SHARED VIRTUAL MEMORY
ACCOMMODATING HETERGENEITY

Kai Li
Michael Stumm
David Wortman
Songnian Zhou

CS-TR-210-89

February 1989

Shared Virtual Memory Accommodating Hetergeneity

Kai Li* Michael Stumm† David Wortman†

Songnian Zhou†

CS-TR-210-89

February 22, 1989

Abstract

Heterogeneity exists in almost every research computing environment. Most operating systems accommodate heterogeneity by using a coherent file system that allows clients to access source files in a transparent way. In order to allow heterogeneous computer systems to cooperate, we have applied the framework of shared virtual memory to a heterogeneous computing environment and explore in detail how to accommodate heterogeneity. We have also designed a shared virtual memory system kernel, Mermaid, upon which one can build system components, programming languages, and application software using heterogeneous computers. We are currently implementing Mermaid on a network of SUN workstations and DEC Firefly multiprocessors.

*Department of Computer Science, Princeton University, Princeton, New Jersey 08544. Supported in part by the National Science Foundation under grants CCR-8814265, by the Information Technology Research Center of Ontario, and by the Digital Equipment Corporation (Systems Research Center and External Research Program).

†Computer Systems Research Institute, University of Toronto, Toronto, Ontario, M5S 1A4, Canada. Supported in part by the Information Technology Research Center of Ontario, and by the Digital Equipment Corporation (Systems Research Center and External Research Program).

1 Introduction

Recent advances in VLSI and communication technology have made distributed, concurrent computation a viable technique for dealing with large problems. How to effectively use resources such as secondary storage, processors, and memories in this type of computing environment has become a main focus of many operating system researchers. One of the difficulties involved is accommodating heterogeneity.

Heterogeneity exists in many computing environments. It is usually unavoidable because a specific hardware and its software system is often designed for a particular application domain. For example, supercomputers are good at computation-intensive applications, but not economical at doing file accesses, and poor at user interfaces. Personal computers and workstations usually have very good user interfaces and provide a good cost performance ratio when manipulating files. In order to obtain short turn-around time, a programming environment may allow programmers to write programs on their personal computers or workstations but execute their programs on more powerful computers such as mainframe computers, parallel computers, or supercomputers. We expect that in the future a typical computing environment will consists of a network of personal computers, workstations, mainframe computers, and perhaps supercomputers. These computers may have different hardware and run different software systems.

The diversity of heterogeneous hardware and software systems has created difficulties in interconnection, filing, authentication, naming, and user interfaces. Previous research has attacked these problems from many angles (see [NHSS87] for a survey). The early effort in this area was the NSW project [Gel77] which provided a standard set of tools for a standard file system on different operating systems. More recent research in this area has concentrated on providing high-level coherence in a network file system of the same kind of operating system while permitting the implementation on different hardware. SUN's NFS, the Athena project at MIT [BLP85], and the Andrew project at Carnegie-Mellon Information Technology Center [MSC*86] are such examples. The Locus operating system developed at UCLA [WPE*83] concentrated on building transparent operating system bridges to integrate computational resources with a high degree of transparency. The Mercury project attacked the heterogeneity problem from the angle of sharing programs in very different languages such as LISP and CLU.

There are also research efforts in doing parallel and distributed computing in a heterogeneous computing environment. The Heterogeneous Computer Systems project at the University of Washington and the Network Computer Systems at Apollo [Apo87] concentrated on heterogeneous remote procedure call mechanisms to support distributed and parallel programs based on the client-server or the message-passing model. *Remote procedure call* (RPC) [Nel81, BN84] is a mechanism for the synchronous language-level transfer of control between two programs in disjoint address spaces where the primary communication medium is a narrow channel [Nel81]. The RPC mechanism allows programmers to worry less about data movement and provides clients with a fairly transparent interface so that remote procedure calls look much like local

procedure calls.

Although an RPC mechanism provides syntax and semantics similar to local procedure calls within the application program's high-level language, passing complex data structures, such as lists, requires much overhead and introduces difficulties in maintaining multiple copies of data. For example, when passing lists, because a list may be circular, a table is needed to keep track of what elements have been sent and perform a table lookup for every element. The problem with passing complex data structures becomes more severe when the data structures are fundamental to a system such as a parallel LISP programming environment [Hal85]. A heterogeneous RPC mechanism [BCL*87] requires additional overhead for marshalling data structures in local representations to standard representations and vice versa.

Shared virtual memory [Li86, LH86] can solve these problems in a homogeneous multiprocessor environment efficiently and conveniently. A shared virtual memory system provides clients with a large, coherent memory address space that is shared by all processors. Each processor can access any memory location in the shared virtual memory address space at any time. In fact, a shared virtual memory system transforms a loosely-coupled multiprocessor into a shared-memory multiprocessor. Therefore, there is no need to pack and unpack the data structures containing pointers in messages. Passing a list data structure simply requires passing a pointer. Programmers can program with the shared-memory model in addition to the client-server model or message-passing model. Shared virtual memory provides more transparency than the RPC mechanism.

The nice properties of shared virtual memory motivated us to investigate the possibility of designing a shared virtual memory system for a heterogeneous environment. This paper explores in detail how to accommodate heterogeneity in a shared virtual memory, designed for a network of heterogeneous computers, and then describes a kernel design, Mermaid, that supports higher level software components using heterogeneous machines. We will also describe how to use Mermaid to implement an object-oriented language and a database system.

We are currently implementing Mermaid on a network of SUN workstations and the DEC Firefly multiprocessors. Our goal is to let users do their programming on their workstations and run their programs on multiprocessors in a transparent way. We also would like to demonstrate how to use such a shared virtual memory system as a base to build system components using heterogeneous machines.

