Extending Distributed Shared Memory to Heterogeneous Environments

Songnian Zhou Michael Stumm Tim McInerney
University of Toronto
Toronto, Canada M5S 1A4

Abstract

Distributed Shared Memory, a high-level mechanism for interprocess communication in distributed systems, is receiving increased attention because of its perceived advantages over message passing mechanisms. In this paper, we take an existing algorithm that implements Distributed Shared Memory due to Li and extend it to a heterogeneous environment. We describe an implementation that runs on Sun and DEC Firefly multiprocessor workstations connected by Ethernet and study related implementation and performance issues. Based on measurements of the applications ported to our system, we conclude that heterogeneous distributed shared memory is not only feasible, but can also be comparable in performance to its homogeneous counterpart.

1 Introduction

Distributed shared memory (DSM) is a mechanism for interprocess communication in distributed systems. In the distributed shared memory model, processes running on separate hosts can access a shared address space through two basic operations:

```
data = read( address );
write( address, data );
```

Read returns the data item referenced by **address**, and **write** sets the contents referenced by **address** to the value of **data**. The underlying distributed shared memory system provides its clients with a shared, coherent memory address space. Each client can access any memory location in the shared address space at any time and see values last written by any client. The primary advantage of DSM is the simpler abstraction it provides to the application programmer, making it the focus of recent study and implementation efforts [4, 6, 7, 9, 10, 11, 12, 13, 16]. The abstraction is one the programmer already understands well, since the access protocol is consistent with the way sequential applications access data. The communication mechanism is entirely hidden from the application writer so that the programmer need not be conscious of data movement between processes and complex data structures can be passed by reference, requiring no packing and unpacking.

In principle, the performance of applications that use DSM is expected to be worse than if they use message passing directly, since message passing is a direct extension to the underlying communication mechanism of the system, and since DSM is implemented as a separate layer between the application and a message passing system. However, several implementations of DSM algorithms have demonstrated that DSM can be competitive to message passing in terms of performance [4, 12, 7]. In fact, for some existing applications, we have found that DSM can result in superior performance. This is possible for two reasons. First, for many DSM algorithms, data is moved between hosts in large blocks. Therefore, if the application exhibits a reasonable degree of locality in its data accesses, communication overhead is amortized over multiple memory accesses, reducing overall communication requirements. Second, many (distributed) parallel applications execute in phases, where each compute phase is preceded by a data exchange phase. The time needed for the data exchange phase is often dictated by the throughput of existing communication bottlenecks. In contrast, many DSM algorithms move data on demand as they are being accessed, eliminating the data exchange phase, spreading the communication load over a longer period of time, and allowing for a greater degree of concurrency.

The most widely known algorithm for implementing DSM is due to Li [11], which is well suited for a large class of algorithms [12]. In Li's algorithm, the shared address space is partitioned into pages and copies of these pages are distributed among the processors, following a multiple-reader/single-writer (MRSW) protocol: Pages that are marked *read-only* can be replicated and may reside in the memory of several hosts, but a page being written to can reside only in the memory of one host.

One advantage of Li's algorithm is that it can easily be integrated into the virtual memory of the host operating system. If a shared memory page is held locally at a host, it can be mapped into the application's virtual address space on that host and therefore be accessed using normal machine instructions for accessing memory. An access to a block not held locally triggers a page fault, passing control to a fault handler. The fault handler then communicates with the remote hosts in order to obtain a valid copy of the data block before mapping it into the application's address space. Whenever a data block is migrated away from a host, it is removed from any local address space it has been mapped into. Similarly, whenever a processor attempts to write to a page for which it does not have a local copy marked as *writable*, a page fault occurs and the local fault handler communicates with the other hosts (after having obtained a copy of the page, if necessary) to invalidate all other copies in the system, before marking the local copy as writable and allowing the faulted process to continue. This protocol is similar to the write-invalidate algorithms used for cache consistency in shared-memory multiprocessors, except that the basic unit is a page instead of a cache line.

In this paper, we extend distributed shared memory to a heterogeneous environment and study the related implementation and performance issues. Our primary motivation for this work is the fact that heterogeneity exists in many (if not most) computing environments. Heterogeneity is usually unavoidable because a specific hardware and its software is often designed for a particu-

CH2878-7/90/0000/0030$01.00 © 1990 IEEE

lar application domain. For example, supercomputers and multiprocessors are good at compute-intensive applications, but often poor at user interfaces. Personal computers and workstations, on the other hand, usually have very good user interfaces. Heterogeneous distributed shared memory is a mechanism by which the advantages of both systems can be obtained in an integrated system, allowing applications to exploit the best of both.

As a practical research effort, we have implemented a heterogeneous distributed shared memory system that runs on a network of SMI Sun workstations and DEC Firefly multiprocessors. The heterogeneity of this distributed system poses a number of challenges that need to be addressed, including different data representations, page sizes, operating systems, etc. We discuss the main issues and solutions of building a heterogeneous distributed shared memory system and describe our implementation in Section 2. Its measured performance is analyzed in Section 3.

