Heterogeneous Distributed Shared Memory

Songnian Zhou, *Member, IEEE*, Michael Stumm, *Member, IEEE*, Kai Li, and David Wortman, *Member, IEEE*

Abstract—**Heterogeneity in distributed systems is increasingly a fact of life, due to specialization of computing equipment. It is highly desirable to integrate heterogeneous hosts into a coherent computing environment to support distributed and parallel applications, so that the individual strengths of the different hosts can be exploited together. Distributed shared memory (DSM), a high-level, highly transparent model for interprocess communication in distributed systems, is a promising vehicle for achieving such an integration.**

This paper studies the design, implementation, and performance of heterogeneous distributed shared memory (HDSM). As a practical research effort, we have developed a prototype HDSM system that integrates very different types of hosts, and have ported a number of applications to this system. Our experience shows that, despite a number of difficulties in data conversion, HDSM is indeed implementable with minimal loss in functional and performance transparency when compared to homogeneous DSM systems.

Index Terms—**Data consistency, data sharing, distributed computer systems, distributed shared memory, heterogeneous computer systems, interprocess communication, parallel computation, performance evaluation, virtual memory systems.**

I. Introduction

DISTRIBUTED shared memory (DSM) is a model for interprocess communication in distributed systems. In the DSM model, processes running on separate hosts can access a shared address space through normal load and store operations and other memory access instructions. The underlying DSM 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 the value 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 [10], [11], [15]–[18], [24], [3], [14], [25]. (See Stumm and Zhou [24] for an overview.) 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 does not have to 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 typically 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 for many applications [5], [18], [11]. 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, DSM algorithms typically 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. One could argue that the above methods of accessing data could be programmed using messages, in effect imitating DSM in the individual applications. Such programming for communication, however, usually represents substantial effort in addition to that for the implementation of the application itself.

The most widely known algorithm for implementing DSM is due to Li [17], [18], which is well suited for a large class of algorithms. In Li's algorithm, known as SVM, the shared address space is partitioned into pages, and copies of these pages are distributed among the hosts, 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.[1] 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 page 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 page before mapping it into the application's address space. Whenever

Manuscript received July 19, 1990; revised June 19, 1991.

S. Zhou, M. Stumm, and D. Wortman are with the Computer Systems Research Institute, University of Toronto, Toronto, Ont., Canada M5S 1A4.

K. Li is with the Department of Computer Science, Princeton University, Princeton, NJ.

IEEE Log Number 9202077.

[1] It is for this reason that Li called this algorithm and the concept it supports *Shared Virtual Memory* (SVM). In this paper, the more general term, DSM, will be used.

a page is migrated away from a host, it is removed from any local address space it has been mapped into. Similarly, whenever a host 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 on which operations occur is a page instead of a cache line. The DSM fault handlers communicate with DSM memory managers, one of which runs on each host. Each DSM memory manager manipulates local virtual page mapping tables according to the MRSW protocol, keeps track of the location of copies of each DSM page it manages, and passes pages to requesting page fault handlers. In this paper, we assume this protocol for supporting DSM.

For parallel and distributed application programming, distributed shared memory can hide communication complexity from the application when used on a homogeneous set of hosts. DSM in homogeneous systems achieves complete *functional transparency*, in the sense that a program written for a shared memory multiprocessor system can run on DSM without change. The fact that no physical memory is shared can be completely hidden from the applications programmer, as can the fact that, to transfer data, messages have to be passed between the hosts. On the other hand, *performance transparency* can only be achieved to a limited degree, since the physical location(s) of the data being accessed affect application performance, whereas in a uniform memory access (UMA) multiprocessor, the data access cost is not affected by its location in the shared memory. In the case of the MRSW protocol, if a page is not available on the local host when being accessed, it has to be brought in from another host, causing extra delay.

In this paper, we study how DSM can be extended to heterogeneous system environments, and to what degree the functional and performance transparency can be maintained. Heterogeneity exists in many (if not most) computing environments and is usually unavoidable because hardware and its software is often designed for a particular application domain. For example, supercomputers and multiprocessors are good at compute-intensive applications, but often poor at user interfaces and device I/O. Personal computers and workstations, on the other hand, usually have very good user interfaces. There exist many applications that require sophisticated user interfaces, dedicated I/O devices, as well as massive computing power. Examples of such applications can be found in CAD/CAM, artificial intelligence, interactive graphics, and interactive simulation. Hence, it is highly desirable to integrate heterogeneous machines into a coherent distributed system, and to share resources among them.

Heterogeneity in a distributed system comes in a number of forms. The hardware architectures of the machines may be different, including the instruction sets, the data representations, the hardware page sizes, and the number of processors on a host (i.e., uni- or multiprocessors). The operating systems, the system and application programming languages and their compilers, the types of distributed file systems, and the communications protocols may also differ.

A number of methods have been proposed to achieve heterogeneous system integration. (See Notkin *et al.* [22] for an overview.) For example, several remote procedure call (RPC) systems enable servers and application software running on hosts of different types to communicate [26], [1], [4], [12]. Such systems typically define a network standard data format for the procedure call and return messages that all hosts follow by converting between their local representations and this standard. Another method for heterogeneous system integration is to build a heterogeneous distributed file system [13], [9], [21], [2], [7]. Again, a file system access interface and data format is defined that all the hosts must follow to share files among them.

Heterogeneous distributed shared memory (HDSM) is useful for distributed and parallel applications to exploit resources available on multiple types of hosts at the same time. For instance, a CAM application controlling a manufacturing line in real time would be able to acquire data through I/O devices attached to a workstation and output results on its bit-mapped display, while doing most of the computation on compute servers. With HDSM, not only can workstations and compute servers be used simultaneously, but multiple compute servers can be used to increase the aggregate amount of computing power available to a single application. A similar effort to provide heterogeneous distributed shared memory is being undertaken by Bisiani and Forin [11] with their Agora system. However, they use a different DSM algorithm (one where the shared data is replicated on all hosts accessing the data). As discussed by Stumm and Zhou [24], we believe that the MRSW protocol performs better for a larger class of applications than the fully replicated algorithm used by Bisiani and Forin. Forin, Barrera, and Sanzi implemented a shared memory server on heterogeneous processors running the Mach operating system [11]. Their work addressed the issues of multiple VM page sizes, and the conversion of basic hardware data types, such as integer, in the context of Mach.

This paper studies the design, implementation, and performance of heterogeneous distributed shared memory. As a practical research effort, we have developed a prototype HDSM system with hosts that differ significantly. Our experience shows that, despite a number of difficulties in data conversion, HDSM can be implemented while retaining functional and performance transparency close to that of homogeneous DSM. Very good performance is obtained for a number of sample applications running on our prototype. In Section II, we discuss the problems that need to be addressed in order to achieve an HDSM system. Although some of the problems are very difficult, in Section III we show that it is possible to build an HDSM system supporting a wide range of applications, using our prototype system as an example. The performance characteristics of our system, as measured by its overhead and the performance of a number of applications running on it, are discussed in Section IV. Finally, concluding remarks are made in Section V.

II. Issues Related to Heterogeneity

Since with DSM, the components of a distributed application share memory directly, they are more tightly coupled than when data is shared through RPC or a distributed file system. For this reason, it is more difficult to extend DSM to a heterogeneous system environment. These difficulties are explained in this section. Our techniques for overcoming them will be discussed in the next section.

