 2 — LAM throughput measurements

It is important to remember that access time is directly proportional to the number of bytes accessed and is unrelated to the size of the data structure being accessed.

6.2.1 Overhead Relative to Data Accesses

LAM overhead as a percentage of total data access time is inversely proportional to the amount of data accessed in the shared structure. Overhead was calculated using the relationship:

$$Overhead_{percent} = \frac{T_{LAM}}{T_{access}} * 100.$$

Where T_{LAM} is the time required to execute lam_acquire() ($T_{acquire}$) plus the time required to execute lam_release() ($T_{release}$). T_{LAM} equals 14.7usec. T_{access} was defined in the previous chapter. The results of this calculation for a variety of access ranges is given in Figure 3.

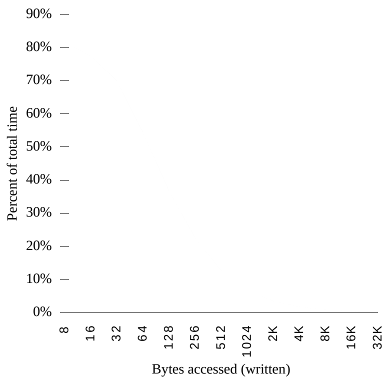

Figure 3 — LAM overhead as a function of shared data access range

6.3 Scaling

Scaling is the ability of a system or resource to accept and complete incremental work. (An alternative definition of scaling has been given as the ability to produce greater precision results in the same period of time given additional resources.) Additional work should be completed in less time until the system or resource is depleted of capacity.

Matrix multiplication is a fairly common utility used to measure parallel systems in general and DSM systems in particular. With this in mind, the code for this matrix multiplication program was derived from that used in the CRL project [Joh95]. Relatively minor modifications were required to replace the CRL interface with the appropriate LAM calls. The matrix sizes were increased to 512x512 in order to increase the runtime. The matrix multiplication program was measured in three environments; running on the local system in one process on one node, running in 1-32 processes across 4 nodes (16 processors) using LAM services, and running in 1-32 processes across 4 nodes using LAM services with the source matrices marked as read only.

Figure 4 is a graph of the multiprocess scaling properties observed running the matrix multiplication program with LAM.

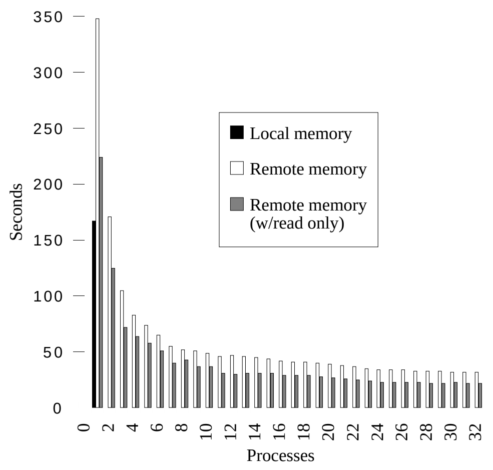

Figure 4 — Multiprocess scaling results

6.3.1 Speedup

Speedup is described by the relationship:

$$Speedup = \frac{T_{ref}}{T_{LAM}}.$$

Where T_{ref} is the reference time. In this case, T_{ref} is either the time required to run the application in local memory or the time required to run the application in LAM memory with one process.

Speedup measurements show that for this matrix multiplication application two or more processes will provide lower overall elapsed time (compared to the local case) and that scaling continued to improve across all 32 processes. Figure 5 is a graph of the speedup obtained using LAM compared to local memory and LAM memory.

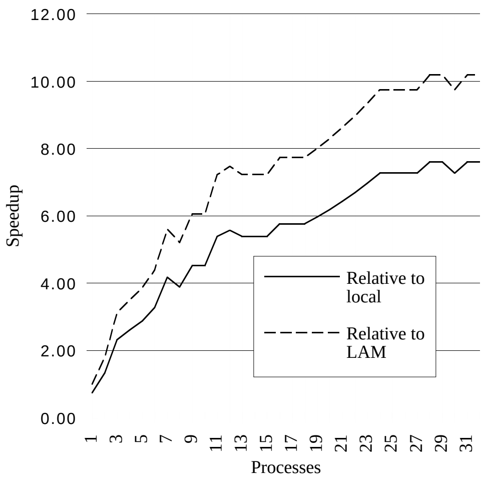

Figure 5 — Speedup

7 Conclusion

In this project a high-performance, hybrid DSM system was implemented and measured.