2 Mermaid: A Prototype

2.1 System Overview

To study the implementation and performance issues of heterogeneous distributed shared memory, we have developed a prototype system, called Mermaid. In general, the architecture, operating system, and language environment on workstations and compute servers may all be different. In order to explore the difficult issues arising from heterogeneity, we wanted to select machines that are sufficiently different. Based on suitability and availability, Sun/UNIX workstations and DEC's experimental Firefly multiprocessors were chosen. Sun-3 workstations are based on M68020 CPU's and run Sun's version of the UNIX operating system. The system programming language is C. On the other hand, the Firefly, developed at DEC's System Research Center, is a small-scale multiprocessor workstation based on DEC's CVAX processors[1] [18]. Each Firefly may have up to 7 processors sharing physical memory. The operating system on Firefly is Topaz, with Modula II+ being the system programming language.

The overall system architecture of Mermaid is similar to that of the IVY system developed at Yale [11]. It consists of 1) a thread management module, 2) a shared memory management module, and 3) a remote operations module, as shown in Figure 1. The *thread management module* provides operations for thread creation, termination, scheduling, as well as synchronization primitives. The *shared memory module* is called to allocate and deallocate shared memory, and to handle page faults. It uses a page table for the shared address space to maintain data consistency. The above two modules are supported by the *remote operations module*, which implements a simple request-response protocol for communication between the hosts.

We chose to implement Mermaid at the user level, as a library package to be linked into application programs using DSM. Although a kernel-level implementation would be more efficient, the difference in performance is not expected to affect applications performance significantly, as evidenced by the low overhead of Mermaid to be discussed in Section 3. More importantly, a user level implementation has a number of advantages. First, it is more flexible and easier to implement; experimentation may be carried out without rebooting the hosts. Second, several DSM packages can be provided to the applications on the same system. Our analysis of the performance of applications using different shared data algorithms revealed that the correct choice of algo-

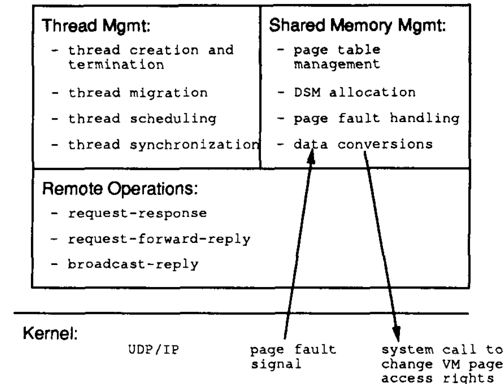

Figure 1: Structure of the Mermaid system

rithm was often dictated by the memory access behavior of the application [16]. It is therefore desirable to provide multiple DSM systems employing different algorithms for applications to choose from. A user-level implementation makes this much easier. Finally, a user-level DSM system is more portable, although some small changes to the operating system kernel are still needed.

For example, Mermaid requires kernel support for setting the access permissions of memory pages from the user level, so that a memory access fault is generated if a non-resident page is accessed on a host. An new system call was added to Sun/UNIX and Topaz for this purpose. A second change to the operating system kernel was to allow user-level fault handlers to be register with the kernel so that they could be invoked on DSM page faults. No other kernel changes were necessary.

The heterogeneous system base of Sun and Firefly systems presented a number of problems in the implementation of Mermaid. We discuss them in the following subsections.

2.2 Basic Support

Communication Substrate Distributed shared memory typically operates in a request-response mode. For instance, when a page fault occurs, the fault handler sends a page request to the shared memory manager, which either supplies the page, or forwards the request to the owner on another host. The most suitable protocol for the remote operations module is therefore a request-response protocol, with forwarding and multicast capabilities. Multicast is used for write invalidation. All the interactions between the memory and thread management modules on different hosts can be supported by this protocol. We considered using the remote procedure call facilities on Sun and Firefly. However, they are incompatible[2], and neither meets our requirements with respect to functionality (e.g., broadcast and forwarding) or performance. For example, data marshaling and unmarshaling are not needed, since the data being transferred (i.e. the pages) is not structured and since data conversions are performed at a higher level.

Being a higher level communications abstraction, distributed shared memory requires the support of an underlying message transport mechanism, which must be available on all of the machine types. The primary requirement of this mechanism is ef-

[1] More powerful machines would have been more appropriate to represent compute servers, had they been available. However, for our purpose, Fireflies are adequate machines to experiment with.

[2] They follow different design specifications, and the procedure call message formats, data representations, etc. are all different.

ficiency, since the specific interface semantics and reliability features needed by the thread and shared memory managers are implemented in the remote operation module of Mermaid. Comparing the two transport protocols implemented on both the Sun and Firefly systems, the stream oriented TCP/IP and the basic, datagram-oriented UDP/IP, the latter is the obvious choice. Unfortunately, only a subset of UDP's functionalities is implemented on the Firefly: message fragmentation and reassembly is not provided. This also makes the corresponding functionalities on the Sun unusable when communicating with Firefly. Since the message size used by Mermaid may be too large to fit in a single packet (e.g. the size of a Sun VM page is 8 Kbytes), we implemented fragmentation and reassembly at the user level.