A. Data Conversion

Data items may be represented differently on various types of hosts due to differences in the machine architectures, the programming languages for the applications, and their compilers. For data types as simple as integers, the order of the bytes can be different. For floating point numbers, the lengths of the mantissa and exponent fields, as well as their positions can differ. For higher level structured data types (e.g., records, arrays), the alignment and order of the components in the data structure can differ between hosts. A simple example, depicted in Table I, presents two data types, an array of four characters and an integer, first in big-endian order and then in little-endian order [6]. This example illustrates the *type dependent* differences in data representation that can arise between hosts.

Sharing data among heterogeneous hosts means that the physical representation of the data will have to be converted when the data is transferred between hosts of different types. In the most general case, data conversion will not only incur run-time overhead, but also may be impossible due to nonequivalent data content (e.g., lost bits of precision in floating point numbers, and mismatch in their ranges of representation). This may represent a potential limitation to HDSM for some systems and applications. The question that needs to be addressed is whether, for a specific set of hosts and programming languages, data conversion can be performed for all or most data types to form a *useful* HDSM system (i.e., a system that supports a large collection of realistic applications).

B. Thread Management

As a means of supporting a shared address space, distributed shared memory usually goes hand in hand with a thread system that allows multiple threads to share the same address space. Such a combination makes programming of parallel applications particularly easy. In a heterogeneous system environment, the facilities for thread management, which includes thread creation, termination, scheduling and synchronization primitives, may all be different on different types of hosts, if they exist at all.

Migrating a thread from one host to another in a homogeneous DSM system is usually easy, since minimal context is kept for the threads. Typically, the per-thread stack is allocated in the shared address space, so the stack need not be moved explicitly. The descriptor, or Thread Control Block (TCB), constitutes a small amount of data that needs to be moved at migration time. In a heterogeneous DSM system, however, thread migration is much more difficult. The binary images of the program are different, so it is hard to identify ''equivalent points of execution" in the binaries (i.e., the places in the different binary program images at which execution can be suspended and later resumed on another host of a different type such that the result of the execution is not affected). Similarly, the formats of the threads' stacks are likely to be different, due to architectural, language, and compiler differences; therefore, converting the stacks at migration time may be very difficult, if not impossible.

TABLE I
Big-Endian and Little-Endian Byte Ordering

Byte	Big-Endian		Little-Endian	
	`int`	`char` array	`int`	`char` array
i	MSB	'J'	LSB	'J'
$i+1$		'O'		'O'
$i+2$		'H'		'H'
$i+3$	LSB	'N'	MSB	'N'

MSB = Most Significant Byte; LSB = Least Significant Byte.

While it is clear that thread migration presents yet another limitation to HDSM, its significance is debatable for two reasons. First, in HDSM, threads can be created and started on remote hosts of any type, thus reducing the need for dynamic thread migration. Second, migration between hosts of the same type is still easy to achieve in HDSM, and, for many applications, this may be all that is required. For an application running on a workstation and a set of (homogeneous) compute servers, for instance, its threads can freely migrate between the compute servers to balance their load.

C. 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 memory management hardware (MMU) can be used to trigger a DSM page fault. In a heterogeneous DSM system, the hosts may have different VM page sizes, presenting both complexity in the page coherency algorithm and opportunity in control of the granularity of data sharing.

D. Uniform File Access

A DSM system supporting an application running on a number of hosts benefits from the existence of a distributed file system that allows the threads to open files and perform I/O in a uniform way. While this is likely to be the case in a modern, homogeneous system, multiple incompatible distributed file systems may exist on heterogeneous hosts, due to the multiplicity of distributed file system protocols currently in existence. A uniform file access interface, encompassing both file names and file operations, should be provided to an HDSM application. One possibility is to choose one of the file systems as the standard and make the other(s) emulate it. It is also possible to define an independent file system structure, and make the native distributed file systems emulate it. Recent research on heterogeneous distributed file system is applicable here [7]. Since heterogeneous distributed file system is a research topic on its own, we will not address it any further in this paper.

E. Programming Languages

The system programming languages used on the heterogeneous hosts may be different. This implies that multiple (more-or-less) equivalent implementations of an HDSM system may have to be done in the various languages. However, applications running on HDSM should not be affected by the language(s) used to implement HDSM, as long as a functionally equivalent application interface is supported by HDSM on all the hosts. If a common application programming language is available on all the hosts, then the same program would be usable on the hosts (with recompilation). Otherwise, multiple (equivalent) implementations of an application would have to be written, increasing the difficulties in using HDSM substantially.

F. Interhost Communication

The realization of HDSM requires the existence of a common communication protocol between the different types of hosts involved. This requirement is not particular to HDSM, however, some common transport protocol must exist for the hosts to communicate in any case. The availability of the OSI and TCP/IP protocols on most systems makes the interhost communication increasingly feasible.

III. MERMAID: A PROTOTYPE

In the preceding section, we identified a number of issues that need to be addressed in order to build an HDSM system. Instead of studying these issues in the abstract, we have taken an experimental approach by designing and building an HDSM prototype, Mermaid, and by studying its performance. We discuss our experience in this section. Although the techniques we used to resolve the issues related to heterogeneity are in the context of Mermaid, we believe that most of them are generally applicable. For the use of Mermaid, please see [19].

A. System Overview

In selecting the types of hosts participating in Mermaid, we wanted to include machines that are sufficiently different, so that the difficult issues arising from heterogeneity can be studied. Based on suitability and availability, SunOS workstations and DEC Firefly multiprocessors were chosen. Sun-3 workstations are based on M68020 CPU's and run Sun's version of the UNIX operating system, SunOS. The system programming language is C. The Firefly, developed at DEC's System Research Center, is a small-scale multiprocessor workstation with up to 7 DEC CVAX processors [27]. Each processor has a direct-mapped 64 kilobyte cache. The caches are coherent, so that all processors within a single Firefly have a consistent view of shared memory. The operating system for the Firefly is Taos [20], an Ultrix with threads and inexpensive thread synchronization. The system programming language is Modula-2+, an augmented version of Modula-2 [23]. Table II highlights the differences between the two types of machines.[2]

To focus on our research problems, we adopted a system architecture for Mermaid similar to that of the IVY system developed by Li [17] that uses a page-based MRSW consistency protocol, as described in Section I. It consists of three modules, as shown in Fig. 1. The *thread management module* provides operations for thread creation, termination, scheduling, as well as synchronization primitives. The *shared memory module* allocates and deallocates shared memory and handles page faults. It uses a page table for the shared address space to maintain data consistency, and performs data conversion at page transfer time, if necessary. The responsibility for managing the pages is assigned to the participating hosts in a round-robin fashion (named *fixed-distributed algorithm* by Li [18]). 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 which will be discussed in Section IV-A. 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 showed that the correct choice of algorithm was often dictated by the memory access behavior of the application [24]. 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 some systems. 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 nonresident page is accessed on a host. It was necessary to add a new system call to SunOS for this purpose (Taos provides such a call). A second change to the operating system kernel was to pass the address of the DSM page that has an access violation to its user-level fault handler. No other kernel changes were necessary for these two host types.

B. Basic Support

Programming Languages: As discussed in Section II-E, it is necessary to choose languages for implementing HDSM and for implementing applications running on HDSM. While it would be simpler to use a single language to implement HDSM, interfacing HDSM to the native operating systems is

[2]The hardware MMU page size on a CVAX is 512 bytes, but the VM implementation on the Firefly uses two MMU pages for one VM page of 1 kilobyte. On the Sun, both the hardware MMU page and the VM page have a size of 8 kilobytes.