The measured results show that this hybrid approach results in good scaling, high throughput and low latency. Additionally, the overhead imposed by LAM is relatively small (see Figure 2).

7.1 Potential Areas for Future Work

The current implementation has a number of shortcomings.

- There is one LAM memory pool. Multiple pools would be useful for prioritized partitioning of LAM shared space.
- The shared data structure identification tags must be known by each participating process. A distributed data structure identification service could be implemented to remove this limitation.

In the current implementation it would be interesting to investigate the potential for more application supplied hints (e.g., a conditional acquire based on whether or not a shared data structure is available). The most direct way to accomplish this would be to port additional DSM applications to LAM with an eye out for any optimizations that would be generally useful.

A strictly consistent, transparent (to the application programmer) DSM system using RM could be implemented. This system could be integrated with the virtual memory subsystem of the operating system so that logical acquire and release operations could be done each time the page was accessed. Notification of the page being accessed could be done through manipulation of the processor page table entry control bits. False sharing [Bol93] could be a problem with this implementation given that the sharing would occur at the page level.

Investigation into a protocol allowing multiple writers (as in Munin [Car91] and TreadMarks [Kel94]) should increase parallelism and improve overall performance and efficiency.

Implementation of increasingly relaxed consistency protocols would improve performance. For example, a lazy entry protocol would take advantage of temporally local references, thereby reducing the incidence of expensive inter-node synchronization events and reducing RM network load.

Recovery in the presence of node failures would be relatively simple to add to LAM since each node could have a copy of all shared data necessary to synchronize a recovering node with its pre-failure state.

7.2 The Importance of Low Overhead to DSM Performance and Acceptance

Low overhead is directly related to increased performance and it is subtly related to the acceptance of the DSM system. Applications written for distributed systems are generally constructed to avoid distributed communications (as much as possible) and the associated drop in performance. If the overhead associated with distributed communications can be driven towards zero then this burden will be removed from the programmer, distributed applications will be easier to produce and therefore are more likely to be produced.

A DSM system imposing a significant additional burden on the programmer is unlikely to be accepted for reasons related to economy — increased software complexity translates directly to extended development cycles and an increased incidence of programming errors, both are costly. The LAM interface is simple, intuitive and provides a relatively straightforward implementation platform for developing or porting applications.

As previously mentioned, in [Kel94] and [Ber93] an effort was made to measure the overhead associated with DSM operations using interfaces requiring a network protocol and operating system intervention in DSM operations. In TreadMarks [Kel94] it was shown that from 3 to 17 percent of the total execution time was spent on communication. In Midway [Ber93] it was found that communication time varied between 6 and 26 percent of the total application time depending on the application and the interconnect media. Reports from the SHRIMP project [Blu95] show that “virtual-memory-mapped communication can reduce the send latency overhead by as much as 78 percent.” In addition to extending processing times this overhead also serves to effectively limit the useful bandwidth of a communication medium. [Min95] noted that a network interface capable of 48MB/s was only able to drive 20MB/s — the other 28MB/s was “lost” to overhead associated with the protocol and data path to the interface.

An interface that is mapped into the address space of the process avoids virtually all of the overhead associated with the operating system. An interconnect media that appears to be memory (operated upon with processor load and store instructions) minimizes the protocol overhead. RM has both properties; it is mapped into the process address space and it is accessed using memory access instructions.

In LAM there is little overhead associated with inter-node DSM communication as long as there is bandwidth available on the RM bus. Assuming similar inherent execution times for the DSM primitives this gives a system with a memory mapped DSM interconnect a significant performance advantage (up to 78 percent [Blu95]) over interconnect devices accessed through the operating system and protocols. This advantage is a result of the lower overhead associated with inter-node communication—load and store instructions will always be much faster than system calls followed by protocol processing.