Thread Support Distributed shared memory provides for an address space shared by several threads of execution, in a location-independent manner. It is therefore important that a thread mechanism be provided to the applications writers using DSM. Being a relatively new facility, however, threads sharing a common address space are not supported by many of the older operating systems, including Sun/UNIX. Mermaid therefore provides a thread module at the user level on the Sun, supporting thread creation, destruction, and suspension operations, and non-preemptive thread scheduling. Threads may be created in an application and later moved to other hosts. Alternatively, threads may be created on remote hosts directly. Since all the threads on a Sun host reside in a single address space, the suspension of one thread by the operating system scheduler (e.g., for *synchronous* I/O) makes other threads non-executable as well. This is typically not a problem, since parallel applications often allocate only one thread on each host. For the Firefly, a system-level thread package is available and is used directly by Mermaid.

Parallel executing threads need a way to synchronize. In principle, this could be supported by atomic instructions on shared memory locations. In practice, however, this would lead to repeated movement of (large) DSM pages between the hosts involved. We therefore implemented a separate distributed synchronization facility that provides for P and V operations and events more efficiently.

2.3 Data Conversion

When data is transferred from a host of one type to a host of another type, then it must be converted before it is accessed on the destination host. The unit of data that must be converted is a page, but the conversion itself must be based on the types of the data stored in the page. A simple example, depicted in Figure 2, demonstrates this. In this figure, two data structures, an integer and an array of four characters, are presented first in big-endian order [5] and then in little-endian order. To convert from big-endian to little-endian (or vice versa), the bytes of the integer must be swapped, but not those of the character array. This complicates one of the stated goals of distributed shared memory, namely transparency. The conversion process cannot simply be delegated to the DSM system because it is type specific and because the DSM system does not a priori know how the memory is being used. Hence, the application (which knows the layout of its memory) must either (i) perform all data conversions itself, or (ii) specify to the system the layout of each page. The first option raises the question of how the user-level conversion routine is called. The second option complicates the interface to the DSM system and requires the DSM system to store the page layout information for each page. Both options reduce the degree of transparency of the distributed shared memory system.

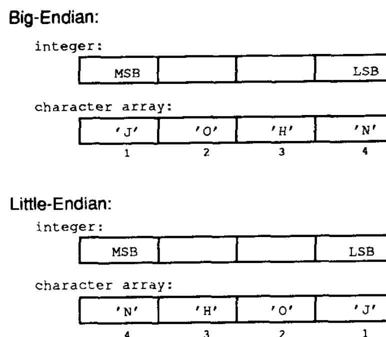

Figure 2: Big-Endian and Little-Endian Byte Ordering

Mermaid Conversion Mechanism The conversion mechanism we use in Mermaid is relatively simple. First, we require that a page contain data of one type only.[3] Second, the application programmer must provide the conversion routine for each user-defined data type in the application. Finally, the appropriate conversion routine is (automatically) invoked by the DSM system whenever a conversion is necessary, i.e. after a transfer between incompatible hosts.

To aid the programmer in achieving the first requirement, a special memory allocating subroutine similar to `malloc` is made available that has an additional argument identifying the type of data that will be stored in that part of memory. This subroutine assigns the allocated memory to pages in such a way that a page contains data of only one type.

The DSM page table entry also keeps additional information identifying the type of data maintained in that page and the amount of data allocated to the page. (the latter information allows for obvious optimizations when only a portion of a page is allocated, i.e. only that portion needs to be transferred and converted.) To allow the DSM system to invoke the correct routine, the application programmer must identify the conversion routines by way of a global (static) table containing a pointer to the appropriate routine for each page type.

The conversion routines are straightforward to write, given conversion routines for the basic data types, such as characters, integers, (single and double precision) floating point numbers. In the case of compound data structures, the conversion routine calls the appropriate conversion routine for each field. In the case of arrays, the conversion routine of the array type is called repeatedly.

Some implementations of DSM will also need to convert pointers a page when is converted. This is necessary if the memory allocated to DSM does not occupy the same sequence of addresses on each host type. To allow the application supplied conversion routine to do this, the type conversion routines are called with an additional argument containing the offset by which pointers must be modified. For example, if the DSM starts at virtual address `start1` on a host of one type and at virtual address `start2` on a host of another type, then pointers that are valid on the first host must be offset by `(start2-start1)` before they are valid on the second host. The DSM system determines the value of this offset during its initialization phase without application intervention. in our implementation, pointer conversions are not necessary, since the memory allocated to DSM starts at the same address on each

[3] This need not be a basic type provided by the programming language, but could be an application defined compound type.

host type.

Conversion Overhead We have found the performance impact of page-based data conversion to be not very significant compared to the other overheads of DSM. (See Section 3.1.) Conversion is necessary only when a page is migrated between hosts of *different* types, and the cost of converting a page is small compared to the overall cost of page migration. The number of necessary conversions can be kept to a minimum by transferring a page from a host of the same type whenever possible.

In general, the conversion process itself is simple. Character strings do not need to be converted. The conversion of integers is a matter of proper byte swapping. Conversion of (single and double precision) floating point numbers may be somewhat more complicated. For example, the IEEE format supports unnormalized numbers and special cases, such as infinity and NAN's, which are not supported by the Vax. These cases can be detected with two additional comparison operations. The positions and lengths of the exponent and matissa fields may be different (such is the case with IEEE and VAX), requiring bit manipulation operations.