TABLE II
HIGHLIGHTS OF THE HETEROGENEOUS FEATURES OF THE SUN AND FIREFLY

Attribute	Sun-3	Firefly
Processor	M68020	CVAX
Number of processors	1	4–6 (user usable) + 1 (I/O)
Byte order	Little-endian	Big-endian
Floating point	IEEE	VAX
VM page size	8 kilobytes	1 kilobyte
Operating system	SunOS 3.5	Taos
System language	C	Modula-2+
Application language	C	Modula-2+, C
Thread management	unavailable	available
Communication protocol	TCP/IP, UDP/IP, Sun RPC	UDP/IP subsets, FF RPC
File system	SUN NFS	RFS

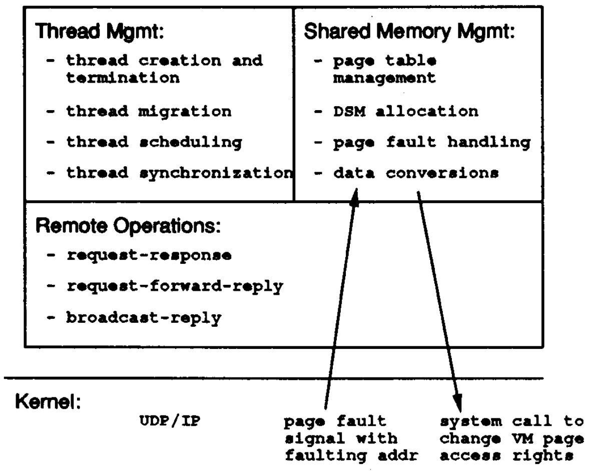


Fig. 1. Structure of the Mermaid system.

easier if the native system implementation languages are used. For Mermaid, we chose the latter approach by having a C implementation for the Sun, and a Modula-2+ implementation for the Firefly. As a result, most of the Mermaid functionalities had to be implemented twice, and, whenever changes are made to Mermaid, both implementations must be modified. Though certainly cumbersome, the modification process has been relatively straightforward.

The situation for application programs is quite different, since it is highly undesirable to force the user to implement an application in multiple languages. We therefore chose C as the common application language. We have ported to Mermaid a number of large, complex applications originally written in C for sequential machines, by only modifying the top-level logic to break the computation into parallel tasks, (without understanding the low-level algorithms employed by the application, which typically constitutes 80–95% of the code). This would have been impossible had multiple languages been required.

Communication Substrate: The distributed shared memory modules typically operate in a request-response mode. For instance, when a page fault occurs, the fault handler sends a page request to the manager for this page, which either supplies the page, or forward 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.

We implemented our own presentation layer protocol in the Remote Operations module following the above requirements, and use it to support all interactions between the memory and thread management modules running on different hosts. This presentation layer protocol is implemented using UDP/IP, a simple, datagram-based transport protocol. Our implementation was complicated by the fact that fragmentation in UDP/IP is not supported on the Fireflies, We did not use the RPC packages available on the Suns and Fireflies, since they are incompatible and do not meet our requirements with respect to functionality, i.e., broadcast and forwarding. Moreover, since data conversion is performed in HDSM, we need not incur the overhead of data marshalling and demarshalling at the RPC level.

Thread Support: Many well established operating systems, including SunOS, do not provide direct support for multiple threads that share a common address space. Mermaid therefore provides a simple thread module at the user level on the Sun. Since all threads in a Sun address space run within a single Unix process, the suspension of one thread by the operating system scheduler (e.g., for *synchronous* I/O) makes the other threads nonexecutable as well. This has not been a problem for the Mermaid applications we ported, since parallel applications often allocate only one thread on each processor. For the Firefly, a system-level thread package is available and is used by Mermaid. Mermaid threads in an address space may be created on one host and later moved to and started on other hosts of any type. Alternatively, threads may be created and started on remote hosts directly. However, no dynamic thread migration facility is provided in the current implementation of Mermaid.

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

C. Data Conversion

When data is transferred from a machine of one type to a machine of another type, it must be converted before it is accessed on the destination machine. In Mermaid, the unit of data that is converted is a page, and the conversion is based on the type of data stored on the page. Our goals for data conversion were to minimize the amount of work the user has to do, to make the conversion method as general as possible, and to achieve good performance. We adopted a three-part conversion scheme for Mermaid. First, the types of data to be allocated in the shared memory are indicated in the memory allocation requests. Second, routines to convert the various types of data are automatically generated by utility software. Finally, a mechanism is built into Mermaid so that the appropriate conversion routine is invoked whenever a page is transferred. We discuss the three parts in more detail in the following sections.

1) Typed Data Allocation: The information about the layout of a page has to be passed to Mermaid so that appropriate conversion can be performed upon page transfer. For this purpose, we provide a special memory allocation subroutine similar to `malloc` in Unix that has an additional argument identifying the type of data being allocated, as shown in Fig. 2(b). When processing such a request, the memory management module of Mermaid records the range of the shared address space allocated for this request, and the data type, in the corresponding DSM page table entry or entries. There is no restriction as to the type or size of data that can be allocated; a structure may be larger than a page. For example, Fig. 2(a) depicts a user-defined structure, `sharedType`, that is allocated by the call in Fig. 2(b).

In principle, multiple types of data could be coallocated in the same page, but this makes keeping track of the data types complicated and the dynamic data conversion inefficient. We therefore made the restriction that a page contain data of one type only,[3] so that information with respect to only one data type needs to be kept in each HDSM page table entry, and only one conversion routine needs to be invoked (which may invoke other routines in turn, as will be discussed below). Multiple allocation requests for the same type of data may be satisfied, fully or partially, by the same page, given that there is space in the page.

Our requirement of allocating only one data type per page may result in more memory usage, since now several pages may be partially filled, rather than at most one. However, the number of pages wasted is limited by the number of distinct data types being allocated in the shared memory. For modern machines with a large main memory, this is typically not a serious problem. Also, as an optimization in our implementation, when a partially filled page is being transferred, only the part with valid data is transferred and converted (if necessary). Despite the increased memory usage, segregating data by types may have the desirable side effect of reducing page contention for some applications, if unrelated data of different types no longer co-reside in a page.

[3]Note that the data type need not be a basic type provided by the programming language, but can be an application defined compound type.

2) Automatic Data Conversion: In addition to data type information, Mermaid also needs a corresponding conversion routine for each type. In Section II-A, we noted that data conversion may not be possible for some types due to differences in data content, size, or alignment. Here, we first assume that conversion is possible, and discuss a general framework for automatically generating the conversion routines. We then study the Mermaid case to expose its limitations.

Conversion Code Generation: In general, a hierarchy of conversion routines must be constructed that partially reflects the data type hierarchy defined in the application program. This hierarchy is partial, because only those types directly or indirectly allocated in the shared memory need conversion routines. For the basic data types defined by the language and supported by the hardware, such as `int`, `short`, `float`, and `double` in C, efficient conversion routines can be provided by the HDSM system itself.[4] For user-defined data types, a conversion routine is invoked that consists of a sequence of calls to lower level routines, mirroring the structure of the data type. If an element is of a basic type, then its HDSM routine is invoked directly. Otherwise, a routine composed for the element is invoked. Ultimately every data type is composed of basic types. Fig. 2(c) gives an example of the conversion code for user-defined, nested data types.

We have constructed a fully automatic conversion routine generator that processes the compiler output for a program and produces all the necessary conversion routines.[5] The conversion routines generated are structural, in that they only specify the names of the lower level routines and the order in which they should be invoked; the same source code may therefore be used on all machines, independent of which machine the routines are generated on. The machine-dependent basic conversion routines provided by HDSM ensure that the conversion is correct on each machine. A number of simple optimizations are made in our current implementation. For example, a single routine is called for an array of data elements, as shown in Fig. 2(c) for the structure `embeddedType`.

