Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems

Angelos Bilas
Dept. of Elec. and Comp. Eng.
10 King's College Road
University of Toronto
Toronto, ON M5S 3G4, Canada
bilas@eecg.toronto.edu

Cheng Liao
Dept. of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544, USA
cliao@cs.princeton.edu

Jaswinder Pal Singh
Dept. of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544, USA
jps@cs.princeton.edu

Abstract

The performance of page-based software shared virtual memory (SVM) is still far from that achieved on hardware-coherent distributed shared memory (DSM) systems. The interrupt cost for asynchronous protocol processing has been found to be a key source of performance loss and complexity.

This paper shows 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.

We find that the performance improvements are substantial, bringing performance on a small-scale SMP cluster much closer to that of hardware-coherent shared memory for many applications, and we show the value of each of the mechanisms in different applications. Application performance improves by about 37% on average for reasonably well performing applications, even on our relatively slow programmable NI, and more for others. We discuss the key remaining bottlenecks at the protocol level and use a firmware performance monitor in the NI to understand the interactions with and the implications for the communication layer.

1 Introduction

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 the three layers that sit between the problem to be solved and the hardware: 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 [9].

Since SVM was first proposed [35], much research has been done in improving the protocol layer by relaxing the memory consistency models [5, 32], by improving the communication layer with low-latency, high-bandwidth, user-level communication [20, 13, 17, 38, 15, 47, 4, 39, 26, 33, 34, 51, 6], and by taking advantage of the two-level communication hierarchy in systems with multiprocessor nodes [18, 7, 46, 8, 40, 50]. Recently, the application layer has also been improved, by discovering application restructurings that dramatically improve performance on SVM systems [28].

However, Figure 1 shows that parallel performance is still not satisfactory. Speedups on a 16-processor, all-software, home-based SVM system running on a Myrinet-connected cluster of Pentium Pro Quad SMPs [40] are still substantially lower than those achieved on hardware-coherent machines for the same problem sizes, even with restructured applications. The processors used by the two systems are different (although of similar generations and with similar clock speeds), so a direct comparison cannot be made, but the implication is clear. This research develops techniques to improve SVM performance on clusters by exploiting the synergy between the protocol layer and a commodity communication layer.

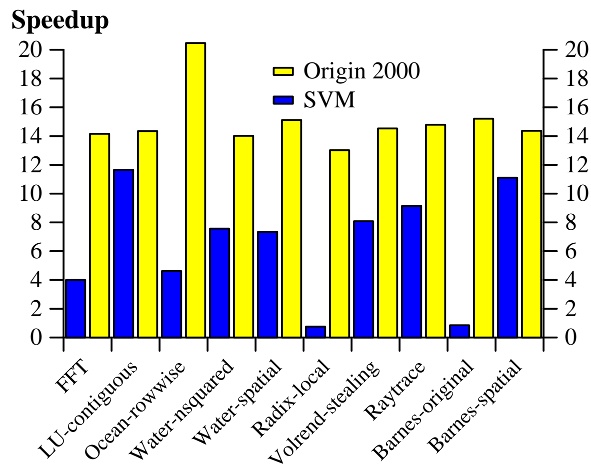

Figure 1: Application speedups for a hardware distributed shared memory (DSM) machine (the SGI Origin 2000) and an SVM system (our Base protocol, see Section 2).

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. Previous work [10] has shown that the cost of interrupts used for asynchronous message handling and/or protocol processing is one of the most important bottlenecks in modern SVM clusters. Using polling instead of interrupts introduces different types of overheads, and the tradeoff between the two methods is unclear for SVM.

The insight behind the communication-layer mechanisms and the synchronous home-based lazy release consistency (GeNIMA) protocol we present in this paper is to decouple protocol processing from message handling. By providing support in the network interface for simple, generic operations, asynchronous message handling 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. For our SVM protocols, they constitute a minimal set of operations needed to avoid involving the processor. We do not examine more sophisticated message handling support, such as scatter-gather, in the NI, though this may help performance further as we shall discuss later.

