Accelerating Shared Virtual Memory via General-purpose Network Interface Support

Angelos Bilas (bilas@eecg.toronto.edu)
Department of Electrical and Computer Engineering
10 King's College Road, University of Toronto, Toronto, ON M5S 3G4, Canada
and
Dongming Jiang (dj@cs.princeton.edu)
Department of Computer Science
35 Olden Street, Princeton University, Princeton, NJ 08544, USA
and
Jaswinder Pal Singh (jps@cs.princeton.edu)

Clusters of symmetric multiprocessors (SMPs) are important platforms for high performance computing. With the success of hardware cache-coherent distributed shared memory (DSM), a lot of effort has also been made to support the coherent shared address space programming model in software on clusters. Much research has been done in fast communication on clusters and in protocols for supporting software shared memory across them. However, the performance of software virtual memory (SVM) is still far from that achieved on hardware DSM systems. The goal of this paper is to improve the performance of SVM on system area network clusters by considering communication and protocol layer interactions.

We first examine what are the important communication system bottlenecks that stand in the way of improving parallel performance of SVM clusters; in particular, which parameters of the communication architecture are most important to improve further relative to processor speed, which ones are already adequate on modern systems for most applications, and how will this change with technology in the future. We find that the most important communication subsystem cost to improve is the overhead of generating and delivering interrupts for asynchronous protocol processing.

Then we proceed to show that by providing simple and general support for asynchronous message handling in a commodity network interface (NI), and by altering SVM protocols appropriately, protocol activity can be decoupled from asynchronous message handling and the need for interrupts or polling can be eliminated. The NI mechanisms needed are generic, not SVM-dependent. We prototype the mechanisms and such a *synchronous home-based LRC* protocol, called *GeNIMA* (GEneral-purpose Network Interface support for shared Memory Abstractions), on a cluster of SMPs with a programmable NI. We find that the performance improvements are substantial, bringing performance on a small-scale SMP cluster much closer to that of hardware-coherent shared memory for many applications, and we show the value of each of the mechanisms in different applications.

Categories and Subject Descriptors: C.2.4 [**Computer Systems Organization**]: Performance of Systems—*Shared Virtual Memory*

General Terms: Design, Performance

Additional Key Words and Phrases: clusters, system area networks, shared virtual memory, applications

Parts of this work have appeared as conference publication in SC97 [9] and ISCA99 [8].

1. INTRODUCTION

As clusters of workstations, PCs or symmetric multiprocessors (SMPs) become important platforms for parallel computing, there is increasing interest in supporting the attractive, shared address space (SAS) programming model across them in software. Supporting a programming model gives rise to a communication architecture that is shown in Figure 1. The lowest layer is the *communication layer,* which consists of the communication hardware and the low level communication library that provide basic messaging facilities. Next is the *protocol* layer that provides the programming model to the parallel application programmer. In this paper we are interested in all–software DSM protocols. In particular, we focus on page–based shared virtual memory (SVM). Finally, above the programming model or protocol layer runs the *application* itself.

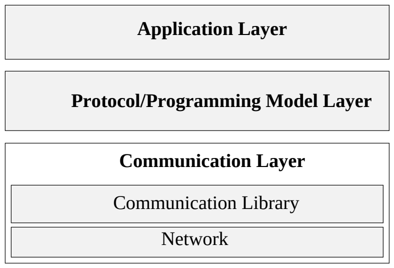

Fig. 1. The layers that affect the end application performance in software shared memory.

In the last few years there has been much improvement of SVM protocols and systems, and several applications have been restructured to improve performance substantially [32; 25; 53; 34]. However, Figure 2 shows that parallel performance is still not satisfactory. Speedups on a 16-processor, all-software, home-based SVM system running on a Myrinet-connected cluster of Pentium Pro Quad SMPs [42] are still substantially lower than those achieved on hardware-coherent machines for the same problem sizes, even with restructured applications. The processors used by the two systems are different (although of similar generations and with similar clock speeds), so a direct comparison cannot be made, but the implication is clear. This difference in performance in many cases is attributed to the the fact that clusters use less aggressive and specialized communication subsystems. Improving communication layers for clusters can help reduce this gap. However, it is not clear which parameters of the communication subsystem are most important.

This paper examines two basic questions: (i) what are the important communication system bottlenecks that stand in the way of effective parallel performance; in particular, which parameters of the communication architecture are most important to improve further relative to processor speed, which are already adequate on modern systems for most applications, and how will this change with technology in the future and (ii) how can we enhance the communication layer and design new protocols that take advantage of these enhancements to alleviate the system bottlenecks. The research results present here are extended versions presented in previous conference publications of the authors [9; 8].

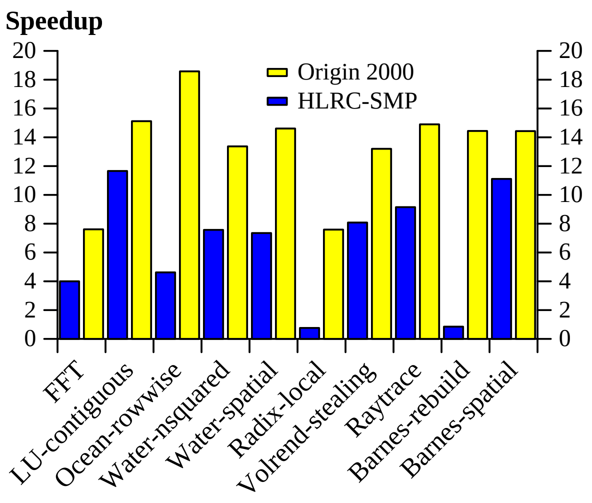

Fig. 2. Application speedups for a hardware distributed shared memory (DSM) machine (the SGI Origin 2000) and an SVM system (our Base protocol).

We examine the first question through detailed architectural simulation using applications with widely different behavior. We simulate a cluster architecture with SMP nodes and a fast system area interconnect with a programmable network interface (i.e. Myrinet). We use a *home-based* SVM protocol that has been demonstrated to have comparable or better performance than other families of SVM protocols. The base case of the protocol, called home-based lazy release consistency (HLRC) does not require any additional hardware support. The major communication performance parameters we consider are the host processor *overhead* to send a message, the network interface *occupancy* to prepare and transfer a packet, the node-to-network *bandwidth* (often limited by I/O bus bandwidth), and the *interrupt cost*. We do not consider network link latency, since it is a small and usually constant part of the end-to-end latency, in system area networks (SAN). We find, somewhat surprisingly, that host overhead to send messages and per-packet network interface occupancy are not critical to application performance. In most cases, interrupt cost is by far the dominant performance bottleneck, even though our protocol is designed to be very aggressive in reducing the occurrence of interrupts.

In SVM systems, nodes exchange both protocol control information and application shared data with messages. Thus, there is a need both for handling messages that arrive asynchronously at the destination node as well as performing protocol operations related to those messages. In current SVM systems the two activities are coupled together and performed by the host processor. For example, SVM protocols manipulate protocol-level page timestamps when a page request message or a page update message is received. Essentially, the sending node asks for some service to be performed by the servicing node by means of asynchronous messages using either interrupts or polling. We find that these interrupts constitute a major system bottleneck. Using polling introduces a number of issues (application code instrumentation, frequency of polling, interactions with application behavior, etc.) which are discussed further in Section 8. Our scheme avoids interrupts and polling all-together.

This paper proceeds to deal with this problem by enhancing the communication layer with general purpose operations and by taking advantage of these operations in the SVM protocol. The insight behind the communication-layer mechanisms and the synchronous home-based lazy release consistency (*GeNIMA*) protocol we present in this paper is to decouple protocol processing from message handling. By providing support in the network interface for simple, generic operations, asynchronous message handling (e.g. data movement and synchronization) can be performed entirely in the network interface without involving the host processor or doing any protocol processing at the time of asynchronous message handling. The protocol is then restructured to perform protocol processing at synchronous points when sending messages rather than when receiving asynchronous requests. This eliminates the need for interrupting the receiving host processor or using polling to handle asynchronous events.

While we use a programmable Myrinet network interface for prototyping, the message-handling mechanisms are simple and do not require programmability. We find that the proposed protocol extensions improve performance substantially for our suite of ten applications. Performance improves by about 38% on average for reasonably well performing applications (and up to 120% for applications that do not perform very well under SVM even afterward).

The rest of the paper is organized as follows. Section 2 presents the overall cluster architecture and the base SVM protocol. Section 3 briefly presents the main characteristics of the applications we use in this work. Section 4 presents our results for the dependence of end application performance on communication layer parameters. Section 5 presents the proposed NI mechanisms, the protocol changes, and the performance results. Section 6 discusses the remaining bottlenecks in the systems, and Section 7 gives a more detailed view of where time is now being spent in the communication layer. Section 8 presents related work and Section 9 presents concluding remarks.

2. CLUSTER ARCHITECTURE

The architecture we are considering is a cluster composed of commodity components. The nodes of the system are small–scale symmetric multiprocessor (SMP) systems. Each node is a “standard” workstation or PC, with a memory bus, a number of memory modules, and a number of processors.

The interconnection network is a low-latency, high-bandwidth system area network (SAN) that plugs into each node at the I/O bus level. The network interfaces do not provide any support for hardware cache coherence. However, for the architectures we consider, the network interface can perform different types of data movement or synchronization functions. Figure 3 presents the general architecture of each node in the system.

When a message is exchanged between two hosts, it is put in a post queue on the network interface at the sending side. In an asynchronous send operation, which we assume, the sender is free to continue with useful work. The network

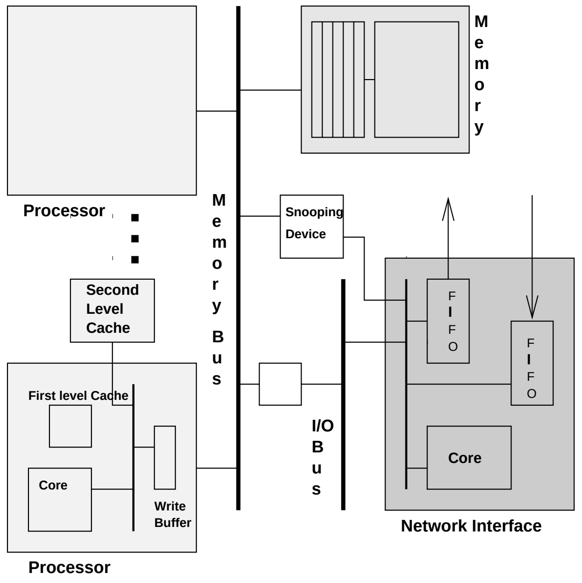

Fig. 3. System node architecture.

interface processes the request, prepares packets, and queues them in an outgoing network queue, incurring an occupancy per packet. After transmission, each packet enters an incoming network queue at the receiver, where it is processed by the network interface and then deposited directly in host memory without causing an interrupt [11; 14]. Interrupts are caused explicitly by messages that require protocol action (e.g. page requests). Thus, the interrupt cost is an overhead related not so much to data transfer but to processing requests.

