Network Interface Support for Shared Virtual Memory on Clusters

Angelos Bilas, Cheng Liao, and Jaswinder Pal Singh

Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544
{bilas, cliao, jps}@cs.princeton.edu

Abstract

Clusters of symmetric multiprocessors (SMPs) are important platforms for high performance computing. A lot of research has been done in building network interconnects and communication layers that deliver low-latency and high-bandwidth communication to the user. With the success of hardware cache-coherent distributed shared memory (DSM), a lot of effort has also been made to support the coherent shared address space programming model in software on clusters. However, performance is still far from that achieved on hardware DSM systems.

In this work we investigate the use of mechanisms in the software communication layer and the underlying network interface to substantially enhance the performance of shared virtual memory (SVM) on clusters of SMPs. We use a real implementation with a programmable network interface as our prototype, but our extensions are general-purpose and can be provided by network interfaces that do not employ a programmable processor.

We examine how the protocol layer can take advantage of each mechanism in the communication layer and be restructured accordingly. The final protocol (SVM-NI) eliminates the need for interrupts and asynchronous protocol handling. For each mechanism, we evaluate the impact on the end performance of ten applications with widely varying characteristics. We demonstrate that substantial improvements in performance can indeed be achieved, and find that different applications need different mechanisms among the ones we use. Application performance improves up to 50% for applications that end up with reasonably good speedups; individual components of execution time targeted by each mechanism are reduced by even higher percentages. Finally, we use a firmware performance monitor, integrated with the communication layer, to understand the drawbacks of the system, to identify interesting tradeoffs in the protocol layer for future exploration, and to identify the remaining bottlenecks in SVM performance that should be addressed next.

1 Introduction

There has been a lot of work recently on supporting page-grained shared virtual memory (SVM) on clusters of symmetric multiprocessors (SMPs). Several SVM systems and protocols have been studied through both implementation and simulation [12, 17, 22, 3, 32, 4, 5, 28]. Previous work has shown that the use of SMP nodes indeed improves performance for software shared memory protocols [19, 4, 30, 29]. In all these cases however, speedups on a 16-processor system were still found to be substantially lower than those expected from the programs on hardware-coherent machines. Figure 1 shows the comparison between an efficient hardware-coherent machine and the all-software HLRC-SMP [28] system for several applications; the processors used by the two systems are different, so a direct comparison cannot be made, but the qualitative result is clear. Thus, it is important to develop techniques for improving SVM performance on clusters.

The end application performance of an 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 application performance. In this paper, we focus on improving performance by enhancing the *functionality* of the system layers, driven by the availability of novel, general-purpose mechanisms in commodity network interfaces. Since we use a real system as our vehicle, the performance parameters of the different layers are held fixed. It is left to other research to examine the related and complementary problem: the impact of improving the cost parameters of the communication and protocol layers, on end performance.

Since the beginning of SVM [24], much excellent research has been done in improving the functionality of the protocol layer either by itself, based on increasingly relaxed memory consistency models [2, 20], or to take advantage of specialized features of a network interface [17, 23, 3]. Recently, several developments have also occurred in communication layers. Commodity network interfaces that support, or can be programmed to support, key data movement and synchronization mechanisms without interrupting the processor have been developed [14, 8]; communication libraries [11, 26, 9, 31, 1, 27] and an industry standard application programmer interface (API) [10] that incorporate

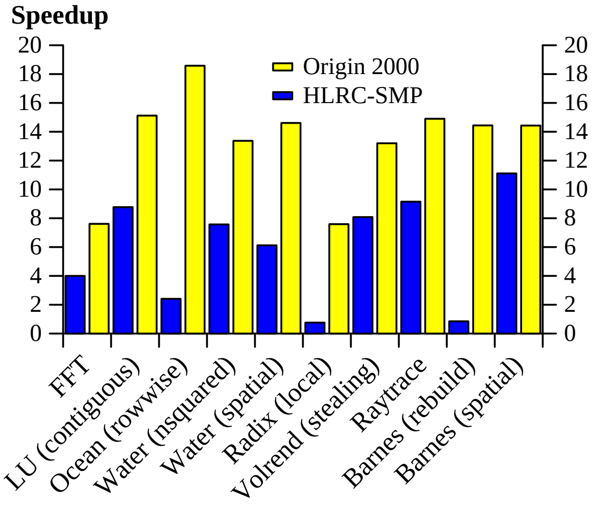

Figure 1: Application speedups for a hardware DSM machine (Origin-2000) and an SVM system (our base protocol).

these mechanisms have also been developed.

The focus of this research is to understand the extent to which general-purpose mechanisms can be used in commodity network interfaces, in conjunction with the appropriate protocol enhancements, to make large improvements in the performance of SVM systems on clusters. Our study is done through real implementation on a cluster of Intel PentiumPro SMPs connected together by a Myrinet network [8]. The SVM system that we start from is HLRC-SMP [4, 28], an all-software home-based lazy release consistency system extended to efficiently support and take advantage of SMP nodes. The all-software HLRC-SMP system uses the network interface only as a pathway for traditional send/receive communication. It does not rely on any specific network interface features in its design, and is applicable to any modern system area network. Communication and protocol processing is managed by the same processors that perform computation.

