Contents

Chapter 1

Experiences with Shared Virtual Memory on System Area Network Clusters: System Simulation, Implementation, and Emulation

Angelos Bilas

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
bilas@eecg.toronto.edu

Jaswinder Pal Singh

Department of Computer Science
Princeton University
Princeton, NJ-08540, USA
jps@cs.princeton.edu

Abstract

Recently there has been a lot of effort in providing cost-effective Shared Memory systems by employing software only solutions on clusters of high-end workstations coupled with high–bandwidth, low–latency commodity networks. The related results have generated a lot of interest as well as many questions about how future SVM clusters will look and what are the related issues in building them.

In this work we first use system simulation to identify system bottlenecks related to architectural features of SVM clusters. Then we use actual system implementation to demonstrate how the most important bottlenecks can be alleviated on todays clusters. Finally we use system emulation to investigate trends in building future clusters with wide nodes and aggressive interconnects. System emulation is also able to point out issues that are either not possible to model in detail or inadvertently overlooked by system simulation.

1.1 Introduction

Recently, there has been a lot of work on providing a shared address space abstraction on clusters of commodity workstations interconnected with low-latency, high-bandwidth system area networks (SANs). The motivation for these efforts is two-fold: (i) System area network (SAN) clusters have a number of appealing features: They follow technology curves well since they are composed of commodity components. They exhibit shorter design cycles and lower costs than tightly-coupled multiprocessors. They can benefit from heterogeneity and there is potential for providing highly-available systems since component replication is not as costly as in other architectures. While these techniques reduce cost, unfortunately they usually lower performance as well. A great deal of research effort has been made to improve these systems for large classes of applications. (ii) The success of the shared address space abstraction: Previous work [39], [1], [21] has shown that a shared address space can be provided efficiently on tightly-coupled hardware DSM systems up to the 128 processor scale. Developers have been writing new software for the shared address space abstraction and legacy applications are being ported to the same abstraction as well. Finally, most vendors are designing hardware cache-coherent machines, targeting both scientific as well as commercial applications.

Our focus in this work is SVM systems. Since SVM was first proposed [40], much research has been done in improving the protocol layer by relaxing the memory consistency models [3], [34], by improving the communication layer with low-latency, high-bandwidth, user-level communication [20], [11], [15], [43], [12], [52], [2], [44], [27], [36], [37], [56], [4], and by taking advantage of the two-level communication hierarchy in systems with multiprocessor nodes [17], [5], [50], [6], [45], [54]. Recently, the application layer has also been improved, by discovering application restructurings that dramatically improve performance on SVM systems [31].

With this progress, it is now interesting to examine what are the important *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. Such studies can hopefully assist system designers in determining where to focus their energies in improving performance, and users in determining what system characteristics are appropriate for their applications.

Then, we use an actual system implementation 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. They also require neither visibility into the node memory system nor code instrumentation to identify memory operations. We prototype the mechanisms and such a *synchronous home-based LRC* protocol, called GeNIMA (GEneral-purpose Network Interface support in a shared Memory Abstraction), on a cluster

of SMPs with a programmable NI, though the mechanisms are simple and do not require programmability.

Finally, we use system emulation to investigate what will be the effect of faster networks and wider nodes on future SVM clusters. Simulation results indicate that the benefits can be substantial. However, the actual system cannot be used to verify this and we would like to go beyond what simulations can achieve, by examining a system as close as possible to future clusters. We use an existing, large–scale hardware cache–coherent system with 64 processors to emulate a future cluster and we port the communication layer and SVM protocol from the actual cluster to this system. We quantify the effects of faster interconnects and wide, cc-NUMA nodes on future cluster design. We find that current SVM protocols can only partially take advantage of faster interconnects and wider nodes and that further research is required in protocol design for future SVM clusters.

Our experience shows that all three methods of studying system behavior and identifying system trends have strengths and weaknesses and that ideally all three should be used. More specifically, simulation is very helpful in studying large ranges of parameters. However, it is rather easy to inadvertently overlook a number of system issues that need to be dealt with in actual systems. System emulation allows a more precise look into the future, however, it is limited in scope by the underlying architecture. Finally, system implementation, although tedious, is invaluable in understanding in depth how to balance complexity and performance tradeoffs and to attracting actual users.

The rest of this work is organized as follows: Section 1.2 presents background information and discusses in more detail our goals and our overall methodology. Section 1.3 presents our simulation methodology and results. Section 1.4 describes our actual system implementation and our results. Section 1.5 discusses our system emulation infrastructure and results. Section 1.6 discusses our overall results at a high level. Finally, Section 1.7 presents related work and Section 1.8 draws our conclusions.

1.2 Overall Methodology and Results

The end performance of a shared virtual memory (SVM) system, or of any implementation of a programming model, relies on the performance and interactions of three layers that sit between the problem to be solved and the hardware (Figure 1.1): the application or program layer, the protocol layer that supports the programming model, and the communication layer that implements the machine's communication architecture. Each of the system layers has both performance and functionality characteristics, which can be enhanced to improve overall performance [7].

1.2.1 System Simulation

This paper first examines these questions through detailed architectural simulation using applications with widely different behavior. We simulate a cluster architec-

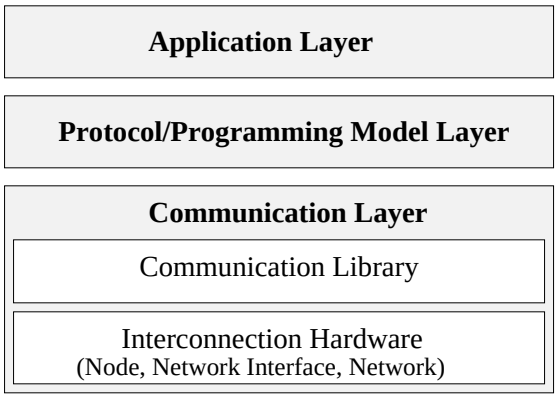

Figure 1.1. The layers that effect the end application performance in software shared memory systems.

ture 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 [56]. The base case of the protocol, called home-based lazy release consistency (HLRC) does not require any additional hardware support. The major 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). After dealing with performance parameters, we also briefly examine the impact of key granularity parameters of the communication architecture. These are the page size, which is the granularity of coherence, and the number of processors per node.

We want to understand how performance changes as the parameters of the communication architecture are varied relative to processor speed, both to see where we should invest systems energy and to understand the likely evolution of system performance as technology evolves. For this, we use a wide range of values for each parameter. 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. Node-to-network bandwidth, typically limited by the I/O bus, is also significant for a few applications, but interrupt cost is important for all the applications we study. These results suggest that system designers should focus on reducing interrupt costs to support SVM well, and SVM protocol designers should try to avoid interrupts as possible, perhaps by using polling or by using a programmable communication assist to run part of the protocol avoiding the need

to interrupt the main processor.

1.2.2 System Implementation

Next, we use system implementation to quantify the benefits from eliminating interrupts. One way to avoid interrupts is to use polling. However, this approach introduces different types of overheads, and the tradeoff between the two methods is unclear for SVM. In our implementation, we redesign the basic protocol operations to avoid interrupts and polling altogether.

In SVM systems, different nodes exchange 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 perhaps performing protocol operations related to those messages. In current SVM systems, the two activities are usually coupled together, using either interrupts or polling to enable the asynchronous protocol processing: Message handling may be performed by the network interface in newer systems, but many types of incoming messages invoke asynchronous protocol processing on the main (host) processor as well. For example, lazy protocols produce and send out page invalidations (write notices) when a lock acquire request arrives and page timestamps are accessed or manipulated when a page request or update message is received.

The insight behind the communication-layer mechanisms and the synchronous home-based lazy release consistency protocol we use in this work (GeNIMA) is to decouple protocol processing from message handling [8]. By providing support in the network interface for simple, generic operations, asynchronous message handling can be performed entirely in the network interface, eliminating the need for interrupting the receiving host processor (or using polling) for this purpose. Protocol processing is still performed on the host processors. However, with the decoupling, it is now performed only at "synchronous" points, i.e. when the processor is already sending a message or receiving an awaited response, and not in response to incoming asynchronous messages. Thus, the need for interrupts or polling is eliminated. The forms of NI support we employ in GeNIMA are automatically moving data between the network and user-level addresses in memory, and providing support for mutual exclusion. They are not specific to SVM, but are useful general-purpose mechanisms.

While we use a programmable Myrinet network interface for prototyping, the message-handling mechanisms are simple and do not require programmability. Unlike some previously proposed mechanisms that also avoid asynchronous message handling in certain situations [37], [27], they support only explicit operations; they do not require code instrumentation or observing the memory bus, or the network to provide global ordering guarantees. Thus, they can more likely be supported in commodity NIs. In fact, many modern communication systems [10], [20], [33], [25] and the recent Virtual Interface Architecture (VIA) standard [14] support some of these or similar types of operations.

We find that: (i) The proposed protocol extensions improve performance sub-

stantially 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). (ii) While speedups are improved greatly by these techniques, they are still not as close as we might like to efficient hardware cache-coherent systems, even at this 16-processor scale. For the applications we examine, synchronization-related cost is the most important protocol overhead for improving performance further. (iii) Analysis with a performance monitor built into the NI firmware shows that although GeNIMA exhibits increased traffic and contention in the communication layer compared to the base HLRC-SMP protocol, it is able to tolerate the increased contention.

1.2.3 System Emulation

Finally, we use system emulation to investigate the effects of faster networks and wide system nodes on future SVM clusters [9]. Traditionally, SAN clusters have been built using small-scale symmetric multiprocessors (SMPs) that follow a uniform memory access (UMA) architecture. The interconnection networks used are faster than LANs and employ user-level communication that eliminates many of the overheads associated with the operating system and data copying. With current advances in technology it will be possible in the near future to construct commodity shared address space clusters out of larger–scale nodes connected with non-coherent networks that offer latencies and bandwidth comparable to interconnection networks used in hardware cache–coherent systems. The shared memory abstraction can be provided on these systems in software both across and within nodes. These nodes will be wider than today's SMPs, employing between 8 and 32 processors. At the same time, the performance characteristics of system area networks are improving. There are already SANs available that offer bandwidth comparable to that of the memory bus on today's commodity SMPs [11] and end-to-end latencies in the order of a few microseconds. These networks are getting closer to providing performance comparable to most tightly coupled interconnection networks that have been used in hardware cc-NUMA systems. The design space for these next generation SANs is not yet fixed; there is a need to determine the required levels of both functionality and performance. Similarly, it is not clear how using wider nodes will impact software shared memory protocol design and implementation as well as system performance.