The shared address space abstraction is provided to the applications in software by the SVM protocol. The protocol we use is a home–based protocol that supports SMP nodes [42]. The protocol uses traditional software diffs (HLRC) to propagate updates to the home node of each page at a release point. The necessary pages are invalidated only at acquire points according to lazy release consistency (LRC). At a subsequent page fault, the whole page is fetched from the home, where it is guaranteed to be up to date according to the lazy release consistency [25]. The protocol for SMP nodes attempts to utilize the hardware sharing and synchronization within an SMP as much as possible, reducing software involvement [5; 42]. The optimizations used include the use of hierarchical barriers and the avoidance of interrupts as much as possible. Interrupts are used only when remote requests for pages and locks arrive at a node. Requests are synchronous (RPC like), to avoid interrupts when replies arrive at the requesting node. Barriers are implemented with synchronous messages and no interrupts. Interrupts are delivered to processor 0 in each node. However, they are handled by a floating process that is scheduled independently by the operating system on any processor in the node.

3. APPLICATIONS

We use 10 applications from the SPLASH-2 [52] application suite (including different versions of the applications). A detailed classification and description of the application behavior for SVM systems with uniprocessor nodes is provided in [27]. The applications can be divided in two groups, regular and irregular.

3.1 Regular Applications

The applications in this category are FFT [2; 52], LU [52] and Ocean [13; 48; 28]. Their common characteristic is that they are optimized to be single-writer applications; i.e. a given word of data is written only by the processor to which it is assigned. Given appropriate data structures, they are single-writer at page granularity as well, and pages can be allocated among nodes such that writes to shared data are almost all local. The applications have different inherent and induced communication patterns [52; 27], which affect their performance and the impact on SMP nodes.

3.2 Irregular Applications

The irregular applications in our suite are Barnes [3; 21; 46; 28], Radix [10; 23], Raytrace [47; 52], Volrend [39; 52; 28] and Water [52]. In this work we use both original versions of several SPLASH-2 applications [27] as well as versions that have been restructured to improve performance on SVM systems [28]. The same restructurings are found to be very important on large-scale hardware-coherent machines [29]. Thus, although they are often substantial and algorithmic, they are not specific to SVM. FFT, LU-contiguous, Ocean-contiguous, Radix-original, Barnes-original, Water-nsquared, and Water-spatial are the original SPLASH-2 applications. These versions of the applications are already optimized to use good partitioning schemes and data structures, both major and minor, for both hardware coherence and release consistent SVM [23]. Barnes-spatial, Ocean-rowwise, Radix-local, Volrend, and Raytrace are the restructured applications from [28]. With a drastic algorithmic change for one phase of Barnes-Hut, Barnes-spatial substantially reduces the amount of lock synchronization. The restructurings for the others are less intrusive and try to improve data assignment, make remote accesses less scattered, or eliminate unnecessary synchronization. The version of Raytrace we use eliminates a lock that assigns unique ids to rays, resulting in less locking. The restructured version of Volrend, Volrend-stealing, uses task stealing, but employs a different initial partition than the SPLASH-2 version.

4. EFFECTS OF COMMUNICATION PARAMETERS

We focus on the following performance parameters of the communication architecture: host overhead, I/O bus bandwidth, network interface occupancy, and interrupt cost. We do not examine network link latency, since it is a small and usually constant part of the end-to-end latency in system area networks (SAN). These parameters describe the basic features of the communication subsystem. The rest of the parameters in the system, for example cache and memory configuration, number of processors, etc. remain constant.

While we examine a range of values for each parameter, in varying a parameter we usually keep the others fixed at the set of *achievable* values. These are the values we might consider achievable currently, on systems that provide optimized operating system support for interrupts. These values represent the actual system we use in Section 5. Interrupt cost, however, is higher in the actual system in the actual system and corresponds to the region around 2500 cycles. The fixed values we use when varying parameters are relatively aggressive, so that the effects of the parameter being varied are observed. In more detail:

Host Overhead is the time the host processor itself is busy sending a message. The range of this parameter is from a few cycles to post a send in systems that support asynchronous sends, up to the time needed to transfer the message data from the host memory to the network interface when synchronous sends are used.

The *I/O Bus Bandwidth* determines the host to network bandwidth (relative to processor speed). In contemporary systems this is the limiting hardware component for the available node-to-network bandwidth; network links and memory buses tend to be much faster.

Network Interface Occupancy is the time spent on the network interface preparing each packet. Packets have variable sizes with the maximum size being equal to the page size (4KBytes). Network interfaces employ either custom state machines or network processors (general purpose or custom designs) to perform this processing. Thus, processing costs on the network interface vary widely.

Interrupt Cost is the cost to issue an interrupt between two processors in the same SMP node, or the cost to interrupt a processor from the network interface. It includes the cost of context switches and operating system processing. Although the interrupt cost is not a parameter of the communication subsystem, it is an important aspect of SVM systems. Interrupt cost depends on the operating system used; it can vary greatly from system to system, affecting the performance portability of SVM across different platforms. We therefore vary the interrupt cost from free interrupts to 50000 processor cycles for both issuing and delivering an interrupt. The achievable value we use is 500 processor cycles, which results in a cost of 1000 cycles for a null interrupt. This choice is significantly more aggressive (about a factor of 4) than what current operating systems provide. However it is achievable with fast interrupt technology [51]. We use it as the achievable value when varying other parameters to ensure that interrupt cost does not swamp out the effects of varying those parameters. Table 1 summarizes the achievable values of each parameter.

Parameter	Range	Achievable	Best
Host Overhead (cycles)	0-10000	600	~0
I/O Bus b/w (Mbytes/MHz)	0.25-2	0.5	2
NI Occupancy (cycles)	0-10000	1000	200
Interrupt Cost(cycles)	0-50000	500	~0

Table 1. Ranges and achievable and best values of the communication parameters under consideration.

4.1 Simulation Testbed

We examine the dependence of system performance on communication parameters using detailed architectural simulation. The simulation environment we use is built on top of augmint [45], an execution driven simulator using the *x86* instruction set, and runs on *x86* systems. The simulation environment and the architectural parameters instantiate a system that follows the general architecture described in Section 2. In particular the simulator tries to model (roughly) a cluster with 4-way PentiumPro (at 200 MHz) nodes interconnected with a Myrinet network (M2M-PCI32C).

The simulated architecture (Figure 3) assumes a cluster of c–processor SMPs connected with a commodity interconnect like Myrinet [12]. Contention is modeled at all levels except in the network links and switches themselves. The processor has a P6-like instruction set, and is assumed to be a 1 IPC processor. The data cache hierarchy consists of a 8 KBytes first-level direct mapped write-through cache and a 512 KBytes second-level two-way set associative cache, each with a line size of 32 Bytes. The write buffer has 26 entries [49], 1 cache line wide each, and a retire-at-4 policy. Write buffer stalls are simulated. The read hit cost is one cycle if satisfied in the write buffer and first level cache, and 10 cycles if satisfied in the second-level cache. The memory subsystem is fully pipelined.

Each network interface (NI) has two 1 MByte memory queues, to hold incoming and outgoing packets. The size of the queues is such that they do not constitute a bottleneck in the communication subsystem. If the network queues fill, the NI interrupts the main processor and delays it to allow queues to drain. Network links operate at processor speed and are 16 bits wide. We assume a fast messaging system [16; 40; 14] as the basic communication library.

The memory bus is split-transaction, 64 bits wide, with a clock cycle four times slower than the processor clock. Arbitration takes one bus cycle, and the priorities are, in decreasing order: outgoing network path of the NI, second level cache, write buffer, memory, incoming path of the NI. The I/O bus is 32 bits wide. The *relative* bus bandwidth and processor speed match those on modern systems. If we assume that the processor has a 200 MHz clock, then the memory bus is 400 MBytes/s.

Protocol handlers themselves cost a variable number of cycles. While the code for the protocol handlers can not be simulated since the simulator itself is not multi-threaded, we use for each handler an estimate of the cost of its code sequence. The cost to access the TLB from a handler running in the kernel is 50 processor cycles. The cost of creating and applying a diff is 10 cycles for every word that needs to be compared and 10 additional cycles for each word actually included in the diff. Computing diffs accounts for cache pollution as well.

The protocol we simulate is close to the base protocol described in 2. With SMP nodes there are many options for how interrupts may be handled within a node. Our protocol uses one particular method: Always deliver to processor 0 and perform protocol processing in the same processor. We also experimented with round robin interrupt delivery and the results look similar to the case where all interrupts are delivered to a fixed processor in each SMP. Overall performance seems to increase slightly, compared to the static interrupt scheme, but as in the static scheme it degrades quickly as interrupt cost increases. Moreover, implementing such a scheme in a real system may be complicated and may incur additional costs.