Limitations The data conversion problem is complex and our solution is not entirely general, although our experience indicates that it is sufficient for many practical applications. We believe our solution is similar in complexity to the solution used by existing heterogeneous RPC schemes [17]. However, our solution does have a number of limitations. First, a memory page may contain data of only one type. Our memory allocator does this for the programmer, but it may lead to increased memory usage due to fragmentation, which in turn can lead to increased paging activity. At the same time, however, it also can reduce thrashing because the probability of page access contention is reduced.

Second, the size of each data type must be the same on each host, and the order of the fields within compound structures must be the same on each host. If this were not the case, then the mapping between pages on the two hosts would not be one-to-one; That is, it may not be possible for a structure to fit on a page in its entirety after the conversion or, conversely, some data from the following page may be needed in order to complete the conversion of the current page. Hence, the compilers used on the different host types must be compatible to some extent.

Third, entire pages are converted even though only a small portion of a page may be accessed before it is transferred away. As mentioned above, however, we have found that the cost of the page translation to be small compared to the overall migration cost. A page-based DSM system performs poorly with this type of access behavior in both the homogeneous and heterogeneous case.

Fourth, our conversion process is not entirely transparent because uses are required to supply the conversion routines for user-defined data types and a table specifying the mapping of data types to conversion routines. It is possible, however, to generate these automatically using a preprocessor on the user program. This is something we are currently looking in to.

Finally, floating point numbers can lose precision when they are converted. Since an application does not have direct control over how many times a page is migrated between hosts of different types and hence converted, the numerical accuracy of results may become questionable. In general, however, we do not consider this to be a problem for many environments; for example, in an environment consisting of workstations and computation servers, data is typically transferred once to the computation servers and then transferred back again at the end of the computation.

2.4 Page Sizes

The unit of data managed and transferred by DSM is a data block, which we call a *DSM page*. In a homogeneous DSM system, a DSM page has usually the same size as a native virtual memory (VM) page, so that the MMU hardware can be used to trigger a DSM page fault. Although a multiple of the native VM page size could be used for DSM pages, this might increase conflicts in data accesses from different hosts.

The situation becomes more complicated when multiple architectures with different page sizes are involved. We may use the largest VM page size for DSM pages. Since VM page sizes are most likely powers of 2, multiple smaller VM pages fit exactly in one DSM page; hence, they can be treated as a group on page faults. The potential drawback of such a *largest page size* algorithm is that more data than necessary may be moved between hosts with smaller VM page sizes. In severe cases, *page thrashing* may occur, when data items in the same DSM page are being updated by multiple hosts at the same time, causing large numbers of page transfers among the hosts without much progress in the execution of the application. While page thrashing may occur with any DSM page size, it is more likely with larger DSM page sizes, as different regions in the same page may be updated by threads on different hosts, causing page transfers that are not necessary with smaller pages. Such data access patterns are referred to as *false sharing*, and is a frequent cause of page thrashing.

One way to reduce data contention is to use the smallest VM page size for the DSM pages. We call the corresponding algorithm the *smallest page size* algorithm. If a page fault occurs on a host with a larger page size, multiple DSM pages will be moved to fill that (large) page. The actual algorithm must differentiate between many cases depending on the type of page fault (read or write), the page sizes of the requesting and the owner hosts, and how the page is currently being shared (what type of hosts have read/write accesses).

Typically, if page thrashing does not occur, more DSM page faults occur on hosts with small VM page sizes, resulting in more fault handling overhead and (small) page transfers. Although intermediate sizes are possible, the above two algorithms represent the two extremes of the page size algorithms. We have implemented both algorithms in Mermaid, and the performance comparison between them will be discussed in Section 3.3.

3 Performance Evaluation

Implementing distributed shared memory in a heterogeneous environment can bring performance benefits to applications, but it may also incur increased overhead. On the one hand, it is now possible to develop and start parallel applications on workstations, and be able to use the computing resources on more powerful machines by moving threads to them. On the other hand, the cost of doing message fragmentation and reassembly at the user level, of handling multiple VM page sizes, and of data conversions may offset the performance gains of parallel execution.

We have performed a number of measurement experiments on our prototype Mermaid system in order to study the impacts of heterogeneity on the performance of distributed shared memory systems. All measurements were performed on Sun3/60 workstations and Fireflies. The measured hosts were idle during the experiments, except for the activities being studied. The results we observed were very stable (except for the page thrashing cases to be discussed in Section 3.3), For all cases, the lowest values are reported.

	Sun	Firefly
Read	1.98	6.80
Write	2.04	6.70

Table 1: Costs of page fault handling (ms).

to / from	Sun	Firefly	Sun	Firefly
Sun	18	27	5.1	7.6
Firefly	25	33	7.3	6.7
page size	8 KB		1 KB	

Table 2: Cost of transferring a page (ms).