To investigate these issues we build extensive emulation infrastructure. System emulation allows direct execution of the same code used on the SAN cluster on top of the emulated communication layer. Using simulation, although advantageous in some cases, tends to overlook important system limitations that may shift bottlenecks to system components that are not modeled in detail. Using emulation on top of an existing, commercial multiprocessor results in a more precise model for future clusters: It uses commercial components that will most likely be used in future nodes and a commercial, off-the-shelf operating system. Moreover, simulators that have been used in this area do not usually simulate operating systems and

system microprocessors in detail due to the large number of nodes involved and the simulation time required. In fact, using a real, commercial OS reveals a number of issues usually hidden in simulation studies and/or custom built systems.

We use an existing, large–scale hardware cache–coherent system, an SGI Origin2000, to emulate such a cluster. We port an existing, low–level communication layer and a shared virtual memory (SVM) protocol on this system and study the behavior of a set of real applications. This communication layer, Virtual Memory-Mapped Communication (VMMC), provides near-hardware performance in exchanging messages [12]. We use VMMC to partition the system in a set of cc-NUMA nodes that communicate with message passing. On top of VMMC we provide an SVM layer that provides a shared address space abstraction across nodes. The protocol we use, *GeNIMA*, has been demonstrated to perform and scale well for a wide range of applications on a real cluster with 64 processors [8]. *GeNIMA* is already tuned for current state-of-the art SAN clusters and takes advantage of network interface support to eliminate asynchronous protocol processing and interrupts. We present results for configurations of 64-processors, which is (to our knowledge) the largest configuration used for software shared memory.

We find that, software shared memory can benefit substantially from faster interconnection networks and can lead in building inexpensive shared memory systems for large classes of applications. However, current SVM protocols can only partially take advantage of faster interconnects and do not address the issues that arise when faster networks and wider nodes are used. Our work quantifies these effects and identifies the areas where more research is required for future SVM clusters.

In the next sections we describe in more detail the results we obtain from system simulation, implementation, and emulation.

1.3 System Simulation

In this Section we use system simulation to examine the impact of the communication layer parameters on system performance and identify future trends.

1.3.1 Simulation Environment

The simulation environment we use is built on top of augmint [48], an execution driven simulator using the *x86* instruction set, and runs on *x86* systems. In this section we present the architectural parameters that we do not vary.

The simulated architecture (Figure 1.2) assumes a cluster of c–processor SMPs connected with a commodity interconnect like Myrinet [11]. 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 [49] has 26 entries, 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

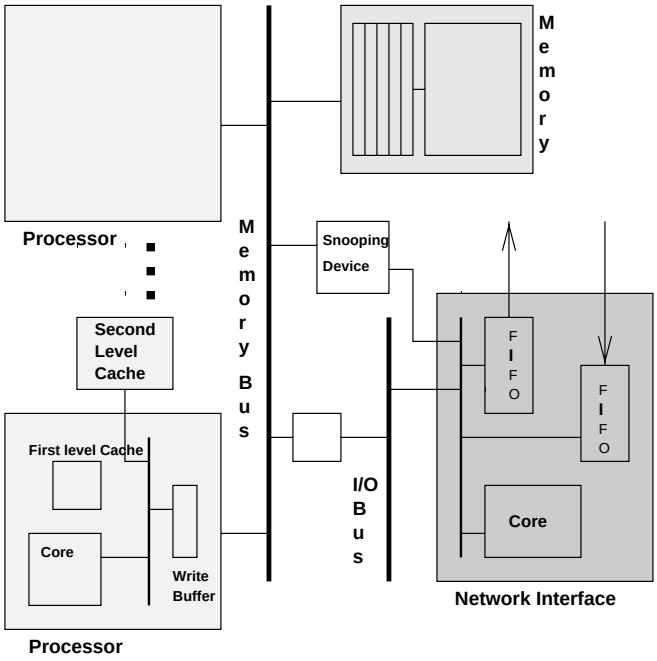

Figure 1.2. Simulated node architecture.

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 [15], [42], [13] 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, 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.

The protocol we use is a home–based protocol, HLRC [56]. The protocol is SMP–aware and attempts to utilize the hardware sharing and synchronization within an SMP as much as possible, reducing software involvement [5]. 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. More complicated schemes (i.e. round robin, random assignment) that result in better load balance in interrupt handling can be used if the operating system provides the necessary support. These schemes however, may increase the cost of delivering interrupts. In this paper we also examine a round robin scheme.

1.3.2 Methodology

As mentioned earlier, 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, total number of processors, etc. remain constant.

When a message is exchanged between two hosts, it is put in a post queue at the network interface. In an asynchronous send operation, which we assume, the sender is free to continue with useful work. The network 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 [10], [13]. Thus, the interrupt cost is an overhead related not so much to data transfer but to processing requests.

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. We choose relatively aggressive fixed values 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 range of values we consider is between 0 (or almost 0) processor cycles and 10000 processor cycles (about 50μs with a $5n$s processor clock). The achievable value we use is an overhead of 600 processor cycles per message.

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. The range of values for the I/O bus bandwidth is from 0.25 MBytes per processor clock MHz up to 2 MBytes per processor clock MHz (or 50 MBytes/s to 400 MBytes/s assuming a 200 MHz processor clock). The achievable value is 0.5 MBytes/MHz, or 100 MBytes/s assuming a 200 MHz processor clock.

Network Interface Occupancy is the time spent on the network interface preparing each packet. 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. We vary the occupancy of the network interface from almost 0 to 10000 processor cycles (about 50μs with a 5ns processor clock) per packet. The achievable value we use is 1000 main processor cycles, or about 5μs assuming a 200 MHz processor clock. This value is realistic for the currently available programmable NIs, given that the programmable communication assist on the NI is usually much slower than the main processor.

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. 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 (0 processor cycles) to 50000 processor cycles for both issuing and delivering an interrupt (total 100000 processor cycles or 500 μs with a 5ns processor clock). 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 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.

To capture the effects of each parameter separately, we keep the other parameters fixed at their achievable values. Where necessary, we also perform additional guided simulations to further clarify the results.

In addition to the results obtained by varying parameters and the results obtained for the achievable parameter values, an interesting result is the speedup obtained by using the best value in our range for each parameter. This limits the performance that can be obtained by improving the communication architecture within our range of parameters. The parameter values for the best configuration are: host overhead 0 processor cycles, I/O bus bandwidth equal to the memory bus bandwidth, network interface occupancy per packet 200 processor cycles and total interrupt cost 0 processor cycles. In this best configuration, contention is still modeled since the values for the other system parameters are still nonzero. Table 1.1 summarizes the values of each parameter. With a 200 MHz processor, the achievable set of values discussed above assumes the parameter values: host overhead 600 processor cycles, memory bus bandwidth 400 MBytes/s, I/O bus bandwidth 100

MBytes/s, network interface occupancy per packet 1000 processor cycles and total interrupt cost 1000 processor cycles.

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

Table 1.1. Ranges and achievable and best values of the communication parameters under consideration.

1.3.3 Applications

In our work, we use the SPLASH-2 [53] application suite. A detailed classification and description of the application behavior for SVM systems with uniprocessor nodes is provided in the context of AURC and LRC in [28]. The applications can be divided in two groups, regular and irregular. The regular applications are FFT, LU, and Ocean. Their common characteristic is that they are optimized to be single-writer applications; 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. Protocol action is required only to fetch pages. The applications have different inherent and induced communication patterns [53], [28], which affect their performance and the impact on SMP nodes. The irregular applications in our suite are Barnes, Radix, Raytrace, Volrend and Water.

1.3.4 Effects of Communication Parameters

In this section we present the effects of each parameter on the performance of an all-software HLRC protocol for a range of values.

Host overhead: Figure 1.3 shows that the slowdown due to the host overhead is generally low, especially for realistic values of asynchronous message overheads. However, it varies among applications from less than 10% for Barnes-space, Ocean-contiguous and Raytrace to more than 35% for Volrend, Radix and Barnes-rebuild across the entire range of values. In general, 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.

Network interface occupancy: Figure 1.4 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

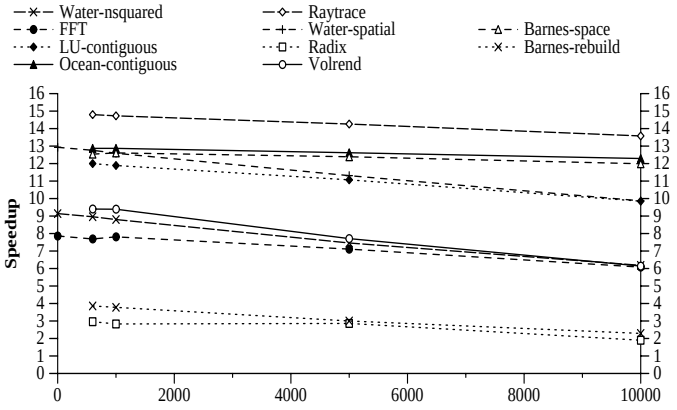

Figure 1.3. Effects of host overhead on application performance. The data points for each application correspond to a host overhead of 0, 600, 1000, 5000, and 10000 processor cycles.

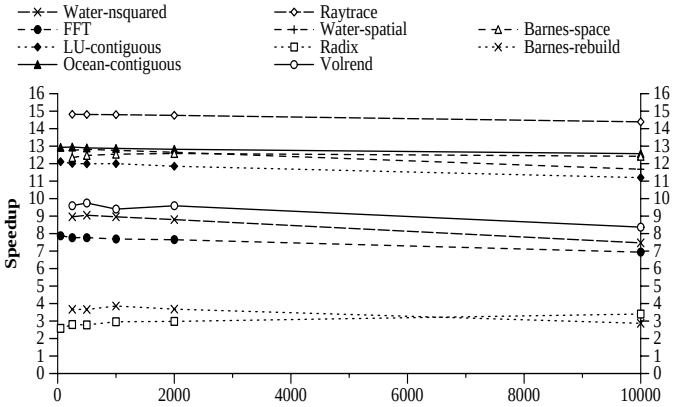

Figure 1.4. Effects of network interface occupancy on application performance. The data points for each application correspond to a network occupancy of 50, 250, 500, 1000, 2000, and 10000 processor cycles.

applications that send a large number of messages. For these applications, slowdowns of up to 22% are observed at the highest occupancy values. The speedup observed for Radix is in reality caused by timing issues (contention is the bottleneck in Radix).