Application	Page Faults		Page Fetches		Lcl Lock Acq		Rmt Lock Acq		Barriers
	1	*4*	1	*4*	1	*4*	1	*4*	1,*4*
FFT (20)	397.12	*251.89*	393.31	*167.17*	0.00	*0.00*	0.00	*0.00*	*1.14*
LU-contiguous (512)	81.36	*56.61*	71.78	*34.94*	0.02	*0.22*	0.27	*0.07*	*19.24*
Ocean-contiguous (514)	647.61	*117.34*	646.97	*24.92*	0.00	*0.76*	2.17	*1.41*	*13.05*
Water-nsquared (512)	69.19	*22.06*	68.26	*19.01*	0.01	*120.36*	203.20	*82.85*	*3.30*
Water-spatial (512)	97.86	*21.42*	93.81	*17.73*	0.01	*1.83*	3.94	*2.16*	*4.19*
Radix (1K)	208.82	*82.73*	203.69	*44.92*	0.10	*0.44*	4.52	*4.11*	*1.04*
Volrend (head)	105.09	*44.06*	104.78	*29.35*	0.00	*29.34*	44.34	*17.64*	*1.61*
Raytrace (car)	89.80	*25.64*	89.79	*25.57*	0.03	*2.21*	4.89	*3.26*	*0.10*
Barnes-rebuild (8K)	211.22	*103.02*	207.72	*90.90*	0.07	*33.92*	127.74	*93.81*	*1.44*
Barnes-spatial (8K)	48.06	*10.43*	46.20	*9.92*	0.00	*0.16*	0.24	*0.07*	*1.79*

Table 2. Number of page faults, page fetches, local and remote lock acquires and barriers per processor per 10^7 cycles for each application for 1 and 4 processors per node.

4.2 Results

Table 2 can be used to characterize the applications. It presents counts of protocol events for each application, for 1, 4 and 8 processors per node (16 processors total in all cases).

Table 3 presents the maximum slowdown obtained for each application by varying each of the parameters under consideration across its range of values. The maximum slowdown is computed from the speedups for the smallest and biggest values considered for each parameter, keeping all other parameters at their achievable values. Negative numbers indicate speedups.

Application	Host Overhead	NI Occupancy	I/O Bus Bandwidth	Interrupt Cost
FFT	22.6%	11.9%	40.8%	86.6%
LU-contiguous	17.9%	7.5%	15.9%	70.8%
Ocean-contiguous	4.5%	2.8%	6.5%	35.2%
Water-nsquared	32.4%	16.6%	10.8%	83.2%
Water-spatial	23.7%	8.5%	8.9%	67.9%
Radix-original	35.8%	-31.8%	77.6%	58.7%
Volrend-stealing	34.7%	12.8%	15.7%	91.3%
Raytrace	8.2%	2.9%	8.9%	52.3%
Barnes-original	40.7%	21.8%	44.8%	80.3%
Barnes-spatial	4.4%	-0.6%	27.5%	59.0%

Table 3. Maximum Slowdowns with respect to the various communication parameters for the range of values with which we experiment. Negative numbers indicate speedups.

4.2.1 *Host Overhead.* Table 3 shows that the slowdown due to the host overhead is generally low, especially for realistic values of asynchronous message overheads. Across the entire range of values, it varies among applications from less than 10% for Barnes-spatial, Ocean-contiguous, and Raytrace to more than 35% for Volrend-stealing, Radix, and Barnes-original. As expected, applications that send more messages exhibit a higher dependency on the host overhead. Note that with asynchronous messages, host overheads will be on the low side of our range, so we can conclude that host overhead for sending messages is not a major performance factor for coarse grain SVM systems and is unlikely to become so in the near future.

4.2.2 *Network Interface Occupancy.* Table 3 shows that network interface occupancy has even a smaller effect than host overhead on performance, for realistic occupancies. Most applications are insensitive to it, with the exception of a couple of applications that send a large number of messages. For these applications, slowdowns of up to 22% are observed at the highest occupancy values we explore. The small increase in speedup observed for Radix is caused by timing issues (contention is the bottleneck in Radix). Generally, the relatively large message (and thus packet) size makes the system insensitive to network interface occupancy.

4.2.3 *I/O Bus Bandwidth.* Table 3 shows the effect of I/O bandwidth on application performance. Reducing the bandwidth across the entire range results in slowdowns of up to 82%, with 4 out of 11 applications exhibiting slowdowns of more than 40%. However, many other applications are not so dependent on bandwidth, and only FFT, Radix, and Barnes-original benefit much from increasing the I/O bus bandwidth beyond the achievable relationships to processor speed today. This does not mean that it is not important to worry about improving bandwidth; as processor speed increases, if bandwidth trends do not keep up, we will find ourselves at the relationship reflected by the lower bandwidth case we examine (or even worse). Our results show that if bandwidth keeps up with processor speed, it is not likely to be the major performance limitation in SVM systems.

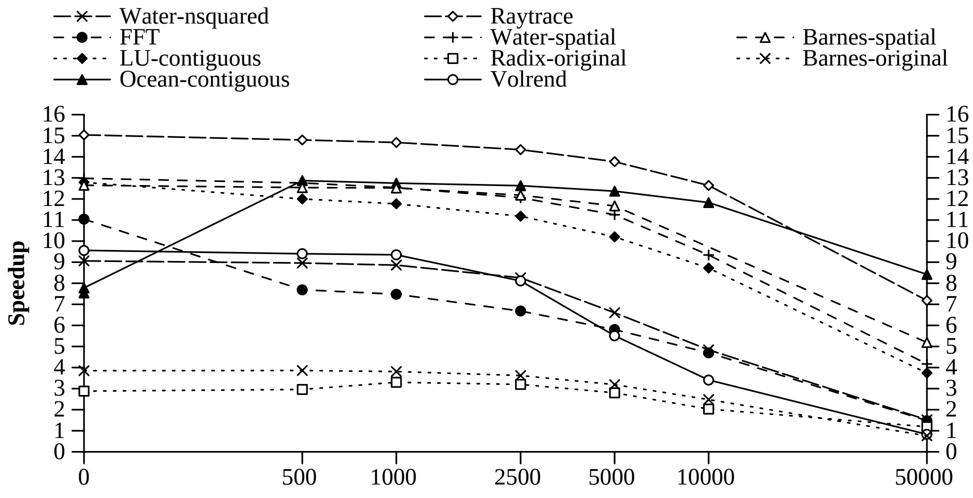

Fig. 4. Effects of interrupt cost on application performance. The six bars for each application correspond to an interrupt cost of 0, 500, 1000, 2500, 5000, 10000, and 50000 processor cycles.

4.2.4 *Interrupt Cost.* Figure 4 shows that interrupt cost is a very important parameter in the system. Unlike bandwidth, it affects the performance of *all* applications dramatically, and in many cases a relatively small increase in interrupt cost leads to a big performance degradation. For most applications, interrupt costs of up to about 2000 processor cycles for each of initiation and delivery do not seem to hurt much. However, commercial systems typically have much higher interrupt costs. Increasing the interrupt cost beyond the 2000-cycle level begins to hurt sharply. All applications have a slowdown of more than 50% when the interrupt cost varies from 0 to 50000 processor cycles (except Ocean-contiguous, where NI queue overflows at 0 interrupt cost result in low speedup).

5. NETWORK INTERFACE AND SVM PROTOCOL EXTENSIONS

The previous section has shown that interrupt cost is the dominant problem in SVM systems. We now describe a minimal set of extensions to the network interface that can be used to remove the need for asynchronous protocol processing from the Base (HLRC) protocol we used in the previous section [5; 42]. We discuss how the resulting message and protocol handling differs from that of the Base protocol. These extensions are prototyped on a real system that consists of 4-way Intel SMPs connected with a system area network.

5.1 Protocol Extensions

In the Base protocol, each incoming message that requires protocol processing causes an interrupt that schedules the protocol process on one of the processors. The incoming requests are (i) page fetches, (ii) lock acquisition, and (iii) diff application at the home. Other requests that require interrupting host processors in some protocols include page home allocation and migration requests. These however, are infrequent and not so critical for common-case system performance. Figure 5 presents an example with both the Base protocol and the final version of *GeNIMA*.

5.1.1 *Remote Deposit.* The communication layer we use (VMMC) already allows for data explicitly transferred to a remote node via a send message to be deposited in specified destination virtual addresses in main memory without involving a remote host processor. This is different from transparently updating remote copies of data structures via memory bus snooping, code instrumentation, or specialized NI support [11; 19; 35; 25]. In our implementation, non-contiguous pieces of data are sent directly to remote data structures with separate messages, and are not packed into bigger messages or combined by scatter-gather support. Many communication systems support this or similar type of operations [11; 19; 31; 24; 15].

In the Base protocol remote data structures are updated by sending updates for non-consecutive fields in a single message. This reduces the number of messages exchanged and results in larger messages. The disadvantage of this approach is that processing is required both at the sending and the receiving sides to pack and unpack the data.

We use the remote deposit mechanism in all our subsequent protocols to exchange small pieces of control information during barrier synchronization and to directly update remote protocol data structures (e.g. page timestamps, barrier control information). In addition, there are two major cases where this operation is used: to propagate coherence information and to remotely apply diffs to application data (*direct diffs*).

(i) First, we use it to propagate *coherence information* in a sender-initiated way rather than in response to incoming messages, without causing interrupts. In traditional interrupt-based lazy protocols coherence information (page invalidations) is transferred as part of lock transfers. When the last owner of a lock hands the lock

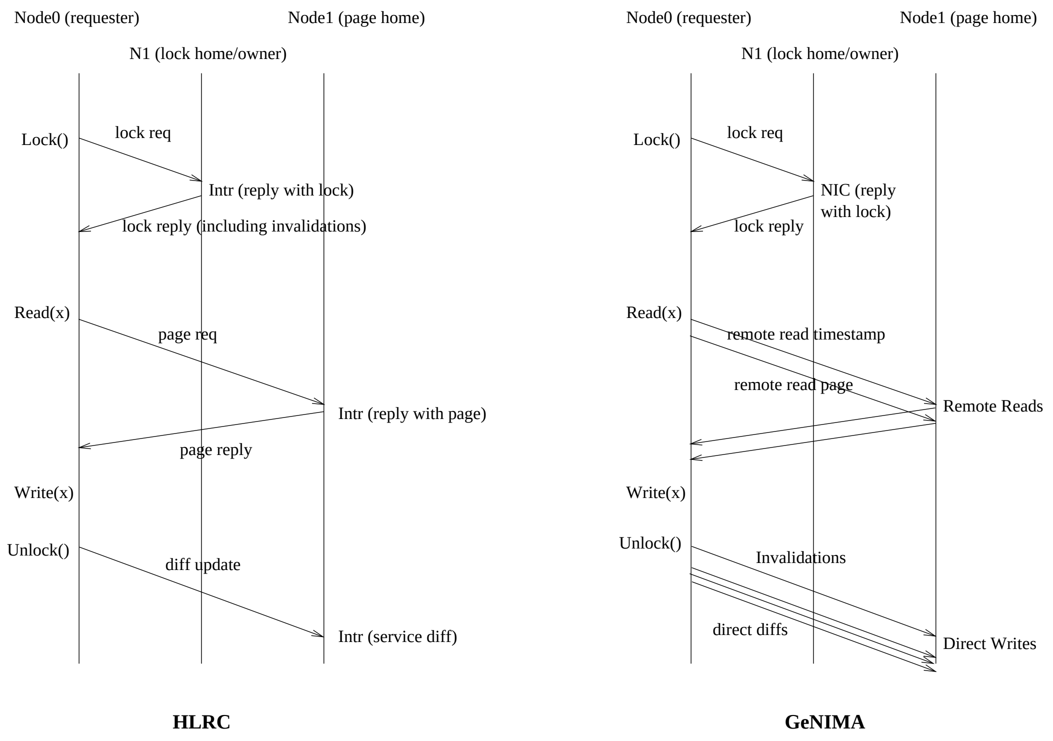

Fig. 5. Protocol example for both the base (HLRC) and final (*GeNIMA*) protocols. On the left, HLRC uses interrupts for asynchronous message handling. On the right, *GeNIMA* uses general network interface support to eliminate interrupts.

to another process, it also sends the page invalidations that the requester needs to maintain the release-consistent view of the shared data. Thus, in the base protocol, when the protocol handler asynchronously services a remote acquire, it sends to the requester both the lock (mutual exclusion part) and the page invalidations (coherence information).

