MultiView and Millipage—Fine-Grain Sharing in Page-Based DSMs

Ayal Itzkovitz, Assaf Schuster
Technion-Israel Institute of Technology

MultiView and Millipage –
Fine-Grain Sharing in Page-Based DSMs

Ayal Itzkovitz Assaf Schuster
Computer Science Department
Technion – Israel Institute of Technology
{ayali,assaf}@cs.technion.ac.il

Abstract

In this paper we develop a novel technique, called MultiView, which enables implementation of page-based fine-grain DSMs. We show how the traditional techniques for implementing page-based DSMs can be extended to control the sharing granularity in a flexible way, even when the size of the sharing unit varies, and is smaller than the operating system's page size. The run-time overhead imposed in the proposed technique is negligible.

We present a DSM system, called MILLIPAGE, which builds upon MultiView in order to support sharing in variable-size units. MILLIPAGE efficiently implements Sequential Consistency and shows comparable (sometimes superior) performance to related systems which use relaxed consistency models. It uses standard user-level operating system API and requires no compiler intervention, page twinning, diffs, code instrumentation, or sophisticated protocols. The resulting system is a "thin" software layer consisting mainly of a simple, "clean" protocol that handles page-faults.

1 Introduction

The basic mechanism for implementing software distributed shared memory systems (DSMs) was described for the first time in a seminal paper by Li and Hudak [15] and implemented in the Ivy system [14]. The method relies heavily on the operating system's virtual memory page protection mechanisms, enforcing a sharing granularity which is equal to the size of the virtual memory page (*page-based* DSM*s*). Page sizes, typically a few kilobytes, are usually much larger than the actual sharing granularity of the applications. Therefore, the main problem that researchers have faced in developing page-based DSMs has been *false sharing*, where two or more hosts use different variables that happen to reside in the same page. False sharing can cause a severe performance degradation of programs running on software DSMs and may even lead to slowdowns.

There have been many attempts to overcome the false sharing problem. An extensive study has been conducted and numerous works on relaxing the memory consistency have been written, including [1, 3, 5, 7, 9, 12, 22, 26], to mention only a few. Relaxing the consistency enhances parallelism and may significantly reduce the required communication for memory synchronization. Memory synchronization is generally controlled by calls to a synchronization primitive or a method which is associated with one. Even when not much work is involved, relaxed consistency models do require that the programmer modify the code and be aware of the semantics of the memory behavior. As a result, DSMs using relaxed consistency models trade the abstraction of the underlying memory system for added efficiency.

A different approach was proposed in the Blizzard and the Shasta systems [18, 19, 20]. In order to circumvent the false-sharing problem and provide fain-grain access, Shasta avoids using the virtual memory protection mechanism. Rather, it instruments binary code, wrapping loads and stores with instructions to check for the availability of the data and to maintain its consistency. The result is a fine-grained DSM, capable of sharing memory blocks of arbitrary size. However, high overhead is introduced by the wrapping instructions, which necessitates aggressive optimization techniques.

In this paper we propose a new method, called MultiView, which allows the efficient implementation of fine-grained DSM. Although MultiView does use the virtual memory protection mechanism, it is capable of manipulating the memory in variable size blocks, called *minipages*, which are smaller than the virtual memory page size. MultiView involves little overhead; it usually requires no modifications by the programmer, nor does it require post compilation or code instrumentation of any kind.

MultiView provides two notable advantages. First, false sharing can be avoided simply by associating variables with individual minipages. The system then manages sharing of program variables rather than full pages. Second, MultiView enables a DSM implementation that completely avoids buffer copying in the DSM layer. For this reason MultiView is well suited for integration with high performance messaging layers like Active Messages [24], FastMessages [16] and the VIA interface.

We have implemented a system named MILLIPAGE - a high-performance fine-granularity page-based DSM. MILLIPAGE uses MultiView both for achieving fine granularity and for enhancing performance. Despite the fact that it uses the virtual memory page protection mechanism (and thus can be viewed as "page-based"), MILLIPAGE supports sharing of memory at any granularity. Furthermore, sharing in small granularity imposes only a negligible overhead in MILLIPAGE.

It has recently been noted, e.g. in [23], that the latest advances in communication speed make the complexity of the underlying DSM protocols a non negligible factor in the overall system performance. A notable aspect of MILLIPAGE is its efficient support of Sequential Consistency with a very simple and "clean" protocol, which leads us to the notion of a *thin-layer* DSM. The key element in thin-layer DSMs is the simplicity of handling a request for shared data. There is no need for page twinning, which consumes memory, nor for diff operations, which occupy the cpu, code instrumentation, which blows up the instruction count, or sophisticated protocols which complicate the system. As a result, thin-layer DSMs are simple to develop and debug, easy to use, and impose little overhead on the local operating system and the communication network, beyond that which is required by the applications.

It was recently shown that reducing the granularity in systems which implement strict consistency may achieve performance comparable to that of systems implementing relaxed consistency memory models [19, 27]. In accordance with these findings, Sequential Consistency was employed in MILLIPAGE: initial performance evaluation shows results comparable or superior to those obtained in systems which employ relaxed consistency models.

MILLIPAGE is fully operational in the Distributed Systems Laboratory at the Technion - Israel Institute of Technology [1]. MILLIPAGE uses the Illinois FastMessages [16] on a cluster of 8 Compaq 300Mhz Pentium II machines, interconnected by a Myrinet switch, and running Microsoft Windows-NT.

The rest of this paper is organized as follows. The following section describes the MultiView technique and how to generate and control minipages. In Section 3 we describe the design of MILLIPAGE; we discuss important issues that affected the DSM architecture, issues which arise from applying the MultiView technique and the integration of fast messaging libraries. Initial performance evaluation of MILLIPAGE is provided in Section 4. Finally, we discuss future work and open research topics.