2 Shared Virtual Memory

A shared virtual memory is a coherent virtual memory address space shared by all processors in a system [LH86, Li86]. Figure 1 shows the system architecture of a shared virtual memory system. Each node in the figure represents a processor and memory of a workstation or a small-scale shared-memory multiprocessor.

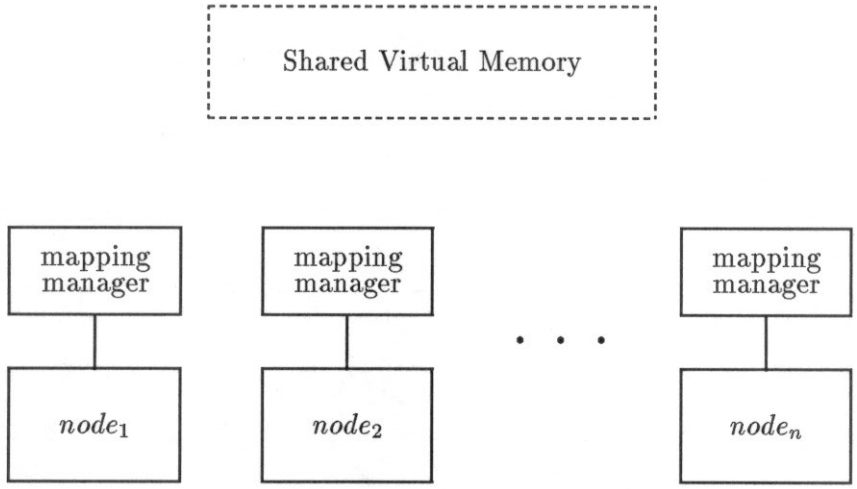

Figure 1: Shared virtual memory architecture.

The shared virtual memory system presents all nodes with a set of coherent shared memory pages. Any node can make memory references to any location of any page at any time. These shared pages are coherent at all times, that is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address.

The memory mapping manager views its local memory as a large cache for its associated processors. Pages that are marked "read-only" can have copies residing in the physical memories of many processors at the same time. But a page currently being written can reside in the physical memory of only one processor. If a processor wants to write a page that is currently residing on other processors, it must get an up-to-date copy of the page and then invalidate all copies on other processors. Like traditional virtual memory [Den80], the shared memory itself is *virtual*. A memory reference may cause a page fault when the page containing the memory location is not in a processor's current physical memory. When this happens, the memory mapping manager retrieves the page from either a disk or the memory of another processor.

The hardware Memory Management Unit (MMU) for implementing traditional virtual memory systems can be used to implement such a system. The protection mechanism of an MMU allows single instructions to trigger page faults and trap the faults in appropriate fault handlers. A program can set the access rights (nil, read-only, writable) in such a way that a memory access that could violate memory coherence causes a page fault, and thus the memory coherence problem can be solved in a modular way in the page fault handlers and their servers. To client

programs, this is completely transparent.

In general, the shared virtual memory system presents clients with the same interface as a tightly-coupled shared-memory multiprocessor. The shared virtual memory system is like the traditional virtual memory system in the sense that it can utilize the capacities of both physical memories and disks in the system.

In order to support programming based on both the shared-memory model and the client-server model, a shared virtual memory system has an integrating thread manager. Threads can also send messages to each other. A thread can address any location in the shared virtual memory address space. Similar to the threads in Topaz [MS87], the cost of a thread context switch, thread creation, or thread termination, is small. For example, the cost of creating a thread is a few procedure calls, because threads share the same address space and therefore there is no need to set up page tables and flush related caches for a context switch. A thread manager provides clients with a set of thread control primitives and a set of traditional synchronization primitives. Threads are transparent in such a system, that is, a thread can run on any processor and can migrate from one node to another at run time, if the destination processor has the same instruction set as the source processor. For example, a thread can be created on a VAX processor based workstation, run for a while and then migrated to a Firefly multiprocessor.

3 Accommodating Heterogeneity

There are many approaches to accommodating heterogeneity in building a shared virtual memory system. Our goal is to design a kernel upon which one can build parallel and distributed systems in a heterogeneous environment efficiently and conveniently. Our approach to the goal is to keep things as simple as possible and to provide only enough funcationality for building higher level system components.

3.1 Page Size

Previous research on shared virtual memory [Li86] shows that a shared virtual memory system can keep its memory space coherent if page-level coherence is maintained. Since the work was based on homogeneous loosely-coupled multiple processors, it implies that there is only one page size. In a heterogeneous system, different machines may have different page sizes for implementing their traditional virtual memory systems. For example, on the SUN-3 workstation, the MMU page size is 8K whereas on the DEC Firefly, it is 512 bytes. The problem is how to accommodate the diversity of different MMU page sizes in implementing a shared virtual memory system.

An obvious solution is to use the largest page size as that of the shared virtual memory system. Since page sizes of MMUs are always powers of two, the largest page size is always a multiple of the smaller ones. For example, on the DEC Firefly, we can group 16 MMU pages

together into an 8K byte page as a shared virtual memory page to match the 8K byte page size on the SUN-3. A page fault detected on a Firefly will cause all 16 512-byte pages to react in the same way as a page on the SUN. The advantage of this approach is its simplicity.

