Supporting Structured Shared Virtual Memory under Mach

Ray Bryant
Paul Carini
Hung-Yang Chang
Bryan Rosenburg

IBM Research Division
Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
{raybry,carini,hychang,rosnbrg} @ watson.ibm.com

Abstract

Distributed Shared Virtual Memory (or DSVM) has been proposed by several researchers as a method for simulating shared memory on a loosely coupled multicomputer. The principle advantage of the DSVM approach is that it provides a shared memory programming model that is familiar to many users, and this simplifies the migration of applications from conventional machines to the loosely coupled environment. However, for use in high-performance parallel processing, the standard DSVM model has several fundamental disadvantages that keep it from performing as well as traditional message-passing models of parallel computation. In this paper, we present a new version of shared virtual memory, called Structured Shared Virtual Memory or SSVM, that overcomes these difficulties. We discuss the implementation of SSVM under the Mach operating system and we present preliminary performance comparisons between this implementation of SSVM and the standard implementation of DSVM under Mach.

Introduction

Shared-memory multiprocessors present a natural programming model to the user, but it is commonly accepted that it is more expensive to build such machines than it is to connect networks of individual machines. The low cost of such networks of machines has led to the rise in popularity of the message-passing paradigm for parallel processing. However, the message-passing model has the disadvantage that conversion of programs from uni-processors or shared-memory multiprocessors can be difficult and time-consuming.

In the distributed programming domain, several researchers have proposed using a *distributed shared virtual memory* or DSVM to provide a shared memory programming paradigm on a network of machines that do not share physical memory. (See, for example, [18][13][6][14][17][20] . See also the survey articles [21][19] .) One rationale for the DSVM model of computation is that it can simplify program development for the distributed environment; it can also simplify the process of migrating programs from uniprocessor to distributed environments. As the speed of communication networks increase, the distinction between distributed and parallel processing begins to blur. It is therefore natural to consider the use of DSVM for high performance parallel processing.

However, high-performance parallel applications written to run on a shared-memory multiprocessor do not, in general, run efficiently in the DSVM environment. Nor are programs written for the latter environment as efficient as carefully balanced message-passing programs for the same hardware. These problems are caused, in part, by the mismatch between the sizes of shared objects and the size of the virtual memory page frame, and by the fact that all data transfer in a DSVM system is demand-driven, limiting the potential for overlapping data transfer with computation.

To solve these problems we present in this paper a new DSVM design, called Structured Shared Virtual Memory or SSVM, and discuss the implementation of SSVM under the Mach operating

system. SSVM overcomes the problems of DSVM for parallel processing by letting the user explicitly define the partitioning of shared memory into subregions that are meaningful to the application, and by letting the user specify the points at which particular objects should be pushed to other machines. Using this information, SSVM can transfer data among processors more efficiently than would be possible under a DSVM system.

In the following sections, we first discuss what we perceive to be the fundamental limitations of traditional DSVM environments for high-performance parallel processing. We then define the SSVM interface and discuss two possible implementation strategies, both based on the Mach [1] operating system. Next, we present the status of our SSVM prototype and initial performance measurements on a system consisting of a network of IBM Risc System/6000 workstations. We compare the performance of the SSVM system with that of the Netmemory server distributed with Mach 2.5 [14]. We then evaluate the appropriateness of the Mach system interface for the implementation of SSVM, and we discuss extensions to the interface that would simplify our implementation or make it more efficient.

Distributed Shared Virtual Memory

For our purposes, a Distributed Shared Virtual Memory (or DSVM) system is a system such as Ivy [18] that provides a simulated shared memory region to a collection of application processes running on machines that do not share real storage. Any process can address any memory location in the DSVM region, and except for performance, the fact that the storage is not really shared is invisible to the application processes that use it. A DSVM system depends on the virtual memory mapping hardware of the underlying machines to detect memory accesses that might result in inconsistent views of the shared region. The process or processes making such accesses can be delayed while the system moves data among machines to provide a consistent picture. A DSVM system can be implemented on top of a native operating system if the underlying system provides necessary facilities for manipulating the virtual memory mapping hardware. Otherwise the DSVM system must be implemented as part of the operating system itself.

A DSVM system uses a coherency protocol to maintain data consistency among machines. Most DSVM implementations use a write-invalidate coherency protocol [19]. Under this protocol, a page in the DSVM region is in one of two states: *read-only* or *writeable*. There can be copies of a read-only page on many processors of the DSVM system. A writeable page is only present on one processor. The DSVM software depends on the virtual memory hardware and on the operating system virtual memory manager to signal the DSVM system when an application process attempts to write a read-only page or attempts to read a writeable page resident on some other processor. When an attempt to write a read-only page is detected, the DSVM software causes all other copies of the page to be discarded ("invalidated") and the page to be converted to a *writeable* page resident on the processor that detected the write operation. When an application process accesses a writeable page resident on another processor, the DSVM system converts the page to a read-only page and copies it to the requesting processor.

DSVM systems using the write-invalidate protocol can present to their users a functional equivalent of shared memory [18][21] . Indeed the write-invalidate protocol is inspired by similar protocols used to manage the processor caches on shared real-memory multiprocessors. However, a DSVM system cannot provide shared memory that has performance equivalent to that provided on a shared real-memory multiprocessor, because

- The granularity of sharing is much larger (page versus cache line) in the DSVM case.
- Inter-processor transfer times are 3-4 orders of magnitude larger in a DSVM system than in a shared real-memory multiprocessor.
- Software overheads for coherency protocol management are much larger than the corresponding hardware overheads for cache coherency.

The fact that the unit of sharing is the entire page in a DSVM system implies that the false sharing problem is much worse in such systems. False sharing occurs when different processors access

different data objects that *happen* to be resident in the same page. The page appears to be shared to the DSVM system even though the original objects were not shared. If the processors attempt to update the objects at the same time, the page can be bounced back and forth between the the processors multiple times before the updates can complete. While this "ping-pong" effect is also observed in the cache consistency case, it does not occur as frequently because cache lines are smaller than pages and are therefore less likely to hold more than one object. Furthermore, the time required for cache-line transfer is infinitesimal compared to a DSVM page transfer. The result is that the performance impact of false sharing in a DSVM system can be severe.

Of course, page thrashing (or cache-line thrashing) can also occur because of true data sharing. Certain hardware versions of DSVM systems (e. g. Plus [7]) solve the page thrashing problem by using a write-update coherency protocol in which updates to the read-only copies of a page are applied every time the page is changed. The write-update protocol is infeasible in software DSVM systems since there is no efficient way to cause the updated data to be copied across the network on an update by update basis.