The mutual exclusion and the coherence information parts can be separated. We will see further motivation for this when we examine servicing lock acquire messages without interrupts, so the host processor at the last owner does not even know about the lock acquire. In *GeNIMA* we *propagate* coherence information eagerly to all nodes at a release, using remote deposit directly into the remote protocol data structures. Invalidations are still *applied* to pages at the next acquire, preserving LRC.

(ii) Second, we use the asynchronous send mechanism with remote deposit to remotely apply diffs and hence update shared *application data* pages at the home nodes. In the Base protocol diffs are propagated to the home lazily, at the next incoming acquire of a lock. Diffs for the same page are packed in a single message and then sent to the home, where they interrupt the processor and are applied to the page by the protocol handler. In *GeNIMA* when the local processor computes a diff, instead of storing it in a local data structure and then sending this diff data structure to the home of the page, it directly sends each contiguous run of different words to the home as it compares the page with its twin. We call this method of computing and applying the differences in shared data *direct diffs*.

Direct diffs save the cost of packing the diff, interrupting a processor at the home of the page and having it unpack and apply the diff on the receive side. However, they may substantially increase the number of messages that are sent, since they introduce one message per contiguous run of modifications within a page rather than one message per page (or multiple pages) that has been modified.

Since synchronization points do not involve interrupts any more (as we shall see shortly), diffs must now be computed at release points rather than incoming acquires. However, if another processor in the same node as the releaser is waiting to acquire the lock next, then no diffs need to be computed. Thus, diff computation is done with a hybrid method that is eager for synchronization transfers across nodes and lazy for transfers within nodes.

In all cases, the remote deposit (send) messages used are asynchronous. Thus, blocking of the sending processor is avoided, except when the post queue between the processors and the network interface is full and must be drained before new requests are posted.

5.1.2 *Remote fetch.* Unlike remote deposit, remote fetch is not provided in the base VMMC. We extend the communication system to support (in NI firmware) a remote fetch operation to fetch data from arbitrary exported remote locations in virtual memory to arbitrary addresses in local virtual memory. Again, the remote fetches must be for contiguous data in our implementation, but there is little occasion for non-contiguous fetches in the protocol.

In the Base protocol page fetches are performed as follows. The requester sends a message to the home node and invokes a protocol handler via an interrupt. The home node performs the necessary protocol processing (i.e. timestamp manipulation) and eventually returns the page to the requester by using the remote deposit mechanism of VMMC.

We use the remote fetch operation to avoid interrupts at page fetches. When a remote page is needed, the local processor first requests the timestamp of the remote page and then immediately requests the page itself. The request messages are asynchronous, so the request for the page is sent before the timestamp arrives. If the timestamp is determined to be incorrect, i.e. the necessary diffs have not been applied at the home, the requester retries.

Another important advantage of the remote fetch operation is that it enhances protocol scalability significantly. When remote deposit is used to transfer pages in VMMC in response to a request, each home node needs to be able to send its pages to every node in the system. With memory mapped communication layers, this requires that each node, as a potential requester, export (and pin) all the shared pages in the entire application, limiting the amount of shared memory. With the remote fetch operation, the requester itself fetches updated page versions from the home, so the requester rather than the home needs to “directly” access remote pages. Thus, each node needs to export (and pin) only those shared pages for which it is the home. Other uses of a remote fetch operation are possible as well, e.g. for fetching protocol data as mentioned earlier.

Remote fetch can also be used instead of remote deposit to avoid interrupts for coherence information propagation: a processor can “pull” the necessary write notices with a point-to-point remote fetch operation at lock acquires rather than pushing it to all nodes at a release. The total traffic is usually about the same in both cases since invalidations for all intervals generally do need to be sent to all nodes in the system at some point, and propagating this information at the releases or at the acquires does not change the number of intervals. However, using remote fetch can reduce the number of messages: If multiple intervals have to be communicated from a releaser to an acquirer, this will be done via a single fetch of all intervals at the acquire, rather than a broadcast operation at each release. Thus, the eager approach increases the remote release cost while the lazy approach increases the remote acquire cost. We choose the former method rather than remote fetch for this purpose, since it spreads out the traffic over a longer period of time throughout the execution of the application. Thus, while the protocol is still lazy in applying invalidations, the coherence information is propagated eagerly.

As mentioned earlier we use remote deposit to implement direct diffs. Interestingly, direct diffs require that pages be fetched with remote fetches and retries rather than by involving the home processor. The reason is that since direct diffs do not interrupt or involve a host processor at the home at diff application, home processors do not know when they have the updated version of a page and are ready to service queued page requests. Remote fetches do not rely on home processors having this knowledge, since the requester retries whenever it fails to fetch the right version of a page. This is why we will present results for direct diffs only after presenting those for remote fetch.

5.1.3 *Network interface locks.* With coherence information propagation already separated from mutual exclusion as described in the discussion of remote deposit, mutual exclusion does not need to be tied to protocol processing. We extend the communication layer to provide support for mutual exclusion in the NI as well. Lock acquisition and release for mutual exclusion per se become communication system rather than SVM protocol operations, and no host processors other than the requester are involved.

In the Base protocol, lock synchronization is implemented as follows: Every lock is statically assigned a home. When a process needs to acquire a lock, it sends a message to the home of the lock. The home forwards the message to the last owner and the owner releases the lock to the requester. The requests at the home and at the last owner are both handled using interrupts, and typically involve protocol activity such as preparing and propagating coherence information as well. The host processor at the home of each lock is in charge of maintaining a distributed linked list of nodes waiting for the lock. The last owner keeps the lock until another processor needs to acquire it.

The implementation of locks in the network interface firmware is similar to the algorithm used in LRC and HLRC. However, no coherence information is involved and the distributed lists for locks are maintained in the network interface processors, without host processor involvement or interrupts. Coherence propagation is decoupled and managed at synchronization points as described earlier. Associated with each lock is one timestamp, which must be interpreted and managed by the protocol. The network interface does not need to perform any interpretation or operations on this timestamp, but the current implementation requires that this piece of information be stored as part of the lock data structure in the network interface and transferred by the NIs among nodes along with the lock.

On the protocol side, each process knows what invalidations it needs to apply at acquires by looking at protocol timestamps that are exchanged with the locks. Flags are used to ensure that invalidations for each interval have reached the node before they are applied. The only requirement in the communication layer is in-order delivery of messages between two processes. There are no requirements for global or other strict forms of ordering.

An alternative to our moving all the functionality for mutual exclusion into the communication layer, including a lock algorithm, is to have the communication layer or NI simply provide remote atomic operations and to build the locking algorithm into the protocol layer while still avoiding interrupts. This makes the NI support simpler, and hence more likely to be implemented in hardware in commodity NIs. It also allows flexibility in the locking algorithm chosen at the protocol level. The performance tradeoffs between the two approaches are unclear, and more investigation is necessary.

VMMC Operation	Cost
1-word send (one-way latency)	14μs
1-word fetch (round-trip latency)	31μs
1-page send (one-way latency)	46μs
1-page fetch (round-trip latency)	105μs
Maximum ping-pong bandwidth	96MBytes/s
Maximum fetch bandwidth	95MBytes/s
Notification	42μs
Remote lock acquire	53.8μs
Local lock acquire	12.7μs
Remote lock release	7.4μs
Host overhead for asynchronous send/fetch	2-3μs

Table 4. Basic VMMC costs. All send and fetch operations are assumed to be synchronous, unless explicitly stated otherwise. These costs do not include contention in any part of the system.

5.2 Experimental Testbed

We implement the network interface extensions on a cluster of Intel SMPs connected with Myrinet. This system follows the general architecture described in Section 2. The nodes in the system are 4-way Pentium Pro SMPs running at 200 MHz. The Pentium Pro processor has 8 KBytes of data and 8 KBytes of instruction L1 caches. The processors are equipped with a 512 KBytes unified 4-way set associative L2 cache and 256 MBytes of main memory per node.

The operating system is Linux-2.0.24. The only operating system call used in the protocol (after the initialization phase) is *mprotect*. The cost of *mprotect* for a single page is about 10-15 μs; coalescing *mprotect* calls for consecutive pages reduces this cost. We use this technique in our protocol when multiple consecutive pages need to be *mprotect*ed[1].

Myrinet [12] is a high-speed system-area network, composed of point-to-point links that connect hosts and switches. Each network interface in our system has a 33 MHz programmable processor and connects the node to the network with two unidirectional links of 160 MBytes/s peak bandwidth each. Actual node-to-network bandwidth is constrained by the 133 MBytes/s PCI bus. All nodes in the system are connected directly to an 8-way crossbar switch.

The communication layer we use in this system is Virtual Memory Mapped Communication (VMMC) for the Myrinet network [14]. VMMC provides protected, reliable, low-latency, high-bandwidth user-level communication. VMMC includes a performance monitor in firmware that allows us to look in detail at the activities in the communication layer at runtime.