2 The MultiView Technique

Unlike object-based DSM systems, which require a specially tailored compiler or binary code instrumentation, page-based DSM systems provide a transparent and portable software layer that can be developed and used on standard platforms. The main disadvantage of page-based DSMs is that the smallest unit of sharing (the granularity) is the page size, which is determined by the operating system and the underlying processor architecture. The size of a page, as large as 4KB for the Intel Pentium and 8KB for the Digital Alpha, is three orders of magnitude larger than the memory addressing granularity. In this section we propose a novel technique, called MultiView, which enables the sharing of memory in fine granularity as small as that of the memory addressing unit.

2.1 The Basic Idea

Consider three variables x, y, and z, whose size is smaller than that of a page. In a page-based DSM system, the memory for these variables may be allocated on the same page, resulting in false sharing. Consider a mapping of the page that contains these variables to three different, non-overlapping, virtual address regions, starting at addresses v_1, v_2, and v_3, so that each of the locations in the page can be viewed via three different virtual addresses. Each of the regions is called a *a view*. Since access permission is controlled through the virtual memory mechanism, it follows that protection and fault handling can now be applied in three different and independent ways, one for each view. Consequently, the mechanism for maintaining page consistency can

[1]Updated information can be found in our web site at http://www.cs.technion.ac.il/Labs/Millipede

also arbitrate between three different policies, each using the access capability setting of one of the views. Figure 1 depicts this situation.

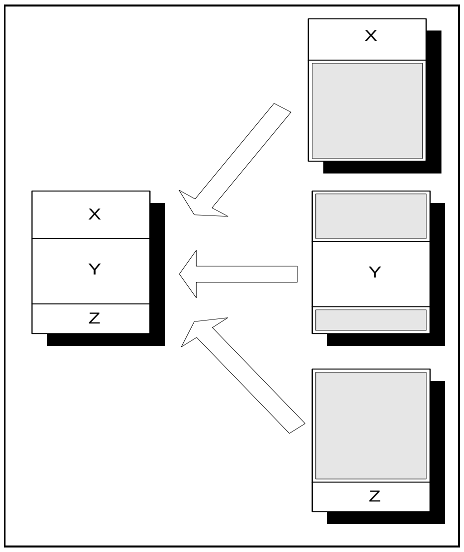

Figure 1: Mapping three virtual views to a single memory page in which three variables reside.

Now suppose the application uses different views to access different variables, say, the variable x is accessed via $v_1 + \text{offset}(x)$, y via $v_2 + \text{offset}(y)$, and z via $v_3 + \text{offset}(z)$. Then, although the variables reside in successive memory addresses on the same physical page, they are seen by the application as if placed in three different (virtual) pages. It is now possible to manage a separate access policy for each variable using its respective view. Moreover, an independent consistency protocol can be implemented for each variable, despite the fact that all three reside on the same page.

2.2 Views, Vpages, and Minipages

Most DSM systems provide consistency guarantees for a single, large, contiguous region of shared addresses. In systems which implement MultiView, the shared space is mapped to several such regions of virtual addresses, called *views*. Consequently, each memory element can be accessed via the virtual memory mapping mechanisms, using any of the views. Since views must be managed in granularity of pages, each view consists of a sequence of virtual memory pages which we call *vpage*s.

MultiView lets the application manage each memory element through a dedicated view, or a vpage, which is said to be *associated* with this element. Because each memory element (in the above example, a variable) can now be managed independently regardless of its size, we call it a *minipage*. Minipage sizes vary; they can be as large as the virtual page size or as small as the basic memory addressing unit.

A minipage is identified by the associated vpage number and a pair $<$ offset, length $>$ which indicates the region inside the vpage where the minipage resides. A protection is controlled for the minipage using the virtual memory mapping mechanisms by manipulating the associated vpage protection. A `NoAccess` protection indicates a non-present minipage, a `ReadOnly` protection is set for read copies, and a writable copy gets a `ReadWrite` protection. Copying minipages between the hosts, invalidating minipage copies, and changing access permissions are all done according to the consistency guarantees and the protocols implementing them.

The basic feature which enables the implementation of a fine granularity DSM using the MultiView method is the independent manipulation of access permissions for minipages which physically share the same memory page, but are accessed by the application via different vpages. For instance, the following protocol implements Sequential Consistency. When a read fault occurs, i.e., when the application attempts to read a minipage for which the associated vpage has a `NoAccess` protection, an accessible copy of that minipage is located in one of the hosts and brought in. Accessing the minipage is then enabled by changing the protection of the associated vpage to `ReadOnly`. Similarly, upon encountering a write fault, a copy of the minipage is retrieved, all other copies are invalidated, and a `ReadWrite` protection is set for the associated vpage.

2.3 Minipages and Views Layouts

Preparing the minipage layout can be done in both static and dynamic fashion. Static layout may divide each memory page into k minipages of equal size. This way, it is easy to calculate the minipage borders when a fault occurs. Static layout may therefore be appropriate for general purpose caching and global memory systems, in order to reduce the page size by a fixed factor [10]. In the dynamic layout each allocation in the shared memory defines its own minipage according to the allocation size, and this minipage is associated with its own vpage. The system should therefore store and maintain a minipage-table (MPT) with the appropriate $<$ offset,length $>$ pair specified for each minipage. Large allocations should still reside in a contiguous region of addresses.

MILLIPAGE design is based on the dynamic layout; it is this option on which we focus in the rest of this paper.

2.3.1 The Privileged View

Multithreaded DSMs commonly use *server threads* for DSM management, and *application threads* for carrying out computation. We call the views that are used by the application threads *application views*.

In addition to the fine granularity capabilities, MULTIVIEW enables another useful feature. An additional, separate view is constructed, called the *privileged view*. The protection of the privileged view is fixed and set to `ReadWrite`. The DSM server threads may use it at all times to access the memory, and are thus not constrained by the DSM memory protections, as determined by the consistency guarantees imposed on the application views.

There are two common DSM operations which highly benefit from using the privileged view. First, atomic minipage updates can now be performed in user-mode. While access is blocked through the application views, the DSM server thread can freely access minipages via the privileged view. Once the modification is complete, the protection in the application views can be reduced, thus enabling the application threads to access the modified memory. In this way multithreading can be employed safely.