Researchers have tried a variety of techniques that attempt to alleviate the page thrashing problem. For example, in Mirage [13] a page remains writeable on a requesting processor for a minimum time interval, and requests for the page by other processors are delayed until this time interval expires. Similarly, in Platinum [10], a NUMA-class shared-memory multiprocessor operating system that is in some ways similar to a DSVM system, a page is "frozen" if the system determines that the page is thrashing between processors. Munin [6] uses type-specific memory coherence; the programmer associates a type (write-once, read-mostly, etc.) with shared data and different coherency protocols are used to manage data of different types. To avoid page thrashing due to false sharing, a programmer can specify the type as *write-many* in which case the system uses a delayed-write policy. Under this policy, Munin does not guarantee a precise simulation of shared memory semantics.

Our approach to solving the page thrashing problem is to adopt a memory coherency model similar to, but more aggressive than, the weak consistency model of memory coherency for shared memory multiprocessors [12][16][19] . The weak consistency model does not require that processors see a consistent view of storage at the completion of each store operation. Instead, the model only requires storage to be consistent after synchronization points. The argument for weak consistency is based on the assumption that correct parallel programs do not access shared memory in an undisciplined manner, but instead use synchronization primitives to ensure that a consistent application-level view of the shared storage is maintained at all times. Thus it is not necessary to keep storage completely consistent between synchronization points because the programmer has already ensured that a processor will not examine storage that is simultaneously being changed by another processor. Programs that do not maintain this discipline are, by definition, faulty. (For the purposes of this paper we ignore such applications as chaotic relaxation that execute properly in spite of concurrent read and write accesses to shared storage.)

We make an analogous argument for DSVM systems: It is not necessary to keep DSVM storage completely consistent at all times. It is enough to present a consistent view of storage at certain points of a DSVM program's execution. A page that is falsely shared because it contains multiple objects can be inconsistent while different processors update the objects, so long as the updates to any particular object are propagated to other processors before the object is "released", where the notion of object release is specific to the programming model of the particular application. The problem, of course, is that without help from the application itself, the DSVM system cannot determine the object boundaries within a page nor can it determine the points at which the updates to an object must be propagated. Structured Shared Virtual Memory includes facilities through which an application can provide this information to the system.

For parallel processing, DSVM systems have the additional disadvantage that they are "demand driven". Data is moved across the network, not when it is first available, but when it is first needed. The fact that a page is required is detected at page-fault time, and while the page fault handler pulls the page across the network, the faulting process is idle. Traditional DSVM systems do not provide

a convenient mechanism that lets an application initiate the transfer of data across the network in anticipation of its actual use. A sophisticated application could conceivably use a separate "scout" thread, marching ahead of the real army, to fault in pages before they are really needed, but programming such an application would not be straightforward.

Now let us compare a DSVM application with a traditional message-passing application running on the same network of machines. The message passing interface allows the user to push data across the network before it is needed at the destination processors. In a well-balanced application, processes might never have to wait for data to be transferred to their processors. DSVM processes, on the other hand, always wait while data is transferred. This fundamental difference makes DSVM systems non-competitive with explicit message passing systems from the standpoint of raw performance, even before the higher overheads of DSVM as opposed to explicit message operations are considered.

On the other hand, synchronization points in an SSVM program indicate when objects have been updated and thus indicate when data should be sent to other processors in the system. The SSVM object synchronization primitive is analogous to the message-passing send primitive, making the SSVM system supply-driven rather than demand-driven. The SSVM interface has the potential to allow the construction of well-balanced, high-performance parallel applications, while the DSVM interface does not.

Structured Shared Virtual Memory

The basic components of the Structured Shared Virtual Memory (or SSVM) interface are facilities that let the programmer:

- specify the objects the program wishes to manipulate.
- specify the subset of the application's processes that are interested in particular objects.
- specify the synchronization points in the program where updates to particular objects are complete.

For our purposes, an *object* is simply an area of storage that is part of the global shared-memory region provided by the SSVM system. (SSVM does not provide an "object-oriented" programming model such as those provided by Choices [17], Clouds [20], or Orca [4].)

SSVM supports two object types, *sequential* objects and *stride* objects. A sequential object is a contiguous sequence of bytes and typically contains an entire user data structure or a portion of a user array. A stride object can be characterized as a collection of identically-sized sequential objects, separated by fixed offsets. Stride objects typically contain slices of user arrays. Associated with each object is a list of application processes that use the object. (We assume that only one application process runs on each node of the multicomputer.) In our current implementation this list must be specified when an object is first declared. Objects are assigned object identifiers which are small integers.

Object synchronization points are specified by explicit procedure calls to the SSVM system: *ssvm_synch(obj_id)*. The *ssvm_synch* call indicates to the SSVM system that the calling process has completed an update of object *obj_id*. Logically, when the *ssvm_synch* procedure returns, the contents of the specified object have been updated on all the processors that were declared to be users of the object. Of course the implementation may delay the actual updates (see the next section), but no process may see the object's old contents once the *ssvm_synch* procedure returns.

At any time, the set of all objects in the SSVM region is called the current *template*. The current template describes how the SSVM storage region is being used by the processors. The objects in the current template cannot overlap, and they must jointly cover the entire SSVM region. The current template may be changed to reflect different phases of a computation, but template change is a relatively expensive operation. As currently implemented, a template change implies a global barrier operation; all processes in the SSVM computation must execute the template change oper-

ation before any of the processes are allowed to proceed. This synchronization is required so that the SSVM servers and the user applications all have the same notion of current template.

SSVM application processes are started independently on separate machines. (In the current implementation, application processes are invoked on multiple RS/6000 workstations using the Unix[1] *rsh* command). Each process builds a local copy of the object and template descriptors and then calls the SSVM initialization routine. As part of *ssvm_initialize*, each process connects to a global SSVM server. The global server checks the consistency of the parameters (including the object and template definitions) supplied by all the processes and then causes a logical copy of the SSVM region to appear in each SSVM process' address space. In the current implementation the SSVM region is mapped at the same virtual address in each process, but this restriction is not fundamental.

Virtual Memory Optimizations for SSVM Synchronization

The implementation of an SSVM system does not require special virtual memory capabilities from an underlying operating system, but such capabilities can be used to build more efficient implementations.