Since we loosely contrast our results with an aggressive hardware cc-NUMA system, an SGI Origin2000 we mention here the base system characteristics: We use an SGI Origin 2000 [36], containing sixty-four 200MHz R10000 processors. The 64 processors are distributed in 32 nodes, each with 512MBytes of main memory, for a total of 16GBytes of system memory. The nodes are assembled in a full hypercube topology with fixed-path routing. Each processor has separate 32KByte, instruction and data caches and a 4MByte unified 2-way set associative second-level cache. The main memory is organized into pages of 16KBytes. The memory buses and the interconnection network support a peak bandwidth of 780MBytes/s for both local and remote memory accesses. The minimum (uncontented) latency for accessing remote memory (in clean state) is about 650ns.

For the SPLASH-2 applications we use to evaluate our extensions we choose problem sizes that are close to the sizes of real-world problems. Table 5 presents the problem sizes along with the uniprocessor execution times. Speedups are computed between the sequential program version (without linking to the SVM library or introducing any other overheads) and the parallel version. The initialization and cold-start phases are excluded from both the sequential and the parallel execution times in accordance with SPLASH-2 guidelines.

5.3 Results

We evaluate four different protocols. Each protocol successively and cumulatively eliminates the use of interrupts in some aspect of the base protocol. The first protocol (DW or direct write) uses the direct deposit mechanism to directly update (write) remote protocol data structures only. The second protocol (RF or remote fetch) extends DW to also use the remote fetch mechanism to fetch pages and their timestamps. The third protocol (DD or direct diff) extends RF to also use the remote deposit mechanism for direct diffs. (We present these results in this order, rather than presenting both DD and DW that use remote deposit first, since direct diffs depend on remote fetch). Finally, the fourth protocol (*GeNIMA*) uses all previous features as well as network interface support for mutual exclusion, eliminating all interrupts or asynchronous protocol processing.

[1]Measuring these costs precisely is difficult since it requires taking into account many other factors as well, e.g. page and cache invalidations, etc.

Application	Problem Size	Uni(sec)	Overall(%)	Data(%)	Lock(%)
FFT	4M points	4.6	52.50	45.37 (44.92)	0.00
LU-contiguous	4096x4096 matrix	935.9	4.63	13.46 (11.20)	0.00
Ocean-rowwise	514x514 ocean	248.3	18.40	21.76 (19.26)	9.21
Water-nsquared	4096 molecules	360.6	21.00	15.26 (46.17)	62.76
Water-spatial	15625 molecules	157.2	6.80	41.60 (41.80)	9.69
Radix-local	4M keys	5.9	90.79	26.76 (27.00)	53.21
Volrend-stealing	256x256x256 cst head	13.2	45.30	43.81 (42.44)	50.44
Raytrace	256x256 car	29.8	39.89	2.52 (50.03)	59.01
Barnes-original	32K particles	47.7	117.65	41.07 (68.25)	1.98
Barnes-spatial	128K particles	219.2	-20.16	40.99 (37.70)	33.84

Table 5. Application statistics. The fourth column represents the overall percentage improvement in each application between the Base protocol and *GeNIMA*. The fifth column is the percentage improvement in data wait time between DW and DW+RF and the sixth column, the percentage improvement for lock time between DW+RF+DD and *GeNIMA*. For remote fetch we also report the percentage improvement between DW and *GeNIMA* in parentheses.

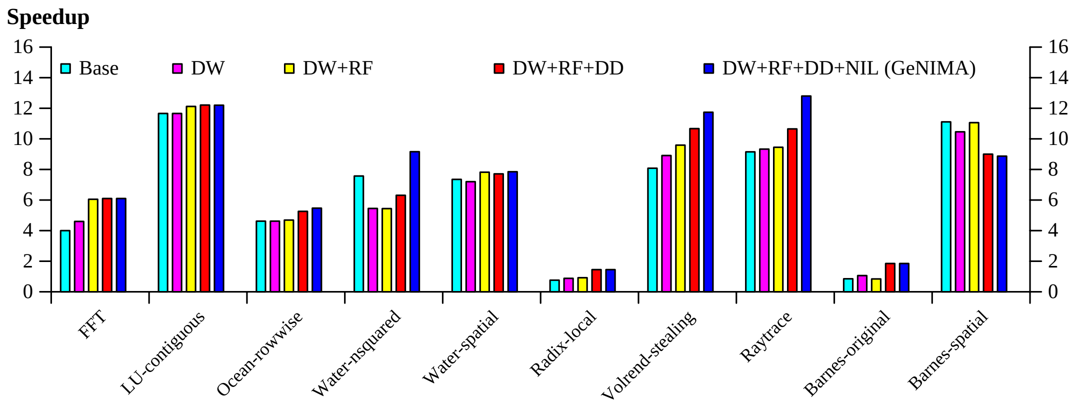

Fig. 6. Application speedups. From left to right the bars for each application are (i) Base, (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

Figures 6 and 7 show the speedups and the average execution time breakdowns respectively, for each protocol. Breakdowns are averaged over all processors. The major components of the execution time we use are: *Compute time* is the useful work done by the processors in the system. This includes stall time to local memory accesses. *Data wait time* is the time spent on remote memory accesses. *Lock time* is the time spent on lock synchronization (in both the lock and unlock routines). *Acq/Rel time* is the time spent in acquire/release primitives used for release consistency, in cases where mutual exclusion (and thus locks) is not necessary. *Barrier time* is the time spent in barriers. Let us examine the results for each protocol.

5.3.1 *Direct writes to remote protocol data structures (DW).* We see that all applications with the exception of Water-nsquared perform either comparably or better with DW than with the base protocol (Figure 7). This is primarily due to the

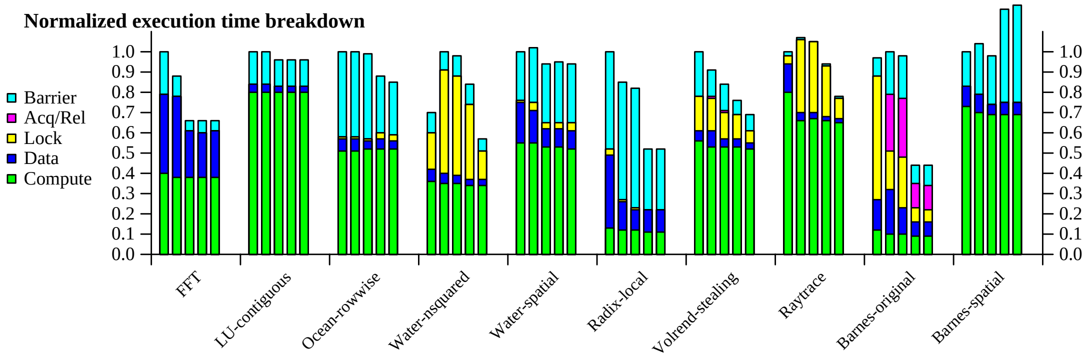

Fig. 7. Normalized average execution time breakdowns for 16 processors. From left to right the bars for each application are (i) Base, (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

removal of message related protocol processing at the sender and the receiver (e.g. copying, packing and unpacking of messages). However, the DW protocol sends more messages than the Base protocol, both because it uses eager propagation and because it uses small messages. This is in fact the reason that Water-nsquared performs worse. This version of Water-nsquared uses fine-grained, per-molecule locks when updating the private forces computed by each process into the shared force array, to reduce inherent serialization at locks. However, this causes the frequency of locks and hence of invalidation propagation to be very large. We find that these messages occupy the queues in the NIs, increasing the time it takes for lock acquire requests to be delivered to the host processor and serviced. We experimented with coarser-grained locks and with staggering lock acquisitions in Water-nsquared. Although the relative costs across nodes changed, overall system performance remained at the same level. However, these techniques require further investigation.

In the final *GeNIMA* protocol (described later in this Section) the effect of this problem is much less apparent and the performance of Water-nsquared improves by 21% compared to the Base protocol. This is because in the final protocol lock acquire messages need not be delivered to host memory but are handled completely in the network interface, so they do not get stuck behind other messages. Thus, the real problem here is not the increased traffic in the network, but the fact that there is one FIFO queue (and one level of priorities) for all messages in the incoming path from the network interface to the host.

As discussed earlier, to reduce the number of messages, invalidations can be "pulled" at acquires with the remote fetch operation rather than pushed with broadcast remote deposit at releases. The cost is increased acquire latency. We experimented with the second approach in a different, WindowsNT version of our system, which has similar performance characteristics, and found no noticeable benefits for *GeNIMA* at the scale of systems we examine here.

5.3.2 *Remote fetches of pages (RF).* Our simple micro-benchmarks show that the uncontended page fetch time is improved with the use of the remote fetch operation from about 200 μs to about 105 μs. Figures 6 and 7 show that all applications benefit from the use of remote fetch even beyond DW, to varying degrees. Especially applications with high data wait times, like FFT, Water-spatial, Radix-local and Barnes-original see a large improvement in performance. The data wait time is reduced up to 45%, and more than 20% for most applications. It is interesting that the 45% improvement in data wait time in FFT comes almost exclusively from the uncontended latency reduction of eliminating the interrupts at the home and the related scheduling effects within an SMP. Using the performance monitor we found that the use of the remote fetch operation does not reduce the contention in the communication layer in this case.

5.3.3 *Remote diff application (DD).* As we see from the execution time breakdowns in Figure 7, direct diffs are particularly useful in the irregular applications that have a lot of synchronization and hence diffs: Radix-local, Barnes-original, Raytrace, Volrend-stealing, and Water-nsquared. The benefits in performance come from eliminating the interrupts (and related scheduling effects in the SMP nodes) as well as from better load balancing of protocol costs, and they come despite the fact the direct diffs use smaller messages than diffs in the Base protocol.

Barnes-spatial performs much worse with DD than without. This is because the number of messages in the network increases by more than a factor of 30 in this case, due to the highly scattered nature of diffs within each page. This problem can perhaps be addressed at the application level by changing the layout of data structures, so that updates to shared data are done more contiguously. At the system level, analysis with the performance monitor shows that the increased number of diff messages indeed results in (i) the send request queue in the NI becoming full and thus stalling subsequent messages from the host (both synchronous and asynchronous), and (ii) increased NI occupancy at the send side. The stalls at the send queue increase the effective overhead at the host processor and lead to less overlapping of communication and computation. The increased NI occupancy at the send side does not seem to have a significant effect on performance. These results agree with the simulation results described earlier, that found NI packet processing occupancy not to be a major bottleneck for SVM on a Myrinet like system.