References

[Aga95] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, Kubiatowicz, B. Lim, K. Mackenzie, D. Yeung, "The MIT Alewife Machine: Architecture and Performance," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[Ahu86] S. Ahuja, N. Carreiro, D. Gelernter, "Linda and Friends," *IEEE Computer*, vol. 19, no. 8, August 1986, pp 26–34.

[Ald95] M. Aldred, I. Gertner, S. McKellar, "A Distributed Lock Manager on Faul Tolerant MPP," *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, January 1995, pp 134-136

[Ber93] B.N. Bershad, M.J. Zekauskas, W.A. Sawdon, "The Midway Distributed Shared Memory System," *Digest of Papers COMPCON*, February 1993, p. 528–537.

[Bir87] K.P. Birman, T.A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, vol. 5, no. 1, February 1987, pp 47–76.

[Bis90] R. Bisiani, M. Ravishankar, "PLUS: A Distributed Shared Memory System," *Proceedings of the 17th International Symposium on Computer Architecture*, vol. 18, no. 2, May 1990, pp 115–124.

[Blu94] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994, pp 142–153.

[Blu95] M. Blumrich, C. Dubnicki, K. Li, M. Mesarina, "Virtual Memory Mapped Network Interfaces," *IEEE Micro*, vol. 15, no. 1, February 1995, pp 21–28.

[Bol93] W. Bolosky, M. Scott, "False Sharing and its Effect on Shared Memory Performance," *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, September 1993.

[Car91] J.Carter, J.Bennett, W.Zwaenepoel "Implementation and Performance of Munin," *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991, pp 152–164.

[Cha89], J. Chase, F. Amador, E. Lazowska, H. Levy, R. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors," *Proceedings of the 12th Symposium on Operating Systems Principles*, December 1989, pp 147–158.

[Esk96] M.R. Eskicioglu, "A Comprehensive Bibliography of Distributed Shared Memory," *IEEE Operating Systems Review*, January 1996.

[Gil96] R. Gillett, "Memory Channel Network for PCI," *IEEE Micro*, February 1996, pp 12–18.

[Gus92] D. Gustavson, "The Scalable Coherent Interface and Related Standards Projects," *IEEE Micro*, February 1992, pp 10–22.

[Hag92] E. Hagersten and A. Landin and S. Haridi, "DDM – A Cache-Only Memory Architecture,", *IEEE Computer*, vol. 25, no. 9, September 1992, pp 241–248.

[Joh95] K. Johnson, M. Kaashoek, D. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[Kus94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy, "The Stanford FLASH Multiprocessor," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994, pp 302–313.

[Kel94] P. Keleher and S. Dwarkadas and A. Cox and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory On Standard Workstations and Operating Systems," *Proceedings of the 1994 Winter USENIX Conference* January 1994, pp 115–131.

[Kon95] L. Kontothanassis, M. Scott, "Distributed Shared Memory for New Generation Networks," *University of Rochester Technical Report 578*, March 1995.

[LeB92] T. LeBlanc, E. Markatos, "Shared Memory vs. Message Passing in Shared-Memory Multiprocessors," *Fourth Symposium on Parallel and Distributed Processing*, December 1992.

[Len92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessey, M. Horowitz, M. Lam, "The Stanford DASH

Multiprocessor," *IEEE Computer*, March 1992, pp 63–79.

[Li89] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, November 1989, pp 321–359.

[Min95] R. Minnich, D. Burns, F. Hady, "The Memory-Integrated Network Interface," *IEEE Micro*, vol. 15, no. 1, February 1995, pp 11–20.

[Nit91] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*. vol. 24, no. 8, August 1991, pp 52–60.

[Now95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, S. Vishin, "The S3.mp Scalable Shared Memory Multiprocessor," *Proceedings of the 24th International Conference on Parallel Processing*, August 1995, pp I:1–10.

[Pro95] J. Protic, M. Tomasevic, V. Multinovic, "A Survey of Distributed Shared Memory Systems," *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, January 1995, pp 74–84.

[Ram88] U. Ramachandran, M. Ahamad, Y. Khalidi, "Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory," *Georgia Technical Institute Technical Report GIT-CS-88/23*, June 1988.

[Vek95] N. Vekiarides, "Fault-tolerant Disk Storage and File Systems Using Reflective Memory," *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, January 1995, pp 103-113.