Secondly, buffering and copying of long messages can be avoided. The privileged view may be used to directly send/receive minipages from/to the user space. While communication activities are taking place using the privileged view, application thread access to the same memory region is forbidden by the protection imposed on the application views. In this way, buffer copying is completely eliminated in the DSM layer. Later, in Section 3.5, we describe how all this is implemented in MILLIPAGE and integrated with the Illinois FastMessages.

2.4 Implementation Issues

We have implemented MULTIVIEW in Windows-NT in a DSM system called MILLIPAGE. Mapping several views of virtual addresses to a shared memory region was accomplished using the *file mapping* API, as follows. First, a *memory object*[2] is created by calling the `CreateFileMapping` API. A memory object is simply a virtual memory region, allocated in the kernel address space of a process, and backed-up by the paging file. This memory object is the shared region on which minipages will be allocated.

Suppose the maximal number of minipages that reside on the same page of the memory object is n. We thus need $n+1$ different views of the memory object: n application views for use by the application threads and one for the privileged view. The views are created using the `MapViewOfFile` API, which is called once for each established view.

For each application, MILLIPAGE keeps a single process on each of the hosts. By carefully configuring the DSM addresses, the views are guaranteed to map to the same addresses in all processes, so there is no need for address translations between the hosts. Once mapped, the protection of vpages (recall that vpages are the virtual pages composing a view) can be manipulated independent of the protection of other vpages that are mapped to the same memory object page. One of the views is made privileged with the protection of all its vpages set to `ReadWrite`. This view is used by the DSM server threads to read and update the memory object.

The construction of the views is performed during system initialization. When the application issues an allocation request, the DSM searches for a suitable region in the memory object, and defines it as a minipage (or a set of consecutive minipages). The DSM associates the newly defined minipage with one of the application views. More precisely, suppose an allocation procedure defines a new minipage $\mathcal{M}$. $\mathcal{M}$ is associated with a certain vpage in one of the views, and an address p in this vpage is returned. During allocation, a new entry is formed in the minipage-table MPT, containing both the offset of p in the vpage and the size of $\mathcal{M}$.

If mapping to $\mathcal{M}$ spans several vpages in the associated view, the above is generalized in a straightforward way.

Figure 2 describes the views configuration and the dynamic allocation of minipages to variables during `malloc` calls.

3 System Design

3.1 Design Goals

MILLIPAGE is a software-only implementation of a fine-granularity strictly-consistent distributed shared memory. Although MILLIPAGE can be seen as a page-based DSM, it is not limited to sharing memory in granularity of full pages.

The main design goals of MILLIPAGE are outlined below. We discuss them throughout this section.

[2] A memory object is called a *memory section* in Windows-NT terminology [21].

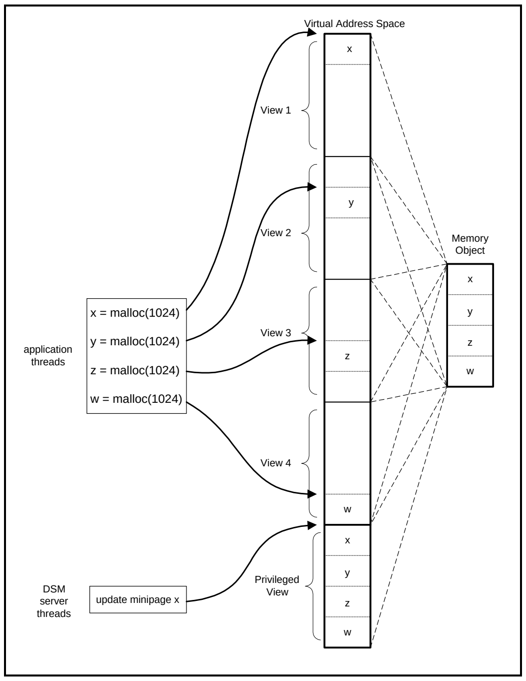

Figure 2: Views configuration and dynamic minipage allocation for independent variables. DSM service and control threads use the privileged view. In general, a memory object may be larger than a page and there may be "gaps" between the views.

- User level software implementation.
- Efficient strict memory model.
- Fine granularity DSM.
- Thin layer DSM.
- Integration with fast networking media and fast messaging packages.
- Multithreaded support to efficiently utilize SMP machines.

3.2 Shared Memory Model

The programming model in MILLIPAGE is Sequential Consistency, which is implemented through the Single-Writer/Multiple-Readers (SW/MR) protocol: at any point in time, for any minipage, there can be either read copies or a single writable copy. Thus, parallel applications run on MILLIPAGE as if they were executing on a physically-shared memory SMP machine; there is no need for either explicit or implicit memory synchronization.

Aside from the initial setting of the maximal number of views and the size of the shared-memory, the allocation process is transparent from the programmer point of view. Some restrictions do apply, however, in the sense that only dynamic allocations can be shared; those are the only allocations managed by the DSM mechanism. For this reason, the application should be written so that the sharing unit is equal to the allocation size.

Allocating from the shared memory is performed via a `malloc`-like API. The returned pointer can point to any of the application view's address spaces (but never to the privileged one). It can then be used in the usual way as if it had been returned by the standard `malloc` call.

3.3 Protocols

An important goal in the design of MILLIPAGE was to encapsulate the DSM functionality in a very thin software layer. This was accomplished by implementing a very simple SW/MR protocol, which we now proceed to describe.

On each host a single MILLIPAGE process is started, running both application and server threads. One of the processes is elected as the *manager*. As part of the manager role, it is in charge of maintaining the directory information of minipage and minipage copy locations, minipage sizes, and the association of view addresses with their minipages. This information is stored in the minipage-table (MPT), which is stored at the manager host.

All requests for missing minipages (resulting from a fault) are sent to the manager, which redirects them to the appropriate hosts. Requests which arrive while an earlier request to the same minipage is still in process are queued in the manager.