I/O bus bandwidth: Figure 1.5 shows the effect of I/O bandwidth on application performance. Reducing the bandwidth 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

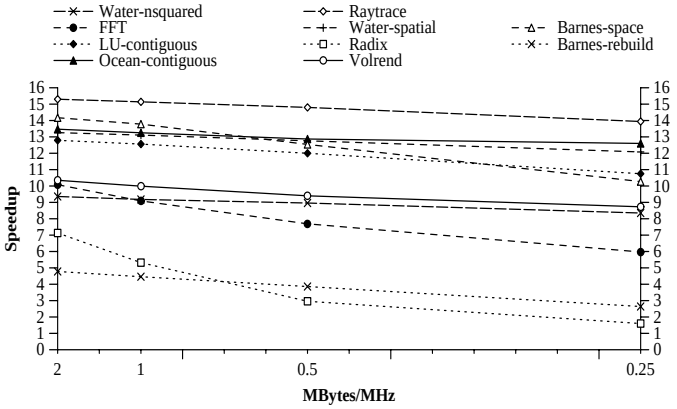

Figure 1.5. Effects of I/O bandwidth on application performance. The data points for each application correspond to an I/O bandwidth of 2, 1, 0.5 and 0.25 MBytes per processor clock MHz, or 400, 200, 100, and 50 MBytes/s assuming a 200 MHz processor.

Barnes-rebuild benefit much from increasing the I/O bus bandwidth beyond the achievable relationships to processor speed today. Of course, 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 quickly find ourselves at the relationship reflected by the lower bandwidth case we examine (or even worse). What it does mean is that if bandwidth keeps up with processor speed, it is not likely to be the major limitation on SVM systems for applications.

Interrupt cost: Figure 1.6 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 this point 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 that exhibits an anomaly since the way pages are distributed among processors changes with interrupt cost). This suggests that architectures and operating systems should work harder to improving interrupt costs if they are to support SVM well, and SVM protocols should try to avoid interrupts as much as possible,

1.3.5 Limitations on Application Performance

In this section we examine the difference in performance between the *best* configuration and an ideal system (where the speedup is computed only from the com-

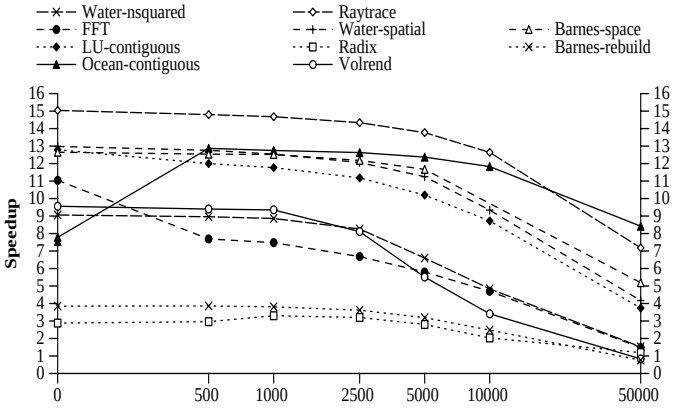

Figure 1.6. Effects of interrupt cost on application performance. The six bars for each application correspond to an interrupt cost of 0, 500, 1000, 2500, 5000, 10000, and 50000 processor cycles.

pute and local stall times, ignoring communication and synchronization costs), and the difference in performance between the *achievable* and the *best* configuration on a per application basis. Recall that best stands for the configuration where all communication parameters assume their best value, and achievable stands for the configuration where the communication parameters assume their achievable values. The goal is to identify the application properties and architectural parameters that are responsible for the difference between the best and the ideal performance, and the parameters that are responsible for the difference between the achievable and the best performance. The speedups for each configuration will be called *ideal, best* and *achievable* respectively. Table 1.2 shows these speedups for all applications. In many cases, the achievable speedup is close to the best speedup. However, in some cases (FFT, Radix, Barnes) there remains a gap. The performance with the best configuration is often quite far from the ideal speedup. To understand these effects, let us examine each application separately.

FFT: The best speedup for FFT is about 13.5. The difference from the ideal speedup of 16.2 comes from data wait time at page faults, which have a cost even for the best configuration, despite the very high bandwidth and the zero-cost interrupts. The achievable speedup is about 7.7. There are two major parameters responsible for this drop in performance: the cost of interrupts and the bandwidth of the I/O bus.

LU: The best speedup is 13.7. The difference from the ideal speedup is due to load imbalances in communication and due to barrier cost. The achievable speedup for LU is about the same as the best speedup, since this application has very low communication to computation ratio, so communication is not the problem.

Application	Best	Achievable	Ideal
FFT	13.5	7.7	16.2
LU	13.7	14.0	18.9
Ocean	10.5	13.0	16.0
Water(nsquared)	9.9	9.0	15.8
Water(spatial)	13.7	13.3	15.8
Radix	7.0	3.0	16.1
Volrend	10.9	9.40	15.4
Raytrace	15.6	14.8	16.4
Barnes(rebuild)	5.9	3.9	15.4
Barnes(space)	14.5	12.5	15.6

Table 1.2. Best and Achievable Speedups for each application

Ocean: The best speedup for Ocean is 10.55. The reason for this is that when the interrupt cost is 0 an anomaly is observed in first touch page allocation and the speedup is very low due to a large number of page faults. The achievable speedup is 13.0, with the main cost being that of barrier synchronization. It is worth noting that speedups in Ocean are artificially high because of local cache effects: a processor's working set does not fit in cache on a uniprocessor, but does fit in the cache with 16 processors. Thus the sequential version performs poorly due to the high cache stall time.

Barnes-rebuild: The best speedup for Barnes-rebuild is 5.90. The difference from the ideal is because of page faults in the large number of critical sections (locks). The achievable speedup is 3.9. The difference between the best and achievable speedups in the presence of page faults is because synchronization wait time is even higher due to the increased protocol costs. To verify this we disabled remote page fetches in the simulator so that all page faults appear to be local. The speedup becomes 14.64 in the best and 10.62 in the achievable cases respectively. The gap between the best and the achievable speedups is again due to host and NI overheads.

Barnes-space: The second version of Barnes we run is an improved version with minimal locking [31]. The best speedup is 14.5, close to the ideal. The achievable speedup is 12.5. The difference between these two is mainly because of the lower available I/O bandwidth in the achievable case. This increases the data wait time in an imbalanced way.

Water-Nsquared: The best speedup for Water-Nsquared is 9.9 and the achievable speedup is about 9. The reason for the not very high best speedup is page faults that occur in contended critical sections, greatly increasing serialization at locks. If we artificially disable remote page faults the best speedup increases from 9.9 to 14.1. The cost for locks in this artificial case is very small and the non-ideal speedup is due to imbalances in the computation itself.

Water-Spatial: The best speedup is 13.75. The difference from ideal is mainly due to small imbalances in the computation and lock wait time. Data wait time is very small. The achievable speedup is about 13.3.

Radix: The best speedup for Radix is 7. The difference from the ideal speedup of 16.1 is due to data wait time, which is exaggerated by contention even at the *best* parameter values, and the resulting imbalances among processors which lead to high synchronization time. The imbalances are observed to be due to contention in the network interface. The achievable speedup is only 3. The difference from the best speedup is due to the same factors: data wait time is much higher and much more imbalanced due to much greater contention effects. The main parameter responsible for this is I/O bus bandwidth. For instance, if we quadruple I/O bus bandwidth the achievable speedup for Radix becomes 7, just like the best speedup.

Raytrace: Raytrace performs very well. The best speedup is 15.64 and the achievable speedup 14.80.

Volrend: The best speedup is 10.95. The reason for this low number is imbalances in the computation itself due to the cost of task stealing, and large lock wait times due to page faults in critical sections. If we artificially eliminate all remote page faults, then computation is perfectly balanced and synchronization costs are negligible (speedup is 14.9 in this fictional case). The achievable speedup is 9.40, close to the best speedup.

Summary: We see that the difference between ideal and best performance is due to page faults that occur in critical sections, I/O bandwidth limitations and imbalances in the communication and computation, and the difference between best and achievable performance is primarily due to the interrupt cost and I/O bandwidth limitations and less due to the host overhead. Overall, application performance on SVM systems today appears to be limited primarily by interrupt cost, and next by I/O bus bandwidth. Host overhead and NI occupancy per packet are substantially less significant, and in that order.

1.3.6 Degree of Clustering

In addition to the performance parameters of the communication architecture discussed above, the number of processors per node is another important parameters that affects the behavior of the system. It plays an important role in determining the amount of communication that takes place in the system, the cost of which is then determined by the performance parameters.

The degree of clustering is the number of processors per node. Figure 1.7 shows that for most applications greater clustering helps even if the memory configuration and bandwidths are kept the same[1]. We use cluster sizes of 1, 4, 8 and 16 processors,

[1]This assumption, of keeping the memory subsystem the same and increasing the number of processors per node is not very realistic, since systems with higher degrees of clustering usually have a more aggressive memory subsystem as well, and are likely to provide greater node-to-network bandwidth.