An SSVM system can be implemented on top of a pure message-passing system, with the bulk of the implementation embodied in a library that is linked with application programs. The library must be able to receive and process messages asynchronously with respect to the rest of the application. (Mach satisfies this criterion because the library can spawn background threads to receive messages asynchronously.) Such an implementation would not even require a global server, provided the individual application processes could locate each other and establish connections among themselves. The *ssvm_initialize* library routine would simply allocate a zero-filled region in the local address space to serve as the SSVM region and would then establish handlers to process messages from the other processes. The *ssvm_synch* routine would simply send the entire contents of the specified object in a message to every other process using the object. A process receiving such a message would copy the new object contents into the appropriate location(s) in the SSVM region before acknowledging the message. The *ssvm_synch* routine would not return until all the acknowledgements were received.

Such an implementation may be acceptable (or even optimal) for small objects, but it can be inefficient for objects that span multiple pages. For example, if this implementation of *ssvm_synch* were applied to a 32-megabyte object, the operation would not return until all 32 megabytes had been copied to all readers of the object. Delaying the sender is bad enough, but the receivers are likely to be delayed as well, waiting at an application-level synchronization point for the sender to "release" the object.

Using virtual memory primitives available in Mach and other systems [3], it is possible to implement *ssvm_synch* in a manner that preserves the semantics of the synchronous implementation but that does not necessarily delay the application processes while the data is actually transferred. For example, *ssvm_synch* might be implemented as follows:

1. *ssvm_synch* issues a remote procedure call (RPC) to the global server specifying the particular object to be updated.
2. The global server communicates with the other users of the object, causing them to temporarily remove all pages containing any part of the object from their address spaces.
3. The global server executes an RPC return to *ssvm_synch*.
4. *ssvm_synch* changes the protection of all pages of the object to read-only on the local machine.
5. *ssvm_synch* initiates the asynchronous transfer of the updated object to the global server.
6. *ssvm_synch* returns to the user.

---

[1] Unix is a registered trademark of AT&T in the United States and other countries.

7. As pages of the object are transferred to the global server, the SSVM runtime routines convert them from read-only back to read-write in the local address space.
8. As pages of the object are transferred from the global server to the destination processors, the pages are restored to the destination address spaces.

This approach preserves the semantics of the synchronous implementation of *ssvm_synch* since the destination processors cannot access the old version of the object once *ssvm_synch* has returned to its caller. Step (4) of the above procedure is required to ensure that the source process does not modify the object again before the original update has been completed. An attempt to modify (source processor) or access (destination processors) a page that has not yet been transferred results in a page fault that is caught by the library linked with the faulting process. At that point the application process is blocked until the required page is available. A sophisticated SSVM implementation might try to change the order in which pages of the object are transferred in response to such page faults. It might even, as a matter of policy, postpone transferring some pages indefinitely, or until such time as an application process requires them.

Clearly, the overhead of this asynchronous implementation of *ssvm_synch* is non-trivial and will only be justified for objects considerably larger than a page. Also, the asynchronous approach will be worthwhile only if the destination processors are usually able to continue computation on other portions of the address space during the execution of the data transfer phase of *ssvm_synch*. Therefore a different approach might be required for stride-type objects that touch large portions of the SSVM region.

In summary, we point out the advantages of the SSVM system over traditional DSVM systems:

- SSVM solves the false sharing problem since objects are not updated on other processors until an *ssvm_synch* call is executed.
- SSVM allows sharing of non-page aligned objects.
- SSVM allows efficient sharing of small objects. (The SSVM system can choose at *ssvm_synch* time to use page-oriented or message-oriented transport mechanisms to send the updates. A DSVM system must always send an entire page.)
- SSVM allows the expression of supply-driven data transfer; equivalently, the interface allows the SSVM system to perform aggressive page prefetching in response to user demands.
- SSVM allows the application of virtual memory optimizations to the problem of data transfer for large objects.

SSVM can potentially let users develop parallel programs for distributed-memory hardware using a shared-memory progamming model without sacrificing the efficiency of a message-passing programming model.

Migrating Programs into the SSVM Environment

One of the advantages of the SSVM paradigm for parallel programming over the traditional message-passing paradigm is that the shared-memory programming model of SSVM lets a programmer convert existing programs to the SSVM environment with relative ease. To justify this claim, we discuss in this section how a user can methodically convert an existing program for a shared-memory multiprocessor to run under the SSVM system. Techniques for converting existing sequential programs to run on traditional shared-memory multiprocessors are well-known [11]. For the purposes of this discussion, we assume the user begins with a correct program for a shared-memory multiprocessor (or for that matter, for a DSVM system). We further assume that the program's shared variables can be collected into a single contiguous region that can be mapped onto the shared-memory region provided by the SSVM system.

The first step in converting the multiprocessor (MP) program is conversion of the synchronization primitives. MP programs are typically coded to a particular application-level synchronization model such as semaphores, locks, or barriers. The application-level primitives are in turn implemented using hardware-specific operations such as *test-and-set* or *compare-and-swap* that atom-

ically read and modify words located in shared storage. The weak consistency model provided by SSVM is not appropriate for these synchronization variables, so the application-level synchronization primitives must be replaced with RPC versions that call the SSVM global server. Because the global server has a copy of the latest updates issued against the SSVM region, the server can execute the necessary atomic instructions against its copy of storage. On return, the application-level RPC stub procedures can copy the updated values of the synchronization variables into the local copy of the SSVM region. Assuming the original MP program never accessed synchronization variables directly, the program should continue to execute correctly after this conversion.

The second part of the conversion process involves creation of the SSVM templates and objects and the insertion of *ssvm_synch* calls. Identifying the objects in the program involves examining each call to an application level synchronization primitive and identifying the area or areas of storage the primitive protects. Each such area must be defined as an SSVM object.

Once the program objects have been defined, the insertion of *ssvm_synch* calls is straightforward: *ssvm_synch* calls are placed before each synchronization primitive that "releases" one or more objects.