In addition to the conversion routines, the generator also generates a table matching the data types that are directly specified in the memory allocation routines to their corresponding conversion routines. This table is used by Mermaid at page transfer time to invoke the appropriate conversion routine.

Feasibility of Conversion: We now address the issue of the feasibility of data conversion. Three problems are involved: 1) the conversion of basic data types, 2) the handling of a data item crossing a page boundary, and 3) the handling of different ordering of fields in a record.

On both the Sun and the Firefly, the ASCII standard is used for characters (`char` in C); hence, no character conversion is needed. Conversion of integers (either the four-byte `int` or the two-byte `short`) is a matter of proper byte swapping.

[4]These are implemented as inline code for efficiency reasons.

[5]It was necessary to work with the compiler output since we were unable to obtain access to the source code or internal documentation for the C compilers on both machines.

```
struct embeddedType {
    double d;                     /* an 8-byte double precision floating number */
    char c[8];                    /* an array of eight ASCII characters */
};

struct sharedType {
    int *p;                       /* a (4-byte) pointer to an integer */
    float b;                      /* a 4-byte single precision floating number */
    struct embeddedType e[16];    /* an array of 16 records declared above */
};
```

(a)

```
ptr = (struct sharedType *) DSM_Alloc((sizeof(struct sharedType) * n, "sharedType");
```

(b)

```
conv_embeddedType(dst, src, numrecords)
  struct embeddedType dst[], src[];
  int numrecords;
{
  register struct embeddedType *dstp = dst, *srcp = src, *srcend = &src[numrecords-1];

  for (; srcp <= srcend; srcp++, dstp++) {
    conv_float64(dstp->d,srcp->d);
    conv_chars(dstp->c,srcp->c, 8);
  }
}

conv_sharedType(dst, src, numrecords)
  struct sharedType dst[], src[];
  int numrecords;
{
  register struct sharedType *dstp = dst, *srcp = src, *srcend = &src[numrecords-1];

  for (; srcp <= srcend; srcp++, dstp++) {
    conv_pointer(dstp->p,srcp->p);
    conv_float32(dstp->b,srcp->b);
    conv_embeddedType(dstp->e,srcp->e,16);
  }
}
```

(c)

Fig. 2. An example of data conversion. (a) Sample data structure with embedded substructure. (b) Sample allocation statement for the second structure in (a). (c) Data conversion routines generated for the structures in (a).

Conversion of floating point numbers is somewhat more complicated. While both the VAX and the IEEE formats of single precision floating point numbers (`float` in C) use 23 bits to represent a 24 bit mantissa, 7 bits to represent the exponent, and 1 bit for the sign, their layout is quite different. In the IEEE format, the bits in the mantissa are stored contiguously, while in the VAX format they are partitioned across bits 0–6 and 16–31. The bias used to represent the exponents differ by one in the two formats. Despite these differences, equivalent conversion is achievable, except for the following special cases. The IEEE format used on the Sun supports unnormalized numbers and special cases, such as infinity and NAN's (not a number), which are not supported by the VAX format on the Firefly. These cases can be detected with two additional comparison operations. The positions and lengths of the exponent and mantissa fields may be different (such is the case with IEEE and VAX), requiring bit manipulation operations.

The VAX and IEEE formats for representing double precision floating point numbers differ more significantly. The IEEE format uses an 11-bit exponent and a 52-bit mantissa, whereas the VAX uses an 8-bit exponent and a 55-bit mantissa. Therefore, the smaller exponent field and the smaller mantissa field of the two representations dictate the range of (floating point) numbers that can be correctly represented on both types of machines.

For pointers, conversion is necessary if the shared address space starts at different virtual addresses on different host types. The HDSM system on each host may obtain the starting address of the shared memory for each host type at initialization time by communicating with each other, without application intervention. Converting a pointer is then a simple matter of adding an offset to the value of the pointer. This is the scheme used in Mermaid.

In Mermaid, all the corresponding basic data types have the same sizes, but their alignment requirements may be different. For the `double` type, for instance, Sun requires only even-byte alignment, whereas Firefly (VAX) requires quad-byte alignment. Thus the size of a compound structure and the alignment of the elements in it may be different on the two machines. Our automatic conversion generator detects this problem and automatically generates a revised structure definition with dummy elements inserted to force data structure elements of interest to have the same alignment on both machines. The application program is then recompiled on both machines with the revised data structure definition and correct alignment of corresponding elements is achieved. This process will result in some wasted storage on the machine with the less strict alignment requirements but it is essential for the operation of HDSM.

In general, it is also possible for a data item to cross a page boundary. If the item is a compound structure, and none of its basic data items crosses the page boundary, then the partial structure on the page to be converted may be copied to a temporary buffer (with the missing part of the structure filled with some default values taken from a template), where the conversion may be performed in-place using the appropriate routine. The partial structure is then copied back to the appropriate location in the page. If, on the other hand, a basic data item crosses a page boundary, then the conversion will need the parts on both (neighboring) pages. One of these pages may be resident on another host, making it necessary for it to be transferred. For certain data types, page-based data conversion may not be possible. Consider, for example, an integer with its first two bytes at the end of one page, and its last two bytes at the beginning of the following page. If byte swapping is necessary in converting an integer, then transferring one of the two pages between hosts with different byte orders can result in the loss of half of the integer, since the two pages held by their owner(s) may end up having the *same* two bytes of the integer. Such a problem does not arise in Mermaid, since we ensure, by forced alignment, that no basic data items cross page boundaries.

A similar problem arises if compilers on different machines ordered the space allocated for the fields of a structure differently. Even if no basic data item crosses a page boundary, the same field in a structure may be located on different pages, depending on the type of machine(s) holding the data. Again, neighboring pages would be needed for conversion, and it is possible to lose some of the data items during conversion. For the C compilers on the Sun and the Firefly, this problem does not exist.

The `union` structure in C allows various formats for a compound data type. Unfortunately, C does not require a tag field to indicate the format being used, thus making automatic conversion of union structures impossible. In Mermaid, we require the user to add a tag field at the beginning of a union structure, and to set its value to indicate the interpretation of the rest of the structure. Such a requirement would not be necessary in more sensibly designed languages, such as Pascal and Modula-2.

3) Dynamic Conversion Mechanism: Once the conversion routines are generated as described above, they can be compiled and linked with the user program on each type of machine, without additional effort from the user. Upon page transfer, the remote machine type is checked, and, if different from the local one, the appropriate conversion routine is invoked. The conversion mechanism in Mermaid uses the data type information stored in the HDSM page table entries, and the table matching the data types to their corresponding conversion routines produced by the code generator described in Section III-C1.

In our current implementation, conversion is always done by the receiving machine. This is desirable for some cases, such as when a master thread distributes input data to multiple worker threads, because the conversion can be performed by the workers in parallel, rather than by the master sequentially. For other cases, such as when the master collects results from the workers, it is better to have the sending machine perform the conversion. Our primary motivation for the current scheme is simplicity and transparency. Since only two types of machine are involved, data is always converted from the foreign format to the native format, rather than using an intermediate, network standard format as in some RPC systems [9].