3.1 Overhead Assessment

Compared to physical shared memory, distributed shared memory has a number of additional overheads. In a user-level implementation, the access rights of the DSM pages have to be set, and DSM page faults have to be passed to the user level. Since data is no longer physically shared, DSM pages need to be moved between hosts upon page faults, typically over a slow, bit-serial network, such as the Ethernet. The allocation of shared memory and thread synchronization and scheduling also introduce overhead, but they are relatively small compared to other overheads. Finally, for heterogeneous systems, the costs of data conversions and the page size algorithm might be significant.

The basic costs of handling a page fault are shown in Table 1. Included are the invocation of the user-level handler, the DSM page table processing, and the request message transmission time. The values of a few milliseconds are considered to be quite small. The costs on the Fireflies are higher, due to the higher user-level message processing cost and the locking of data structures, which is necessary in shared memory multiprocessors.

Table 2 shows the costs of transferring 8 KB and 1 KB pages between hosts. The higher cost when the Firefly is involved is partially due to user level message fragmentation and reassembly processing. The costs for 8 KB transfers are only about three times that of 1 KB transfers. This is due to the fixed costs of message transport. Hence, in the absence of page thrashing, larger DSM pages incur fewer page faults and lower data transfer overhead.

For heterogeneous shared memory, the cost of data conversion must be considered. We measured the costs of converting a page of integers, shorts, or floating point numbers (single and double precision) on a Firefly (See Table 3). For integers and shorts, only byte swapping is needed, whereas for floating point numbers, extra checks for extreme values in the IEEE representation (e.g., underflow, negative zero, NAN) are also necessary. In all of the cases, the conversion costs are only a fraction of the page transfer costs. We also measured the conversion costs of user-defined data

	8 KB page	1 KB page
int	10.9	1.3
short	11.0	1.3
float	21.6	2.7
double	28.9	3.6

Table 3: Costs of data conversions (ms).

	Sun→Sun		Ffly→Sun		Sun→Ffly		Ffly→Ffly	
	R	W	R	W	R	W	R	W
R/M→O	26.4	26.7	47.7	48.3	56.3	47.8	46.5	46.4
R→M/O	29.6	27.9	50.9	51.6	58.6	59.4	49.6	49.1
R→M→O	31.7	31.3	54.7	55.5	61.9	61.3	54.4	53.6

R = Requester host; M = Manager host; O = Page Owner
R/M: Requestor and Manager are on the same host.

Table 4: End-to-end page fault delays for 8 KB pages (ms).

structures, and found them to be comparable to the above. For example, converting an 8 KB page containing records of three integers, 3 floats, and 4 shorts took 19.6 milliseconds on a Sun3/60.

To consider the combined effects of the above overhead costs, we show the end-to-end page fault delays for different types of hosts under different scenarios in Table 4. (In this implementation, each page has a fixed manager that can identify the owner and the copy set of the page. A request to transfer a page is always passed by the manager.) The cost for (integer) data conversion is included when the two hosts (Requestor and Owner) are of different types. The measurements are based on the largest page size algorithm, hence the values are for 8 KB pages only. The costs for read and write page faults were found to be very similar. The DSM page fault delay is comparable to that of a VM page fault involving a disk seek. The costs of page faults involving both Sun and Firefly are very comparable to the homogeneous cases. As with traditional VM, if the DSM fault rate is not excessive, the application's performance under distributed shared memory may be close to that under physical shared memory.

3.2 Evaluation of Applications Performance

While the above cost measurements are useful in assessing the performance penalty of distributed shared memory, the most direct measure of DSM performance is the execution times of applications. One of the applications we implemented on Mermaid is a parallel version of matrix multiplication (MM) in which the computation of the rows in the result matrix is performed by a number of threads running simultaneously, each on a separate processor. The result matrix is write-shared among the threads, whereas the two argument matrices are read-shared only, hence can be replicated. At the end of the computation, pieces of the result matrix are transferred (implicitly) to the master thread, which creates and coordinates the activities of the slave threads, but performs no multiplication itself. We used 256 × 256 integer matrices for our measurements. Had floating point matrices been used (which would be the case in numerical applications), the amount of parallelizable computation would increase, so the relative overheads would be lower, and the speedup would be better.

Another application we converted to run under Mermaid is a program that detects flaws in printed circuit boards (PCB). Two digital images (front- and back-lit) of a PCB are taken by a camera, digitized, and then stored as large matrices. The software then checks all the geometric features on the board, such as conductors, wire holes, and spacing between them. If design rule voilations are found, they are high-lighted in red in a third image, which is displayed on a color workstation, so that a human decision may be made as to whether this PCB has to be discarded. The amount of computation involved in the rule checking is substantial: on a Sun3/60, it takes about five minutes to process a 2 cm × 16 cm area. A suitable computing environment for on-line PCB inspection is a workstation with bit-mapped display, coupled with compute servers on which the checking software runs in par-

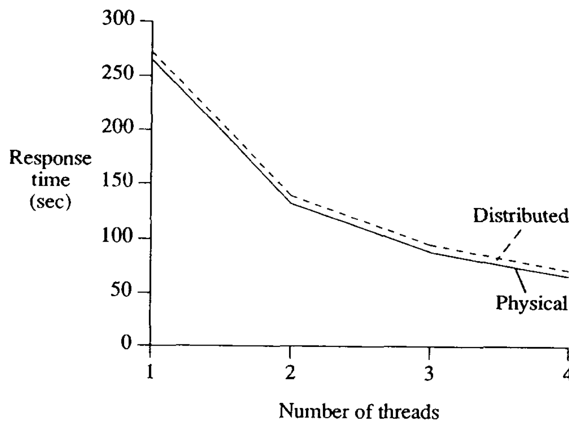

Figure 3: Response times of matrix multiplication when executed on one or multiple Fireflies.