In the following example, barrier #2 in process 1 releases object A, while the same barrier in process 2 releases object B.

```
Process 1                Process 2
-----------              -----------
. . .                    . . .
barrier #1;              barrier #1;
. . .                    . . .
update object A          update object B
. . .                    . . .
barrier #2;              barrier #2;
. . .                    . . .
access object B          access object A
. . .                    . . .
barrier #3;              barrier #3
. . .                    . . .
```

We therefore insert *ssvm_synch* calls before barrier #2 in both processes as follows:

```
Process 1                Process 2
-----------              -----------
. . .                    . . .
barrier #1;              barrier #1;
. . .                    . . .
update object A          update object B
. . .                    . . .
ssvm_synch(A);            ssvm_synch(B);
barrier #2;              barrier #2;
. . .                    . . .
access object B          access object A
. . .                    . . .
barrier #3;              barrier #3
. . .                    . . .
```

Now when process 2 accesses object A after passing barrier #2, it will see the correct value because process 2 cannot pass barrier #2 until process 1 reaches the same barrier, and process 1 cannot reach the barrier until its *ssvm_synch* of object A returns. When the *ssvm_synch* call returns, the updated value of object A will have been propagated (logically, at least) to process 2.

If process 2 were to access object A between barriers #1 and #2, it might or might not see the changes made by process 1, and the SSVM program would not behave correctly. But we know that process 2 cannot access object A between barriers #1 and #2 because if it did, the original MP program would not have been correct.

Note that our SSVM interface does not include any notions of locking that are directly associated with the SSVM objects (cf. the access modes associated with segments in the distributed shared memory controller operations provided in Clouds [20]). Instead we assume that synchronization is handled independently since the user began with a correct program for a shared memory MP. (However, as a performance optimization it may be useful to combine the RPC's for the *ssvm_synch* operation and the subsequent application-level synchronization primitives.)

The effort needed to convert an existing MP program to the SSVM environment will be greater than the effort required to convert the same program to a traditional DSVM environment, because the latter environment does not require the identification and definition of objects or templates. But the effort should still be considerably less than that required to convert an MP program to a message-passing environment, and the performance improvement available under the SSVM system should justify the additional effort.

Furthermore, the initial conversion of an MP program to the SSVM environment need not make sophisticated use of the SSVM facilities. The program can be converted quickly and then effort can be devoted to optimizing just those portions of the program that are performance critical. For the message-passing paradigm on the other hand, the entire program must be converted before any of it can be run. Our experience with RP3 [9] justifies this claim. We found that the initial process of migrating programs into the parallel programming environment was relatively simple. Time could then be spent restructuring those parts of the program that were crucial to obtaining good performance on the parallel machine.

We hope that parallelizing compilers such as PTRAN [2] will eventually reduce or eliminate the programming effort required to identify and define templates and objects. Alternatively, we believe that the explicit data partitioning statements of languages such as Fortran-D [15] can be used to generate the template and object information.

Approaches to Implementing SSVM under Mach

Thus far we have described the SSVM interface without discussing its implementation in any detail. Before we discuss the Mach implementation, let us consider the requirements an SSVM implementation places on the underlying operating system. SSVM software must be able to perform all of the following functions:

1. It must be able to apply asynchronous updates to user address spaces.
2. It must be able to suspend and resume the application process.
3. It must be able to manipulate page protection in the user address space.
4. It must be able to detect when the application takes a protection exception and be able to restart the user after correcting the protection exception.

Of these requirements (1) and (2) are required for any implementation of the *ssvm_synch* primitive; (3) and (4) are required for the asynchronous implementation of *ssvm_synch*.

Under Mach there are two basic approaches to supporting the SSVM system: An implementation can use the memory object (external pager) interface [22], or it can use the Mach exception mechanism [8]. Since we are in the business of supporting shared virtual memory, and since the standard shared virtual memory implementation under Mach uses the memory object interface, we expected the memory object interface would be the method of choice. We found that the choice is not so clear cut, as we will explain later in this section. In later subsections we outline implementations using both methods, but we begin by describing the common features of the two implementations.

Common features of the SSVM implementations. Both SSVM implementations use a single global synchronization and data server, and both rely on an SSVM library linked with each application process. In both cases, the global data server maintains a copy of the entire SSVM region by applying all *ssvm_synch* operations to its own copy of the SSVM region. Therefore, the global data server can always provide the latest "correct" values for a page.

The SSVM library contains application-level synchronization primitives, and the *ssvm_initialize* and *ssvm_synch* procedures. The *ssvm_initialize* routine is responsible for establishing a connection with the global server and for passing the application's template and object definitions to the server. The *ssvm_synch* routine is reponsible for collecting object data and shipping it to the global server. Sequential objects are sent directly. Stride-type objects are first copied to sequential storage and then sent. The data must be copied from its sequential form back into the SSVM region by the global server and by each recipient application task.

When the global server receives an RPC requesting an *ssvm_synch* operation, it forwards the request to each application process that is a user of the object. While the call from the application to the global server is a true RPC, the calls from the global server to the destination processes are sent as unidirectional messages so that multiple simultaneous transfers from the global server can be initiated. These messages are explicitly acknowledged by the destination processes, and the global server completes the original RPC request once it has received all the acknowledgements.

Supporting SSVM using the memory object interface. The Mach memory object interface is intended to allow the construction of memory servers that reside outside of the Mach kernel [22]. The interface is defined in [22] and [5]. Here we review the aspects of the interface that are important for this discussion.

A memory object server (or external pager) is a user mode program that provides the contents of pages for a particular memory object. A memory object is represented by a *memory object port* which is a port whose receive rights belong to the memory object server. The memory object is mapped into a task by including the memory object port in a Mach *vm_map* system call. The *vm_map* call also establishes the virtual address of the object in the task. Once the object has been mapped, a reference to a page in the *vm_map*'ed region that cannot be resolved by the kernel results in an RPC to the memory object server's *memory_object_data_request* entry point. The server does whatever is necessary to bring the data for the page into storage and then returns the data to the kernel using a *memory_object_data_provided* call.

Other routines of interest to us here are:

memory_object_lock_request
: This call from the server to the kernel allows the memory object server to reclaim pages from the kernel's page cache, to change the level of protection on objects in the cache, or to cause pages to be flushed from the cache. If a page has been modified and the server requests that it be removed from the cache, the page is delivered back to the server via a *memory_object_data_write* call.

memory_object_data_unlock
: This call from the kernel to the server occurs when an access has been made to a page that is not permitted by the current page protection. This request from the kernel asks the server to either change protection (with a *lock_request* call) to allow the requested access or to deny this mode of access.

memory_object_data_write
: This call from the kernel to the server indicates that a particular page has been modified but now needs to be removed from the kernel's page cache. Such an eviction can happen as a result of a *lock_request* call made by the server, or because the kernel's supply of free page frames has run low.