4) Limitations: A Summary: The data conversion problem is complex. Our experience indicates that our solution is sufficient for many practical applications in the context of Mermaid, and we believe that it is similar in complexity to the solution used by existing heterogeneous RPC schemes [26]. However, as pointed out above, our solution does have a number of limitations:

1) Some functional transparency is sacrificed by requiring the programmer to specify the type of data being allocated. This is usually only a small annoyance.
2) Floating point numbers can lose precision when being 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. However, we do not consider this to be a practical 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. The initial (floating point) data and final results are not likely to be in the extreme ranges, or nonnumbers. During the computation phase, data pages may be transferred among the (homogeneous) compute servers without conversion.
3) Entire pages are converted even though only a small portion of a page may be accessed before it is transferred away. However, we have found that the cost of page conversion to be small compared to the overall migration cost (to be discussed in Section IV-A). Applications that access only a few data items of a page between page migrations will perform poorly in both the homogeneous and heterogeneous cases when using a page-based MRSW algorithm. A DSM system based on the MRSW algorithm performs poorly with this type of access behavior in both the homogeneous and the heterogeneous cases.
4) An additional tag field is required in union structures in languages such as C and Modula-2; the value of the tag must be set by the application to indicate the interpretation of data in the structure.

5) The order of the fields within compound structures must be the same on each host.
6) A memory page may contain data of only one type, which may be a compound type containing multiple data items of other types.

Of the above limitations, the first three are "hard" in that they are limitations to our design. The fourth limitation is particular to the C language. The fifth is not a problem in any of the systems we know of, including the Sun and the Firefly. The last limitation is not necessary, but is desirable for efficiency.

D. Page Sizes

In a heterogeneous system with machines supporting different VM page sizes, choosing a size for the DSM page becomes an important issue. We may use the largest VM page size for DSM pages. Since VM page sizes are most likely powers of 2, multiple small 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, due to increased *false sharing*, where nonoverlapping regions in the same page are shared and updated by threads on different hosts, causing repeated page transfers. False sharing should be contrasted to real sharing, in which a number of data items are shared and updated by multiple hosts. While performance degradation due to real sharing is hard to avoid, performance degradation due to false sharing can often be reduced by using smaller DSM pages.[6]

One way to reduce data contention is to use the smallest VM page size for the DSM pages. If a page fault occurs on a host with a larger page size, multiple DSM pages are moved to fill that (larger) page. If a fault occurs on a host with a small (DSM) page and no host with a large page size is sharing the data, then only this small (DSM) page needs to be obtained. We call this the *smallest page size* algorithm.[7]

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 DSM page 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 IV-C2.

[6]False sharing can also be reduced by rearranging memory layout, so that data that would be falsely shared if placed on the same page is assigned to different pages.

[7]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).

IV. Performance Evaluation

We have performed a number of experiments on our prototype Mermaid system in order to study the impacts of heterogeneity on the performance of distributed shared memory systems. Along the way, some performance aspects of distributed shared memory in general are also studied. In the following, we first discuss a number of overhead measurements, followed by the response time measurements of three Mermaid applications. We then assess the performance impact of DSM page size algorithms and page thrashing. 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 IV-C2). The Mermaid prototype is not fine-tuned to achieve optimal performance, since our goal is not to push the performance of HDSM to its limit, but to assess its practical value in terms of its performance and ability in supporting applications.

A. Overhead Assessment

Compared to physical shared memory, distributed shared memory has a number of additional overheads. 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. 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. The allocation of shared memory, thread scheduling, and thread synchronization also introduce overhead, but they are relatively small compared to the communication overheads. Finally, for heterogeneous systems, the costs of data conversion and the page size algorithm must be added.

The basic costs of handling a page fault in Mermaid are shown in Table III. Included are the invocation of the user-level handler, the identification of page fault type (read or write), the HDSM page table processing, and the request message transmission time.[8] The delay in transferring the page over the network is not included. The values of a few milliseconds are considered to be quite small. The costs on the Fireflies are higher, due to the higher overhead of the page fault handling mechanism for access violation (about 4.5 ms or higher). The operating system kernel, the "nub," of Taos version 72.4 on the Fireflies, considers access violation fault to be a rare case.

Table IV shows the costs of transferring 8 kilobyte and 1 kilobyte 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 kilobyte transfers are only about three times that of 1 kilobyte transfers, due to the fixed portion of the cost of the message transport. Hence, in the absence of page thrashing, larger DSM pages incur fewer page faults and lower data transfer overhead.

[8]To collect the data for this and subsequent overhead measurements, the Mermaid system was slightly modified so that a large number of the same operation (e.g., 100 000) are performed in a sequence, and the total elapsed time measured.

TABLE III
COSTS OF PAGE FAULT HANDLING (ms)

	Sun	Firefly
Read	1.98	6.80
Write	2.04	6.70

TABLE IV
COSTS OF TRANSFERRING A PAGE (ms)

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

The measured costs of converting a page of integers, shorts, floating point numbers (single and double precision), and the user-defined structure of Fig. 2(a) on a Sun3/60 and a Firefly are shown in Table V. In all of the cases except that of `double` on Fireflies, the conversion costs are substantially lower than those of page transfer. The cost of converting an 1 kilobyte page is approximately 1/8 of that for an 8 kilobyte page. It is interesting to note that the overhead for converting the user-defined structure with an embedded structure is not much larger than that for the basic types. We also measured several other user data structures and found their conversion costs to be comparable.

To consider the combined effects of the overhead costs discussed above, we show the end-to-end page fault delays for different types of hosts in Table VI. Three different scenarios are considered, depending on the locations of the host on which the thread triggering the page fault resides (Requester), the host acting as the manager for the page (Manager), and the host currently having ownership of the page (Owner). While the Requester and the Owner are always different (otherwise there would not be a page fault in the first place), the Requester and the Manager, or the Manager and the Owner may be the same host. The cost for (integer) data conversion is included when the Requester and Owner hosts are of different types. The measurements are based on the largest page size algorithm, so the values are for 8 kilobyte pages only. The costs for read and write page faults were found to be very similar. The HDSM page fault delay is comparable to that of a VM page fault involving a disk seek. The costs of page faults involving both the Sun and the Firefly are very comparable to the homogeneous case of Firefly, but higher than that of Sun, partly due to user-level message fragmentation and reassembly. As with traditional VM, if the HDSM fault rate is not excessive, the application's performance under distributed shared memory may be close to that under physical shared memory.

B. Evaluation of Application Performance

1) Three Sample Applications: While the cost measurements above 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

TABLE V
COSTS OF DATA CONVERSIONS (ms)

page / data type	8 kilobyte page		1 kilobyte	
	Sun	Firefly	Sun	Firefly
`int`	5.01	7.75	0.63	1.00
`short`	6.53	10.0	0.82	1.15
`float`	9.72	13.9	1.22	1.68
`double`	11.6	29.0	1.46	3.63
user structure	6.61	21.6	0.67	2.15

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 divided into a number of groups of adjacent rows equal to the number of threads, and assigned to the threads. The result matrix is write-shared among the threads, whereas the two argument matrices are only read-shared, and can hence 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. Except where noted, the experiments discussed below use 512×512 matrices of `double`[9] numbers.

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 sample PCB are taken by a camera, digitized, and then transferred to a workstation to be stored as large matrices. The software then checks the geometric features on the board, such as conductors, wire holes, and spacing between them. If design rule violations are found, they are high-lighted in a third image, which is displayed on the workstation, so that a human decision may be made to rectify the problem. The amount of computation involved in the rule checking is substantial: on a Firefly, it takes about 11 min to process a 2 cm $\times$ 32 cm area using a sequential version of the software. Obviously, speeding up the execution would make the feedback on the manufacturing line more timely, reducing the number of boards that may have to be discarded. 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 parallel. We therefore used Mermaid as a prototype for such a system. Our version of the PCB software has a master thread that runs on a Sun, divides the board area into stripes, and creates threads on the Fireflies to check them.[10] All data including the raw and processed images, and the data structures containing the design rules and the flaw statistics are allocated in the DSM space, and are properly converted when transferred between the Sun master and the Firefly slave threads. For our measurements, an area of 2 cm $\times$ 32 cm is used.