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 [34, 26], 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 [12, 20, 31, 24] and the recent Virtual Interface Architecture (VIA) standard [16] support some of these or similar types of operations.

We find that: (i) The proposed protocol extensions improve performance substantially for our suite of ten applications. Performance improves by about 38% on average for reasonably well performing applications (and up to 120% for applications that do not perform very well under SVM even afterward). (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. Control messages (e.g. lock requests) getting stuck behind data traffic in NI queues is a significant source of performance loss in some applications.

The rest of the paper is organized as follows. Section 2 presents the proposed NI mechanisms and the protocol changes needed to take advantage of them. Section 3 presents the performance results for each NI mechanism and the related protocol changes on our platform and examines the remaining bottlenecks. Section 4 looks in more depth at the behavior of the communication layer. Section 5 discusses the main limitations of this work and some directions for future work. Section 6 presents previous and related work. Section 7 presents concluding remarks.

2 Network Interface and SVM Protocol Extensions

Base protocol: The base SVM system that we use is HLRC-SMP [7, 40], an all-software home-based lazy release consistency system extended to efficiently support and take advantage of SMP nodes. Both HLRC-SMP and older LRC protocols use software twinning and diffing to solve the multiple-writer problem, but with different schemes for propagating and merging diffs (updated data). The main difference in HLRC-SMP is that each shared page is allocated to a home node, and that all updates to this page are always propagated to the home. When the updated version of a page is required in a node different than the home of the page, the full page is fetched. Thus, the home of each shared page functions as a centralization point for data updates and requests on this page. The protocol uses any number of compute processes in each node (here, four per each four-processor node), and an additional floating process to handle protocol requests. Hardware coherence and synchronization are exploited within a node, while still exploiting laziness via separate page tables, and performance is most often better than that with uniprocessor nodes [7, 40].

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.

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.

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 struc-

tures via memory bus snooping, code instrumentation, or specialized NI support [12, 20, 34, 26]. 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 [12, 20, 31, 24, 16].

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

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

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

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

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

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

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. Again, the remote fetches must be for contiguous data in our implementation, but there is little occasion for non-contiguous fetches in the protocol.

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

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

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

Interestingly, direct diffs require that pages be fetched with remote fetches and retries rather than by involving the home processor. The reason is that since direct diffs do not interrupt or involve a host processor at the home at diff application, home processors do not know when they have the

updated version of a page and are ready to service queued page requests. Remote fetches do not rely on home processors having this knowledge, since the requester retries whenever it fails to fetch the right version of a page. This is why we will present results for direct diffs only after presenting those for remote fetch.

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.

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

The implementation of locks in the network interface firmware is similar to the algorithm described earlier. 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 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. Flags are used to ensure that invalidations for each interval have reached the node before they are applied. The only requirement in the communication layer is in-order delivery of messages between two processes. There are no requirements for global or other strict forms of ordering.

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

3 Performance Results

3.1 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[1].

Myrinet [13] 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 are connected directly to an 8-way switch.

The communication library that we use on top of the Myrinet network is VMMC [15]. 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.

VMMC includes a performance monitoring tool [36] that gathers network packet-level data. The core of the monitor is firmware run by the NI processor. Monitoring the system at the NI firmware level provides low-overhead access to crucial determinants of performance bottlenecks.

Finally, since our research infrastructure is currently using Windows NT as the operating system, we have ported

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

Application	Problem Size	Uniproc Time(sec)	Overall(%)	Data Time(%)	Lock Time(%)
FFT [2, 48, 49]	4M points	4.6	52.50	45.37 (44.92)	0.00
LU-contiguous [48, 49]	4096x4096 matrix	935.9	4.63	13.46 (11.20)	0.00
Ocean-rowwise [14, 45, 28]	514x514 ocean	248.3	18.40	21.76 (19.26)	9.21
Water-nsquared [48]	4096 molecules	360.6	21.00	15.26 (46.17)	62.76
Water-spatial [48]	15625 molecules	157.2	6.80	41.60 (41.80)	9.69
Radix-local [11, 23, 49]	4M keys	5.9	90.79	26.76 (27.00)	53.21
Volrend-stealing [37, 48, 28]	256x256x256 cst head	13.2	45.30	43.81 (42.44)	50.44
Raytrace [44, 48]	256x256 car	29.8	39.89	2.52 (50.03)	59.01
Barnes-original [3, 22, 43]	32K particles	47.7	117.65	41.07 (68.25)	1.98
Barnes-spatial [28]	128K particles	219.2	-20.16	40.99 (37.70)	33.84