A request which arrives at the manager contains only the faulting address. The manager looks it up in the MPT and stores the translation information (the minipage base address, its size, and its address in the privileged view) in the message header where the appropriate space has been reserved. When the message is forwarded, it carries the translation information.

The manager-centered design significantly simplifies the DSM layer for the non-manager processes. Whenever a fault occurs, the fault handler issues a request and sends it directly to the manager. No computation or local search in any data structure is required. The thread then waits on an event while its request is serviced.

When the reply arrives, it is handled by a DSM server thread, which receives the message in two stages: first, the message header arrives, containing the original request and the translation information. Next, the minipage contents are received directly at the appropriate address in the privileged view, as specified in the request header. When the receive operation completes, the protection for the minipage is set and the faulting thread is signaled to continue its execution.

The manager's role is essentially to mark and forward requests to hosts, and to maintain the MPT. If a read copy is requested, the manager updates the minipage copyset and forwards the request. If an exclusive write copy is requested, the manager first chooses one of the hosts in the copyset, instructs all others to invalidate their copies, and then forwards the request to the remaining one. This host will then invalidate its copy and send the minipage directly to the host where the fault occurred. The pseudo-code of the *complete protocol* is given in Figure 3.

Once a fault is served and the faulting thread wakes up, it sends an additional ack message to the manager. Although this additional message might seem to reduce performance, it actually solves a few potential problems. First, a possible livelock caused by race conditions on two or more threads is eliminated. This is reminiscent of the delta mechanism [6], which ensures that a page remains in the host for a certain amount of time before its removal is permitted. Second, the potential need for message queuing in the non-manager hosts is eliminated. A host which receives a request is never in the process of acquiring the same minipage, nor has it given the minipage away. Hence, a request which arrives at a non-manager host can always be served immediately, completely eliminating the need for buffers.

Since all the messages which are sent to and by the manager are small (32 bytes in our current implementation), reading and writing them to and from the network does not involve much overhead, leaving the manager highly responsive.

3.4 The MILLIPAGE Library

MILLIPAGE is implemented as a win32 library on Windows-NT. It exports several APIs for the use of application development. Its interface includes an initialization routine, spawning local and remote threads, a memory allocation routine, and common synchronization calls such as barriers and locks.

Applications can be compiled with any standard compiler. They should then be linked with the MILLIPAGE and the FM libraries. The final executable integrates the DSM run-time system with the application code and can be started concurrently on several hosts. Figure 4 shows the process of preparing an application to run on top of MILLIPAGE. Since MILLIPAGE is multithreaded and its architecture supports multithreaded applications, only a single instance of the application should be executed on each host, even if this host is a multi-processor (SMP) machine.

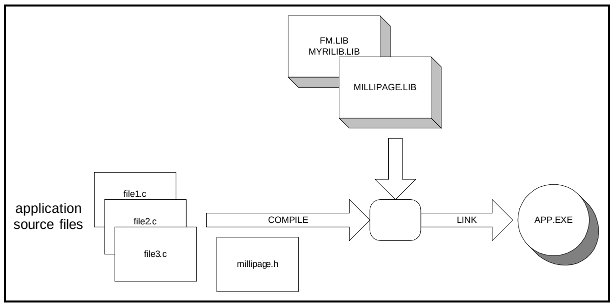

Figure 4: Preparing an application for execution on top of MILLIPAGE. The final executable includes both the application code and the MILLIPAGE libraries.

3.5 FastMessages

MILLIPAGE uses FM on Myrinet as its communication layer. FM was developed at the University of Illinois as a low latency messaging layer, working on fast networking media such as Myrinet. We measured a roundtrip delay of 25 usec for small messages (200 bytes) and 180 usec for 4 KB messages. FM achieves network bandwidth higher than 1 GB/sec on our switched Myrinet LAN.

FM provides a reliable and FIFO ordered messaging service. Its high performance is due to two main features. First, it does not switch between user mode and kernel mode, but rather transfers data directly to and from the user space. Second, it minimizes buffer copying of messages. When a send operation is initiated by the user process, FM verifies that there is sufficient space in the network buffers at the target network adapter. Then the message is read directly from the user space, through the local adapter to the target network adapter card. Using DMA, the message is copied at the receiver side from the buffers of the network adapter card to the FM reserved (and pinned) memory. The receiver should then poll in order to check that the message has arrived and can be processed by its handler.

Although we measured low latencies and high bandwidth for messages that were sent using FM, we confronted a problem caused by FM's polling policy. When coordination of the send-receive operations is possible, e.g., in message-passing interfaces such as MPI and PVM, the sender thread and the receiving

```
On Read or Write Fault
pmsg→event = myEvent;
pmsg→type = {READ or WRITE}_REQUEST;
pmsg→from = ME;
pmsg→addr = fault_addr;
Send pmsg to manager;
wait(myEvent);

Handle Read Request
if (access(pmsg→fault_addr) == ReadWrite)
    then set access(pmsg→fault_addr) to ReadOnly
pmsg→type = READ_REPLY;
Send pmsg to pmsg→from;
Send(pmsg→privbase,pmsg→pgsize) to pmsg→from;

Handle Write Request
Set access(pmsg→fault_addr) to NoAccess
pmsg→type = WRITE_REPLY;
Send pmsg to pmsg→from;
Send(pmsg→privbase,pmsg→pgsize) to pmsg→from;

Handle Read or Write Reply
Recv(pmsg→privbase,pmsg→pgsize);
Set access(pmsg→fault_addr) to ReadOnly/ReadWrite
SetEvent(pmsg→event);

Handle Invalidate Request
Set access(pmsg→fault_addr) to NoAccess
pmsg→type = INVALIDATE_REPLY;
Send pmsg to manager;
```

```
Manager: Translate(pmsg)
pmsg→base = get_minipage_base(pmsg→fault_addr);
pmsg→pgsize = get_minipage_size(pmsg→fault_addr);
pmsg→privbase = addr2priv(pmsg→base);

Manager: Handle Read Request
Translate(pmsg);
p = find_replica(pmsg→fault_addr);
Forward pmsg to p;

Manager: Handle Write Request
Translate(pmsg);
forall p in replicas of pmsg→fault_addr {
    pmsg→type = INVALIDATE_REQUEST;
    Send pmsg to p;
}
Manager: Handle Invalidate Reply
if got less than (#replicas - 1) replies then return;
pmsg→type = WRITE_REQUEST;
p = find_replica(pmsg→fault_addr);
Forward pmsg to p;
```