Note that in general the memory object server has no direct access to the user address space. In particular, the server has no direct way to deliver pages to the user address space before the user has touched them. The Mach 2.5 memory object interface lets the server supply more than one

page in response to a *data_request* from the kernel, but pages must be contiguous and must start at the offset requested by the kernel.[2]

Also note that the memory object interface is (understandably) page oriented. There is no simple way to update part of a page in the application address space. (If the page is already in the kernel's cache, it can be reclaimed by the server using a *lock_request* call. The page can then be updated with the new data and cached in the server address space until the user again accesses the page. There is no way to return the page to the application address space before the user next touches it.)

Another problem with the memory object interface is that it is slow, at least from the standpoint of parallel processing. Measurements on our Mach 2.5 system on the RS/6000 indicate that it requires approximately a millisecond to resolve an application page fault even when the memory object server is on the same machine as the application and when the required data is already in the server's address space. (Remember that we are trying to compete with message-passing applications that can transfer data from local message queues into a user address space in a few hundred instructions!) While an overhead of 1 ms. during the processing of a page fault that requires 30 ms. to fetch the data from a disk may not be excessive, it is very large when the page can be fetched across a fast network in 3 ms. or less.

To minimize the effect of these problems, our memory object implementation of SSVM is more complicated than we would like. The global SSVM server does not itself serve as the memory object server for the application processes. Instead we run a separate memory object server on each node of the SSVM system. Each individual server defines a memory object that serves as the SSVM region for the application task running on the same node as the server. The local memory object servers communicate with the global SSVM server using standard Mach IPC; the messages exchanged between these servers are of arbitrary size and need not be page aligned. *ssvm_synch* operations initiated on remote nodes are forwarded, not to the application processes directly, but to the local memory object servers.

The local memory object servers let the global SSVM server push new pages and page updates out to the machines running application processes, but the memory object interface by itself does not let the local servers move the data into the client address spaces before the clients need it. To circumvent this problem, each local memory object server maps the SSVM region, not only into its client's address space, but into its own address space as well. The result is that the memory object server and the application process share the storage that backs the memory object. Now when the local server gets an *ssvm_synch* request and the associated data, it can *bcopy* the data directly into the SSVM region. Of course, in doing so, it might cause page faults that result in *memory_object_data_request* calls to itself, and it must therefore take care to avoid a variety of possible race conditions and deadlocks. In particular, the server may have to suspend the application process to keep it from using out-of-date data that becomes temporarily visible. Unfortunately, the cost of handling these phantom *data_request* calls (about a millisecond per page) negates the benefits of pushing data into the application address space before it is needed.

Supporting SSVM using the Mach exception mechanism. The Mach exception mechanism [8] is a facility that allows general user-level handling of exceptional conditions. Associated with each thread in the Mach system is a special port called the thread's *exception port*. Initialized by the system to PORT_NULL, a program can designate a particular port to serve as a thread's exception port. When a thread whose exception port has been defined encounters an exceptional condition (illegal instruction, protection violation, divide by zero, etc.), the kernel suspends the thread and issues a *catch_exception_raise* RPC on the thread's exception port. (It is the program's responsibility to establish a listening thread for the exception port that will receive this RPC.) Inside *catch_exception_raise*, the program can examine the state of the suspended thread and determine whether the exceptional condition can be cleared and whether to resume or to terminate the thread.

[2] Even this facility does not work in the Mach 2.5 kernels available to us. Our kernels hang or crash if a server tries to provide more than one page in response to a *data_request*. Joseph Barrera at CMU has indicated to us that fixes for this problem are available from CMU.

If the exception can be cleared, the handler can restart the thread that raised the exception by returning KERN_SUCCESS from *catch_exception_raise*. If the exception cannot be cleared, the handler can cause the thread to be terminated by returning KERN_FAILURE.

The Mach exception mechanism allows an implementation of SSVM that includes just one server, the SSVM global server; local servers on the application nodes are not required. In this implementation, more of the SSVM functionality is handled directly by the SSVM library linked with the application processes. The *ssvm_initialize* library routine creates the SSVM region using the standard Mach *vm_allocate* primitive. It then uses the Mach *vm_protect* call to reduce the protection on the entire SSVM region to VM_PROT_NONE (no access allowed). Finally, it establishes an exception port for the application thread and creates a new thread to handle exceptions raised by the application. Another thread is created to listen for incoming *ssvm_synch* requests from the global server.

Now when the application touches an inaccessible page in the SSVM region, it generates a protection exception which is caught by the handler thread. By examining the faulting virtual address, the handler can determine whether the fault was taken on a page in the SSVM region. If so, it calls the global SSVM server to obtain a current copy of the required page. The handler changes the page protection (again using *vm_protect*) to VM_PROT_ALL (all access allowed), copies the current data into the page, and returns KERN_SUCCESS to allow the application thread to continue.

Implementation of *ssvm_synch* is much simpler in this environment than in the memory object version. As before, *ssvm_synch* is an RPC from the application to the global server. The global server forwards the requests directly to the appropriate application tasks. Recall that *ssvm_initialize* created a thread in each application address space to listen for such requests. The global server returns to the originating *ssvm_synch* routine once it has received acknowledgements from all the helper threads. The asynchronous version of *ssvm_synch* uses *vm_protect* to make pages inaccessible in the target address spaces and read-only in the originator's address space.

Since all data is delivered directly to the application tasks (instead of to separate local servers) it can simply be copied from the RPC buffers to the SSVM regions where it belongs. Before starting the copy, the program examines the state of each target page and reprotects those that are not writeable. As in the memory object implementation, it is important to suspend the application thread before making the pages writeable. Otherwise the application may see out-of-date values before they are replaced.

Comparison of the two implementation approaches. The SSVM implementation based on the Mach exception mechanism was simpler and easier to develop than the memory object version. It has fewer components and fewer complicated interactions among components. Furthermore, the system overheads are lower in the exception port implementation for some common SSVM operations. For example, consider the time required to insert a previously absent page into an application task's SSVM region once the contents of the page have been delivered to the application's machine. In the exception port implementation, this time is roughly 700 microseconds on an RS/6000 Model 530 running Mach 2.5. The equivalent time in the memory object implementation is roughly 1200 microseconds, largely because it is impossible to avoid a *data_request/data_provided* trip through the memory object interface, even with the SSVM region mapped directly into the local server's address space. The difference is more dramatic for multi-page transfers, because much of the exception port overhead is constant, independent of the number of pages involved.

The previous times were for inserting a page into the application's address space before the application thread needed it. If the application actually faults on an absent page, the overhead (not counting the page transfer time) of the two implementations is approximately the same (about a millisecond in each case). This overhead may be acceptable given the 11 milliseconds currently required to fetch a page from the global server. It may no longer be acceptable when the page fetch time is reduced to the anticipated 3 millisecond range.