Table 1: 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.

the final GeNIMA protocol on the same hardware but under Windows NT. However, intermediate protocol versions are not available under Windows NT. This port involves a number of non-trivial changes in the communication and the protocol layers, motivated in part by the use of threads rather than separate processes within a node. Some experiments to understand performance issues in the later stages of the GeNIMA design are performed under Windows NT.

3.2 Applications

To evaluate the system we use both original versions of several SPLASH-2 applications [48] as well as versions that have been restructured to improve performance on SVM systems [28]. The same restructurings are found to be very important on large-scale hardware-coherent machines [29], so they are not specific to only SVM. Table 1 presents basic facts for the applications we use and summarizes the overall performance improvements. FFT, LU-contiguous, Ocean-rowwise[2], 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 [28]. 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 presents the problem sizes along with the uniprocessor execution times. Speedups are computed between the sequential program version (without linking to the SVM library or introducing any other overheads) and the parallel version. The initialization and cold-start phases are excluded from both the sequential and the parallel execution times in accordance with SPLASH-2 guidelines.

3.3 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 2 and 3 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 3). 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. This is in fact the reason that Water-nsquared performs worse. This version of Water-nsquared uses fine-grained, per-molecule locks when updating the private forces computed by each process into the shared force array, to reduce inherent serialization at locks. However, this causes the frequency of locks and hence of invalidation propagation to be very large. We find that these messages occupy the queues in the NIs, increasing the time it takes for lock acquire requests to be delivered to the host processor and serviced. We modified Water-nsquared to use coarse-grained locks, and we found that although lock overhead is reduced, overall performance does not improve due to increased serialization.

In the final GeNIMA protocol (described later in this

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

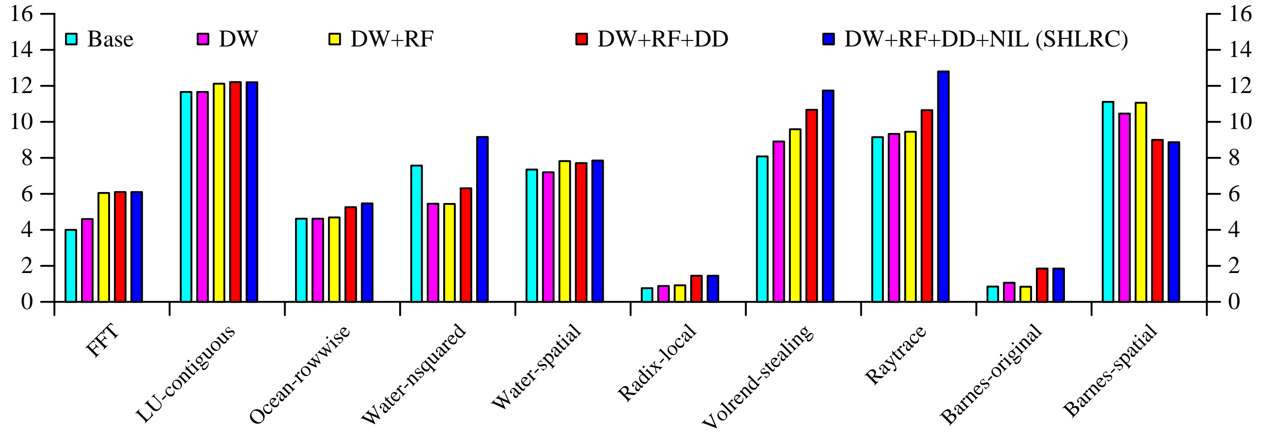

Figure 2: Application speedups. From left to right the bars for each application are (i) Base, (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