For some application domains, using large page sizes is good because the average cost of maintaining memory coherence is low. For other application domains, it may cause more memory contention than when using smaller page sizes. In a typical loosely-coupled multiprocessor, the startup cost of sending a packet is relatively high, so that using a large page size can amortize the cost of startups. The cost of the startup consists of software protocols and operating system overhead. If these overheads are acceptable, relatively large memory units are possible in a shared virtual memory. On the other hand, the larger the memory unit, the greater the chance for contention. Although the shared virtual memory storage management may allocate memory in a smart way to reduce the contention, but it may introduce inefficient use of memory. So, the possibility of contention indicates the need for relatively small memory units.

To accommodate the contention problem, we can allow for use of different page sizes in a shared virtual memory system. Using different page sizes requires changing the memory coherence algorithm in the page fault handlers and their servers on the machines using smaller page sizes. For convenience in the following discussion, let us call the largest page in a system an SVM page or simply a page and call the pages on machines with smaller page sizes subpages. For example, if the largest page in the system is 8K bytes, then the SUN-3's page is an SVM page and the Firefly's page is a subpage of an SVM page. In order to maintain the address space coherent at all times, memory needs to be coherent at the SVM page level. Hence, we can manage ownership at the SVM page level, but maintain read-only copies at the subpage level. For processors with subpage sizes, a read fault can be less expensive since only one subpage needs to be transfered, as opposed to all subpages that make up an SVM page. More detailed discussion on different algorithms can be found in [Li88b].

3.2 Code and Data

In a homogeneous environment, all processors execute the same instruction set, so threads in the same shared virtual memory address space can migrate dynamically from one processor to another at any time [Li88a]. In a network of heterogeneous machines, the situation is quite different. Different hardware executes different instruction sets and the same hardware may be running different operating systems which will also require different code [Coh81]. Threads cannot simply migrate from one machine to another.

A simple way of accommodating different instruction sets is to let thread management module understand machine types. A thread created on a particular type of machine can be restricted to only migrate to machines of the same type. With this approach, either clients will need to plan well where threads should be created, or the system will need to supply an "intelligent" thread creation routine that can decide where threads should be created. In either case, if the machine type of a thread is determined at run time, then the image of

the program must include machine instructions for every machine type. This replication of executable programs is unavoidable if the program is to execute efficiently on different hardware. Thus, the executable image should have multiple code segments as shown in Figure 2.

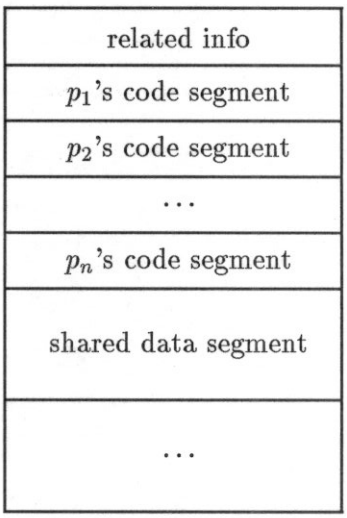

Figure 2: An object code layout

Machines of different types may have different data representations for atomic elements like integers, reals and characters and for composite objects like arrays and records. Conversion of a block of data from its representation on one machine to an equivalent representation on another machine requires complete knowledge of the types used to create the block of data. This knowledge must either be encoded with the data or provided in some auxilliary tables. The information necessary to describe the data objects used in a program could be easily generated during the compilation of the program. Usually the conversion for atomic elements is straight forward and can be efficiently implemented by small utility functions that are optimized for speed. Conversion of composite objects can be accomplished by repeatedly applying the atomic element conversion routines to the contents of composite objects. With efficient conversion routines for atomic data types among all the machines in the environment, a compiler can generate the conversion code for a composite object for all machines during one compilation.

In order to provide clients with a more transparent interface, the system may need to convert shared data into appropriate representations on demand. The basic idea for doing such data conversions is to separate data pages from code pages and allow each page to have different representations for different machines when it has multiple read copies. Conversion of data representation is an unavoidable cost of using a heterogeneous computing environment. An attempt is made to organize the data such that each page contains data items of one type only. The system, if it understands this type, can then convert the page when it migrates, if necessary. We call this method *convert-on-reference*.

This method requires data type information about each page. One way is to let the page

table of the shared virtual memory address space include a new field *type* for each entry. This field indicates the data representation type of the page. According to the page type and the current machine type, an appropriate data conversion routine will be invoked when a read or write page fault occurs. If the value of the field *type* is nil, no conversion will be performed. The shared virtual memory system has a number of built-in conversion routines for most data types.

The main advantage of this method is that it avoids data conversions when a page moves among machines of the same type. The main disadvantage is that the convertion mechanism has to know the data types of each page. This is not a problem in many language implementations, such as LISPs, functional languages, and object-oriented languages, that have sophisticated storage managements in which each storage allocation unit has an associated data type. The storage allocator usually puts data items of the same type on the same page for the convenience of garbage collection and for saving space for tag bits. It is easy to use the same data type information for the conversion mechanism.

3.3 Thread Migration

In order to allow threads to migrate among heterogeneous machines, we can define *migratable points* in threads. A migratable point of a thread is the point in the program where it is safe to migrate the thread to a machine of another type. For the convenience in implementation, one can treat different migratable points with different implementations. The very beginning of a program is always a migratable point that allows creation of the program on any machine. The shared virtual memory system is not required to do any work because there are multiple code segments in the executable image.

The existance of other migratable points in a program depends on its structure, the language used, and its implementation on the different machines. Two desirable properties for migratable points are that the program's state is not changing asynchronously (i.e. no input/output is in progress) and that the amount of program state at the migration point is relatively small. As a specific instance, a server program usually has migratable points at the beginning of each service. For example, the server program structure shown in Figure 3 will have a migratable point at the beginning of each iteration. We call such a migratable point *iterative migratable point* and there is one such a point for each thread.