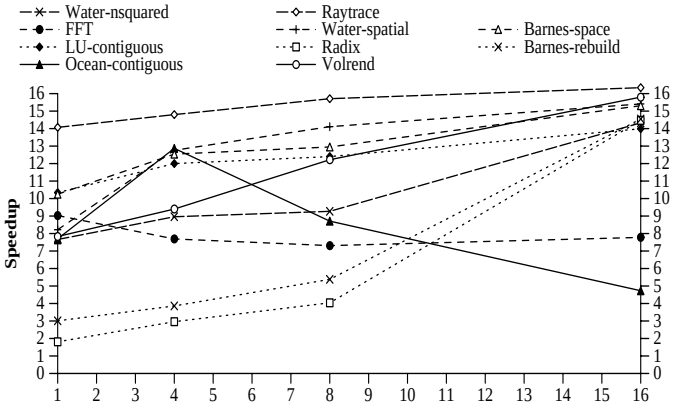

Figure 1.7. Effects of cluster size on application performance. The data points for each application correspond to a cluster size of 1, 4, 8, and 16 processors per node.

always keeping the total number of processors in the system at 16. These configurations cover the range from a uniprocessor node configuration to a cache-coherent, bus-based multiprocessor. Typical SVM systems today use either uniprocessor or 4-way SMP nodes. A couple of interesting points emerge. First, unlike most applications, for Ocean-contiguous the optimal clustering is four processors per node. The reason is that Ocean-contiguous generates a lot of local traffic on the memory bus due to capacity and conflict misses, and more processors on the bus exacerbate this problem. On the other hand, Ocean-contiguous benefits a lot from clustering because of the communication pattern. Thus when four processors per node are used, the performance improvement over one processor per node comes from sharing. When the system has more than four processors per node, the memory bus is saturated and although the system benefits from sharing, performance is degrading because of memory bus contention. Radix and FFT also put greatly increased pressure on the shared bus. The cross-node SVM communication however, is very high and the reduction in it via increased spatial locality at page grain due to clustering outweighs this problem. The second important point is that the applications that perform very poorly under SVM do very well on a shared bus system at this scale. The reason is that these applications either exhibit a lot of synchronization or make fine grain accesses, both of which are much cheaper on a hardware-coherent shared bus architecture. For example, applications where the problem in SVM is page faults within critical sections (i.e. Barnes-rebuild) perform much better on this architecture. These results show that bus bandwidth is not the most significant problem for these applications at this scale, and the use of hardware coherence and synchronization outweighs the problems of sharing a bus.

1.4 System Implementation

We now describe a minimal set of extensions to the network interface that can be used to remove the need for asynchronous protocol processing, and how the resulting message and protocol handling differs from that of the Base protocol. We also present a detailed evaluation of these extensions.

1.4.1 Network Interface and SVM Protocol Extensions

Base protocol: The base SVM system that we use is HLRC-SMP [5], [45], an all-software home-based lazy release consistency system extended to efficiently support and take advantage of SMP nodes. The Base (HLRC-SMP) protocol uses the communication layer only as a fast interrupt-based messaging system. Each protocol request causes an interrupt that schedules the protocol process on one of the processors. The incoming protocol requests that require interrupts in the base protocol 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.

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 [10], [20], [37], [27]. 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 [10], [20], [33], [25], [14].

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 other remote protocol data structures (e.g. page timestamps, barrier control information). In addition, there are two major cases where this operation is used:

(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. However, mutual exclusion and coherence information 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. 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.

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.

Remote fetch: 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.

The remote fetch operations is primarily used to avoid interrupts when fetching updated version of shared pages. Remote pages are fetched in the Base protocol by sending a request to the home node. In the new protocol, the requester uses remote fetch operations to directly access the required data without interrupting the remote (home) node.

Although, the remote fetch operation can also be used instead of remote deposit to avoid interrupts at coherence information propagation, we use an eager propagation scheme to achieve the same effect. Thus, invalidations are eagerly "pushed" at release points to all nodes, as opposed to lazily being "pulled" at acquires. The former (and current) approach increases the remote release cost while the latter approach increases the remote acquire cost. We choose the former method since it results in smaller messages that are easier to pipeline in the node and NI and 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.

Network interface locks: With coherence information propagation already separated from mutual exclusion as described above, the communication layer is extended to provide support for mutual exclusion in the NI as well. Lock acquisition and release for mutual exclusion become communication system rather than SVM protocol operations, and no host processors other than the requester are involved.

The implementation of locks in the network interface firmware is similar to the algorithm used in the base protocol. 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 as described earlier. Associated with each lock is one timestamp, which is interpreted and managed by the protocol. The network interface does not need to perform any interpretation or operations on this timestamp, but the current

Application	Problem Size	Uniproc Time(sec)	Overall % Improvement	Data % Improvement	Lock % Improv
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	256^3 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 1.3. 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.

implementation requires that this piece of information be stored and transfered as part of the lock data structure in the network interface. 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. 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.

1.4.2 Experimental Testbed

We implement the network interface extensions on a cluster of Intel SMPs connected with Myrinet. 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 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[2].

Myrinet [11] 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 four nodes

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

are connected directly to an 8-way switch.

The communication library that we use on top of the Myrinet network is VMMC [12]. VMMC provides protected, reliable, user-level communication. VMMC uses variable size packets, and the maximum packet size is 4 KBytes. Thus messages smaller than 4 KBytes will be transferred with one packet. Data transfers from the host memory to the NI and from the NI to the network are pipelined even for the same packet. VMMC uses three software queues in each network interface: One for requests posted from the host processor, one for outgoing packets, and one for incoming packets. No other software queues are used in the communication layer.

Four major stages can be identified in the path from the sender to the receiver, dividing latency into four components: *SourceLatency* is the time between the first appearance of the send request for each packet in the NI's request queue and the completion of the DMA of the packet's data into the NI's memory. *LANaiLatency* is the time between the end of SourceLatency and the end of the NI's insertion of the packet into the network. *NetLatency* is the time between the end of SourceLatency and the point when the receiving NI gets the last word of the packet. *DestLatency* is the time between the last word's arrival at the destination NI and the completion of the destination NI DMA into its host's memory.

The one-way latency for one-word messages is about 18μs and the maximum available bandwidth about 95 MBytes/s. The overhead for an asynchronous send operation is about 2μs. The basic feature of VMMC that we use in this work is the remote deposit capability. Also, we have extended VMMC with the remote fetch and locking support described earlier. The resulting time for one 4 KByte page fetch operation is about 110μs (about 40μs for one word), as opposed to about 200μs without the remote fetch operation and one compute process per node.

1.4.3 Applications

To evaluate the system we use both original versions of several SPLASH-2 applications [53] as well as versions that have been restructured to improve performance on SVM systems [31]. Table 1.3 presents basic facts for the applications we use and summarizes the overall performance improvements. FFT, LU-contiguous, Ocean-rowwise[3], Barnes-original, Water-nsquared, and Water-spatial are the original SPLASH-2 applications. Radix-local and Barnes-spatial are restructured versions of the original SPLASH-2 applications [31]. The version of Raytrace we use eliminates a lock that assigns unique ids to rays, resulting in less locking. The restructured version of Volrend uses task stealing but the initial task assignment is such that it results in better load balancing and reduces the need for task stealing. The problem sizes we choose are close to the sizes of real-world problems. Table 1.3 presents the problem sizes along with the uniprocessor execution times.

[3]When using 4-way SMP nodes this version is practically equivalent to Ocean-contiguous in SPLASH-2.

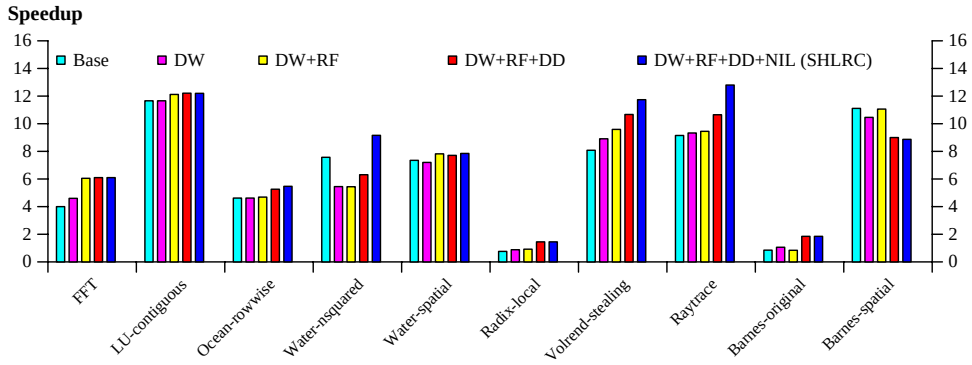

Figure 1.8. Application speedups. From left to right the bars for each application are (i) Base, (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

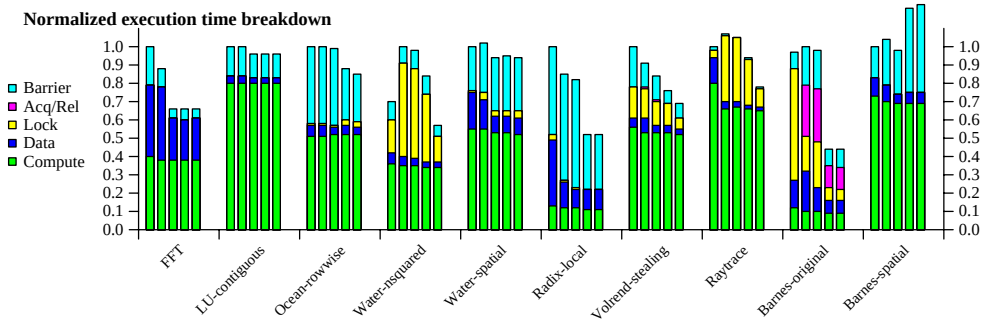

Figure 1.9. Normalized average execution time breakdowns for 16 processors. From left to right the bars for each application are (i) Base, (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

1.4.4 Results

We evaluate four different protocols. Each protocol successively and cumulatively eliminates the use of interrupts in an 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.

Figures 1.8 and 1.9 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. *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.

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 1.9). This is primarily due to the 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, both because it uses eager propagation and because it uses small messages.

Remote fetches of pages (RF): We see in Figures 1.8 and 1.9 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 and the related scheduling effects within an SMP.

Remote diff application (DD): As we see from the execution time breakdowns in Figure 1.9, direct diffs are particularly useful in the irregular applications that have a lot of synchronization and hence diffs: Radix-local, Barnes-original, Raytrace, Volrend 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.

Network interface locks (NIL): This version includes all NI extensions and is the final version of the protocol (GeNIMA). Compared to the previous version, GeNIMA substantially improves the performance of applications that use locks frequently. Table 1.3 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.

Summary: Overall, we see from Figure 1.10 that GeNIMA, which leverages all our general-purpose NI extensions to eliminate interrupts and asynchronous proto-

col 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. Our results also show that all three mechanisms are important in different applications, so all should be supported in the NI if possible.