A third application we used to study Mermaid performance is a partial differential equation solver that uses the Successive

[9] 64-bit, double precision floating point numbers.

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

TABLE VI
END-TO-END PAGE FAULT DELAYS FOR 8 KB PAGES (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:
R/M: Requester and Manager are on the same host.
M/O: Manager and Owner are on the same host.

Over-Relaxation method (SOR). Data is represented as a large two-dimensional matrix. With the boundary values fixed, the internal values are then iteratively updated to be the average of the values of their four neighbors, until they converge. While this application is again based on large matrices (so we can partition the computation along with the data by assigning groups of adjacent rows of the matrix to the threads on various Fireflies), its data access behavior is quite different from the above two applications. The threads update values in their regions asynchronously with each other. The entries in the matrix are updated many times, and, for each iteration, the neighboring entries in the neighboring rows are needed. Thus, page sharing occurs in every iteration, and the number of iterations generally grows with the size of the matrix. Furthermore, to reach a decision on global convergence, the local convergence condition of each thread needs to be recorded in a shared array, and is checked by all threads. Given the simple model of data sharing used in Mermaid, it is interesting to see if page thrashing can be avoided, and good speedup can be achieved. For the experiments described below, a matrix of 128 × 816 of `double` numbers is used.

2) Physical Versus 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 threads are either allocated to the processors on the same Firefly or to multiple Fireflies, (with one thread on each). The speedups of MM for both cases are shown in Table VII, for up to a maximum number of four threads. The slightly worse performance for the distributed case is due mainly to the cost of transferring pages between the machines. As also observed with the PCB and SOR applications, the penalty for running in a distributed system depends on the costs of data distribution and replication and the costs of data consistency and less so on the costs of data conversion. The distribution and replication costs are determined by the underlying communication and data conversion costs, whereas data conversion costs also depend on the applications' data access behaviors.

3) Heterogeneous Versus Homogeneous Shared Memory: To assess the effect of heterogeneity, we measured the response times of the three sample applications with a number of threads running on one or multiple 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 both

TABLE VII
SPEEDUPS OF MATRIX MULTIPLICATION WHEN EXECUTED ON ONE OR MULTIPLE FIREFLIES

	number of processors			
	1	2	3	4
`physical shared memory`	1.00	1.98	3.00	3.97
`distributed shared memory`	0.97	1.91	2.87	3.32

the master and the slave threads run on Fireflies, very little performance difference 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 appropriate data conversions. No data conversion is needed for the homogeneous case. This is further evidence that data conversion does not add significant overhead to HDSM.

4) Application Performance with HDSM: The speedup curves of the MM, PCB, and SOR applications running on Mermaid are shown in Figs. 3, 4, and 5. The speedups are computed with respect to a sequential execution on a Firefly. For all the data points, the master thread is located on a separate Sun3/60. One to four Fireflies are used, and the numbers of threads allocated to each are approximately balanced. Better performance was observed using the largest page size algorithm for MM and PCB, while for SOR the smallest page size algorithm produced the best results (to be discussed further in Section IV-C2). For MM, performance improvements are observed as more and more threads are added to the computation, up to 18, the maximum number of Firefly processors available, for a maximum speedup of 12.

For PCB, there are two additional limitations to speedup: First, the volume of data to be transferred is very high (about 5 megabytes for each image), incurring substantial synchronous delays to remote worker threads as parts of the images are faulted in from the master. Second, the overlapping areas of the images must be processed by two threads, and represent extra computation, which grows as more threads are used. Despite these limitations, good speedup (up to 10 using 14 threads on four Fireflies) were still observed. Hence, the checking can now be completed in 65 s on four Fireflies, instead of 11 min in the sequential case. In some cases, super-linear speedup was observed. This is due to a reduction in the VM working set, as the data is partitioned among more and more processors where CPU caching becomes more effective. This observation is analogous to one made by Li with respect to main memory and VM paging [17].

The performance of SOR is comparable to that of MM,

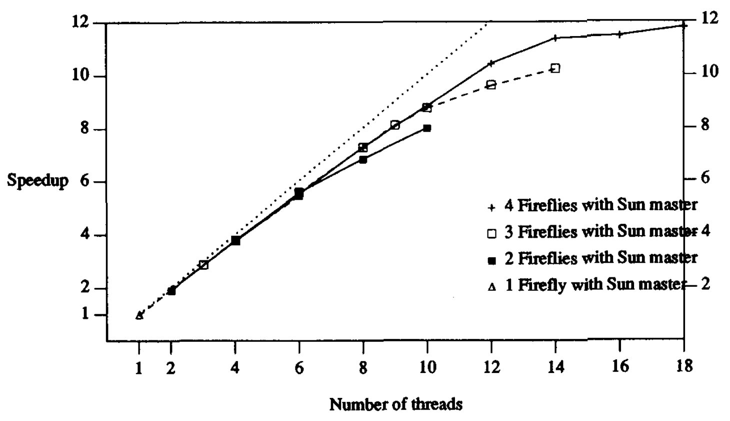


Fig. 3. Matrix multiplication with master on Sun and slaves on one to four Fireflies.

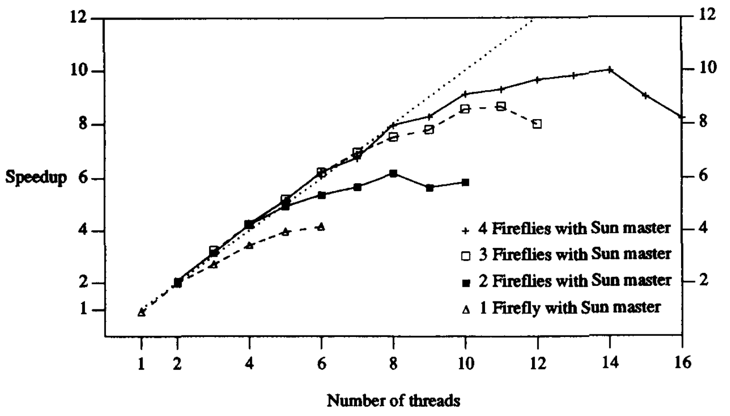


Fig. 4. PCB with master on Sun and slaves on one to four Fireflies.

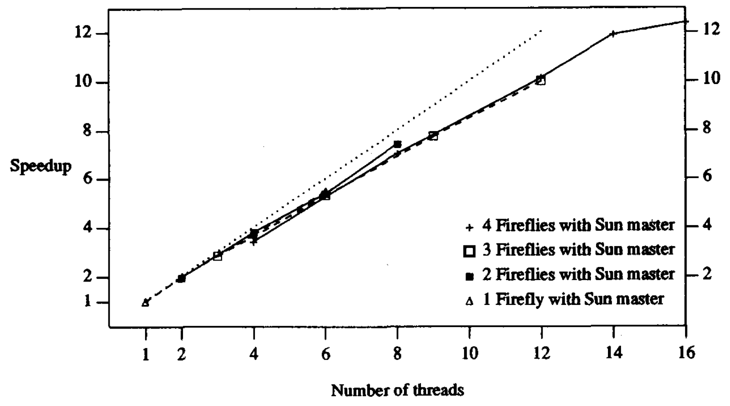


Fig. 5. SOR with master on Sun and slaves on one to four Fireflies.

despite the large numbers of iterations made by the threads, causing many page faults on the pages that are shared by threads running on different Fireflies. Thrashing does not usually happen, since an iteration for each thread takes 150 ms or more, whereas a fault on an 1 kilobyte page only takes approximately 15 ms.