To migrate a server program at its iterative migratable point among machines of k types, the system needs to initialize states, including global data structures and stacks, such that the program can migrate immediately at its iterative migratable point without any complicated state conversion and transfer. An easy way to achieve such a task is to start up k instances of the program, one on a machine of each type. One of the instances will continue running while the other $k-1$ instances will be suspended at the iterative migratable point, waiting for future migration operations. When migrating, if the destination processor has an instance of the program suspended at the migratable point, the migration operation entails simply a

```
main()
{

    while ( 1 ) {
        IterativeMigrateCheck();  /* migratable point */

        ...                       /* actual code */
        }
}
```

Figure 3: A server program structure.

resume operation on that instance of the program. If the destination processor does not have a suspended instance of the program at the migratable point, the system finds a suspended instance on a machine of the same type as the destination processor, resumes that instance on such a machine, and then migrates it to the desitination processor.

Another kind of migratable point is called *procedural migratable point*. Every procedure call in a program can be specified (or identified) as a procedural migratable point. To describe the implementation of procedural migratable points, we can divide them into three classes. The first class is the set of procedures (callees of migratable points) that reference only local variables and global variables. The second class is the set of procedures that reference lexically scoped variables including arguments, local variables, outer-scope variables, and global variables. The third class is the most general kind that references not only lexically-scoped variables but also dynamically-scoped variables.

A procedural migratable point of the first class is semantically much like a remote procedure call in the sense that a transfer of control is specified explicitly and the callee procedure does not require any initial stack states since it does not reference any outer-scope variables. The implementation of such procedural migratable points is easy in the shared virtual memory system. It can simply suspend the current thread (caller), start the execution of the procedure (callee) on the desitination processor, and resume the caller thread when the execution of the procedure is complete. It is more or less like coroutine except that the execution of the procedure is on another processor. It is more convenient than an RPC for writing many parallel programs because thread migration allows the thread to reference global variables at will without any marshalling.

The second class of procedural migratable points allows references to outer-scope variables that are statically defined. For languages that are lexically scoped, all procedures belong to this class. The implementation for such procedual migratable points is more complicated since data conversion on stack is not trivial. For simplicity, one might consider restricting this class to allow references to outer-scope variables that appear only in the procedure arguments, in addition to local variables and global variables. Such a restricted class is still stronger than the

remote procedure call mechanism because it allows passing arguments by reference in addition to passing arguments by value and allows references to global variables; whereas a remote procedure call mechanism allows passing arugments only by value and disallow any references to global variables. Like the remote procedure call mechanism, for the restricted class, the implementation of passing arugments between the caller and callee requires compiler assistance such that "in" arguments will be put onto the stack of the destination processor properly and "out" arguments will update corresponding variables in the caller's scope. In addition, the implementation needs to convert argument data on the fly.

The implementation of unrestricted second class of procedural migratable points is difficult, because its implementation requires complete knowledge of the data conversion of the stacks of between any two machines. How to implement efficiently is still an open research problem.

Althought the third class of procedual migratable points is the most general class, its implementation is very difficult. The system has to understand not only how to convert data on stack among machines, but also how to convert data in dynamic scopes. One may argue that dynamic scoping has become less important in programming languages since most programming languages are moving towards static scoping. How to implement the third class of procedual migrable points is still an interesting problem.

The discussions above have always assumed that migratable points are specified explicitly in programs. It is a challenge how to automatically identify migratable points for programs for each kind, in particular, how to automatically distinguish different classes of procedual migratable points. Also, it would be interesting to find other kinds of migratable points. These are our future research topics.

3.4 Communication and I/O

The implementation of a shared virtual memory system requires a common communication protocol to implement its remote operations and page transfers. To design a module that implements such a protocol, we need a common underlying protocol to accommodate the diversity of machines. Such an underlying protocol always exists since we assume the target heterogeneous machines are already connected.

If more than one common protocol is available, we should choose the one that is simple to use, yet efficient. For example, in most research environments, the Internet Transmission Control Protocol (TCP) is commonly used. This protocol provides for reliable, stream-oriented inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer networks. It appears to be easy to use, but its performance will probably be inadequate. On the other hand, the User Datagram Protocol (UDP) is common and has low overhead. Such a protocol uses the Internet Protocol (IP) as the underlying protocol and does not provide for reliable communication. For efficiency purpose, one would probably want to implement the remote operation module for the shared virtual memory system with such

a protocol. Sharing I/O resources is very important. Most machines have their own displays, disks, and tape drives. To allow clients to use the shared virtual memory in a transparent way, we should allow a thread to accept input from the keyboard of a workstation, to execute programs on a powerful machine, and then display results on the display of the workstation again.

It is necessary for a shared virtual memory system to use secondary storage such as disks and tape drives in a transparent way. When implementing a database system on top of a shared virtual memory system, for example, one can view disks as concurrent secondary storage or a massive and high-performance storage system. Most operating systems nowadays use files to access secondary storage such as disks and tape drives. A basic requirement of file sharing in a shared virtual memory system is to have a consistent view of all files on different file systems. This is much more difficult than implementing an FTP program which simply transfers and translates files. In fact, translating typed files can be rather complex itself. To simplify the design while providing efficient sharing, one may consider using files of only one type and maintaining consistency via the coherent shared virtual memory system.

4 Mermaid System

To bring the ideas of accommodating heterogeneity into reality, we have designed a system, called Mermaid, for a network of SUN-3 workstations, and DEC Firefly multiprocessors. Mermaid is a shared virtual memory kernel that can support system components, using both kinds of machines.