There are different ways to deal with this problem in the system itself: (i) By increasing the size of the post queue, such that most of the time there is enough space for all diff requests. (ii) By adding a scatter-gather operation in the NI. The host processor can use this operation to send in one message all scattered update "runs" of the local copy to the home copy of each shared page. This approach would greatly reduce the number of messages and the contention at the post queue, but would increase the NI occupancy at both the sending and receiving sides. The NI is assumed to be relatively slow compared to the host processor (as it very slow in our system), and a scatter-gather operation would require additional processing in the NI to pack data from different virtual locations to the same message on the send side and to unpack them on the receive side. It would also require fast fine-grained access to local memory from the NI, which we do not have due to the interposition of an I/O bus. For these reasons, and because we are examining a minimal set of extensions needed to avoid interrupts or polling, we do not use scatter-gather. (iii) By increasing the pipelining between successive messages in the outgoing path of the NI. Then messages can be picked from the post queue as previous messages are sent so the queue is drained faster. We have experimented with the third approach in the WindowsNT version of our system and we have found that indeed this greatly reduces contention in the post queue (the resulting speedup for Barnes-spatial increases from 8.87 to 12.21).

5.3.4 *Network interface locks (NIL).* This version includes all NI extensions and is the final version of the protocol (*GeNIMA*). Our simple micro-benchmarks show that the uncontended time for lock acquisition is reduced from about 220 μs (in the Base protocol) to about 54 μs. Compared to the previous version, *GeNIMA* substantially improves the performance of applications that use locks frequently. Table 5 shows that lock time is reduced up to about 60%. These improvements come from the elimination of interrupts, and also from the fact that lock messages do not need to be delivered to host memory. As discussed earlier, the latter results in shorter service times for lock messages since they need not wait for other messages to be delivered to the host first. Interestingly, Barnes-original does not benefit from network interface support for locks despite its very high lock time. In particular, we find that lock acquire time is very imbalanced across different processors, and that the largest component of lock acquire time comes from contention in either the SVM protocol or the communication system. This issue bears further investigation. Finally, *GeNIMA* makes task stealing in Volrend more effective than with previous protocols, and it improves the computational load balance too. In previous studies [28], it was found that task stealing is not effective in Volrend because of the high cost of locks as well as the dilation of critical sections.

5.3.5 *Summary.* Overall, we see from Figure 8 that *GeNIMA*, which leverages all our general-purpose NI extensions to eliminate interrupts and asynchronous protocol processing, improves application performance by on average 38% for applications that end up performing quite well and up to 120% for applications that still do not perform very well under SVM. The improvements in individual overhead components of execution time are even larger. Several applications that performed in mediocre ways now perform much better, even well, on a 16-processor system. The only application that performs worse in *GeNIMA* than in the Base protocol is Barnes-spatial, due to the direct diffs problem discussed earlier. Eliminating direct diffs causes Barnes-Spatial to perform better than in the Base protocol too (with a speedup of 12.2). Our results also show that all three mechanisms are important in different applications (i.e. locks for Volrend-stealing, remote fetch for FFT, direct diffs for Raytrace, remote deposit for enabling the decoupling of mutual exclusion from coherence propagation and direct diffs), so all should be supported in the NI if possible.

6. REMAINING BOTTLENECKS

Unfortunately, despite all the improvements in *GeNIMA*, Figure 8 shows that the resulting performance is still not quite where we would like it to be to compete with efficient hardware coherence on several applications. Let us now examine what the most significant remaining bottlenecks are.

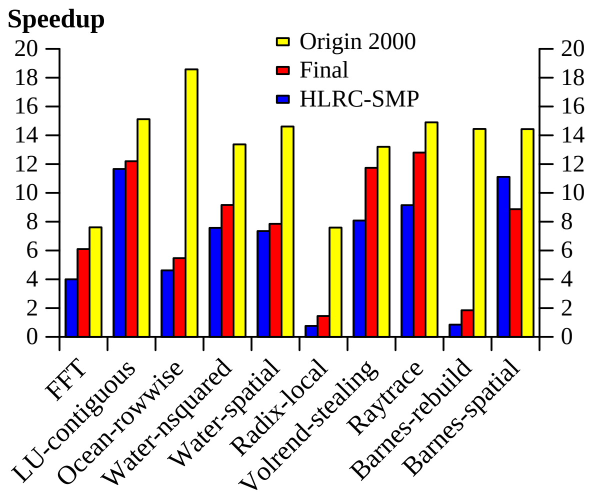

Fig. 8. Application speedups for a hardware DSM machine (Origin-2000), and for the Base and the final *GeNIMA* protocols.

6.1 Data wait time

Figure 7 shows that *GeNIMA* exhibits high data wait time for three applications: Barnes-original, FFT, and Radix-local. Barnes has low inherent bandwidth requirements, but it exhibits scattered accesses to remote addresses at very small granularity and incurs high fragmentation overheads due to the page granularity of SVM. FFT on the other hand, exhibits coarse-grained memory access patterns, but has high inherent bandwidth demands. If we compare the data wait time in FFT under *GeNIMA* to what it would be with uncontended remote fetch operations, the increase is less than 30%. The rest of the data wait time is due to the still-remaining uncontended cost of the remote fetch operation. Thus, FFT would benefit from bandwidth increases in the communication layer. Radix exhibits both these problems to a higher extent, as well as a lot of false write-sharing due to the page granularity.

6.2 Lock synchronization

Previous work has identified locks and their dilation to be a major performance problem for SVM for many applications [26; 28]. The restructured versions of these applications dramatically reduce the number of locks and hence their effect on performance. Moreover, a large part of the additional improvement in lock synchronization overheads comes from the fact that lock messages are handled in the network interface and do need to wait in queues to be delivered to the host processor and handled there by the protocol handlers. In *GeNIMA* the applications that still suffer from high lock synchronization costs are the unrestructured Water-nsquared and Barnes-original. Both exhibit fine grain locking, and despite the dramatic reduction in lock overhead costs due to the NI support, the lock costs as well as the dilation of critical sections remain very high compared to hardware cache coherent systems.

6.3 Barrier synchronization

For the restructured or other applications that are not dominated by locks (except for FFT), the time spent in barriers emerges as the most significant remaining bottleneck. Barrier time can be divided into two parts: the wait time at the barrier and the cost of protocol processing (including page invalidation or *mprotect* cost) and communication. Separating these tells us whether major improvements require improving protocol processing costs at barriers or better load balancing of computation, communication and protocol costs. Table 6 shows the portion of time spent in barriers for each application and the portion of the barrier time that is devoted to protocol processing (the third column). While for LU-contiguous, Water, Volrend-stealing, and Barnes-original both imbalances and protocol costs are significant, for FFT, Radix-local, and Barnes-spatial most of the barrier cost is in fact protocol processing time. Protocol processing time can be reduced mostly by protocol level modifications or by faster communication and *mprotect* support. For example, reducing the amount of laziness in the protocol could cause protocol processing (i.e. propagation and application of invalidations) not to be deferred exclusively to synchronization points (with remote deposit support and low-overhead messaging, it may become feasible to send out invalidation notices when a page changes its protection rather than waiting for the release, more akin to hardware coherence protocols). However, such protocol modifications may increase other costs. In *GeNIMA* we have optimized the amount of overlapping between communication and protocol processing at barriers to reduce waiting time. However, we have not considered NI support for barrier synchronization since the actual communication costs are relatively low.

Application	Barrier	Barrier Protocol	*mprotect*
FFT	7.6%	87%	32.4%
LU-contiguous	13.5%	30%	15.1%
Ocean-rowwise	15.7%	50%	8.6%
Water-nsquared	10.5%	20%	14.1%
Water-spatial	30.5%	37%	23.9%
Radix-local	57.7%	94%	51.9%
Volrend-stealing	11.5%	35%	13.1%
Raytrace	20.6%	20%	15.7%
Barnes-original	22.7%	19%	30.5%
Barnes-spatial	39.0%	82%	19.7%

Table 6. Barrier time. The second column (Barrier) is the portion of the execution time that is spent in barriers. The third column (Barrier Protocol) shows how much of the barrier time is spent for protocol processing. The last column (Barrier *mprotect*) shows the percentage of the total SVM overhead time (including barrier, lock, and data wait time) spent in *mprotect*.

6.4 *mprotect* cost

For most applications, the cost of *mprotect* is an issue primarily to the extent that it contributes to the protocol cost at barrier synchronization; in many applications,

Application	SourceLat	LANaiLat	NetLat	DestLat
Water-nsquared	1.7/10.4	2.2/13.8	1.9/13.2	3.2/5.1
Barnes-original	-/8.6	-/12.6	-/12.4	-/6.0
Volrend-stealing	-/7.2	-/8.8	-/7.8	-/4.6
Raytrace	1.8/7.3	2.4/8.2	2.2/5.3	3.5/2.4
FFT	1.8/2.4	3.1/3.8	3.0/3.2	4.2/5.5
Ocean-rowwise	1.5/-	2.4/-	2.0/-	4.3/-
Water-spatial	1.8/4.6	3.2/6.9	3.4/6.2	4.7/4.6
Radix-local	1.8/4.3	3.5/1.3	3.4/5.3	4.6/7.1
Barnes-spatial	1.9/3.2	3.6/5.5	3.6/4.3	5.2/5.8

Table 7. Ratios of average time to uncontended time for each network or NI stage in the path from the sender to the receiver, for small messages in the Base protocol and *GeNIMA* (reported as Base/*GeNIMA*)

a lot of shared pages need to be invalidated at barriers between major phases of computation. Table 6 shows that in certain cases (e.g. Radix) the cost of *mprotect* is a very large component of the protocol costs. Reducing the cost of *mprotect* is not straightforward, mainly because the operating system needs to be involved. We coalesce *mprotect* system calls to multiple contiguous pages into one call, and have experimented with *mprotect*ing more pages than necessary to further reduce the number of *mprotect* calls (e.g. *mprotect* all pages in contiguous range when more than a certain threshold of them need to be *mprotect*ed), but more basic improvements may be necessary.