Figure 3: The complete protocol in MILLIPAGE. Note the simplicity of the DSM layer: no buffer copying, queuing, table lookup, or translation of any kind are required, except at the manager.

thread can be co-scheduled to achieve good timing of the polling action. Unfortunately, such coordination is impossible in DSM systems, since (mini)page faults occur in an unpredictable manner. When a thread faults and sends a (mini)page request to another process, this process will commonly be busy in application-related computation. In this situation, frequent polling will slow down the computation, whereas infrequent polling will cause large delays in receiving and handling the request. Since both responsiveness and efficiency have a major impact on DSM performance, we had to address this problem in our system design. We proceed below to describe our solutions.

3.5.1 Communication and DSM Server Threads

Application threads run as native kernel threads of the operating system. When started, they invoke a wrapper routine that installs the MILLIPAGE exception handler and calls the original main thread routine. Aside from this, the application code is executed as is and application threads experience no interference unless they fault.

In addition to the application threads, MILLIPAGE spawns three threads that are in charge of maintaining the memory consistency and servicing requests: two DSM server threads and a millisecond timer thread. One DSM server thread, called the *poller*, is constantly polling for messages in a busy loop. It promptly serves received messages, then continues to poll. The *poller* runs at a low priority; it does not consume cpu cycles when required by application threads. Another thread, called the *sweeper*, differs from the *poller* only in that it waits on an event before it issues a single poll.

Ideally, we would let the *sweeper* sleep for periods of a few hundred microseconds, maximizing responsiveness without overloading the cpu. However, high resolution timers are provided in Windows-NT only through *multimedia timers*, of which the finest resolution is 1ms. Therefore, a third thread, the *timer*, was used to set up an event to wake up the *sweeper* once in every millisecond. For higher accuracy, and since it consumes, over a period of a millisecond, very little time, the priority of the *sweeper* was set to exceed that of the application threads. Note that the multimedia *timer* thread runs in high priority as well.

Our measurements showed that the deviation in the time between timer events is extreme: most of them appear either within several tens of microseconds, which overloads the cpu, or take several milliseconds, which degrades responsiveness. We have found that this anomaly of the Windows-NT timers has been recently reported in [11], where a standard deviation of 955 microseconds was measured for 1ms timers on similar hardware - nearly equal to the mean! Since polling is in the core of our system, the timer inaccuracy significantly affected the responsiveness of the DSM server threads, and thus the overall performance of MILLIPAGE.

4 Performance Evaluation

In this section we discuss the performance implications of using MultiView and the performance of the MILLIPAGE system. Our testbed environment consists of a network of eight Pentium II 300Mhz uniprocessor machines, running Windows-NT Workstation 4.0 SP3. Each machine has 128 Mbytes of RAM. The architecture page size is 4 KB. The cluster is interconnected by a switched Myrinet LAN [4]. The Myrinet drivers were taken from the HPVM release 1.0 for NT [8].

4.1 MultiView Limitations

In order to measure the overhead of using MultiView we used a standalone test application which is not related to the MILLIPAGE system. In this way the overhead of MultiView could be distinguished from that of other implementation-related sources.

Our test application allocates an array of characters (bytes). The array resides in minipages of equal size. The number of minipages in each page is equal to the number of views. The main application routine iteratively traverses the array, reading each element (from first to last) exactly once in each iteration. We experiment with two parameters: the size of the array (which represents the size of the shared memory), N, and the number of views n.

As expected, the total size of committed memory increases with the size of the allocated region, independent of the number of views. We were limited, however, by the size of the available virtual address space, which stands at about 1.63 GB. Thus, we could only experiment with $n \leq 1.63\text{Gig}/N$ (e.g., for $N = 16$MB we could set up to $n = 104$ views).

For $N = 512$KB we hardly noticed any overhead. For $1 \leq n \leq 32$ the measured overhead is always less than 4% for 512KB $\leq N \leq$ 16MB. Note that $n = 32$ means a sharing granularity of 128 bytes.

In order to study the limitations of the MultiView technique we also experimented with a very large number of views, up to 1664. The results show

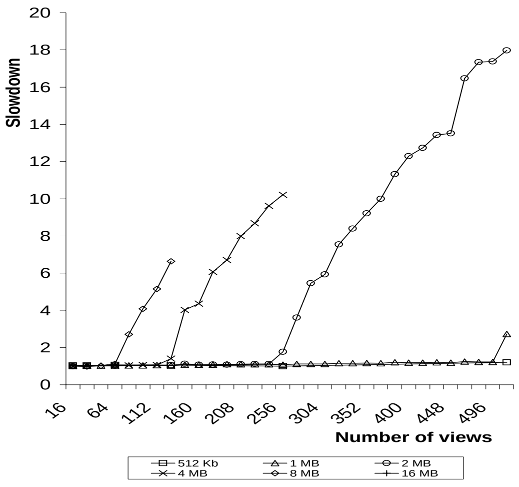

Figure 5: Overheads of MultiView. Note that the breaking-points where the overheads become substantial depend on the size of the shared memory in an inversely linear fashion.

that, at certain *breaking-points*, the overhead of using many views becomes substantial and may lead to severe slowdown. Figure 5 summarizes the results up to 512 views (minipages of size 8 bytes). Taking a closer look at the graphs, we could make several observations. First, for each N, beyond the breaking-point, the overhead increases linearly with the number of views. Second, beyond their respective breaking-points, the graphs for all N increase with the same slope. Third, the breaking-points themselves depend on N in an approximately inversely-linear fashion: they appear at the points where $n \cdot N = 512$ (N in MB). Finally, when we tried to allocate a large N and use only a fraction of it, the breaking-point appeared earlier than in the case where only the accessed fraction was allocated.