A user can edit a program on her SUN workstation, compile her programs there, and run her program under a Unix shell. All or part of her program may execute transparently on the Firefly multiprocessors. During the execution, the SUN workstation can be used to input to and output from the program running on the Firefly multiprocessors. The program can use both the Unix/Ultrix NFS files and the Topaz RFS files.

The first prototype Mermaid we are currently implementing is a user-mode implementation designed for experimentation. The structure of the system is the same on both target machines as shown in Figure 4. The implementations are quite different, because the operating system interfaces are different.

The memory mapping module implements the shared virtual memory mapping, and its coherence algorithm in particular. The thread management provides clients with a set of primitives for thread control and thread synchronization. The storage management is responsible for shared virtual memory allocation and garbage collection. The file package provides clients with a transparent view of the heterogeneous file systems. The remote operation module provides a mechanism for all remote operations of other four modules. These five modules will eventually move into the kernel.

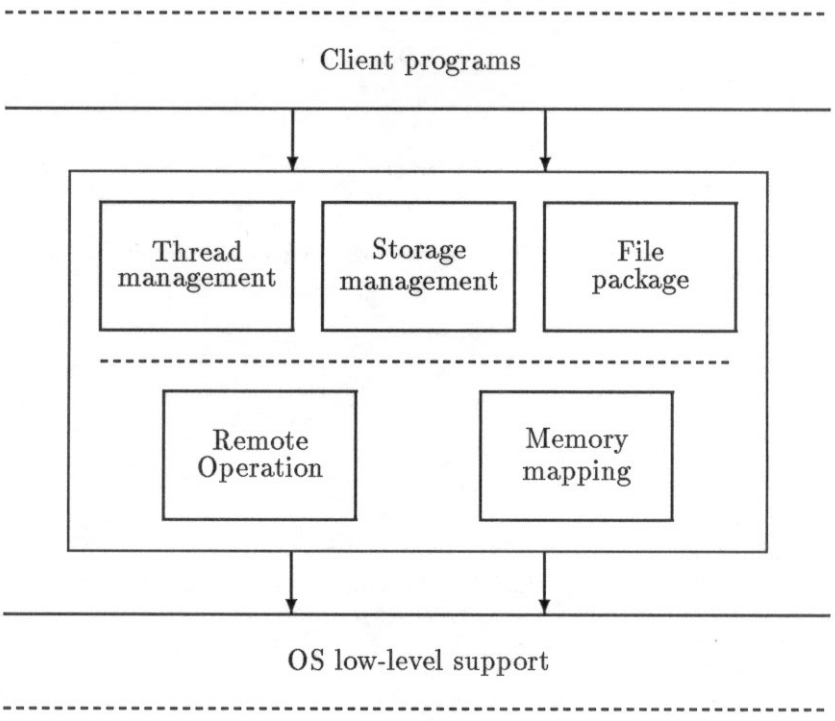

Figure 4: System Structure.

4.1 Heterogeneous Environment

A SUN-3 workstation [Sun86] is a Motorola 68020 based machine with usually 4M bytes or more memory. The SUN wokstations run the SUN Unix operating system with the Unix Network File System (NFS). Most SUNs do not have local disks and instead access file servers over the Ethernet.

The DEC Firefly [TS87] is an experimental shared-memory multiprocessor developed at the DEC Systems Research Center. Each Firefly consists of up to seven MicroVAX 78032 or CVAX processors, each with a floating point unit and a 16 KByte cache. The caches are coherent, so that all processors see a consistent view of up to 32 megabytes of shared memory. The operating system on the Firefly is called Topaz whose file system currently is Remote File System (RFS). The only user-level communication mechanism is the remote procedure call (RPC) embeded in Modula-2+ programming language.

The main reason for selecting these two kinds of machines is that their architectures are substantially different. The main differences are:

- Different processors that execute different machine code.
- Different MMU page sizes. The SUN-3's MMU has a page size of 8K bytes; whereas the Firefly uses the standard VAX MMU which supports 512-byte pages.

- Different byte order. The byte order of a word and a long word on the two systems are completely different.
- Different floating point representation. The SUN-3 uses the IEEE standard; whereas the Firefly uses the DEC representation.

The only common data representation is character or byte stream. Both machines use the ASCII representation for characters and have the same byte stream representation in memory.

The SUN workstations and the Fireflies are connected by an Ethernet. They all implement the same UDP/IP protocol which gives us a simple way to accommodate the heterogeneity of interconnection.

4.2 Shared Virtual Memory Mapping

A memory mapping manager implements the mapping between a local memory and the shared virtual memory address space. Other than mapping, their chief responsibility is to keep the address space *coherent* at all times; that is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address.

Each user address space is divided into two portions. The shared virtual memory address space is in the high portion and the private memory is in the low portion. For simplicity, the data structure of the page table is a vector of records and each record is a table entry. The whole table is stored in the private memory. Since our initial implementation is in user space, the page table is used only by the page fault handlers and their servers. This table is a mirror data structure of the page table used by the MMUs. Clearly, when the memory mapping manager moves into kernel mode, this table will be merged with the MMU page table.

The memory mapping manager uses the access protection faulting mechanism provided by the MMU hardware to keep the shared virtual memory address space coherent. For simplicity, we decide to use a uniform page size of 8K bytes for both the SUNs and the Fireflies in our first implementation. Such a prototype will serve as a base for implementing the coherence algorithms with subpages.