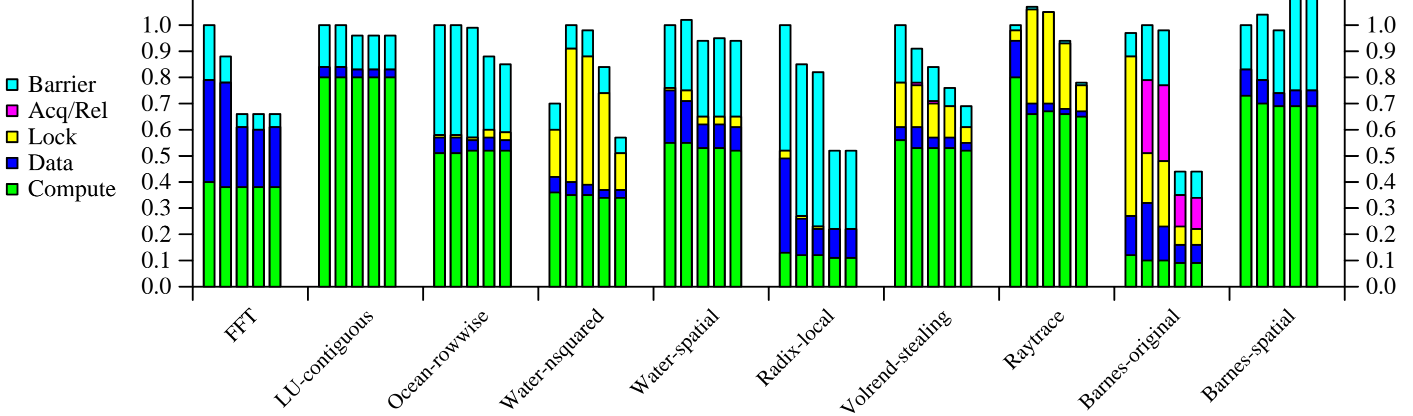

Figure 3: Normalized average execution time breakdowns for 16 processors. From left to right the bars for each application are (i) Base, (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

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

To reduce the number of messages, invalidations can be "pulled" at acquires with the remote fetch operation, rather than pushed with broadcast remote deposit at releases, which increases acquire latency. We experimented with the second approach in the Windows NT version of our system, which has similar performance characteristics, and found no noticeable benefits for GeNIMA at the scale of systems we examine here.

Remote fetches of pages (RF): We see in Figures 2 and 3 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. Using the performance monitor we found that the use of the remote fetch operation does not reduce the contention in the communication layer in this case.

Remote diff application (DD): As we see from the execution time breakdowns in Figure 3, 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.

Barnes-spatial performs much worse with DD than without. This is because the number of messages in the network increases by more than a factor of 30 in this case, due to the highly scattered nature of diffs within each page. This problem can perhaps be addressed at the application level by changing the layout of data structures, such that updates to shared data are done more contiguously. At the system level, analysis with the performance monitor shows that the increased number of diff messages indeed results in (i) the send request queue in the NI becoming full and thus stalling subsequent messages from the host (both synchronous and asynchronous), and (ii) increased NI occupancy at the send side. The stalls at the send queue increase the effective overhead at the host processor and lead to less overlapping of communication and computation. The increased NI occupancy at the send side does not seem to have a significant effect on performance. These results agree with previous simulation results [10], that found NI packet processing occupancy not to be a major bottleneck for SVM on a Myrinet like system.

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

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 shows that lock time is reduced up to about 60%. These improvements come from the elimination of interrupts, and also from the fact that lock messages do not need to be delivered to host memory. As discussed earlier, the latter results in shorter service times for lock messages since they need not wait for other messages to be delivered to the host first. Interestingly, Barnes-original does not benefit from network interface support for locks despite its very high lock time. We find that in some processors the time per lock acquire is very high (while in others it is low), and in those cases the largest component of lock time comes from contention in either the SVM protocol or the communication system. This effect is not present in the NT version of our system, where lock time does indeed improve. Finally, GeNIMA makes task stealing in Volrend more effective than with previous protocols, and it improves the computational load balance too. In previous studies [28], it was found that task stealing is not effective in Volrend because of the high cost of locks as well as the dilation of critical sections.