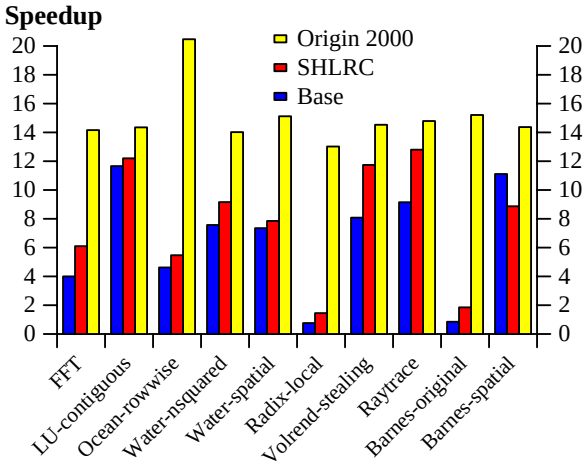

Figure 1.10. Application speedups for a hardware DSM machine (Origin-2000), and for the Base and the final GeNIMA protocols.

1.4.5 Remaining bottlenecks

Unfortunately, despite all the improvements in GeNIMA, Figure 1.10 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, especially those that have not been restructured. Let us now examine what the most significant remaining bottlenecks are.

Data wait time: Figure 1.9 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, there is an increase of less than 30% in the data wait time. The rest of the overhead is due to the still-remaining uncontended cost of the remote fetch op-

eration. Thus, FFT would benefit from bandwidth increases in the communication layer. Radix exhibits both these problems to a heightened extent, as well as a lot of false write-sharing due to the page granularity.

Lock synchronization: Previous work has identified locks and their dilation to be a major performance problem for SVM for many applications [29], [31]. The restructured versions of these applications dramatically reduce the number of locks and hence their effect on performance. In GeNIMA the applications that 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.

Barrier synchronization: For the restructured or other applications that are not dominated by locks, 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 1.4 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, Water, Volrend, 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 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.

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, a lot of shared pages need to be invalidated at barriers between major phases of computation. Table 1.4 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

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

range when more than a certain threshold of them need to be *mprotect*ed)), but more basic improvements may be necessary.

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 the memory and bus subsystems of the SMP nodes.

1.5 System Emulation

1.5.1 Emulation infrastructure

To emulate a cluster on top of an SGI Origin2000 we provide the layered architecture depicted in Figure 1.1 [19]. We implement a user-level communication layer that has been designed for low-latency, high-bandwidth cluster interconnection networks. We then port on this communication layer an existing SVM protocol that has been optimized for clusters with low-latency interconnection networks. At the highest level we use the SPLASH-2 applications. In the next few sections we describe our experimental platform in a bottom–up fashion.

Hardware platform: The base system used in this study is an SGI Origin 2000 [38], containing sixty-four 300MHz R12000 processors and running Cellular IRIS 6.5.7f.

The 64 processors are distributed in 32 nodes, each with 512 MBytes of main memory, for a total of 16 GBytes of system memory. The nodes are assembled in a full hypercube topology with fixed-path routing. Each processor has separate 32 KByte instruction and data caches and a 4 MByte unified 2-way set associative second-level cache. The main memory is organized into pages of 16 KBytes. The memory buses support a peak bandwidth of 780 MBytes/s for both local and remote memory accesses.

To place our results in context, we also present results from the cluster used in the implementation section. These results were originally presented in [30]. A direct comparison of the two platforms is not possible due to the large number of differences between the two systems. Our intention is to use the actual cluster statistics as a reference point for what today's systems can achieve.

Communication layer: The communication layer we use in this system is Virtual Memory Mapped Communication (VMMC) [12]. We implement VMMC on the Origin2000 (VMMC-O2000) providing a message passing layer on top of the hardware cache-coherent interconnection network. VMMC-O2000 provides applications with the abstraction of variable-size, cache-coherent nodes connected with a non-coherent interconnect and the ability to place thread and memory resources within each cc-NUMA node. VMMC-O2000 differs from the the original, Myrinet implementation [12] in many ways:

The original implementation of VMMC makes use of the DMA features of the Myrinet NIs: data are moved transparently between the network and local memory without the need for intervention by the local processors. The Origin2000's *Block Transfer Engine* offers similar, DMA–like services but user-level applications do not have access to this functionality. VMMC-O2000 instead uses Unix shared–memory regions to accomplish the sharing of memory between the sender and receiver, and *bcopy(3C)* to transfer data between the emulated nodes.

Asynchronous send and receive operations are not implemented in VMMC-O2000. Asynchronous transmission on the cluster was managed by the dedicated processor on the Myrinet NI; the Origin offers no such dedicated unit. Although similar possibilities do exist—for instance using one of the two processors in each node as a communication processor—these are beyond the scope of this work and we do not explore them here. Asynchronous operations in the SVM protocol are replaced with synchronous ones.

VMMC-O2000 inter-node locks are implemented as ticket-based locks. Each node is the owner of a subset of the system–wide locks. At the implementation level, locks owned by one node are made available to other nodes by way of Unix shared–memory regions.

Since nodes within the Origin system are distributed cc-NUMA nodes, they do not exhibit the symmetry found in the SMP nodes that have been used so far in software shared memory systems. For this reason, we extend VMMC-O2000 to provide an interface for distributing threads and global memory within each cluster node. Compute threads are pinned to specific processors in each node. Also, global

memory is placed across memory modules in the same node either in a round-robin fashion (default) or explicitly by the user.

Table 1.5 shows measurements for the basic operations of both VMMC and VMMC-O2000. We see that basic data movement operations are faster by about one order of magnitude in VMMC-O2000. Notification cost is about the same, but is not important in this context, as *GeNIMA* does not use interrupts. The cost for remote lock operations are significantly reduced under VMMC-O2000.

VMMC Operation	SAN Cluster	Origin2000
1-word send (one-way lat)	14μs	0.11μs
1-word fetch (round-trip lat)	31μs	0.61μs
4 KByte send (one-way lat)	46μs	7μs
4 KByte fetch (round-trip lat)	105μs	8μs
Maximum ping-pong bandwidth	96 MBy/s	555 MBy/s
Maximum fetch bandwidth	95 MBy/s	578 MBy/s
Notification	42μs	47μs
Remote lock acquire	53.8μs	8μs
Local lock acquire	12.7μs	7μs
Remote lock release	7.4μs	7μs

Table 1.5. 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.

Overall, VMMC-O2000 provides the illusion of a clustered system on top of the hardware cache-coherent Origin 2000. The system can be partitioned to any number of nodes, with each node having any number of processors. Communication within nodes is done using the hardware–coherent interconnect of the Origin without any VMMC-O2000 involvement. For instance, an 8-processor node consists of 4 Origin 2000 nodes, each with 2 processors that communicate using the hardware–coherent interconnect. Communication across nodes is performed by explicit VMMC-O2000 operations that access remote memory.

Protocol layer: We use *GeNIMA* [8] as our base protocol. *GeNIMA* uses general-purpose network interface support to significantly improve protocol overheads and narrow the gap between SVM clusters and hardware DSM systems at the 16-processor scale. The version of the protocol we use here is the one presented in [30] with certain protocol-level optimizations to enhance performance and memory scalability.

For the purpose of this work, we port *GeNIMA* on the Origin2000 (*GeNIMA*-O2000), on top of VMMC-O2000. The same protocol code runs on both WindowsNT and IRIX. This involved modifying three parts of the system: (i) thread creation, (ii) memory allocation and (iii) page fault handling. Otherwise, the core

protocol code is exactly the same on both platforms. Additionally, *GeNIMA* was extended to support a 64-bit address space.

A number of architectural differences arise as a result of faster communication, the cc-NUMA nodes, and contention due to wider nodes. To address these issues extend and optimize *GeNIMA*-O2000 as follows:

GeNIMA uses *mprotect* calls to change the protection of pages and invalidating local copies of stale data. *mprotect* is a system call that allows user processes to change information in the process page table. Since *mprotect* is a system call, it requires entering the kernel and can be expensive. Moreover, *mprotect* calls require invalidating TLBs of processors within each virtual node, and thus the cost is affected by the system architecture. More information on the cost of *mprotect*s will be presented next. *GeNIMA* tries to minimize the number of *mprotect*s by coalescing consecutive pages to a single region and changing its protection with a single call.

We enhance data sharing performance by adding page prefetching to the page fault handler. When a process needs to fetch a new shared page, the subsequent N pages are also read into local memory. Empirically, we determined that N=4 offers the best prefetching performance on *GeNIMA*-O2000. Ideally, this approach can reduce both the number of `mprotect()` calls and the number of page protection faults by increasing demands on network bandwidth. The average cost of an *mprotect* call and the overhead of a page fault on the Origin2000 are relatively high: $\approx$150μs and $\approx$37μs, respectively). Thus, prefetching not only takes advantage of the extra bandwidth in the system, but alleviates these costs as well. The actual prefetch savings are dependent on the applications' particular access pattern. In order to determine when prefetching is useful, "false" prefetches are tracked in a history buffer. A prefetch is marked as "false" when the additional pages that were fetched are not used. If a page has initiated a "false" prefetch in the past, the protocol determines that the current data access pattern is sufficiently non-sequential and disables prefetching for this page the next time it is needed.

We enhance barriers in two ways. The first change involves serializing large sections of the barrier code, while the second involves a more efficient ordering of the work to be performed.

This first change is surprisingly counter-intuitive. In the original *GeNIMA* implementation (*GeNIMA*-Myrinet) all processes within a node cooperate to perform the tasks of processing incoming diffs and updating the protection of the shared pages. One process in each node is selected as the barrier *leader*, while the others are designated as *followers*; the followers assist the leader by performing much of the barrier processing in parallel. *GeNIMA*-O2000 was changed to prohibit the followers from assisting in the processing — resulting in a complete serialization of the barrier-related work in each node. This approach eliminates concurrent *mprotect*s by processes in the same address space; this savings is significant for large nodes, as the cost of a single *mprotect* call roughly doubles for each additional processor issuing simultaneous requests (Figure 1.13). This approach is specific to *GeNIMA*-O2000, but is important for systems that will employ wide compute nodes.

The second change introduces an ordering step prior to the processing of the page invalidations. We use the *quick-sort* algorithm to order the updates by page number; groups of consecutive pages are then serviced together and the entire block is updated with a single *mprotect*.