The system interface on the Firefly is sufficient to support a user-mode shared virtual memory implementation. Unfortunately, there is no system call on the SUN Unix that allows a client program to set the access mode (nil, read-only, or writable) for each page. We have added in such a call to the kernel. SUN Unix also does not return a faulting address to the user address space when there is a protection fault. This has also been fixed. Unlike the VAX architecture, the Motorola 68020 does not provide information about whether an instruction will modify memory when a protection fault occurs. For example, instruction `addl x, (y)` may cause a read fault on reading location `(y)`, even though the instruction actually wants to write to `(y)`. in contrast, the VAX architecture indicates to the fault handler that an instruction will write to a page. Therefore, on the SUN, this may lead to a read page fault followed immediately by

a write fault. In order to avoid this situation, we decided to analyze the faulting instruction on the SUN to determine whether an instruction with a read page fault will lead to a write fault.

Initially, all processors set the protection of all pages in the shared virtual memory address space portion to nil. Allocated memory pages will change their protections to writable. The memory mapping manager uses the protection fault handlers to maintain the shared virtual memory address space coherent.

To keep the Sun space coherent, we are implementing the dynamic distributed manager algorithm [LH86]. Briefly, it keeps track of the ownership of all pages in each processor's local page table, using a field called *probOwner* in each page entry. The value of this field can be either the true owner or the "probable" owner of the page. The information that it contains is just a hint; it is not necessarily correct at all times, but if incorrect it will at least provide the beginning of a sequence of processors in which the true owner can be found. Initially, the *probOwner* field of every entry on all processors is set to some default processor that can be considered the initial owner of all pages. As the system runs, each processor uses the *probOwner* field to keep track of the last change of the ownership of a page. This field is updated whenever a processor receives an invalidation request, relinquishes ownership of the page, or forwards a page fault request. The atomicity of each update is guaranteed by using a lock field in each entry of the page table.

The shared virtual memory mapping contains the implementation for *convert-on-reference*. To identify the data type of each page, a data type field is included in each entry of the page table. When a page fault occurs, the shared virtual memory mapping mechanism will check the data type against the current machine to decide what data conversion should be done. The first implementation contains only the conversion routines for 16-bit and 32-bit integers.

4.3 Thread Management

The basic requirement of the thread management module is to provide clients with a uniform interface such that clients view their threads and their shared virtual memory address space as if they were provided by a tightly-coupled shared-memory multiprocessor. Thread management provides basic primitives for thread creation, termination, migration, and synchronization.

Our first concern is how to integrate the thread control mechanism on SUN Unix and Firefly Topaz. Since there is no thread implementation on the SUN Unix, we have two design choices. The first is to build a thread mechanism in the kernel as Mach did [RTY*87]. The second is to build a thread scheduling mechanism inside a Unix process. Such a thread scheduling mechanism can be either preemptive or nonpreemptive. We decide to do the latter and postpone the former to when we move the code into the kernel. One may worry about the lack of I/O overlaps in such a scheme. In other words, a thread doing a sysnchronous I/O will block all other threads on the same processor. To overcome this drawback, we decided to implement our own file interface supported by the remote operation module. Such an interface allows the

system to run another thread while the current thread is blocked on a file operation. This scheme still has the drawback, however, that a local VM page fault will block all the SVM threads in the same address space.

The Firefly Topaz provides clients with a thread interface. Threads in different address spaces are scheduled by the same scheduler. We also have two design choices to implement shared virtual memory threads. One is to have another level of scheduling and use a fixed number of Topaz threads. We then bind a SVM thread to a Topaz thread when the SVM thread is scheduled to run. In this case, Topaz threads can be viewed as virtual processors. Another design choice is to use a Topaz thread for each SVM thread. The latter method is more difficult than the former, but we choose it because this approach allows thread scheduling on all I/O operations.

The second concern is what the synchronization primitives should be. Thread synchronization primitives must use atomic instructions in their implementation. In order to accommodate the diversity of atomic instructions on different machines, we need to select a set of primitives that can be implemented efficiently on all machines. For example, binary P/V operations, wait/signal, or eventcounts can be implemented equally efficiently on almost all machines types. We decided to use the thread interface provided by Topaz as a base from which we chose binary P/V and wait/signal. This set of primitives is easy to use and can be implemented efficiently on both SUNs and VAXes.

Whether to allow semaphores to migrate is another concern. To allow migration, we essentially have to implement a coherence algorithm for the semaphores. We can implement migratable semaphores with remote operations or the shared virtual memory address space [Li88a]. The shared virtual memory address space is a simple approach, but in our environment this may cause a lot of overhead since the SVM page size is 8K bytes. For this reason, we decided to use remote operations to implement migratable semaphores.

The thread management also contains mechanisms for thread migration at iterative migratable points and procedural migratable points. Clients are required to specify migratable points explicitly in their programs. The Mermaid system implements the restricted second class migratable points which allow the callee procedures to reference arguments, local variables, and global variables.

4.4 Storage Management

The storage management module consists of an allocator and a collector. Since garbage collection algorithms for a shared virtual memory environment is still an open research problem and this system is designed for simplicity, we will not implement a sophisticated collector. Instead we will provide for explicit free operations. The allocator uses a "first fit" algorithm with one-level centralized control. One processor will be appointed to the centralized memory manager. To reduce memory contention, the memory allocator aligns allocated chunks to page

boundaries.

A two-level memory management would be a more efficient approach, where each processor has a local allocator maintaining a large chunk of memory allocated from the central memory allocator. This large chunk of memory is used for the local memory allocations. When a local memory allocator runs out of free memory, it will allocate another large chunk from the central allocator. We plan to use this approach in our next implementation.

4.5 Remote Operation Module

The remote operation module provides the shared virtual memory system with an efficient mechanism for handling all remote operations in other modules. Reliability and efficiency are the main goals of this module.