The same application has been studied in the Amber system at the University of Washington [8]. In Amber, an object-oriented approach is used for parallel application support. For the SOR application, the Amber implementation partitions the matrix into groups of rows called section objects, and stores one section object on each of the Fireflies. Multiple computation threads on each Firefly work on the local section object in parallel, and a separate communication thread on each Firefly sends the boundary rows of the section object to the neighboring Fireflies as the rows are updated. Thus, the potential of data contention in Mermaid is avoided in Amber, and the computing threads never have to stop for data. With the limited number of processors we have, however, our results are very comparable to those of Amber's on problems of similar sizes. (A 122 × 842 matrix was used with Amber.) Programming an application on Mermaid, on the other hand, is much simpler than on Amber, because communication is completely hidden in the shared memory system model.

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 well explored (in the sense that physical shared memory is used if present and distributed shared memory is used otherwise). Also, the number of machines involved does not affect the performance of any of the three applications much, as evidenced by the close clustering of the curves in Figs. 3, 4, and 5. The feasibility and performance potential of HDSM systems are therefore clearly demonstrated by these experiments, at least for certain classes of applications.

Besides the data sizes used in the above experiments, we also measured the performance of the three applications with several other data sizes to force the same DSM pages to be shared among multiple hosts. No significant changes to the speedup values were found from those presented above.

C. Effects of the 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,[11] the second, MM2, assigns rows to threads in a round-robin fashion. MM2 is expected to have more data contention on its DSM pages and is intended to represent the class of applications with this behavior. By using matrix multiplication for both, we are able to eliminate other factors affecting the performance of parallel applications, such as scalability and the size of data sets.

1) Effects of Page Size Algorithms and Locality: We compared the performance of MM1 using the largest page size algorithm to that using the smallest page size algorithm, and found that there is a small but definite degradation in performance when using smaller page sizes, due to an increased number of page faults on the Fireflies (see Fig. 6).

Since MM2 divides the result matrix into rows for the slave threads (4 kilobyte, or 512 `double` floats each), and since the smallest page size algorithm operates on 1 kilobyte pages, we expected the degradation of MM2 over MM1 using this smallest page size algorithm to be very small, which we verified experimentally (results not presented here). The degradation is slightly greater with 256 × 256 matrices of integers, where one DSM page of 1 kilobyte holds one row of the matrix (256 integers). Using integers instead of double precision floating point numbers, together with the smaller matrices, accentuates the importance of communication and locality.

2) Thrashing: The most likely case for thrashing is MM2 with the largest page size algorithm, where an 8 kilobyte page is shared by up to 8 threads running on several Fireflies. We ran MM2 with various numbers of threads on two or three Fireflies. The corresponding execution times we observed fluctuated greatly, even between consecutive runs of identical setup. Speedup relative to the sequential case was rarely observed, while execution times up to 10 times of that of the sequential case were measured. Examination of the detailed statistics of the numbers of page faults and transfers revealed that a large number of pages were being transferred between the Fireflies; the performance degradation and unpredictable fluctuations were 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, large DSM page sizes may generate less overhead and better performance. If data in small ranges of the DSM space are updated by separate threads, however, performance may degrade greatly using large pages due to false sharing, and small page sizes are more likely to provide stable performance.

Our experience with a number of applications shows that small, seemingly minor changes to an implementation of an application may result in very different data sharing patterns and drastically different performance. MM1 versus MM2, using the largest page size algorithm, is such an example. For the SOR application, we initially implemented the algorithm so that each thread, during each iteration, updated the data elements in its portion of the matrix from top to bottom, thus sharing data with neighboring threads in every iteration. The resulting performance was unsatisfactory due to the frequent read–write sharing. We then changed the algorithm such that each thread updated the data elements in its portion of the matrix from top to bottom to top. Performance is improved substantially because the number of times the boundary rows are worked on is reduced by half, and the amount of computation in between such shared data zones is doubled. Consequently, data movement between machines and the possibility of data contention are reduced, and better speedups are observed (as shown in Fig. 5).

V. Concluding Remarks

In this paper, we discussed the main issues and solutions of building a DSM system on a network of heterogeneous machines. As a practical research effort, we designed and implemented an HDSM system, Mermaid, for a network of Sun workstations and Firefly multiprocessors, and we ported a number of applications to Mermaid. We conclude that heterogeneous DSM is indeed feasible. From a functional point of view, we showed that little transparency need be lost due to heterogeneity. The most important problem is data

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

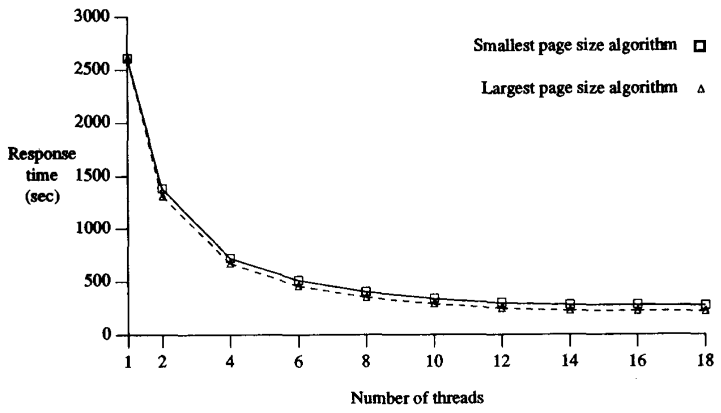


Fig. 6. Response times of MM1 using the largest (smallest) page size algorithms.

conversion. Our solution requires that the user specify the type of data being allocated in the HDSM space, which is usually natural to the programmer. For different representations of floating point numbers, equivalent data conversion may be impossible for extreme values. However, with the increasing use of the IEEE floating point standard, this may be considered to be a passing problem. We were able to easily integrate our HDSM 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.

From a performance point of view, we again showed that little transparency is lost due to heterogeneity; that is, our heterogeneous DSM implementation performs comparably to an equivalent homogeneous DSM system. Overall, we have found that the cost of data conversion does not substantially increase the cost of paging across the network. Other aspects of heterogeneity, such as accommodating different page sizes and user-level processing of messages, also do not contribute significantly to the DSM overhead. The presence of multiple VM page sizes on different types of machines presents applications with the opportunity of selecting the DSM page size according to their data access patterns; we noticed substantial performance gains in using suitable DSM page sizes.

Our measured performance results corroborate the results of other researchers in that distributed shared memory can be competitive to the direct use of message passing, 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.

Although our prototype Mermaid system integrates only two types of hosts, we believe that the techniques we developed to accommodate heterogeneity are easily extensible to more than two types of hosts, without significant additional overhead. For conversion of user-defined data types, the same conversion routines can be used on all machines since the routines only contain structural information. However, for the basic types, separate conversion routines need to be written for each (ordered) pair of machines, with a total of $N \times (N - 1)$ routines for each basic data type allocated in the HDSM space. For the implementor of the HDSM system, this is a one-time only effort and is transparent to the application programmer. In contrast, defining a network standard data format would decrease the conversion coding effort, but increase the run-time conversion overhead.