Overall, we see from Figure 4 that GeNIMA, which leverages all our general-purpose NI extensions to eliminate interrupts and asynchronous protocol processing, improves application performance by on average 38% for applications that end up performing quite well and up to 120% for applications that still do not perform very well under SVM. The improvements in individual overhead components of execution time are even larger. Several applications that performed in mediocre ways now perform much better, even well, on a 16-processor system. The only application that performs worse in GeNIMA than in the Base protocol is Barnes-spatial, due to the direct diffs problem discussed earlier. Eliminating direct diffs causes Barnes-Spatial to perform better than in the Base protocol too. Our results also show that all three mechanisms are important in different applications, so all should be supported in the NI if possible.

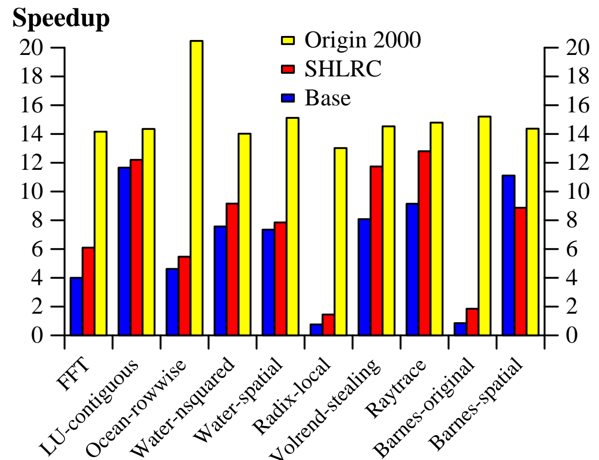

Figure 4: Application speedups for a hardware DSM machine (Origin-2000), and for the Base and the final GeNIMA protocols.

3.4 Remaining bottlenecks

Unfortunately, despite all the improvements in GeNIMA, Figure 4 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 3 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 operation. 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 [27, 28]. 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 2 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 2 shows that in certain cases (e.g. Radix) the cost of `mprotect` is a very large component of the protocol costs. Reducing the cost of `mprotect` is not straightforward, mainly because the operating system needs to be involved. We coalesce `mprotect` system calls to multiple contiguous pages into one call, and have experimented with `mprotect`ing more pages than necessary to further reduce the number of `mprotect` calls (e.g. `mprotect` all pages in contiguous range when more than a certain threshold of them need to be `mprotect`ed)), but more basic improvements may be necessary.

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 2: 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`.

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.

4 Communication Layer Implications

Due to many complex interactions in the system, the mechanisms we use often affect other, seemingly unrelated, aspects of performance. The approach taken in GeNIMA creates a tradeoff: On one hand, the number of messages and the total traffic in the system are both increased compared to the Base protocol, potentially leading to increased contention. On the other hand, the traffic is spread over larger time intervals, and there is more overlapping of communication, computation and message handling due to both the use of asynchronous messages as well as the small size of messages and the resulting pipelining between the host and the NI. In this section, we use the performance monitoring tool implemented in the NI [36] to examine the network interface activity in detail for both the Base protocol and GeNIMA.

Tables 3 and 4 quantify the effect of contention in the Base and GeNIMA protocols for small messages (up to 256 bytes) and large messages respectively. Each column repre-

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

Table 3: Ratios of average time to uncontended time for each network or NI stage in the path from the sender to the receiver, for small messages in the Base protocol and GeNIMA (reported as Base/GeNIMA)

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

Table 4: Ratios of average time to uncontended time for each network or NI stage in the path from the sender to the receiver, for large messages in the Base protocol and GeNIMA (reported as Base/GeNIMA).

sents one stage of the path from the sender to the receiver (described in Section 3.1), which can be individually measured by the monitor firmware and software [36]. In all stages, queuing and contention is included in the measurements. Note that in VMMC there is no explicit receive operation; thus, there is no receive stage in the message transfer pipeline that involves the host processor, even in the Base protocol.