The remote operation module uses the standard timeout-retransmission protocol to support the following three kinds of remote operations:

- simple request/reply,
- request/forward/reply, and
- multicast request/reply.

A page can be included as part of any request or reply message. All remote operations can be either synchronous or asynchronous.

The request/forward/reply mechanism allows a processor that received a request to forward it to another processor. For example, processor 1 can send a request to processor 2, processor 2 forwards the request to processor 3, and so on until processor k performs the operation and sends a reply back to processor 1. There are no intermediate replies involved in the operation. This mechanism is particularly useful for implementing the dynamic distributed manager algorithm.

The multicast request/reply mechanism has three reply options: a reply from any receiving processor, replies from all receiving processors, or no reply at all. The first option is useful for locating some interesting information. The second option is used for implementing invalidation operations. The third option is used for broadcasting approximate information.

The SUNs and the Fireflies are connected to an Ethernet in our computing environment. Both systems use the User Datagram Protocol (UDP), whose underlying protocol is the Internet Protocol (IP). Both systems have their own remote procedure call (RPC) mechanisms implemented on top of the UDP protocol. Althought they are quite efficient, they are very different and tied to different programming languages. Also, the RPC mechanisms do not support asynchronous, forwarding or multicast requests. For these reasons, we decided to implement our own remote operation module using the UDP protocol.

The remote operation modules on both systems are similar. The only difference is the way they hook up with their thread schedulers. The main goal of this module is to multiplex remote

operations among threads and maximize the utilization of processors and the network. A thread performing a synchronous remote operation should be suspended until the completion of the remote operation, so that other threads can run in the mean time.

4.6 File Access

File access is important to many applications. One goal of our system is to allow clients to build system components in a more transparent way. Hence, in our design, we allow any thread to access any file in the system at any time. That is, a thread running on either a SUN or a Firefly can access both RFS files and NFS files.

The obvious way to achieve this is to implement our own file system using the facility provided by both file systems. There are many design choices. For example, the question of whether we want to implement the file system at the level of individual machines or at the level of file systems? We chose the file system level for simplicity and we will implement it as a user library package.

This package contains mainly two things: a naming module and a file operation module. The naming module provides a translation mechanism for a given file name. The basic rules used in the naming module are:

- Use a hierachical name space. The general form is `/FileSystem/Machine/DirectoryAndFileName`.
- If the machine field is missing, then the name service will pick a machine name from the available machines that run the specified local file system.

For example, `/nfs/hoskin/f/li/foo.txt` means the file `foo.txt` is an NFS file residing in the directory `/f/li` and that it can be accessed through machine `hoskin` first. And the name `/nfs/f/li/foo.txt` refers to the same file, but the naming service may translate into `/nsf/king/f/li/foo.txt` where `king` is a machine name picked by the service.

The file operation module contains primitives such as create, open, close, read and write. Create, open, and close are implemented by remote operations. These operations are always performed on the machines of the file systems specified. An open file operation is implemented by mapping the file into the shared virtual memory space, in a form of lazy evaluation. We use 8K bytes as a unit of each mapping invocation. A close operation is implemented by unmapping the file. Read and write operations are implemented as reads and writes from and to the mapped portions in the mapped shared virtual memory space. We expect such a design to provide us with good performance in both fetching and caching [LLD*83, Che88].

4.7 Linker, Loader, and Shell

The basic tools needed for the shared virtual memory system to accommodate heterogeneity are a linker, a loader and a simple shell. The linker takes different machine code object files in Unix object file representation, and links them together into an executable image with multiple code segments and data segments. In our current implementation, an image has only two code segments (68020 and VAX) and three data segments (two private and one shared).

The loader can load an executable image into the shared virtual memory address space and execute it. To execute such an image, an initial thread on each kind of machine will start at its code segment.

The simple shell provides clients with a minimum interface to the file package and the execution of heterogeneous code images. The initial set of commands in the shell is a subset of the Unix shell csh without programming ability. The file names in the shell will be translated by the simple naming service.

4.8 Restrictions

The shared virtual memory system outlined above supports many system components using heterogeous machines in a transparent way. Since it is designed based on the philosophy of simplicity, it has a number of restrictions.

The first restriction is that threads cannot migrate freely among different types of machines. Although our current design allows the restricted class of procedural migratable points which is stronger than the RPC approach, the callee procedures are allowed to reference only arguments, local variables and global variables. Outer scope variables rather than arguments are not allowed. We are still doing research on how to simplify the implementation of other class of migratable points.

For simplicity, our first implementation does not include all atomic data types for convert-on-reference data conversion. Since the MC68000 and the VAX have the same byte representation [Coh81] and the system implements byte order swapping for words and longwords, but they have the same byte representation [Coh81], it is sufficient for many applications.

5 How to Use Mermaid

In this section, we discuss what we can build with the Mermaid system by discussing two examples. In both examples, we will try to explain how systems that require data sharing can be implemented on the shared virtual memory and do their own data conversion. Since we have not built any of these examples yet, the discussion serves only for further understanding of our shared virtual memory system.

5.1 An Object-oriented System

An object in a typical object-oriented system consists of a set of data items and some access methods associated with it. The only way to access the data of an object is via its access methods.

To implement an object-oriented system on the shared virtual memory, every access method can check the data representation of an object before performing its operation. A tag in each object can be used to record its current data representation. If the data representation is compatible with the current machine type, no conversion will be necessary. For example, when a thread running on a VAX accesses an object whose data is stored using the SUN representation, the system will first perform a data conversion such as byte-order swapping, translating the IEEE floating point representation into the VAX floating point representation, and so on. If the data is already in the VAX data representation, no conversion is needed. This is in contrast to the heterogeneous RPC mechanism in which conversion to a standard data representation is always enforced.