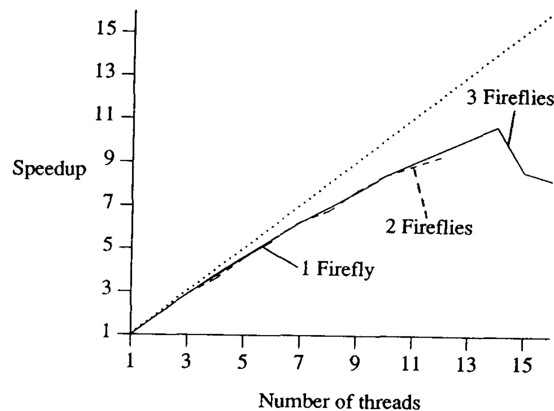

Figure 4: Matrix mult. with master on Sun and slaves on one or more Fireflies.

allel. We therefore used Mermaid as a prototype of such systems. Our version of the PCB software has a master thread that runs on a Sun, divides the area into stripes, and creates threads on the Fireflies to check them [4]. The raw and processed images, as well as data structures containing the design rules, the flaw statistics, etc, are allocated in the DSM space, and are converted properly when transferred between the Sun master and the Firefly slave threads. For our measurements, an area of 2 cm $\times$ 16 cm is used.

Overhead of DSM initialization and thread operations: We compared the execution times of MM and PCB using a sequential implementation to those using DSM with a single slave thread and found the difference to be close to zero on both the Sun and the Firefly; hence, we conclude that the overheads of shared memory allocation, thread creation and synchronization are very low.

Physical vs. distributed shared memory: Since the Firefly is a multiprocessor, we are able to compare the performance of physical shared memory to that of distributed shared memory. The same number of slave threads are either allocated to the processors on the same Firefly or to multiple Fireflies, with one thread on each, and the master thread on yet another Firefly. The response times for both cases are shown in Figure 3. The slightly higher execution times for the multiple Firefly case are mainly due to the cost of transferring pages between the machines. For multiplication of large matrices, performance penalty of distributed memory is minimal. More generally, the penalty depends on the costs of data distribution and replication and the costs of data consistency The former is determined by the underlying communication and data conversion costs, whereas the latter also depends on the applications' data access behaviors.

Heterogeneous vs. homogeneous shared memory: To assess the effect of heterogeneity, we measured the response times of matrix multiplication with a number of threads running on Fireflies, and the master thread running on a Sun3/60. This is a representative configuration of heterogeneous distributed shared memory that takes advantages of both the user-friendly programming environment on a workstation, and the computing power of the background server hosts. Compared to the similar case in which the slave and the master threads run on Fireflies, very little performance degradation is observed. In the first case, pages of the matrices are moved from the Sun to the Fireflies and the result matrix is then moved back to the Sun after the computation; all data movements are accompanied by integer conversions. No data conversion is needed for the homogeneous case.

Scalability of heterogeneous shared memory: The scalability of heterogeneous distributed shared memory, in the case of MM and PCB applications, is shown in Figures 4 and 5. One to four Fireflies are used, and the numbers of threads allocated to each are approximately balanced. For MM, performance improvements are observed as more and more threads are added, up to 14. Beyond that point, the communication overheads begins to becomes more significant. For PCB, there are two additional limitations to speedup: First, the amount of computation for each stripe is unbalanced; hence the larger threads determine the overall response time. Had we used a PCB area of greater height (e.g., 10 cm instead of 2 cm), multiple stripes of the board could have been assigned to the threads in an interleaved fashion to balance the work. Second, the overlaps of the areas represent extra computation, which grows as more threads are used. Despite these limitations, very good speedup (up to 7 using 10 threads) were still observed. Instead of taking six minutes on a Sun, the checking can be completed in 44 seconds on three Fireflies.

It is interesting to note that physical shared memory is treated as a special case of distributed shared memory in Mermaid; the two types of memory are fully integrated throughout the heterogeneous system base, and the performance potential of such a system is fully explored (in the sense that physical shared memory is used if present and distributed shared memory is used otherwise). The feasibility and performance potential of heterogeneous DSM systems are clearly demonstrated by these experiments.

3.3 Effect of page size algorithms and page thrashing

To assess the effects of page thrashing due to data contention among threads, and to study the relationship between page size and thrashing, we conducted a number of experiments using two different implementations of matrix multiplication. The first, MM1, assigns large groups of rows of the result matrix to each thread[5], the second assigns rows to threads in a round-robin fashion (MM2). MM2 is expected to have more data contention on its DSM pages and is intended to represent the class of applications with such characteristics. By using matrix multiplication

[4]Small overlaps of the stripes are necessary so that features on the borders are checked properly.