In Tables 3 and 4 there are two numbers per stage for each application, separated by a slash: one for the Base protocol and one for GeNIMA. Each number is the ratio of the average time spent by a message in the corresponding stage to the time that would have been spent in the same stage in uncontended transfers. Thus, each number is a ratio that shows the average effect of contention incurred in that stage.

Table 4 shows that large messages behave very similarly in the two protocols; contention is very small in the NI in both cases. This may be because large messages are often page transfers, which afford the least overlap between that message and other activities. For small messages, however, GeNIMA greatly increases contention at the NI or network for almost all applications, and for all stages except the last one (data delivery to host memory). Moreover, in the same protocol, applications that were found to perform poorly tend to have higher contention than others. The most apparent cases are Water-nsquared and Barnes-original.

Thus, GeNIMA performs much better despite incurring higher contention for small messages. This means that be-

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

Table 5: Speedup on 32 processors or both our system and the SGI Origin2000. The data presented for Barnes-spatial is for 32K bodies (as opposed to 128K bodies in the Linux version).

sides the improvements from removing interrupts, scheduling problems, etc. from the host processors, the system can better tolerate the higher latencies due to contention. This increased tolerance comes from the fact that almost all communication layer operations used in the protocol are asynchronous, so the processor directly incurs only the small post overhead[3], and that the bandwidth in the system is adequate in most cases [1]. Moreover, GeNIMA takes advantage of current technology trends that make it easier to improve system bandwidth than latency. Ordering and data integrity is guaranteed mostly by ensuring that the necessary conditions are met at the protocol level.

Finally, it is interesting to see how GeNIMA performs on larger scale systems. Table 5 presents data for GeNIMA on 32 processors for the Windows NT version of our system. We see that many applications scale reasonably well up to 32 processors (and in fact perform even better for larger problem sizes). We are currently pursuing this research further to larger scale, to judge the potential of SVM and identify the key bottlenecks.

5 Limitations and Future Work

Although we have presented a detailed analysis of results, this work has certain limitations. The most important ones are the following:

First, we have presented results for only one problem size and one system configuration for each application, mainly for space reasons. Varying these would give a more complete picture of the system and the key performance factors. We experimented with larger problem sizes, and performance of most applications indeed improves as the problem size increases. The impact of the NI support and protocol extensions tends to decrease somewhat for several applications if system size stays the same (and to increase with smaller problem sizes unless load imbalance dominates), but it remains substantial for the problem sizes we can run. We are currently investigating how the performance and bottlenecks scale with system size.

Second, the NI support we have examined aims at eliminating interrupts and asynchronous protocol processing. However, it can be enhanced with more sophisticated operations like scatter-gather, as well as with multicast or broadcast support in the NI now that coherence information is be-

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

ing propagated eagerly. The effects of these enhancements are worth exploring. Alternative support for some operations can also be provided (e.g. atomic operations instead of full lock support).

Third, we have examined one set of tradeoffs about laziness/eagerness issues in SVM protocols. While it applies invalidations lazily, in certain cases GeNIMA is more eager in propagating information than other protocols that have been proposed. It is unclear how much further this can be taken with modern communication layers. For instance, propagating invalidations when the actual protection changes occur upon first write, instead of at release or acquire points, would result in an even more eager protocol. The effects on system performance of this and other tradeoffs (e.g. centralized vs. $O(p^2)$ barriers) with modern communication layers need more investigation.

Another limitation is that our programmable NI processor is very slow (33MHz) compared to the compute processors. Faster programmable support or hardwired support for these operation may improve parallel SVM performance much further, and may also make sophisticated features like scatter-gather more attractive. Also, it is interesting to investigate how well our approach can be supported on clusters of DSM nodes, an attractive method for building truly large-scale shared address space systems but in which there may be many NIs per node.