6.5 Memory bus contention and cache effects

For two applications, FFT and Ocean, the aggregate "compute time" (which includes stall time on local memory) in the parallel execution increases compared to the execution time of the sequential run, despite the fact that the per-processor working set in the parallel execution is smaller than the uniprocessor working set in the sequential execution. For both FFT and Ocean, the increase is due to contention on the SMP memory bus caused by the misses from the four processors within each SMP node. This problem increases with problem size and with the number of processors used in each node. Whether it is a problem in general depends on whether the application has a lot of capacity misses and on the memory and bus subsystems of the SMP nodes.

7. COMMUNICATION LAYER IMPLICATIONS

Due to many complex interactions in the system, the mechanisms we use often affect other, seemingly unrelated, aspects of performance. The approach taken in *GeNIMA* creates a tradeoff: On one hand, the number of messages and the total traffic in the system are both increased compared to the Base protocol (e.g., due to the more eager propagation in *GeNIMA*), potentially leading to increased contention. On the other hand, the traffic is less bursty (not confined to synchronization points) over larger time intervals, and there is more overlapping of communication, computation and message handling due to both the use of asynchronous messages as well as the spreading of traffic throughout program execution. In this section, we use the performance monitoring tool implemented in the NI [37] to examine the impact

Application	SourceLat	LANaiLat	NetLat	DestLat
Water-nsquared	1.1/1.5	1.0/1.3	1.1/1.4	1.1/1.3
Barnes-original	-/2.0	-/1.5	-/2.1	-/1.5
Volrend-stealing	-/1.7	-/1.3	-/1.5	-/1.4
Raytrace	1.1/1.1	1.2/1.2	1.2/1.2	1.4/1.2
FFT	1.2/1.4	1.0/1.0	1.3/1.3	1.2/1.1
Ocean-rowwise	1.2/-	1.2/-	1.2/-	1.4/-
Water-spatial	1.1/1.4	1.2/1.3	1.3/1.4	1.4/1.3
Radix-local	1.3/1.6	1.3/1.3	1.3/1.5	1.5/1.3
Barnes-spatial	1.2/1.6	1.3/1.3	1.3/1.4	1.4/1.3

Table 8. Ratios of average time to uncontended time for each network or NI stage in the path from the sender to the receiver, for large messages in the Base protocol and *GeNIMA* (reported as Base/*GeNIMA*).

of this tradeoff by looking at the network interface activity in detail for both the Base protocol and *GeNIMA*.

Tables 7 and 8 quantify the effect of contention in the Base and *GeNIMA* protocols for small (up to 256 bytes) and large messages respectively (over all types of messages). Each column represents one stage of the path from the sender to the receiver (described in Section 5.2), which can be individually measured by the monitor firmware and software [37]. In all stages, queuing and contention is included in the measurements. Note that in VMMC there is no explicit receive operation; thus, there is no receive stage in the message transfer pipeline that involves the host processor, even in the Base protocol.

In Tables 7 and 8 there are two numbers per stage for each application, separated by a slash: one for the Base protocol and one for *GeNIMA*. Each number is the ratio of the average time spent by a message in the corresponding stage to the time that would have been spent in the same stage in uncontended transfers. Thus, each number is a ratio that shows the average effect of contention incurred in that stage.

Table 8 shows that large messages behave very similarly in the two protocols; contention is very small in the NI in both cases. This may be because large messages are often page fetches, for which the processors usually stall and which therefore afford the least overlap between that message and other activities. For small messages, however, *GeNIMA* greatly increases contention at the NI or network for almost all applications, and for all stages except the last one (data delivery to host memory). Moreover, in the same protocol, applications that were found to perform poorly tend to have higher contention than others. The most apparent cases are Water-nsquared and Barnes-original.

Thus, *GeNIMA* performs much better despite incurring higher contention for small messages. This means that besides the improvements from removing interrupts, scheduling problems, etc. in the host processors, the system can better tolerate the higher latencies due to contention. This increased tolerance comes from the facts that almost all communication layer operations used in the protocol are asynchronous, so the processor directly incurs only the small post overhead[2], and

[2]Certain messages that exchange flags are synchronous. However, these are usually one word messages with very small post overhead.

Application	Speedup (32 procs)	
	SVM	SGI Origin2000
FFT	5.55	26.36
LU-contiguous	16.49	24.73
Ocean-rowwise	5.93	30.98
Water-nsquared	14.07	24.65
Water-spatial	7.75	25.45
Radix-local	1.74	21.68
Volrend-stealing	18.64	23.88
Raytrace	17.48	26.86
Barnes-original	1.05	25.57
Barnes-spatial	23.99	24.22

Table 9. Speedup on 32 processors or both our system and the SGI Origin2000. The data presented for Barnes-spatial is for 32K bodies (as opposed to 128K bodies in the Linux version).

that the bandwidth in the system is adequate in most cases [1]. Thus, *GeNIMA* takes advantage of current technology trends that make it easier to improve effective system bandwidth than latency. Ordering and data integrity is guaranteed by ensuring that the necessary conditions are met at the protocol level.

It is also interesting to see how *GeNIMA* performs on larger scale systems. Table 9 presents data for *GeNIMA* on 32 processors for the WindowsNT version of our system. We see that many applications scale reasonably well up to 32 processors (and in fact performance improves for larger problem sizes). We are currently pursuing this research further to judge the potential of SVM in constructing larger scale systems.

Finally, in this work we explore the two extreme points in a spectrum of possible configurations. We start from an HLRC protocol that uses interrupts for protocol processing [42] and we eliminate the use of all interrupts in the protocol. It is also possible to use hybrid schemes where interrupts are used for protocol processing in some cases and not in others. For instance, diffs could sometimes be propagated with the use of interrupts and some times with the direct diffs mechanism. Similar possibilities exist with fetching pages and acquiring locks). Such schemes could try to minimize both the cost of interrupts as well as the number of messages exchanged. However, it is no clear what is the overhead of choosing which method to use in each case, and we do not consider such schemes in this work.

8. RELATED WORK

The impact of individual communication architecture parameters on performance has been studied for different architectures. In [38], the authors examine the impact of communication parameters on end performance of a network of workstations with the applications being written in Split-C on top of Generic Active Messages. They find that application performance demonstrates a linear dependence on host overhead and on the gap between transmissions of fine grain messages. Applications were found to be quite tolerant to latency and bulk transfer bandwidth. [22] finds that the occupancy of the communication controller is critical to good performance in DSM machines that provide communication and coherence at cache line granularity. Overhead is not so significant in that study (unlike in [38]). For SVM, we find host overhead and NI occupancy to not be very important, since their cost is usually amortized over page granularity. However, we find interrupt cost and I/O bus bandwidth to be very important.

The dependency of software shared memory on communication layer parameters was studied in [9], which represents the first part of this paper. The limitations of and synergies between the different layers of shared memory clusters, both for SVM as well as fine–grained software DSM were studied in [7]. The authors in [50] examine the impact of network total order, broadcast, and remote-write capability on a family of shared memory protocols. They find that latency is more important than remote writes, broadcast, or total ordering. The difference with our results comes from the significant differences in the protocols used (directory vs. no directory, etc.). Also, in this work we break the communication path between the server and the receiver into a set of stages, and treat each stage separately. Thus, latency in our work refers only to wire latency, whereas in [50] latency is an end-to-end metric including host overhead and packet processing cost. Various types of hardware support to accelerate protocols have been examined for SVM in [25] and [35], and for fine–grained software DSM in the Typhoon–zero prototype [41]. In [30], Karlsson et al. find that the latency and bandwidth of an ATM switch is acceptable in a clustered SVM architecture. In [33] a Lazy Release Consistency protocol for hardware cache-coherence is presented. In a different context, they find that applications are more sensitive to the bandwidth than the latency component of communication.

Previous efforts to avoid interrupts other than for write propagation have mainly focused on using polling on the main processor to handle asynchronously arriving messages and requests. Using polling instead of interrupts for page-based SVM could reduce the cost of handling asynchronous requests [34; 54]. However, it introduces a number of new issues: (i) It requires fairly intrusive instrumentation mechanisms that are not needed with our scheme. If asynchronous requests are to be handled in a timely fashion, instrumentation of the application is likely to be necessary (e.g. polling at back edges); (ii) the optimal frequency of polling may vary with applications and may lead to significant polling overhead; (iii) polling is not a very portable solution, since it depends on the ISA of the processor under consideration and since it affects performance portability as well. Our scheme increases the portability of protocols, since it eliminates interrupts and relies only on standard NI operations. Moreover, by eliminating interrupts and polling, it reduces the need to tweak performance parameters and it makes performance more portable across platforms with different operating systems.

Related work in network interface support for SVM has discussed how NIs can be used for several purposes: (i) Fast communication to improve the performance of traditional send and receive communication. This type of support has been exploited in many SVM projects [17; 25; 34; 53; 34; 43; 42; 44] and is also used in our base system, HLRC-SMP [42]. (ii) Protocol processing in the network interface. This choice lies at the other end of the spectrum. The network interface can be used not only to avoid interrupting the compute processor but also to perform full-blown protocol processing, including diff creation and application and the management of timestamps and write notices. This approach was taken in [53]. [18] reserves a compute processor in an SMP node for protocol processing. The amount of protocol processing involved in SVM systems with SMP nodes was examined also in an earlier simulation study [30] and other research, and is found to be small compared to other system overheads. (iii) Transparent remote data and synchronization handling that can be utilized by protocols to alleviate key bottlenecks. In this case the remote compute processor is not involved in handling message requests, but remains responsible for all protocol processing and SVM-specific operations. This is the approach we have taken. Previous work in this area relies on more specialized network interface and/or network support. The Automatic Update Release Consistency (AURC) [25] protocol takes advantage of automatic write propagation to a remote node's memory based on write-through caching and snooping writes from the memory bus in the SHRIMP network interface [11] to avoid diff computation and application in a home-based SVM protocol. The Cashmere system [34] uses the fine-grained remote write capability of the DEC Memory Channel network interface, where code instrumentation is used to propagate relevant writes (of application or protocol data) to a remote node, also in a home-based protocol. A different type of coarse- or variable-grained remote fetch support has been examined through simulation [35], but not in real implementation. More sophisticated support to accelerate specific protocol operations has also been examined in simulation, such as hardware diff engines in [4]. Support for AURC with write-back caches has also been designed and evaluated through simulation in [6]. A discussion on how the remote write access capabilities of VM-based networks can be used in SVM systems is provided in [20].