The original *GeNIMA* protocol uses the *test-and-set* algorithm to implement inter-node locks. Although this approach works well for systems with small–scale SMP nodes, it is not adequate for systems with larger–scale, cc-NUMA nodes and leads to poor caching performance and increased inter-node traffic in *GeNIMA*-O2000. We have experimented with a number of lock implementations. Overall, ticket–based locks turn out to be the most efficient. For this reason we replace these locks w with a variant of the *ticket-based* locking algorithm that generates far less invalidation traffic and offers FIFO servicing of lock requests to avoid potential starvation.

Applications layer: We use the SPLASH-2 [53], [31] application suite. 64-bit addressing and operating system limitations related to shared memory segments prevent some of the benchmarks from running at specific system configurations. We indicate these configurations in our results with 'N/A' entries. FFT and LU 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 [24]. Radix, SampleSort, Volrend are the restructured applications from [31]. The version of WaterSpatial we use (WaterF) is the restructured version from [30]. Furthermore, we restructure FFT (FFTst) to stagger the transpose phase among processors within each node. This allows FFT to take advantage of the additional bandwidth available in the system. Since FFTst outperforms the original version, we only present results for this optimized version.

1.5.2 Impact of Fast Interconnection Networks

In this Section we discuss the impact of faster interconnection networks on the performance of shared virtual memory. We use system configurations with 4 processors per node since most of the work so far with SVM on clusters has used Quad SMP nodes and we would like to place our results in context. This allows us to loosely compare our results with an actual cluster [30]. The two platforms cannot be compared directly due to a number of differences in the micro-processor and memory architectures they use, they offer invaluable insight in the needs of future clusters. A more detailed analysis of our results is presented in [19].

Figure 1.11 presents execution time breakdowns for each application for *GeNIMA*-Myrinet and *GeNIMA*-O2000 for 32– and 64–processors. Table 1.6 presents speedups and individual statistics for the applications run on *GeNIMA*-O2000. In particular, FFT is bandwidth–limited on the cluster. By modifying the original application to stagger the transpose phase, the speedup of FFTst is quadrupled with an impressive 23.8 on the 32-processor system.

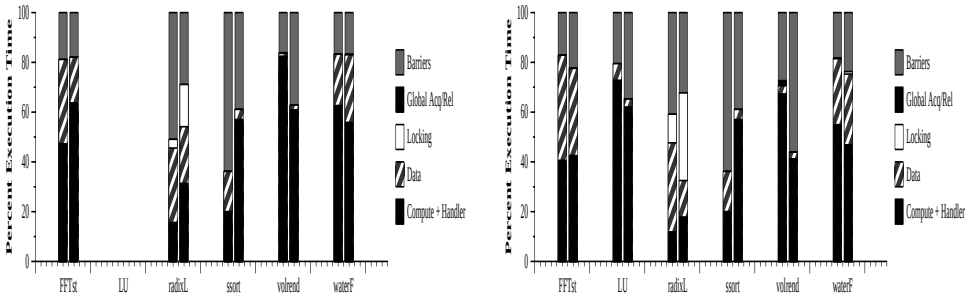

Figure 1.11. Execution time breakdowns for each application. The leftmost graph provides breakdowns for a 32-processor system; the rightmost graph provides breakdowns for a 64-processor system. The left bar in each graph refers to *GeNIMA*-Myrinet, whereas the right bar refers to the *GeNIMA*-Origin2000. (FFT is the original version under *GeNIMA*-Myrinet and the staggered version under *GeNIMA*-O2000.)

	FFTst	LU	radixL	ssort	volrend	waterF
32 Proc (8x4)	23.8	N/A	1.9	6.7	17.8	9.2
64 Proc (16x4)	26.6	32.9	1.0	N/A	23.1	14.5

Table 1.6. Parallel speedup of benchmarks running under *GeNIMA*-O2000.

Remote data wait: Overall, the faster communication layer and our optimizations provide significantly improved data wait performance. The direct remote read/write operations in *GeNIMA* eliminate protocol processing at the receive side (the page home) and make it easier for protocol performance to track improvements in inter-

	Data Time		**Barrier Time**		**Lock Time**		***mprotect* Time**		***diff* Time**	
	Myri	O2000	Myri	O2000	Myri	O2000	Myri	O2000	Myri	O2000
FFTst	33.9	18.3	18.9	17.9	–	–	15.0	6.1	0.1	0.6
LU	N/A	N/A	N/A	N/A	–	–	N/A	N/A	N/A	N/A
radixL	29.9	22.8	30.9	28.9	3.5	17.0	16.7	6.2	18.4	10.3
ssort	N/A	N/A	N/A	N/A	–	–	N/A	N/A	N/A	N/A
volrend	1.1	1.9	16.1	37.2	–	–	0.5	2.5	0.3	2.5
waterF	20.6	27.2	16.7	16.7	0.1	0.2	14.6	11.1	0.2	0.7

Table 1.7. Performance of *GeNIMA* on the SAN Cluster (Myri) and on the emulated system (O2000). The results are for for 32-processor systems (8 nodes, 4 processors/node). All results are indicated as percentages of total execution time.

	Data Time		Barrier Time		Lock Time		*mprotect* Time		*diff* Time	
	Myri	O2000	Myri	O2000	Myri	O2000	Myri	O2000	Myri	O2000
FFTst	42.2	35.1	17.1	22.4	–	–	6.8	6.6	0.4	0.2
LU	6.7	3.2	20.5	34.8	–	–	1.3	1.4	0.0	0.3
radixL	35.8	14.7	40.8	32.3	11.5	35.2	7.1	10.2	14.7	4.4
ssort	16.1	4.1	63.8	38.9	–	–	14.4	2.1	12.8	0.3
volrend	3.3	2.6	27.3	56.1	–	–	1.1	7.0	0.7	0.8
waterF	26.5	28.4	18.3	23.7	0.3	1.0	10.6	11.2	0.7	0.8

Table 1.8. Performance of *GeNIMA* on the SAN Cluster (Myri) and on the emulated system (O2000). The results are for for 64-processor systems (16 nodes, 4 processors/node). All results are indicated as percentages of total execution time.

connection network speed. In the majority of the applications data–wait time is reduced to at most 20% of the total execution time. Also, although a direct comparison is not possible, data wait time is substantially improved compared to the SAN cluster. Applications where prefetching is effective receive an additional benefit in reduced page-fault interrupt costs: anywhere from 30% to 80% of the total page faults are typically eliminated in prefetching, with the reduction in page faults resulting in a savings of 15% to 20% in total execution time on *GeNIMA*-O2000.

Barrier synchronization: The faster communication layer and our barrier–related optimizations result in moderately improved barrier synchronization performance for most of the applications. Protocol costs are the main overhead and faster interconnects are unlikely to benefit barrier costs in future clusters, unless they are accompanied by protocol, application, and/or operating system changes as well to improve *mprotect* costs. The barrier cost on *GeNIMA*-O2000 for most applications is less than 30% of the total execution time. With 64 processors, barrier cost does not scale well with the problem sizes we use here. Future work should address the impact of problem size on the scalability of these overheads.

Lock synchronization: Lock synchronization costs are generally effected by the cost of invalidating pages and the resulting dilation of the critical sections. For this reason, although lock acquires and releases have lower overheads in *GeNIMA*-O2000 (in Table 1.5), the higher *mprotect* costs dominate (Tables 1.7, 1.8). Applications that rely heavily on lock synchronization such as radixL exhibit higher lock costs with *GeNIMA*-O2000. This change is due to radixL's small critical regions (several dozen loads and stores) and the larger locking overheads (under contention) on *GeNIMA*-O2000: more time is spent acquiring and releasing the locks than is actually spent inside the critical region. The result is that the application spends a comparatively small percentage of its time inside in the critical region, thus reducing the likelihood that acquires are issued while the lock is still local. Developing techniques to limit the effect of *mprotect* cost on lock synchronization is critical for further reductions in lock synchronization overheads.

1.5.3 Impact of Wide, CC-NUMA Nodes

In this section we attempt to answer the question of future performance on software shared memory clusters by demonstrating the performance of GeNIMA-O2000 on wide (8- and 16-processor) cc-NUMA nodes. A more detailed analysis of our results is presented in [19]. Table 1.9 summarizes the parallel speedups for the 4-, 8- and 16-processor configurations. Table 1.11 shows the major protocol costs as percentage of the total execution time at the 64–processor scale.

	Parallel Speedup					
	32 Processors			**64 Processors**		
	4	8	16	4	8	16
FFTst	23.8	16.3	5.7	26.6	28.7	12.4
LU	N/A	18.5	N/A	32.9	32.5	N/A
radixL	1.9	1.8	N/A	1.0	1.2	N/A
volrend	17.8	17.7	16.1	23.1	24.8	17.6
waterF	9.2	4.9	1.3	14.5	8.5	2.7

Table 1.9. Parallel speedups for 64-processor configurations with varying node widths.

	Avg. *mprotect* Cost		
	4	8	16
FFTst	127.8μs	199.5μs	622.1μs
LU	55.4μs	43.6μs	N/A
radixL	325.8μs	296.4μs	N/A
volrend	404.2μs	402.5μs	569.4μs
waterF	87.1μs	163.4μs	674.9μs

Table 1.10. Average *mprotect* cost for 64-processor configurations with varying node widths.