6 Related Work

Related work in network interface support for SVM has discussed how NIs can be used for several purposes:

Fast communication to improve the performance of traditional send and receive communication. This type of support has been exploited in many SVM projects [18, 26, 33, 51, 46, 41, 40, 42] and is also used in our base system, HLRC-SMP [40].

Protocol processing in the network interface. This choice lies at the other end of the spectrum. The network interface can be used not only to avoid interrupting the compute processor but also to perform full-blown protocol processing, including diff creation and application and the management of timestamps and write notices. This approach was taken in [51], where an Intel Paragon was used to move protocol processing into the network interface processor. Each node on the Paragon has two processors one of which is used as a communication processor. The performance benefit in that case was found to be small. 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 [19], where it was found that the benefit of this approach is small due to poor utilization of that processor, especially compared to the 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 [30] and other research, and is found to be small compared to other system overheads.

Transparent remote data and synchronization handling that can be utilized by protocols to alleviate key bottlenecks. In this case the remote compute processor is not involved in handling message requests, 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) [26] 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 [12] to avoid diff computation and application in a home-based SVM protocol. The Cashmere system [33, 46] 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. The network supports these remote writes and also provides global ordering. In our approach, on the other hand, the NI supports only *explicit* communication operations performed by the protocol library, supports data movement in both directions (deposit and fetch) as well as mutual exclusion, and does not have to support global ordering. A different type of coarse- or variable-grained remote fetch support has been examined through simulation [34], 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 [25], and to often disappear or worse with SMP nodes due to the bus traffic imposed by write-through caching [8].

More sophisticated support to accelerate specific protocol operations has also been examined in simulation, such as hardware diff engines in [6]. Support for AURC with write-back caches has also been designed and evaluated through simulation in [8], 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 [21].

Finally, 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 [33], 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 [52], with similar results. Our approach avoids the need for interrupts or polling altogether.

7 Conclusions

We have used network interface support to decouple asynchronous message handling from protocol processing and to thereby eliminate the need for expensive interrupts or polling in SVM protocols. In particular, we have implemented NI support for general-purpose, *explicit* data movement and synchronization operations that are not specific

to SVM, and altered the SVM protocol to take advantage of these operations. In the final GeNIMA protocol, asynchronous message handling is done entirely in the NI, and protocol processing is done on the host processors but only at synchronous points with respect to application and protocol execution. The protocol propagates information more eagerly in some cases, but the need for asynchronous protocol processing and the related interrupts (or polling) is eliminated without requiring the NI to be tightly integrated in the node or to observe memory operations in it.

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

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

8 Acknowledgments

We thank Dongming Jiang for providing us with the data for the Origin 2000. We are grateful to the members of the PRISM group at Princeton, in particular Liviu Iftode and Rudrajit Samanta, for useful discussions. We also thank Sanjeev Kumar for his help with proof-reading the paper, and the reviewers for their useful comments.

References

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

[2] D. H. Bailey. FFTs in External or Hierarchical Memories. *Journal of Supercomputing*, 4:23–25, 1990.

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

[4] 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.

[5] 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.

[6] 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.

[7] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation. In *IMA Workshop on Parallel Algorithms and Parallel Systems*, Nov. 1996.

[8] 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.

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

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

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

[12] 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 Annual Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

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

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

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

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

[25] 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.

[26] 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.

[27] 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.

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

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

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

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

[32] P. Keleher, A. 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.

[33] 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.

[34] 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.

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

[36] C. Liao, M. Martonosi, and D. W. Clark. Performance monitoring in a Myrinet-connected SHRIMP cluster. In *Proc. of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Aug. 1998.

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

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

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

[40] 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.

[41] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 125–136, Jan. 1998.

[42] 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.

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

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

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

[46] 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.

[47] 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.

[48] 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 Annual Symposium on Computer Architecture*, May 1995.

[49] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating message-passing in cache-coherent multiprocessors. In *Proceedings of Architectural Support For Programming Languages and Operating Systems*, 1994.

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

[51] 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.

[52] 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.