[5]MM1 is the implementation of MM being used so far.

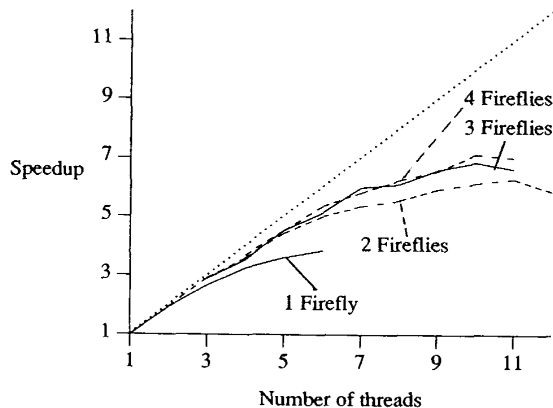

Figure 5: PCB with master on Sun and slaves on one or more Fireflies.

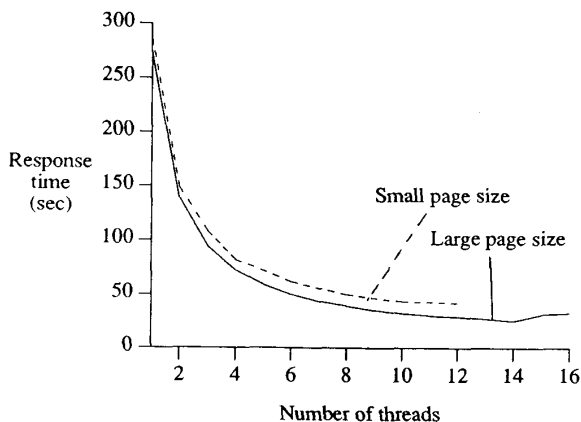

Figure 6: Response times of MM1 using large (small) page size algorithms.

for both, we are able to eliminate other factors affecting the performance of parallel applications, such as scalability and the size of data sets.

Effects of page size algorithms: Figure 6 compares the performance of MM1 using the large page size algorithm to that using small page size algorithm. There is some definite degradation in performance due to increased number of page faults on the Fireflies, throughout the range of the number of processors.

Effects of locality - small page size algorithm: Since MM2 divides the result matrix into rows for the slave threads (1 KB, or 256 integers each), and since the small page size algorithm also operates on 1 KB pages, we expected the degradation of MM2 over MM1 using this algorithm to be small, which is the case, as shown in Figure 7.

Thrashing: The most likely case for thrashing is MM2 with the large page size algorithm, where an 8 KB page is shared by up to 8 threads on a number of Fireflies. We ran MM2 with various numbers of threads running on two or three Fireflies. The corresponding execution times we observed fluctuated greatly, even between consecutive runs with the same setting. Speedup relative to the sequential case was rarely observed, and execution times up to 10 times of that of the sequential case were measured. Examining detailed statistics of the numbers of page faults and

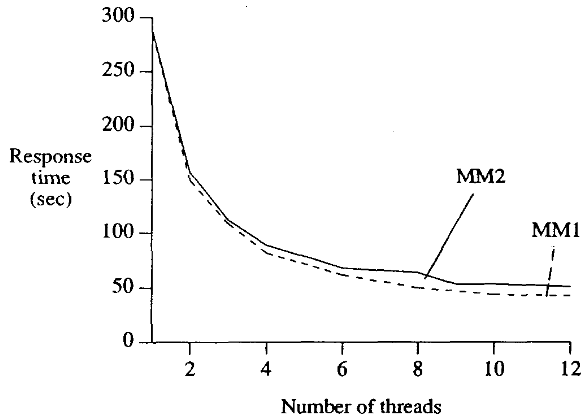

Figure 7: Response times of MM1 and MM2 using the small page size algorithm.

transfers revealed that a very large number of pages were transferred between the Fireflies. The performance degradation and unpredictable fluctuations are clearly due to page thrashing.

From the above experiments, it may be concluded that, if the locality in the application's data accesses is very good, larger DSM page sizes generate less overhead and better performance. If there are substantial data interferences, however, performance may degrade greatly using large pages, and small page sizes are more likely to provide stable performance.

4 Related Work

Efforts in accommodating heterogeneity in distributed computer systems have been the focus of research for some time; see [15] for an overview. Most often, heterogeneity is accommodated by standardizing a system-wide file system interface or by standardizing communication protocols, including remote procedure call protocols. For example, an early effort in accommodating heterogeneity was the NSW project [8], which provided a standard set of tools for a standard file system on different operating systems. More recent examples of file systems for heterogeneous environments include Sun's NFS [17], the Athena project at MIT [2] and the Andrew project at Carnegie-Mellon University [14]. The Heterogeneous Computer Systems project at the University of Washington [3], Sun [17] and Apollo [1] have all developed remote procedure call packages that allow communication between hosts of different types.

Initial efforts to provide distributed shared memory for loosely-coupled distributed systems are due to Cheriton [4] and Li [11]. Li's algorithm is probably one of the most studied distributed shared memory algorithms [9, 7, 12]. Forin et.al.[7] designed and implemented a shared memory facility for heterogeneous environments; however, they assume a homogeneous operating system base (MACH) from the outset.