Local data placement: The NUMA effects of the nodes in *GeNIMA*-O2000 are negligible up to the 4-processor level. However, as wider nodes are introduced, processors in each node exhibit highly imbalanced compute times. Generally, applications that exhibit significant intra–node sharing of global data that are fetched from remote nodes, such as FFTst and waterF, incur highly imbalanced compute times. On the other hand, applications that either exhibit little intra–node sharing of global data, and/or have low data wait overheads overall, such as LU, volrend, and radixL, are not greatly affected. To explain this behavior we need to examine what happens when pages are fetched from remote nodes. Certain processors in each node fetch more shared pages than others. Due to the *bcopy* method used to

	Data Time (%)			Barrier Time (%)			Lock Time (%)			*mprotect* Time (%)		
	4	8	16	4	8	16	4	8	16	4	8	16
FFTst	35.1	21.1	17.8	22.4	28.7	19.0	–	–	–	6.6	9.9	12.6
LU	3.2	3.2	N/A	34.8	35.0	N/A	–	–	–	1.4	0.9	N/A
radixL	14.7	13.5	N/A	32.3	31.4	N/A	35.2	28.2	N/A	10.2	4.4	N/A
volrend	2.8	2.6	1.8	56.1	52.8	62.1	–	–	–	7.0	2.3	1.0
waterF	28.4	17.6	12.2	23.7	31.6	44.8	1.0	0.2	0.0	11.2	10.7	11.1

Table 1.11. Protocol overhead for 64-processor configurations with varying node widths. (Percentages indicate percent of total execution time.

fetch pages in VMMC-Origin, new pages are placed in the cache of the processor that performs the fetch operations. Thus, due to the NUMA node architecture, processors in each node exhibit varying local memory overheads depending on the page fetch patterns. An examination of data access patterns in Figure 1.12 shows that in FFTst processors with the lowest execution times also perform the highest number of remote data fetches. The processor that first fetches each page places data in its second–level cache, resulting in higher local memory access overheads for the rest of the processors in the same *GeNIMA*-O2000 node. Modifying FFTst, such that processors in each node compete less for fetching pages, reduces imbalances and improves average compute time from 45% to 60% of total execution time (Figure 1.12). However, processors in each node still exhibit varying memory overheads. Thus, dealing with NUMA effects in wide nodes is an important problem that future SVM protocol and possibly application design has to address.

Remote data wait: In contrast to local memory access overheads, remote data wait time is reduced across applications by up to 57%. Table 1.11 shows the percentage of execution time associated with remote data fetches, and each of the other components of the protocol overhead. The number of remote fetches are reduced on 16-processor nodes by 19% on average over 4-processor nodes. This leads to an overall reduction in remote fetch times of 15% to 20% for all benchmarks, except LU where data wait time is small and accounts only for about 3% of the total execution time.

Lock synchronization: Lock synchronization also benefits from wider nodes, achieving significant reductions in overhead with 8–processor nodes, and showing a moderate improvement at the 16-processor level. Local lock acquires and releases are very inexpensive in *GeNIMA* (equivalent to a few instructions). Wider nodes incur more local than remote acquires and all aspects of lock overhead are improved with 8-processor nodes.

Barrier synchronization: In contrast to the gains observed in remote data access and lock synchronization, barrier performance is generally unchanged with 8-processor nodes. This is mainly due slightly higher intra– and inter–node imbalances in compute times. With wide nodes, barrier performance is dominated by

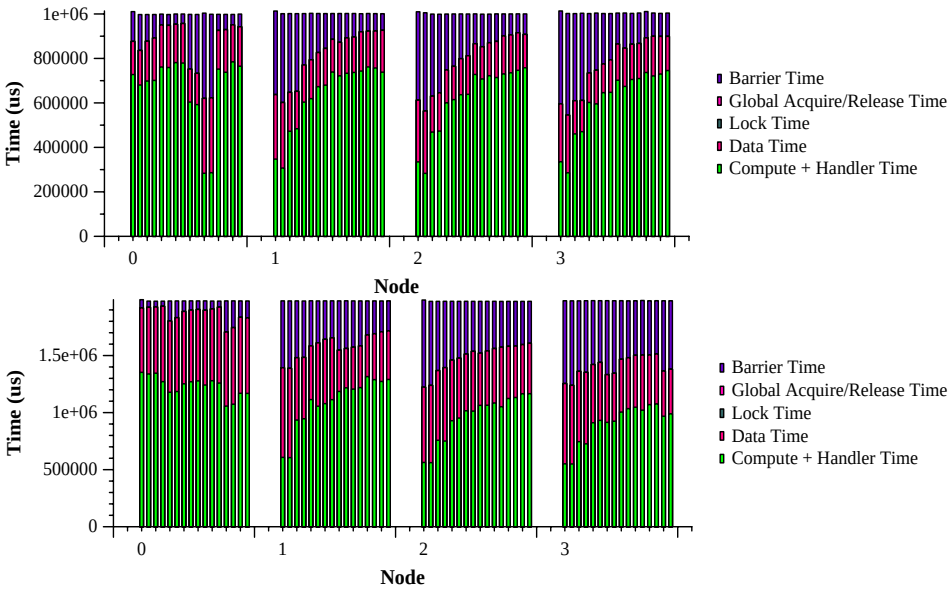

Figure 1.12. Breakdown of protocol overhead in FFTst, before (top) and after (bottom) balancing.

higher imbalances and high *mprotect* costs, as explained next.

mprotect **costs:** On the cluster *mprotect* calls typically cost between 40–80μs. In contrast, the cost of downgrading a page (e.g., moving from a *read-write* to an *invalid* state in a barrier or unlock) is between 250μs and 1ms in *GeNIMA*-O2000. The cost of *mprotect* on IRIX stems from four different sources: (i) broadcasting TLB invalidations to multiple processes in the same share group; (ii) changing the protection on large portions of the address space, necessitating the modification of multiple page table entries; (iii) changing the protection of a page to a state that differs from its neighbors, resulting in the breaking of the larger protection region into three smaller ones; and, (iv) locking contention in the kernel while executing multiple, unrelated processes (in our case, in *mprotect* code). Although we ensure that (i)–(iii) are minimized in *GeNIMA*-O2000, *mprotect* costs remain higher the comparable costs in the cluster. This leads us to suspect that (iv) is the reason for the additional overhead[4]. Figure 1.13 shows the average cost as a function of the number of processors issuing *mprotect* calls. We see that the overhead for downgrading (invalidating) pages increases with the number of processors. This aspect however of *GeNIMA*-O2000, bears further investigation.

[4]To verify this, we perform a set of experiments that are somewhat intricate so we do not describe them fully. However, an effort was made to try and capture the desired system behavior.

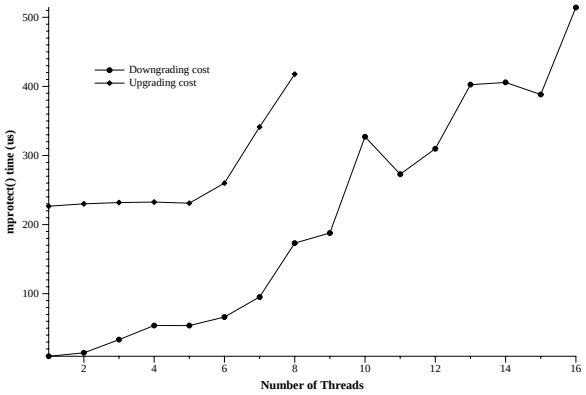

Figure 1.13. Average *mprotect* cost for both upgrading and downgrading single pages. Threads in this experiment are competing with each other; although they *mprotect* different pages, they belong to the same share group. This results in TLB invalidations for all processes, as well as contention in kernel data structures.

1.6 Discussion on Methodology

Comparing the simulation, implementation, and emulation results, the observations are interesting:

Simulation is able to identify the major SVM bottleneck on modern clusters: (i) Interrupts: Reducing the cost of interrupts in the system can improve performance significantly. (ii) System bandwidth: Providing high bandwidth is also important, to keep up with increasing processor speeds. (iii) Degree of clustering: Up to the scale we examine, adding more processors per node helps in almost all cases. In applications where performance does not increase quickly with the cluster size, scaling of other system parameters, such as memory bus and I/O bandwidth, can have the desirable effects.

Then, we redesign the SVM protocol to avoid all asynchronous protocol processing and provide an implementation on a state–of–the–art cluster. The implementation shows that our simulation results with respect to interrupts are accurate and that large performance gains can be achieved. Finally, to see how future clusters will actually look, we explore the use of faster networks and wide nodes based on the simulation results that indicate bandwidth and clustering to be the other two major system parameters. Since actual clusters do not provide the desired architectural features we use system emulation.

In two out of the five applications we use (FFT, Volrend) we see that emulation at the 32–processor scale results in parallel efficiencies (parallel speedup/ideal speedup) very close to the ones predicted by the simulator at the 16–processor scale, if interrupt and bandwidth related overheads are eliminated. The parallel efficiency for FFT and Volrend as computed by the simulation results at the 16–processor

scale (Table 1.2) are: 0.83 and 0.70 respectively. The same numbers calculated from the emulation results (Table 1.6, using an ideal speedup of 32) are: 0.75 and 0.74 respectively. Also, although we were not able to run LU at the 32–processor scale, the performance at 64–processors suggests that the efficiency at 32–processors would be close to the 0.72 of the simulation results for the ideal configuration.

In the other two applications (radixL, waterF), the emulation results are far from the simulation results. This is mainly due to the fact that these applications are affected more by the operating system (mainly *mprotect* contention) and processor architecture (data placement and memory bus contention) side effects that are not modeled by the simulator in detail. Finally, the emulation results reveal that at larger than 32–processor scales with cc-NUMA nodes, existing SVM protocols do not provide the expected performance, even with aggressive interconnects and further research is required.

Overall, we find that simulation is successful in identifying future trends. It is able to perform impressively detailed predictions for cases that are modeled properly. On the other hand, our simulation infrastructure inadvertently oversees certain aspects of the system architecture. On the other hand, system implementation and emulation help understand and address issues that may not seem as important at a first glance and provide the certainty that no aspect of the system architecture has been overlooked. They are also able to quantify precisely the benefits of protocol and architectural changes, the remaining overheads, and to verify the next set of problems to solve.

Finally, in doing this work we find that restructuring applications is an area that can make a big difference. Understanding how an application behaves and restructuring it properly can dramatically improve performance far beyond the improvement in system parameters or protocols [31]. This however, is not always easy, and, unfortunately, not many tools are available in parallel systems to help easily discover the cause of bottlenecks and obtain insight about application restructuring needs, especially when contention is a major problem as it often is in commodity-based communication architectures. Architectural simulators are one of the few tools that can currently be used to understand how an application behaves in detail.

1.7 Related Work

Parts of this work have been previously presented in [9], [8], [19].