By timing the exception path through the kernel, we have determined that for the exception port SSVM implementation, nearly half of the 1 millisecond overhead is due to suspending the application thread, invoking the handler thread, and returning back through the kernel to resume the application thread. To reduce this part of the overhead, we propose the notion of a *page fault reflector* analogous to the system-call reflector that is part of Mach 3.0. A task would register an address range and the entry point of a page-fault handler with the kernel. When the kernel detects a page fault or protection violation in the identified address range (the range would be set to the boundaries of the SSVM region) the kernel would simply resume the faulting thread but with its program counter set to the specified entry point. Information describing the fault would be pushed on the faulting thread's stack. The page fault handler would clear the fault by adjusting the protection of the faulting page and fetching current data from the global SSVM server. It would then return directly to application code without going back through the kernel. Using this approach we expect to reduce the 1 millisecond overhead mentioned above by a factor of 2.

Status and Performance Measurements

As of this writing, we have completed initial SSVM implementations using both the memory object and the exception port methods. The implementations differ slightly from those described previously in that:

1. They use a separate synchronization server rather than a server integrated with the global data server.
2. They implement only the synchronous version of *ssvm_synch*.
3. They do not yet support demand-driven page fetching. The SSVM regions are always fully instantiated.
4. The memory object implementation does not handle kernel page-out requests for the SSVM region.
5. The exception port implementation does not include the "page fault reflector".

Item (1) implies that migration of MP programs into our current environment is less transparent that it might be. Item (2) implies that for large objects we are paying an overhead at *ssvm_synch* that is higher than necessary. Item (3) will not be a problem until we have an asynchronous implementation of *ssvm_synch*. Item (4) only applies if the problem size is so large that it cannot reside in main storage, a situation we have not yet encountered. Item (5) is a performance enhancement that will become more important when our page transfer times are smaller than they are now.

Our test environment is illustrated in Figure 1. The SSVM Testbed consists of four IBM Risc System/6000 Model 530 workstations connected by IBM Risc System/6000 Optical Links to a Risc System/6000 Model 930 server in a "star" configuration. The Risc Systen/6000 Optical Link is a 200 Mb/s. optical communications link; we run TCP/IP across this link with a nominal page transfer time between machines of 11 ms. For these experiments, the Model 930 server was used solely as a host for the synchronization server and the SSVM global data server (or for the Mach Netmemory server); the Model 930 did not take part in the computations we describe below. Each of the RS/6000 workstations in the SSVM Testbed was running the X126 version of the Mach 2.5 kernel.

Computations are distributed on the Testbed using the Unix *rsh* command. One copy of the test program runs on each of the 4 Model 530's in the configuration. As part of initialization, the test programs connect to the various servers required for a particular experiment. Command line parameters are used to assign logical processor numbers to the test programs. The machines ran in multiuser mode during these test runs, but they were dedicated to these experiments; no users were running on the machines during the timing tests.

Our initial test programs are *mp_matmul*, an MP matrix multiply program, and *odd_even*, a program designed to cause page thrashing in a traditional DSVM environment.

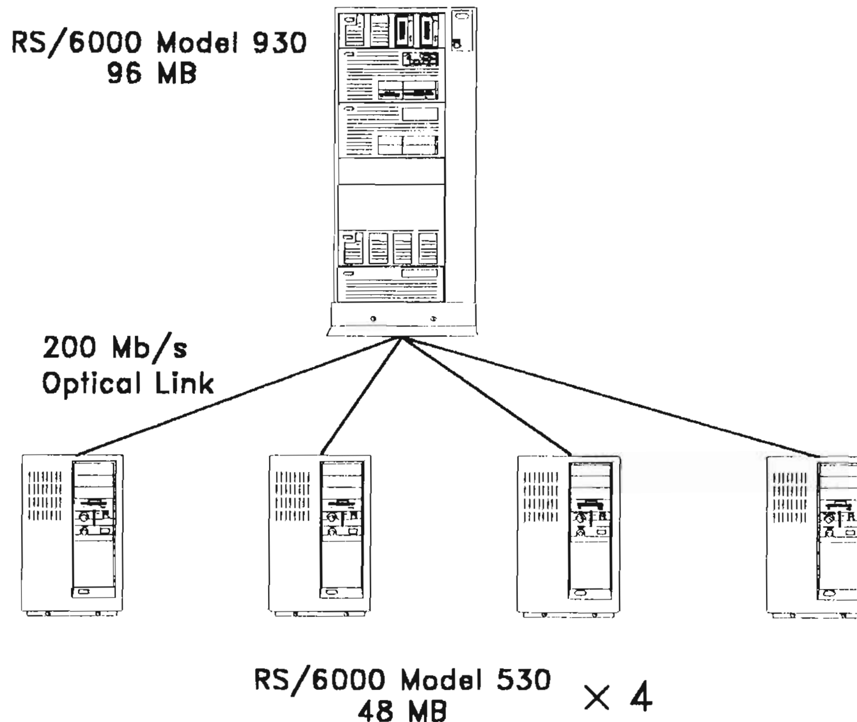

Figure 1. The SSVM Testbed

mp_matmul: This program implements a standard multiprocessor matrix multiply of M by M matrices: $C \leftarrow A \times B$. All three matrices are stored in row-major order. The computation is distributed in a straightforward manner: M/N rows of A and all of B are copied to each of the N processors (N is 4 in our case). Each processor then computes M/N rows of the product matrix C.

The computation consists of 4 major phases separated by barriers:

1. Initialization
2. Multiplication
3. Verification
4. Termination

During the initialization phase, process 1 initializes the A and B matrices. During the multiplication phase, each process uses the B matrix and its M/N rows of the A matrix to compute the corresponding M/N rows of the C matrix. During the verification phase, process 1 examines the product matrix to validate the result. When checking is complete the programs terminate.

During the computation, one slice of the A matrix moves through the Netmemory or SSVM system from process 1 to each of the other processes. The entire B matrix moves from process 1 to the other processors. At the end of the computation, one slice of the C matrix from each of the processes moves back to process 1 to be checked for correctness.

For the SSVM experiments, the matrices are partitioned into objects as follows:

The A matrix is divided into N objects, each consisting of M/N rows of the matrix.

The B matrix is also divided into N objects, each consisting of M/N columns of the matrix. These are stride-type objects because the matrix is stored in row-major order.