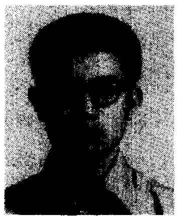
Acknowledgment

T. McInnerny was responsible for most of the Mermaid implementation work, and conducted the measurements reported in this paper. M. Lalovic and S. Fink separately implemented two versions of the data conversion routine generator software. A. Yip performed an initial porting of the PCB software to Mermaid running on Suns. The SOR application is based on a parallel implementation for a Firefly by R. Unrau. Comments by the anonymous referees helped us to improve the presentation of this paper. Partial support for this work was generously provided by the Information Technology Research Center of Ontario and Digital Equipment Corporation.

References

[1] "Network computing systems reference manual," Tech. rep., Apollo Computer Inc., Chelmsford, MA, 1987.
[2] E. Balkovich, S. Lerman, and R. P. Parmelee, "Computing in higher education: The Athena experience," *Commun. ACM*, vol. 28, no. 11, pp. 1214–1224, 1985.
[3] J. Bennet, J. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. PPoPP*, Mar. 1990, pp. 168–176.
[4] 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 Trans. Software Eng.*, vol. SE-13, no. 8, pp. 880–894, 1987.
[5] D. R. Cheriton, "Preliminary thoughts on problem-oriented shared memory: A decentralized approach to distributed systems," *ACM Oper. Syst. Rev.*, vol. 19, no. 4, Oct. 1985.
[6] D. Cohen, "On holy wars and a plea for peace," *IEEE Comput. Mag.*, vol. 14, no. 10, 1981.
[7] C. Pinkerton *et al.*, "A heterogeneous distributed file system," in *Proc. Tenth IEEE Int. Conf. Distributed Comput. Syst.*, 1990.
[8] J. Chase *et al.*, "The Amber system: Parallel programming on a network of multiprocessors," in *Proc. Twelfth ACM Symp. Oper. Syst. Principles*, 1989.
[9] R. Sandberg *et al.*, "Design and implementation of the Sun network filesystem," in *Proc. 1985 Summer USENIX Conf.*, 1985.

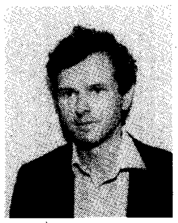
[10] B. D. Fleisch, "Mirage: A coherent distributed shared memory design," in *Proc. 12th ACM Symp. Oper. Syst. Principles*, Dec. 1989, pp. 211–223,
[11] A. Forin, J. Barrera, M. Young, and R. Rashid, "Design, implementation, and performance evaluation of a distributed shared memory server for Mach," in *Proc. 1989 Winter USENIX Conf.*, Jan. 1989.
[12] K. Geihs and U. Holberg, "Retrospective on DACNOS," *Commun. ACM*, vol. 33, no. 2, pp. 439–448, Apr. 1990.
[13] D. P. Geller, "The national software works: Access to distributed files and tools," in *Proc. ACM Nat. Conf.*, Oct. 1977, pp. 39–43,
[14] M. Kaashoek, A. Tanenbaum, S. Hummel, and H. Bal, "An efficient reliable broadcast protocol," *ACM Oper. Syst. Rev.*, vol. 23, no. 4, pp. 5–19, Oct. 1989.
[15] R.E. Kessler and M. Livny, "An analysis of distributed shared memory algorithms," in *Proc. 9th Int. Conf. Distributed Comput. Syst.*, June 1989.
[16] O. Krieger and M. Stumm, "An optimistic approach for consistent replicated data for mulitcomputers," in *Proc. 1990 HICSS*, 1990.
[17] K. Li, "Shared virtual memory on loosely coupled multiprocessors," Ph.D. dissertation, Dep. Comput. Sci., Yale Univ. 1986.
[18] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989.
[19] T. McInnerny, M. Stumm, and S. Zhou, "Mermaid user's guide and programmer's manual," Tech. rep., Computer Systems Research Institute, Univ. Toronto, Sept. 1990.
[20] P. R. McJones and G. F. Swart, "Evolving the UNIX system interface to support multithreaded programs," Tech. Rep. 21, Systems Research Center, Digital Equipment Corp., Sept. 1987.
[21] 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," *Commun. ACM*, vol. 29, no. 3, pp. 184–201, 1986.
[22] D. Notkin, N. Hutchinson, J. Sanislo, and M. Schwartz, "Heterogeneous computing environments: Report on the ACM SIGOPS workshop on accommodating heterogeneity," *Commun. ACM*, vol. 30, no. 2, pp. 142–162, Feb. 1987.
[23] P. Rovner, "Extending Modula-2 to build large integrated systems," *IEEE Software*, vol. 6, pp. 46–57, Nov. 1986.
[24] M. Stumm and S. Zhou, "Algorithms implementing distributed shared memory," *IEEE Comput. Mag.*, vol. 23, no. 5, May 1990.
[25] ——, "Fault tolerant distributed shared memory," in *Proc. IEEE Int. Conf. Parallel Distributed Comput.*, Dec 1990.
[26] "Networking on the Sun workstation," Tech. rep., Sun Microsystems Inc., Mt. View CA, 1985.
[27] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, "Firefly: A multiprocessor workstation," *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 909–920, Aug. 1988.



Songnian Zhou (S'83–M'87) received the B.S. degree from Northeastern University, Boston, MA, in 1982, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1984 and in 1987, respectively.

Since 1987, he has been on the faculty of the Departments of Computer Science and Electrical Engineering at the University of Toronto, where he is currently an Associate Professor. He is a member of the Computer Systems Research Institute. His research interests include parallel computation, multiprocessor operating system, distributed systems, computer networks, and performance evaluation.

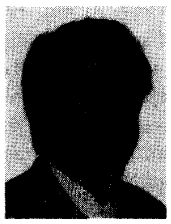
Dr. Zhou is a member of the IEEE Computer Society and the Association for Computing Machinery.



Michael Stumm (M'87) received the diploma in mathematics and the Ph.D. in computer science from the University of Zurich, Switzerland, in 1980 and 1984, respectively.

Since 1987, he has been on the faculty of the Departments of Electrical Engineering and Computer Science at the University of Toronto, where he is currently an Associate Professor. He is a member of the Computer Systems Research Institute. His research interests are in the area of computer systems.

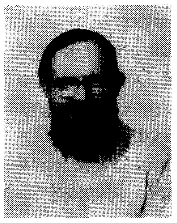
Dr. Stumm is a member of the IEEE Computer Society and the Association for Computing Machinery.



Kai Li received the B.S. degree from Jilin University, China, in 1977, the M.S. degree from the Graduate School of the University of Science and Technology, China, in 1981, and the M.S. and Ph.D. degrees in computer science from Yale University in 1983 and 1986, respectively.

Since 1986, he has been on the faculty of the Department of Computer Science at Princeton University, where he is currently an Associate Professor. His research interests include parallel and distributed systems, parallel programming, and computer architecture.

Dr. Li is a member of the IEEE Computer Society and the Association for Computing Machinery.



David Wortman (S'65–M'70) received the B.S.E.E. degree from Yale University in 1961, and the M.Sc. and Ph.D. degrees in computer science from Stanford University in 1968 and 1972, respectively.

Since 1970 he has been on the faculty of the Department of Computer Science at the University of Toronto where he is currently a Professor. He is a member of the Computer Systems Research Institute at the University of Toronto where he pursues research in the areas of advanced compilation techniques, software engineering, and computer architecture. His most recent research has been on the design and implementation of high-performance concurrent and distributed compilers.

Dr. Wortman is a member of the IEEE, IFIP Working Group 2.4, the IEEE Computer Society and the Association for Computing Machinery.