Our simulation work is similar in spirit to some earlier studies, conducted in [41], [23], but in different context. In [41], 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. For SVM, we find these parameters to not be so important since their cost is usually amortized over page granularity. Applications were found to be quite tolerant to latency and

bulk transfer bandwidth in the split-C study as well. In [23], Holt et al. find 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 there (unlike in [41]) since it is very small. In [32], Karlsson et al. find that the latency and bandwidth of an ATM switch is acceptable in a clustered SVM architecture. In [35] a Lazy Release Consistency protocol for hardware cache-coherence is presented. In a very different context, they find that applications are more sensitive to the bandwidth than the latency component of communication. Several studies have also examined the performance of different SVM systems across multiprocessor nodes and compared it with the performance of configurations with uniprocessor nodes. Erlichson et al. [16] find that clustering helps shared memory applications. Yeung et al. in [54] find this to be true for SVM systems in which each node is a hardware coherent DSM machine. In [5], they find that the same is true in general for all software SVM systems, and for SVM systems with support for automatic write propagation.

Unlike our implementation, 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. For page-based SVM, this approach has been investigated with SMP nodes in Cashmere-2L [36], where the compute processors poll the network queues for incoming messages via code instrumentation on program back-edges. They find that polling and interrupt based asynchronous protocol processing performs comparably on their system. A similar polling method was also used and compared with interrupts in [57], with similar results. Our approach avoids the need for interrupts or polling altogether. Related work in network interface support for SVM has discussed how NIs can be used for several purposes: (i) To improve the performance of traditional send and receive communication. This type of support has been exploited in many SVM projects [17], [27], [36], [56], [50], [46], [45], [47] and is also used in our base system, HLRC-SMP [45]. (ii) To perform SVM protocol processing. 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 [56], where an Intel Paragon was used to move protocol processing into the network interface processor. In system area networks with programmable NI processors, the NI processor is usually much slower compared to the compute processor than in the Paragon and it does not have good enough access to main memory to perform protocol processing efficiently. In these cases, a compute processor in an SMP node can be reserved for protocol processing alone. This approach was examined in [18], where it was found that the benefit of this approach is small due to poor utilization of that processor, especially compared to a system that uses all processors both for computing and protocol processing. The amount of protocol processing involved in SVM systems with SMP nodes was examined through an earlier simulation study [32] and other research, and is found to be small compared to other system overheads. To

transparently handle remote data and synchronization. In this case the remote compute processor is not involved in handling message requests, e.g. to simply provide or deposit data or to obtain a lock, but remains responsible for all protocol processing and SVM-specific operations, e.g. maintaining timestamps, invalidating pages. This is the approach we have taken. Previous work in this area relies on more specialized network interface and/or network support, whereas the mechanisms we implement are broader and more commodity-oriented. The Automatic Update Release Consistency (AURC) [27] 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 [10] to avoid diff computation and application in a home-based SVM protocol. The Cashmere system [36], [50] uses the fine-grained remote write capability of the DEC Memory Channel network interface. The memory bus is not snooped by the NI, but 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 [37], but not in real implementation. The use of fine-grained, implicit write propagation has costs (snooping, write-through caching, or instrumentation overhead) and the benefits over all-software home-based protocols have been found to be present but not large for AURC [26] and to often disappear or become worse with SMP nodes due to the bus traffic imposed by write-through caching [6]. 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], resulting in a small performance advantage for most applications over all-software protocols on systems with SMP nodes. A discussion on how the remote write access capabilities of VM-based networks can be used in SVM systems is provided in [22].

Our emulation work bears similarities with the MGS system [55]. The MGS system examined clustering issues for SVM systems and in particular different partitioning schemes on Alewife, a hardware cache-coherent system [1]. Unlike our work, the authors use a TreadMarks-like protocol and they find both synchronization and data movement across nodes to be a problem. The SVM protocol they use is tuned for traditional clusters with slow interconnection networks (they make use of interrupts and assume high overheads in communication initiation). The SoftFLASH system [17] provided a sequentially consistent software shared memory layer on top of 8–processor SMP nodes. They find that the cost for page invalidations within each node is very high.

1.8 Conclusions

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, overall, the achievable application performance today is limited primarily by interrupt cost and then by node to network bandwidth.

Host overhead and NI occupancy appear less important to improve relative to processor speed. If interrupts are free and bandwidth high relative to the processor speed, then the achievable performance approaches the best performance in most cases.

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. We have prototyped these extensions in the programmable network interface of the Myrinet network and evaluated their impact on application performance on a network of Intel Pentium Pro SMPs. We found that: (i) The proposed communication and protocol extensions improve performance substantially for our suite of ten applications. Overall, 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). Data wait time improves up to 45% and lock time up to 60%. (ii) Different applications benefited greatly from different NI features, indicating that all three should be supported. (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. We find synchronization cost to be the most important protocol overhead for improving overall application performance further.

Then, we have examined the implications of building software shared memory clusters with interconnection networks that offer latency and bandwidth comparable to hardware cache–coherent systems and wide, cc-NUMA nodes. We used an aggressive hardware cache-coherent system to investigate the impact of both of these architectural features on SVM performance. We ported a shared memory protocol that has been optimized for low-latency, high-bandwidth system area networks on a 64–processor Origin 2000 and provided a number of optimizations that take advantage of the faster interconnection network. Our approach eliminated the need for simulation and allowed direct execution of the same code used on the SAN cluster on top of the emulated communication layer. Using simulation, although advantageous in some cases, tends to overlook important system details that may shift bottlenecks to system components that are not modeled in detail. We found that SVM protocols can only partly take advantage of faster communication layers. In the applications we examine data wait time is typically reduced to less than 15% of the total execution time. Moreover, certain bandwidth intensive applications that were performing adequately or even poorly on current SAN clusters can achieve higher speedups; in particular, FFT, after restructuring that takes advantage of the additional system bandwidth, achieves an impressive 23.6 speedup on 32 processors. Barrier costs do not scale well to the 64–processor level for some applications for the problems sizes we examine. Finally lock synchronization costs are not able to benefit from the faster lock acquire and release times, due to the increased cost of

mprotect calls. Using wide (8– and 16–processor) cc-NUMA nodes has an impact on intra-node synchronization and data placement. Most importantly, the NUMA features of the nodes result in imbalances and the increased number of processes per node results in higher *mprotect* costs. These effects combine to reduce system performance and need to be addressed in protocol design and implementation.

In terms of the methodology, we find that system simulation, implementation, and emulation are complementary approaches and one cannot replace others. Our simulation results have successfully revealed high–level trends. The implementation was able to precisely quantify the benefits from eliminating asynchronous protocol processing and to demonstrate that simple NIC support is adequate to achieve this. System emulation verified and quantified the importance of bandwidth for system performance in systems that do not yet exist and will be build in the future but more importantly it revealed a number of side effects that have been inadvertently overlooked by our simulation infrastructure. Overall, combining the three approaches has provided invaluable insight and has resulted in extensive infrastructure that can be used for future work in the area.

1.9 Acknowledgments

We would like to thank the numerous people people that have contributed directly or indirectly to various pieces of this work: Yuqun Chen, Courtney Gibson, Liviu Iftode, Dongming Jiang, Scott Karlin, Sanjeev Kumar, Kai Li, Cheng Liao, Brian O'Kelley, Rudrajit Samanta, Limin Wang, Bill Wichser, Xiang Yu, and Yuanyuan Zhou.

1.10 Bibliography

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[2] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado*, December 1995.

[3] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Seattle, Washington, Mar. 1990.

[4] R. Bianchini, L. Kontothanassis, R. Pinto, M. D. Maria, M. Abud, and C. Amorim. Hiding communication latency and coherence overhead in software dsms. In *The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 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. 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] 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, Apr. 1994.

[11] 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, Feb. 1995.

[12] 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*, Aug. 1997.

[13] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, pages 388–396, April 1997.

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

[15] T. Eicken, D. Culler, S. Goldstein, and K. 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.

[16] A. Erlichson, B. Nayfeh, J. P. Singh, and K. Olukotun. The benefits of clustering in shared address space multiprocessors: An applications-driven investigation. In *Supercomputing '95*, pages 176–186, 1995.

[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] C. Gibson and A. Bilas. Shared virtual memory clusters with next–generation interconnection networks and wide compute nodes. In *Proceedings of the 8th International Conference on High Performance Computing (HiPC). Hyderabad, India.*, Dec. 2001.

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

[21] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. Devries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, P. McHardy, S. Srblijic, M. Stumm, Z. Vranesic, and Z. Zilac. The NUMAchine Multiprocessor. In *The 2000 International Conference on Parallel Processing (ICPP2000)*, Toronto, Canada, Aug. 2000.

[22] 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 computing. In *Newsletter of the IEEE CS Technical Committee on Computer Architecture*, pages 28–33, Mar. 1997.

[23] 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.

[24] 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.

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

[26] L. Iftode, M. Blumrich, C. Dubnicki, D. Oppenheimer, J. P. Singh, and K. Li. Implementation and performance of shared virtual memory protocols on shrimp. In *Seventh Workshop on Scalable Shared Memory Multiprocessors (in conjunction with the 25th Annual International Symposium on Computer Architecture)*, June 1998.

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

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

[29] 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.

[30] D. Jiang, B. Cokelley, X. Yu, A. Bilas, and J. P. Singh. Applicaiton scaling under shared virtual memory on a cluster of smps. In *The 13th ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[31] 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.

[32] 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*, Feb. 1996.

[33] 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.

[34] P. Keleher, A. 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, Jan. 1994.

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

[36] 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.

[37] 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*, Feb. 1996.

[38] 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.

[39] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of the Stanford DASH multiprocessor. Technical Report CSL-TR-89-403, Stanford University, December 1989.

[40] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[41] 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, Nov. 1996.

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

[43] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for myrinet. In *Supercomputing '95*, 1995.

[44] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance. In *PC-NOW Workshop, held in parallel with IPPS/SPDP98, Orlando, USA*, March 30 – April 3 1998.

[45] 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.

[46] 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, Jan. 1998.

[47] 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, Mar. 1996.

[48] 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.

[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, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, Oct. 1997.

[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, Sept. 1993. USENIX.

[52] H. Tezuka, A. Hori, and Y. Ishikawa. PM: a high-performance communication library for multi-user parallel environments. Technical Report TR-96015, Real World Computing Partnership, 1996.

[53] S. 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.

[54] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: a multigrain shared memory system. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1996.

[55] D. Yeung, J. Kubiatowicz, and A. Agarwal. Multigrain shared memory. *ACM Transactions on Computer Systems*, 18(2):154–196, May 2000.

[56] 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*, Oct. 1996.

[57] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. Toonen, I. Schoinas, M. 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.