Our first explanation is that the slowdown is related to the enormous increase in TLB misses, and to the size of the cache. The number of active PT entries (not MPT) at the breaking points becomes 128K. The TLB size in the Pentium II is 64 data entries and 32 code entries. A PTE is four bytes in size, and the PTEs are cachable. A TLB miss and a 1st level cache miss cause a single stall; an increase there may thus explain the 1–4% slowdown we experienced with smaller N and n, but cannot possibly explain the slowdown that appears beyond the breaking-points.

The size of the 2nd level cache in our machines is 512KB, thus the breaking-points occur precisely when the PTEs can no longer be cached there. The cache is physically tagged. Attempting to access a new minipage causes the virtual memory translation mechanism to search for a new PTE, which, when the cache is congested, causes a miss. In the extreme situations that we tested, beyond the breaking-points, the cache misses caused by the missing PTEs dominate the cache activity.

We note here that other factors might also have affected the performance of MultiView. One example is a possible performance degradation caused by overloading the operating system's internal data structures, keeping track of extensive data mapping. Another example is the quicker cache exhaustion that may occur in virtually tagged caches.

As we describe later in this section, the applications in our benchmark suite all use less than 32 views and thus the overhead they experience is negligible.

4.2 Basic Costs in Millipage

A major goal in millipage's design was to minimize the dsm protocol time by implementing a thin protocol layer. Table 1 shows the measured times (costs) of some basic operations that are part of the dsm protocols in millipage.

operation	μs
access fault	26
get protection	7
set protection	12
header message send/recv (32 bytes)	12
a data message send/recv (0.5 KB)	22
a data message send/recv (1 KB)	34
a data message send/recv (4 KB)	90
minipage translation (MPT lookup)	7

Table 1: Cost of basic operations in millipage.

The time it takes to bring in a page for reading in millipage is 204 μs for minipages of size 128 byte, and 314 μs for minipages of size 4 KB. The difference in arrival times for a minipage request arriving in a single hop as opposed to two hops was slight. The time it takes to bring in a page for writing in millipage is 212–366 μs for 128 bytes minipages, and 327–480 μs for 4KB minipages. These times vary according to the number of read copies that should be invalidated prior to serving the write request.

A *barrier* between 1 to 8 hosts takes 59–153 μs (linearly in the number of hosts) and a *lock* followed by an *unlock* operation takes 67–80 usec.

In addition, we measured diff creation time in our setting. Our measurements show that a run-length diff operation (as described in [5]) for 4KB page takes 250 μs and decreases linearly with the size of the page. Obviously, this time is not negligible, and would have dominated the overhead if it were required in the dsm protocol.

4.3 Applications

This section presents the results of parallel execution of five benchmark applications on the millipage system. Our application suite consists of: Water-nsquad (WATER) and LU-contiguous (LU) from - SPLASH-2 [25]; Integer-Sort (IS) from the NAS parallel benchmarks [2]; Successive Over Relaxation - (SOR) and the Traveling Salesperson Problem (TSP) from the Treadmarks [13] benchmark applications.

Table 2 summarizes application information such as data sets, shared memory size, and the sharing granularity. As can be seen, different applications naturally use minipages of different sizes, which in turn dictates the number of views as explained earlier in Section 2.

The code for memory allocation in three of the applications was slightly modified in order to equate the allocations and the sharing units.

In the original code for WATER, all the molecules are stored in a single array (VAR) and are referenced via pointers. We altered the main function so that each molecule will be allocated separately.

IS allocates a shared portion of memory where the keys reside. The array is relatively small and is divided into regions of equal size where each host is in charge of another region. We modified the allocation routine to have these regions allocated separately and thus reside in different minipages.

TSP allocates a global memory structure that contains an array of tours. Each tour (`TourElement`) is of size 148 bytes and each tour is manipulated exclusively by one of the tasks. We extracted the array out of the global memory structure, leaving there only a pointer. We then allocated each tour independently so that each one resides in a separate minipage.

There was no need to modify SOR, as it uses a matrix which is allocated row by row. The granularity of a row is suitable as the sharing unit, so the size of a row may determine that of a minipage. Similarly, it was not necessary to modify LU, as it

	Input Set	Shared Mem. Size	Num. views	Sharing Granularity	Barr.	Locks
SOR	32768x64 matrices	8 MB	16	a row, 256 bytes	21	-
IS	2^{23} numbers, 2^9 values	2 KB	8	256 bytes	90	-
WATER	512 molecules	336 KB	6	a molecule, 672 bytes	29	6720
LU	1024x1024 mat., 32x32 blocks	8 MB	1	a block, 4 KB	577	-
TSP	19 cities, recursion level 12	785KB	27	a tour, 148 bytes	3	681

Table 2: Application Suite.

builds a matrix by allocating sub-blocks, each of size $32 \times 32 \times |int| = 4$KB. Since the granularity of these sub-blocks is suitable as the sharing unit, the size of a minipage may be set equal to that of a 4KB page.

4.3.1 Speedups

Figure 6 summarizes the speedups on MILLIPAGE for each of the applications, when executing on 1 to 8 processors.

When examining the results, one should keep in mind that because of the FM polling problem and the large-grain timers described in Section 3.5, our system suffers from relatively high delays in servicing minipage requests. Since the resolution of the operating system timers is constrained to 1 ms (and is extremely inaccurate, see Section 3.5.1 and [11]), we experienced an average delay of about 750 μs for minipage requests. Only about a third of the delay comes from the DSM layer (see Section 4.2 above), while the rest is due to the slow response of the server thread: an average of more than 500 μs.