9. CONCLUSIONS

This paper has examined how the performance of software shared memory clusters of SMPs interconnected with system area networks can be improved by protocol and communication layer co-design.

We have examined the effects of communication parameters to a family of SVM protocols. Through detailed architectural simulations of a cluster of SMPs and a variety of applications, we find that most applications are very sensitive to interrupt cost, and a few would benefit from improvements in bandwidth relative to processor speed as well. Unbalanced systems with relatively high interrupt costs and low I/O bandwidth can result in substantial losses in application performance. In these cases we observe slowdowns of more than 90% (a factor of 10 longer execution time). However, most applications are not sensitive to host overhead and network interface occupancy. Overall, our results show that SVM systems can benefit significantly from the availability of faster network interfaces. We are currently investigating this direction by emulating a cluster on top of a hardware cache-coherent system that provides faster communication than today's state-of-the-art clusters.

We have used network interface support to decouple asynchronous message handling from protocol processing and to thereby eliminate the need for expensive interrupts or polling in SVM protocols. In particular, we have implemented NI support for general-purpose, *explicit* data movement and synchronization operations that are not specific to SVM, and altered the SVM protocol to take advantage of these operations. In the final *GeNIMA* protocol, asynchronous message handling is done entirely in the NI, and protocol processing is done on the host processors but only at synchronous points with respect to application and protocol execution.

The protocol propagates information more eagerly in some cases, but the need for asynchronous protocol processing and the related interrupts (or polling) is eliminated without requiring the NI to be tightly integrated in the node or to observe memory operations in it.

We have prototyped these extensions in the programmable network interface of the Myrinet network—though they are simple enough to not require programmability—and evaluated their impact on application performance on a network of Intel Pentium Pro SMPs. This system exhibits the characteristics of the achievable configuration used in the simulation results with higher interrupt cost. We found that: (i) The proposed communication and protocol extensions improve performance substantially for our suite of ten applications. Application performance improves by 38% on average for reasonably well-performing applications (and up to 120% for applications that do not perform very well under SVM). Similarly, the specific components of execution time targeted by the individual mechanisms improve substantially: data wait time improves up to 45% and lock time up to 60%. Several applications that originally performed in mediocre ways now perform much better, even well, on a 16-processor system. (ii) Different applications benefited greatly from different NI features, indicating that all three should be supported. These features also provide more flexibility in the choice of efficient protocol operations and management. (iii) While speedups are improved greatly by these techniques and are much closer to those on hardware-coherent systems for most applications, they are still not as close as we might like even at this 16-processor scale. On our applications and modern systems, we find synchronization cost to be the most important protocol overhead for improving overall application performance further. (iv) Analysis with a firmware performance monitor in the NI shows that *GeNIMA* indeed exhibits increased traffic and contention in the communication layer due to its more eager propagation of information; however, most messages exchanged are asynchronous and the system is able to tolerate the increased contention and to improve overall system performance with current communication bandwidths and latencies.

While our simple NI extensions suffice to eliminate interrupts and polling, alternative and more sophisticated extensions are possible, even within our target scope of explicit operations that don't require the NI to observe memory operations. These may improve performance further given appropriate system characteristics, and we plan to explore them in the future.

10. ACKNOWLEDGMENTS

We thank Hongzhang Shan for making available to us the improved version of Barnes. We also thank the members of the PRISM group at Princeton, in particular Liviu Iftode, Kai Li, Rudrajit Samanta, Limin Wang, and Yuanyuan Zhou for useful discussions. We gratefully acknowledge the support of NSF and DARPA.

REFERENCES

[1] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *Proceedings of Supercomputing 98, Orlando, FL*, November 1998.

[2] D. H. Bailey. FFTs in External or Hierarchical Memories. *Journal of Supercomputing*, 4:23–

25, 1990.

[3] J. E. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[4] R. Bianchini, L.I Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding communication latency and coherence overhead in software dsms. In *The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[5] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. *Supporting a coherent shared address space across SMP nodes: An application-driven investigation*, volume 105, pages 19–59. Springer-Verlag New York, Inc., November 1998.

[6] A. Bilas, L. Iftode, and J. P. Singh. Evaluation of hardware support for shared virtual memory clusters. In *The 12th ACM International Conference on Supercomputing (ICS'98)*, July 1998.

[7] A. Bilas, D. Jiang, Y. Zhou, and J.P. Singh. Limits to the performance of software shared memory: A layered approach. In *The 5th IEEE Symposium on High-Performance Computer Architecture*, February 1999. Also Princeton University Tech. Report No. TR-576-98.

[8] A. Bilas, C. Liao, and J. P. Singh. Accelerating shared virtual memory using commodity ni support to avoid asynchronous message handling. In *The 26th International Symposium on Computer Architecture*, May 1999.

[9] A. Bilas and J. P. Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *Proceedings of Supercomputing 97, San Jose, CA*, November 1997.

[10] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[11] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 142–153, April 1994.

[12] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[13] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, April 1977.

[14] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, August 1997.

[15] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proceedings of Hot Interconnects V Symposium*, Stanford, August 1997.

[16] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pages 256–266, May 1992.

[17] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: analyzing the performance of clustered distributed virtual shared memory. In *The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, Oct 1996.

[18] B. Falsafi and D. A. Wood. Scheduling communication on an SMP node parallel machine. In *The 3nd IEEE Symposium on High-Performance Computer Architecture*, pages 128–138, 1997.

[19] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, February 1996.

[20] N. Hardavellas, G. C. Hunt, S. Ioannidis, R. Stets, S. Dwarkadas, L. Kontothanassis, and M. L. Scott. Efficient use of memory-mapped network interfaces for shared memory com-

puting. In *Newsletter of the IEEE CS Technical Committee on Computer Architecture*, pages 28–33, March 1997.

[21] L. Hernquist. Hierarchical N-body methods. *Computer Physics Communications*, 48:107–115, 1988.

[22] C. Holt, M. Heinrich, J. P. Singh, , and J. L. Hennessy. The effects of latency and occupancy on the performance of dsm multiprocessors. Technical Report CSL-TR-95-xxx, Stanford University, 1995.

[23] C. Holt, J. P. Singh, and J. Hennessy. Architectural and application bottlenecks in scalable DSM multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[24] R. W. Horst and D. Garcia. ServerNet SAN I/O Architecture. In *Proceedings of Hot Interconnects V Symposium*, Stanford, August 1997.

[25] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving release-consistent shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[26] L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[27] L. Iftode, J. P. Singh, and Kai Li. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1996.

[28] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[29] D. Jiang and J. P. Singh. Does application performance scale on cache-coherent multiprocessors: A snapshot. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, May 1999.

[30] M. Karlsson and P. Stenstrom. Performance evaluation of cluster-based multiprocessor built from atm switches and bus-based multiprocessor servers. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[31] M. G. H. Katevenis, E. P. Markatos, G. Kalokerinos, and A. Dollas. Telegraphos: A substrate for high-performance computing on workstation clusters. *Journal of Parallel and Distributed Computing*, 43(2):94–108, 15 June 1997.

[32] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[33] L. I. Kontothanasis, M. L. Scott, and R. Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In *Supercomputing '95*, November 1995.

[34] L. I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr., S. Dwarkadas, and M. L. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 157–169, June 1997.

[35] L. I. Kontothanassis and M. L. Scott. Using memory-mapped network interfaces to improve t he performance of distributed shared memory. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[36] J. P. Laudon and D. Lenoski. The sgi origin2000: A scalable cc-numa server. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, June 1997.

[37] C. Liao, M. Martonosi, and D. W. Clark. Performance monitoring in a myrinet-connected shrimp cluster. Submitted for publication, 1998.

[38] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effect of communication latency, overhead, and bandwidth on a cluster architecture. Technical Report CSD-96-925, Berkeley, November 1996.

[39] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory MIMD architectures. In *Proceedings of the Boston Workshop on Volume Visualization*, October 1992.

[40] S. Pakin, M. Buchanan, M. Lauria, and A. Chien. The Fast Messages (FM) 2.0 streaming interface. Usenix'97, 1996.

[41] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 34–43, New York, May22–24 1006. ACM Press.

[42] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based svm protocols for smp clusters: Design, simulations, implementation and performance. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture, Las Vegas*, February 1998.

[43] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *The 4th IEEE Symposium on High-Performance Computer Architecture*, pages 125–136, January 1998.

[44] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lucas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. Implementing fine-grain distributed shared memory on commodity smp workstations. Technical Report 1307, University of Wisconsin-Madison, March 1996.

[45] A. Sharma, A. T. Nguyen, J. Torellas, M. Michael, and J. Carbajal. Augmint: a multiprocessor simulation environment for Intel x86 architectures. Technical report, University of Illinois at Urbana-Champaign, March 1996.

[46] J. P. Singh, A. Gupta, and J. L. Hennessy. Implications of hierarchical N-body techniques for multiprocessor architecture. *ACM Transactions on Computer Systems*, May 1995. To appear. Early version available as Stanford Univeristy Tech. Report no. CSL-TR-92-506, January 1992.

[47] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(6), june 1994.

[48] J. P. Singh and J. L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[49] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *The 3nd IEEE Symposium on High-Performance Computer Architecture*, Feb 1997.

[50] R. Stets, S. Dwarkadas, L. Kontothanassis, , U. Rencuzogullari, and M. L. Scott. The effect of network toral order, broadcast, and remote-write capability on network-based shared memory computing. In *The 6th IEEE Symposium on High-Performance Computer Architecture*, January 2000.

[51] D. Stodolsky, J. B. Chen, and B. Bershad. Fast interrupt priority management in operating system kernels. In USENIX Association, editor, *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures: September 20–21, 1993, San Diego, California, USA*, pages 105–110, Berkeley, CA, USA, September 1993. USENIX.

[52] S.C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1995.

[53] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.

[54] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B.R. Toonen, I. Schoinas, M.D. Hill, and D. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.