5 Concluding Remarks

Our research motivation for extending distributed shared memory to accommodate heterogeneity stems mainly from two observations. First, most networked computing environments are becoming more and more heterogeneous due to the increased specialization of computing machines; workstations are employed mainly for their user interfaces, while powerful, parallel systems serve as

compute servers. Second, as a mechanism for interprocess communication in homogeneous systems, distributed shared memory has a number of advantages over its message passing counterparts such as RPC; in particular, it provides clients with a more transparent interface and programming environment for distributed and parallel applications. It is therefore interesting to consider heterogeneous DSM in an attempt to translate as many of these advantages to heterogeneous systems as possible.

In this paper, we discussed the main issues and solutions of building a distributed shared memory system on a network of heterogeneous machines. As a practical research effort, we have designed and implemented a DSM system, Mermaid, for a network of Sun workstations and Firefly multiprocessors, and we ported several applications to this system. In doing so, we were also able to gain insight into distributed shared memory systems in general.

We conclude that heterogeneous distributed shared memory is indeed feasible. ¿From a performance point of view, we showed that the execution times of (at least some) applications running on Mermaid are comparable to those running on a homogeneous DSM system, because the cost of data conversion does not substantially increase the overall cost of paging across the network. Moreover, we were able to easily integrate our distributed shared memory system into the physical shared memory system on the Firefly, allowing the programmer to exploit both physical and distributed shared memory using one and the same mechanism.

One of the main advantages of distributed shared memory is the high degree of transparency it provides of the underlying communication structure. However, as in homogeneous distributed shared memory, *performance* transparency cannot be achieved completely in all cases, because of the need to transfer data over a bit-serial network, and the interactions between the data access patterns of the application and the DSM algorithm used. In the heterogeneous case, we also find it more difficult to achieve *functional* transparency, mainly because of different data representations. Our scheme requires the user to specify the data type when allocating data in the DSM space, and to compose conversion routines for user-defined data types using system-provided routines for the basic types. We claim that for the application programmer, this added complexity is no larger than that required for many existing heterogeneous RPC systems. We are currently working on automatic generation of the conversion routines at compile time, which appears to be feasible.

Our measured performance results also corroborate the results of other researchers in that distributed shared memory can be competitive to the direct use of message passing, at least for a reasonably large class of applications. In some cases, they actually outperform their message passing counterparts, even though the shared memory system is implemented in a layer on top of a message passing system. However, we demonstrated that the size of a DSM page can have an important effect on performance; too large a DSM page size is more likely to cause page thrashing, severely affecting performance.

Acknowledgements

We are grateful to Kai Li who participated in the initial phases of this project and contributed to some of the software. David Wortman participated in many of the discussions on the design and implementation of Mermaid. Adrian Yip performed an initial portiñg of the PCB software to Marmaid.

References

[1] Network Computing Systems Reference Manual. Technical report, Apollo Computer Inc., Chelmsford, Mass., 1987.

[2] E. Balkovich, S. Lerman, and R.P. Parmelee. Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 28(11):1214–1224, 1985.

[3] B.N. Bershad, D.T. Ching, E.D. Lazowska, H.Sanislo, and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, 1987.

[4] D.R. Cheriton. Preliminary Thoughts on Problem-Oriented Shared Memory: A Decentralized Approach to Distributed Systems. *ACM Operating Systems Review*, 19(4), October 1985.

[5] D. Cohen. On Holy Wars and a Plea for Peace. *IEEE Computer*, 14(10), 1981.

[6] B.D. Fleisch. Distributed Shared Memory in a Loosely Coupled Distributed System. In *Proc. ACM SIGCOMM '87 Workshop*, 1987.

[7] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach. In *Proc. 1988 Winter USENIX Conf.*, January 1989.

[8] D.P. Geller. The National Software Works: Access to Distributed Files and Tools. In *Proc. ACM National Conference*, pages 39–43, October 1977.

[9] R.E. Kessler and M. Livny. An Analysis of Distributed Shared Memory Algorithms. In *Proc. 9th Intl. Conf. on Dist. Comp. Sys.*, June 1989.

[10] O. Krieger and M. Stumm. An Optimistic Apoproach for Consistent Replicated Data for Mulitcomputers. In *Proc. 1990 HICSS*, 1990.

[11] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University Department of Computer Science, 1986.

[12] K. Li. IVY: Shared Virtual Memory System for Parallel Computing. In *Proc. Intl. Conf. on Parallel Computing*, 1988.

[13] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1986.

[14] J.H. Moris, M. Satyanarayanan, D.S.H. Rosenthal M.H. Conner, J.H. Howard, and F.D. Smith. Andrew: A distributed Personal Computing Environment. *Communications of the ACM*, 29(3):184–201, 1986.

[15] D. Notkin, N. Hutchinson, J. Sanislo, and M. Schwartz. Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity. *Communications of the ACM*, 30(2):142–162, February 1987.

[16] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5), May 1989.

[17] Networking on the Sun Workstation. Technical report, Sun Microsystems Inc., Mt. View California, 1985.

[18] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.