Despite the above, our initial experience with MILLIPAGE shows encouraging results. IS and SOR achieved speedups that are close to linear: the efficient resolution of false sharing led to a relatively small communication volume. WATER's performance was comparable to that achieved in reduced consistency systems. This performance was achieved by *chunking* molecules in larger minipages, a method that we describe later in this section.

LU achieved relatively good results, mainly due to the thin DSM layer which reduces the protocol overhead. In addition, in order to minimize the large minipage service delays explained above, we inserted two prefetch calls during the LU computation. We believe that this prefetch mechanism will not be needed once the FM polling problem is resolved, and/or the operating system timer resolution is refined.

False sharing was resolved in TSP, except for a single data race for updating the minimal tour found so far. Although the modification of this variable is protected by means of mutual exclusion, it is frequently read through an unprotected section. We changed a single code line in which this variable is updated, so that it pushes readable copies of the new value to all hosts. It is instructive to consider the minipage allocation size, which is equal to that of a tour element: 148 bytes. Providing granularity of 128 or 256 bytes ("cleaner" numbers that divide a page size) may involve a large increase in false sharing due to the pattern in which TSP assigns tours to processors: two adjacent tours are often assigned to two different processors.

4.4 Chunking

In the tradeoff between false sharing and aggregation we found that it is sometimes better to use granularity larger than the sharing unit size, with the cost of some additional false sharing. The main reason is the longer time it takes to bring in a relatively large portion of the memory when fine granularity is employed. Aggregation may reduce the number of DSM protocol calls, number of messages, page protection changes, and other sources of overhead, thus improving performance.

In our test applications we found WATER to behave much worse when we allocated each molecule in a separate minipage, than when several of them were chunked into a larger minipage. At the beginning of each iteration of WATER, each processor brings in the entire molecules' structure, namely, the *read phase*. When the allocation is done in granularity of molecules, the read phase takes a long time to complete due to the large number of minipage-faults that should be served. Despite the fact that false sharing is avoided for the computation which follows the read phase, this phase has a major impact on degrading the speedup. We therefore set a switch in MILLIPAGE, called *chunking level*, that makes MILLI-

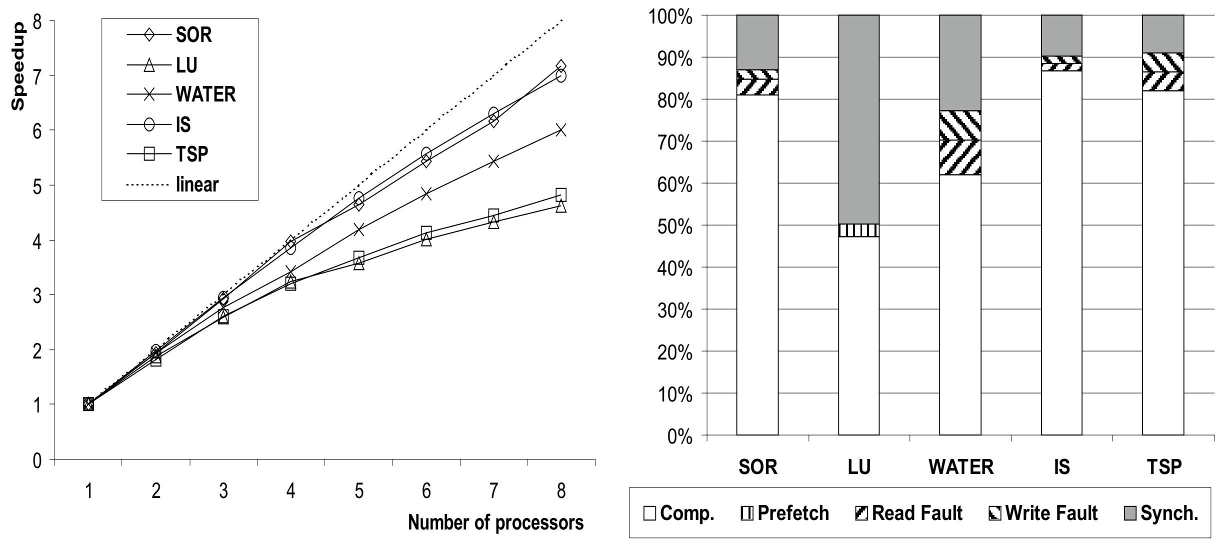

Figure 6: Summary of performance results. The breakdown graph on the right (for eight hosts) proves that the efficient resolution of false sharing results in a low communication volume, which shows itself in small total service times for faults. The total service time will further decrease once the polling and timer resolution problems are solved (see Section 3.5).

PAGE aggregate allocations in larger minipages.

Increasing the chunking level causes our manager to report more *competing requests*, i.e., requests for a certain minipage that are enqueued while a previous request is being served. When there was no chunking, 21 competing requests were reported. This surprised us at first, as false-sharing had been completely eliminated, and we expected no competing requests whatsoever. However, [17] already reported that there is a Write-Read data-race in WATER. Apparently this data-race is the cause for the competing requests reported by our manager.

We experimented with chunking in WATER for the shared molecules structure, setting the chunking level to increase from 1 to 6. We also ran WATER with no false-sharing control, so molecules were allocated in the traditional way; i.e., in minipages of a page size, disregarding minipage boundaries. As expected, the number of competing requests increases with the chunking level, reaching up to 601 when no false-sharing control is employed. From Figure 7 we conclude that the best performance is achieved for a chunking level of 4 or 5.

It is interesting to compare our findings with those of Shasta, as reported in [19]. They also found WATER to benefit from a relatively coarse granularity. However, they set the granularity level to 2048 bytes, while we found the optimum in a larger granularity level, 2688 or 3360 bytes. We expect that when the FM polling problem is solved (see Section 3.5), the optimal chunking level will decrease, in which case we may find it closer to that found by Shasta.

5 Discussion and Future Work

Prior to this work, the notion of a page-based DSMs which use page protection mechanisms was perceived as contradicting that of sharing data in fine granularity, and tightly coupled with that of false-sharing. Although relaxed consistency memories are successful in solving the problem, they require complicated protocols which introduce new sources of overhead.