The *C* matrix is divided into N^2 objects, each consisting of an M/N by M/N square submatrix of *C*. These are also stride-type objects. Each *C*-object can be computed from one *A*-object and one *B*-object.

In all cases, the user-list associated with each object is defined to include just those processes that require access to the object.

This partitioning into objects is actually more fine-grained than necessary for this problem. (The *B* matrix could be a single object, and the *C* matrix could be partitioned along the same lines as the *A* matrix.) The finer granularity lets the processes initiate transfers (via *ssvm_synch*) before the initialization and computation phases are complete. This optimization provides little benefit given our current synchronous implementation of *ssvm_synch*, but it could be important when an asynchronous version is available.

Results: We ran the *mp_matmul* test program using matrix sizes of 256, 512, and 1024 in each of three environments, the standard Mach Netmemory environment and our two implementations of the SSVM environment. Each experiment was run three times, and the average times are shown below in Table 1.

Matrix size		Netmemory	SSVM mem. obj.	SSVM exc. port
256x256:	initialize	3.2 s	9.1 s	8.7 s
	multiply	14.3 s	6.3 s	6.6 s
	check result	3.7 s	0.1 s	0.1 s
	total	21.2 s	15.5 s	15.4 s
512x512:	initialize	12.3 s	34.2 s	36.5 s
	multiply	93.0 s	63.0 s	62.3 s
	check result	14.7 s	0.4 s	0.3 s
	total	120.0 s	97.6 s	99.1 s
1024x1024:	initialize	53.3 s	138.8 s	147.8 s
	multiply	590.5 s	468.0 s	466.3 s
	check result	88.0 s	1.4 s	1.2 s
	total	731.8 s	608.2 s	615.3 s

Table 1: Times for Matrix Multiply Example

The results show that the SSVM approach is 20 to 30 percent faster than the DSVM approach as embodied in the Mach Netmemory server. The reason is largely that the SSVM system batches page transfers while the Netmemory case fetches individual pages across the network. (This is an instance where the *data prefetch* advantage of the SSVM approach is clearly profitable.)

For the matrix sizes used above, false sharing does not occur since no page of the *C* matrix contains objects that are written by more than one processor. Experiments with matrices of other sizes failed to exhibit performance degradation attributable to page thrashing. In this program, false sharing can only occur at the boundaries of the objects in the *C* matrix, and the order of computation is such that it is unlikely that two processors ever try to write the same page at the same time.

Under the Netmemory server, the initialization phase does not include any data transfer, the multiplication phase includes the time needed to distribute the *A* and *B* matrices to all the processes, and the verification phase includes the time needed to fetch the result matrix *C* back to process 1. Under SSVM, the initialization phase is longer because it includes the time needed to distribute the source matrices. The multiplication phase includes the time needed to return the result matrix to process 1. The verification phase is much faster under SSVM because it involves no data transfer at all.

For comparison, we observe that the uniprocessor times for the matrix multiply results given above are 9.5 seconds, 219 seconds, and 1746 seconds, respectively. Since the computational cost increases with M^3, one would expect these times to differ by a constant factor of 8. But the smaller matrix benefits much more from the RS/6000 data cache, so the factor of 8 is not seen there. The ratio between the uniprocessor times for the 512 and 1024 dimension matrices is 8 as expected. We are not using strip mining or other cache-oriented optimization techniques. Using these uniprocessor times, the speedups obtained on 4 processors under SSVM are calculated to be 0.66, 2.23, and 2.85, respectively, for the three different matrix sizes. While this is better than the Netmemory speedups (0.45, 1.70, and 2.39, respectively) there is still room for significant improvement.

odd_even: This test program is deliberately designed to cause page-thrashing in the Netmemory server case and is intended to be a worst-case comparison of DSVM versus SSVM performance. In the *odd_even* test case, each of the N processors in the system updates a portion of a one dimensional, double-precision array. Processor 1 updates elements 0, N, 2N, etc., processor 2 updates elements 1, N+1, 2N+1, etc., and so forth for the other processors. The time a processor spends between updates is an adjustable parameter. When the entire array has been updated, processor 1 inspects all the entries to ensure they have the expected values. As in the *mp_matmul* test cases, the initialization, computation, and checking phases of the program are separated by barriers. An SSVM stride-type object is defined to hold the data updated by each processor.

Results: We ran this program on our 4-processor Testbed with a total of 1000 data items (250 per processor), and with delays of 0 ms. and 10 ms. between updates. With no delays between updates, the program ran in about 600 ms. under the Netmemory environment and under both implementations of the SSVM environment. Page thrashing was not apparent in the Netmemory case because each processor, after obtaining a shared page, had time to complete all of its updates before the page could be claimed by another processor.

The story is different when the processors sleep 10 ms. after updating each data element. In this case the program should run in approximately 2.5 seconds (250 items, 10 ms. per item), and indeed under the two SSVM environments the program ran in 2.7 seconds. Under the Netmemory server, however, the program required 26 seconds. This example is artificial, but it shows that the effects of false sharing can be severe in a traditional DSVM environment.

Planned Performance Improvements: We intend to pursue a number of performance optimizations. At the moment, we are running TCP/IP over the optical links in the SSVM Testbed. We know that the optical link hardware is capable of transferring pages between machines at approximately 1 ms. per page. The TCP/IP version gives us 11 ms. per page. We plan to eliminate the TCP/IP protocol by using a custom protocol directly between Mach *netmsgservers* on the machines of our Testbed. We hope to achieve a page transfer time of 3 ms. or less using this approach.

We also need to implement the asynchronous version of *ssvm_synch*. Particularly in the larger matrix multiply cases, we are paying a heavy overhead for using the synchronous version of *ssvm_synch*.

Finally, for large data copies, it would be more efficient to use the Mach *vm_copy* and *vm_deallocate* routines than to use *bcopy*. Initial measurements indicate that the break even point is around 20 pages (80 KB). Copies larger than this should use the Mach primitives instead of *bcopy*.

We hope to be able to report on the results of these improvements as well as to discuss some additional applications at the Mach Symposium in November, 1991.

Suggested Changes to the Mach Interface for SSVM Support

So far, we have attempted to achieve the best possible performance for the SSVM system using the existing Mach system interface. In this section we discuss some enhancements or changes to the Mach interface that we think would improve the performance or simplify the implementation of the SSVM system.