Further optimization to minimize the need for data conversion is possible. Instead of using a tag to record the data representation of the entire object, a tag could be used for each data item, so that data conversions can be performed at data item level. This approach requires the knowledge of which access method will access which data items, and also requires additional space for tags.

Implementing a distributed, object-oriented system based on the RPC mechanism or a message-passing system for a heterogeneous environment can be very complicated. In addition to data conversions, a distributed implementation requires locating the object before invoking an access method [ABLN83], and maintaining the consistency of objects if the system supports caching. Furthermore, the overhead of passing complex data structures can be very high and it would be rather complicated to implement collectable storage management.

The main advantage of using the shared virtual memory system to build an object-oriented system over using the heterogeneous RPC mechanism is that the property of coherent memory greatly simplifies the implementation of the object-oriented system. Using the shared virtual memory system, the shared virtual memory mapping mechanism provides clients with the same shared-memory space as that on a tightly-coupled multiprocessor. The difficulties of locating objects, consistency of multiple copies, and marshalling complex data structures are non-existant.

5.2 A Database system

It is well-known that building a distributed database is not trivial [RBF*77, SN77], especially on a heterogeneous environment. A database implementation may require both data conversions and the coherence and consistency of the database. Although we can use an object-oriented

system to implement a database to avoid doing any client-level data conversion, the approach may not be appropriate since some database systems require fast response time.

If it is sufficient to use byte vectors to hold data in a database implementation, then the shared virtual memory system will not require database implementors to do any extra work in terms of data conversion. If sharing byte vectors does not suffice, then we can partition a database into different portions and let each partition have its own data representation. A database query may migrate from one machine to another in order to access appropriate relations without performing any data conversion. This would greatly improve the performance if the database is large. A user query may start on a workstation, migrate to a supercomputer, and finally come back to the workstation with its result. This type of query processing can be conveniently implemented with the thread management in our system.

The shared virtual memory system reduces the problem of implementing a database system on multiple machines to a multi-user-centralized database system problem. Although database designers need to work out the details for each individual system design, including such problems as recovery scheme and logging, the difficulties of concurrency control and distributed database consistency no longer exist.

6 Conclusions

Our research motivation in accommodating heterogeneity stems mainly from the observation that most workstation systems have better user interfaces than supercomputers. We also believe that as communiation technology improves, parallel processing and sharing of resources, such as processors, memories, and secondary storage, in a research computing environment will become more important. Our research effort is directed towards this goal.

We have discussed the main issues and solutions of building a shared virtual memory system on a network of heterogeneous machines. Compared with related work in heterogeneous RPC, the shared virtual memory approach provides clients with a more transparent interface and a programming environment fitting the shared-memory model of parallelism.

As a practical research effort, we designed a shared virtual memory system for a network of SUN workstations and Firefly multiprocessors. Although this design is simple, it provides a more transparent way of building systems in a heterogeneous environement than with the heterogeneous remote procedure call paradigm.

We are currently implementing the Mermaid system presented in this paper. Part of the implementation is already operational. We plan to do experiments and measurements on the system in the near future. We feel the research has given us a lot of insight into parallel and distributed computing in a heterogeneous environment in general.

References

[ABLN83] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden System: A Technical Review. Technical Report 83-10-05, University of Washington, October 1983.

[Apo87] Apollo. *Network Computing Systems Reference Manual.* Apollo Computer Inc., Chelmsford, Mass., 1987.

[BCL*87] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo, and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[BLP85] E. Balkovich, S. Lerman, and R.P. Parmelee. Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 28:1214–1224, November 1985.

[BN84] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Che88] David R. Cheriton. The Unified Management of Memory in the V Distributed System. Draft, 1988.

[Coh81] Danny Cohen. a Plea for Peace. *IEEE Computer*, pages 49–54, October 1981.

[Den80] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[Gel77] D.P. Geller. The National Software Works: Access to Distributed Files and Tools. In *Proceedings of the ACM National Conference*, pages 39–43, October 1977.

[Hal85] Robert H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, pages 501–538, October 1985.

[LH86] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors.* PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[Li88a] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.

[Li88b] Kai Li. Memory Coherence with Non-uniform Block Sizes. In preparation, 1988.

[LLD*83] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, 1983.

[MS87] P.R. McJones and G.F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. Tech Report 21, DEC Systems Research Center, September 1987.

[MSC*86] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith. Andrew: A Distributed Perosnal Computing Environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[Nel81] Bruce J. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University, May 1981.

[NHSS87] D. Notkin, N. Hutchinson, J. Sanislo, and M. Schwartz. Heterogeneous Computing Envirnments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity. *Communications of the ACM*, 30(2):162–142, February 1987.

[RBF*77] J.B. Rothnie, P.A. Bernstein Jr., F. Fox, N. Goodman, M. Hammere, T.A. Landers, C. Reeve, D. Shipman, and E. Wong. Introduction to System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 5(1), 1977.

[RTY*87] R.F. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architecture. In *Second International Conference on Architectural Support for Programming Lanugages and Operating Systems*, pages 31–41, October 1987.

[SN77] M. Stonebraker and E. Neuhold. A Distributed Database Version of INGRES. In *2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1977.

[Sun86] Sun. *Sun-3 Architecture: A Sun Technical Report (revised version)*. Sun Microsystems, Inc., August 1986.

[TS87] C.P. Thacker and L.C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[WPE*83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proceedings of the ninth Symposium on Operating Systems Principles*, pages 49–70, October 1983.