In this paper we proposed a new method, called MULTIVIEW, that unbinds the (seemingly) tight connection between page-based DSMs and the need to share data in granularity of pages. We describe one realization of MULTIVIEW in a DSM system called MILLIPAGE. In addition to the MULTIVIEW implementation, the design of MILLIPAGE gives rise to other important issues such as the notion of a thin-layer DSM and the integration of fast messaging layers.

The MULTIVIEW technique and its implementation in MILLIPAGE open a new avenue of research directions. Work in these directions is already under way. We proceed in this section to mention only a few of them.

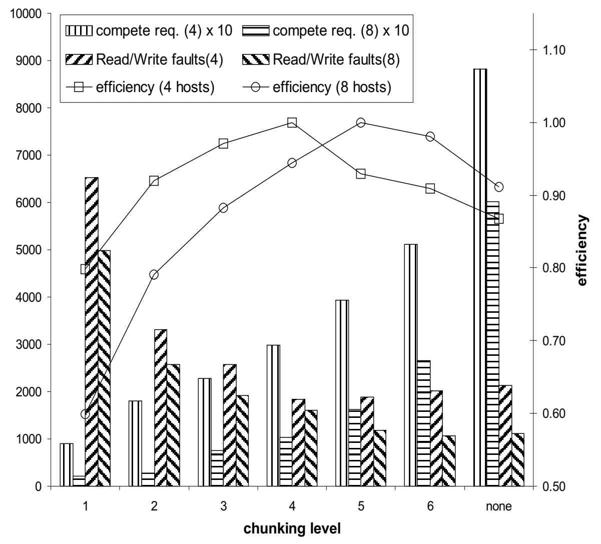

Figure 7: The effect of chunking in WATER. The optimal speedup is obtained at the chunking level of 4 for four hosts and 5 for eight hosts. Note the opposite tendencies of false-sharing and remote requests, represented by the number of competing requests and the number of Read/Write faults, respectively. The efficiency, which is given relative to the best chunking level, is determined by this tradeoff.

Reduced-Consistency Protocols

When the minipages defined for a certain application are larger than the sharing unit, i.e., the chunking level is set higher than one (see Section 4.4), performance may benefit from employing reduced-consistency protocols such as Lazy Release Consistency [12, 26]. Thus, chunking reduces the overhead involved in fine-grain operation, while false-sharing is eliminated through the reduced consistency protocol. The overhead involved in the reduced consistency protocol itself is small compared to that measured in traditional page-based systems, due to the smaller page size.

Compiler Work

While we chose to use only the operating system API, we believe that much can still be done through the compiler. A compiler can map and re-map variables to views, optimize accesses through those views, minimize PT usage, reduce TLB misses, etc. Large static variables can be distributed carefully among views in order to tune the granularity level with the access pattern.

Composed-Views

Complex data structures (such as multi-dimensional arrays) may be stored in groups of minipages. It might be helpful for an application to access these structures using different views at different stages. Higher level views may be associated with groups of lower level views, or groups of minipages. Obviously, the access permissions to such a *composed-view* should be set to the least of the access permissions of its components.

One good example where composed-views (or compiler work) can be used is the chunking-level in WATER, as discussed in Section 4.4. Clearly, the read phase in WATER could benefit from a coarse grain operation mode, whereas the later write phase would accelerate in a fine grain mode due to the elimination of false-sharing. Hence, replacing the current compromise by arbitration between fine-grained and coarser-grained views would speed up the computation.

Access Locality in the Page Table

The main weakness of MULTIVIEW is the lost of locality in using PTEs due to the unusual memory layout (see Section 4.1). Nevertheless, locality is not completely lost, but is preserved across views. A future work for integrating minipages inside the operating system may consider altering the PT data structure to exploit this memory layout.

Global Memory Systems

Recently, there has been a lot of research in utilizing remote memories for storing memory pages, thus establishing an additional stage in the memory hierarchy. It has been shown that using subpages as the transfer units leads to substantial performance boost [10]. We believe that MULTIVIEW can be used for implementing subpages in global memory systems.

Acknowledgments

We thank the referees, whose comments helped us improve the paper. We are grateful to Roy Friedman for his valuable advises throughout this work. We thank Evan Speight for providing us with benchmark applications from Brazos. We cheer the HPVM team from Illinois for developing and releasing the stable Myrinet drivers. Thanks to their drivers, we have gotten our Myrinet LAN working in just a short time.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, pages 26–37, February 1996.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames, August 1991.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, October 1991.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[7] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA'93)*, pages 144–155, May 1993.

[8] HPVM 1.0 for Windows NT 4.0 on x86. CSAG, University of Illinois. http://www-csag.cs.uiuc.edu/projects/hpvm/sw-distributions/.

[9] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.

[10] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. Evans II, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reduing Network Latency Using Subpages in a Global Memory Environment. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 258–267, October 1996.

[11] M. B. Jones and J. Regehr. Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. In *Proceedings of the Eighth International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'98)*, pages 107–110, July 1998.

[12] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[13] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[14] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[15] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, August 1986.

[16] S. Pakin, V. Karamacheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60–73, 1997.

[17] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, October 1996.

[18] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[19] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, October 1996.

[20] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI)*, pages 297–306, October 1994.

[21] David A. Solomon. *Inside Windows-NT, 2nd Edition.* Microsoft Press, 1998.

[22] E. Speight and J.K. Bennett. Brazos: A Third Generation DSM System. In *First Usenix-NT Workshop*, pages 95–106, August 1997.

[23] M. Swanson, L. Stroller, and J. B. Carter. Making distributed shared memory simple, yet efficient. In *Proc. of the 3rd Int'l Workshop on High-Level*

Parallel Programming Models and Supportive Environments, pages 2–13, March 1998.

[24] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, May 1992.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, June 1995.

[26] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, October 1996.

[27] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed consistency and coherence granularity in dsm systems: A performance evaluation. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, June 1997.