Directed out-of-line data. The first change we would like concerns Mach IPC and out-of-line data. Currently, out-of-line data included with a Mach message shows up at an arbitrary location in the receiver's address space. In SSVM we are constantly sending parts of large structures (a few rows of a matrix, for example) and these fragments must end up at the correct virtual addresses in the destination. At present we handle this problem by copying the data after it is received. The RS/6000 can copy a page in about 100 microseconds, but this is still a hundred microseconds of overhead we would like to avoid. The point is that we know precisely where we wish the data to appear in the destination address space, and we would like to be able to specify (direct) that the out-of-line data accompanying a Mach message be deposited at a specified address. The authorization for such directed sends might be another kind of "port right" that a destination task can give out to a trusted set of other tasks.

Asynchronous insertion of pages into memory objects. For our purposes, a major shortcoming of the Mach 2.5 memory object interface is that it does not let a memory object server asynchronously instantiate arbitrary pages of a client's mapped memory object. It provides all the facilities necessary to handle client-generated page faults, but it does not provide the facilities a server needs to preclude the necessity for such faults. We are hoping this shortcoming has been corrected in Mach 3.0.

Double mapping of anonymous memory regions. In Mach, storage that is backed by an external memory object server can be mapped at more than one location in a client's address space. The same capability is not available for storage that is simply obtained from the kernel via *vm_allocate*. The exception port implementation of SSVM would be simpler if the *vm_allocate*'d SSVM region could be mapped a second time at another location in the application address space. The access privileges available to the application via the original mapping could then differ from the privileges available to the SSVM library via the second mapping. We could obtain the desired double-mapping functionality by including local external memory object servers in the exception port implementation of SSVM, but we would rather avoid this complication.

Concluding Remarks

In this paper we presented a new shared virtual memory programming model that allows the construction of more efficient parallel programs for multicomputers than did previous shared virtual memory implementations. We defined our model (called Structured Shared Virtual Memory, or SSVM), and discussed its implementation using the Mach memory object and exception port interfaces. We found that the implementation using the exception port interface was simpler, and although performance of the two implementations was similar, we believe the exception port implementation can be more easily optimized. We then presented initial performance comparisons between SSVM and the standard implementation of distributed shared virtual memory under Mach (known as the Netmemory server [14]). The SSVM approach was seen to be 20-30 percent faster than the Netmemory approach. Finally, we suggested changes to Mach kernel interface that would make support of SSVM under Mach simpler or more efficient. Further work is necessary, both in experimenting with a wider range of applications and in enhancing the SSVM implementation.

Our current SSVM implementations run under Mach 2.5. Mach 3.0 offers us several advantages, including

- improved IPC performance.
- enhanced memory object interface that appears to support asynchronous insertion of pages into client address spaces.
- shorter path lengths in the Mach exception mechanism.
- direct kernel support for inter-machine IPC.

As we complete our port of Mach 3.0 to the RS/6000, we intend to move our SSVM environment onto that platform in order to exploit these changes.

Acknowledgements

This work would not have been possible without the assistance of Bob Fitzgerald, Dan Poff, Marie Butrico and Kati Rader, all of the IBM Research Division. Bob Fitzgerald and Dan Poff are principally responsible for the existence of a Mach port to the Risc System/6000 and Bob Fitzgerald, in particular, solved a number of problems related to running Mach on the Model 930 in our test environment. Maria Butrico and Kati Rader provided us with the driver code from AIX/v3 for the Risc System/6000 Two Port Optical Link adapter, and patiently assisted us in the port of this code from AIX/v3 to Mach.

References

[1] M. Accetta et al., "Mach: A New Kernel Foundation for Unix Development," *Usenix Association Proceedings*, Summer 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *The Journal of Parallel and Distributed Computing*, vol. 5, no. 5, pp. 617-640, Oct 1988.

[3] A. Appel and K. Li, "Virtual Memory Primitives for User Programs," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 96-107, Baltimore, Md. Order Number 556910: ACM, April 1991.

[4] H. Bal, M. Kaashoek, and A. Tannenbaum, "A Distributed Implementation of the Shared Data-Object Model," *Proceedings of the Distributed and Multiprocessor Systems Workshop*, pp. 1-19, Usenix Assocation, 1990.

[5] R. Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rashid, A. Tevanian, Jr., and M. Young, "Mach Kernel Interface Manual", Pittsburgh: School of Computer Science, Carnegie Mellon University, Unpublished manuscript, August 1990.

[6] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proceedings 1990 Conference Principles and Practice of Parallel Programming*, pp. 168-176, New York, N. Y.: ACM Press, 1990.

[7] R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared Memory System," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 115-124, Los Alamitos, Cal. Order No. 2047: IEEE CS Press, 1990.

[8] D. Black, D. Golub, R. Rashid, A. Tevanian, and M. Young, "The Mach Exception Handling Facility," *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices*, vol. 24, no. 1, pp. 45-56, Department of Computer Science, Rice University, P. O. Box 1982, Houston, Texas, May 1988.

[9] R. M. Bryant, H.-Y. Chang, and B. S. Rosenburg, "Operating System Support for Parallel Programming on RP3," *IBM Journal of Research and Development*, To appear: November 1991.

[10] A. Cox and R. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum," *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 32-44, December 1989.

[11] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/Fortran," *Parallel Computing*, no. 7, pp. 11-24, 1988.

[12] M. Dubois and C. Scheurich, "Memory access dependencies in shared-memory multiprocessors.," *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 660-673, June 1990.

[13] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 211-223, December 1989.

[14] A. Forin, J. Barrera, M. Young, and R. Rashid, "The Shared Memory Server," *Proceedings of the 1989 Winter Usenix Conference*, pp. 229-243, 1989.

[15] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, "Fortran D Language Specification", Houston, Texas: Department of Computer Science, Rice University, P. O. Box 1982, Technical Report COMP TR90-141, December 1990.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennesy, "Memory consistency and event ordering in scalable shared memory multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15-26, May 1990.

[17] G. Johnston and R. Campbell, "An Object-Oriented Implementation of Distributed Virtual Memory," *Proceedings of the Distributed and Multiprocessor Systems Workshop*, pp. 39-57, Usenix Assocation, 1990.

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

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

[20] U. Ramachandran and M. Khalidi, "An Implementation of Distributed Shared Memory," *Proceedings of the Distributed and Multiprocessor Systems Workshop*, pp. 21-38, Usenix Assocation, 1990.

[21] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, vol. 23, no. 5, pp. 54-64, May 1990.

[22] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, B. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 63-76, Austin, Texas: ACM, November 1987.