Starting from this system, we use the commodity Myrinet network interface [8], which includes a programmable processor, to implement general data transfer and synchronization mechanisms one by one and examine their cumulative impact on end performance. Our extensions to the network interface are general purpose extensions; some of these mechanisms are available in the Virtual Interface Architecture (VIA) industry standard API [10]. We leverage the VMMC communication library [9] that provides the software communication architecture needed to exploit some of the features. Also, since taking advantage of some of the mechanisms either requires or is helped greatly by changes to the protocol layer as well, with each mechanism we discuss and implement the relevant protocol changes. Performance is achieved through the synergy of the two system layers, though driven by the communication layer. We do not modify the application layer in this paper, relying on the application restructuring for SVM that was performed in [18].

The specific network interface mechanisms that we examine include support for remote write and remote fetch operations without interrupting the remote processor, and for mutual exclusion (locking) in the network interface. Protocol operations like performing diffs and keeping track of versions and coherence information are not performed in the network interface but rather on the main processor; the network interface is assumed to be (and, in our case, is) slow at these activities. The focus is only on general purpose data movement and synchronization mechanisms. These mechanisms do not require a programmable processor on the network interface; we use a programmable network interface to only prototype and evaluate the mechanisms on a real platform. Since previous work [6] has shown that interrupts and scheduling can be a major problem for SVM, and many efforts have been made to deal with protocol handler scheduling on SMP nodes [13], we propose a protocol that eliminates the need for asynchronous protocol processing. Each of our mechanisms reduces the need for interrupts and asynchronous protocol handling; the final protocol (SVM-NI) does not use interrupts for protocol processing at all, and in fact eliminates the need for asynchronous protocol processing.

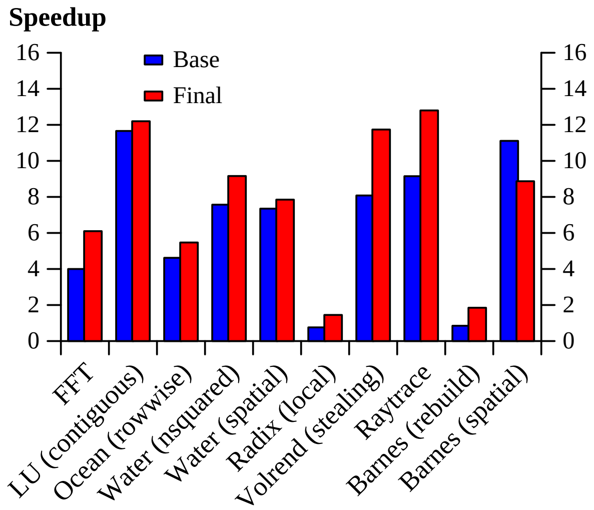

Figure 2: Application speedups. The left bar is the base protocol and the right bar the protocol with the NI extensions.

We find that implementing these mechanisms in the network interface, together with the relevant protocol changes, improves performance substantially on the cluster of SMPs for our suite of ten applications. Several applications that originally performed in mediocre ways now perform much better, even well, on a 16-processor system, with most applications now yielding speedups close or more than 10 on 16 processors (see Figure 2). Application performance improves up to 50% for reasonably well performing applications (and up to 120% for applications that do not perform very well under SVM even afterward). 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%. The results indicate that providing support for these features in a network interface is indeed important for SVM performance and for improving clusters as a computational infrastructure. Among the mechanisms we implement, different ones are important for different applications, indicating that they should all be supported and leveraged. While speedups are improved tremendously by these techniques, they are still not quite as good as those on hardware-coherent systems, even at this 16-processor scale. We use a firmware performance monitor, that is integrated with the communication layer, to understand the drawbacks of the system and identify some protocol tradeoffs that bear further investigation. Most importantly, we use the monitor to investigate the causes of the key remaining overheads for SVM performance after all these enhancements, and hence the areas that should be addressed for the next major improvements in bringing SVM performance closer to that of hardware coherence.

2 Approach and Related Work

As shown in Figure 1, for many applications the performance of shared virtual memory (SVM) on clusters is far from the performance that hardware DSM systems provide. This is despite the fact that system area networks (SANs) [8, 14, 16] provide increasingly low latency (on the order of a few microseconds today) and high bandwidth (on the order of 100 MBytes/s) in hardware, and communication software delivers almost all of this performance to user programs. However, network interfaces also implement increased functionality compared to traditional local area networks (LAN), which can be exploited to enhance SVM performance. For example, the network interface in SHRIMP [7] and the DEC Memory Channel [14] provide some form of automatic propagation of writes to remote memory. Furthermore, SAN networks like Myrinet have started to provide programmable processors on the network interface that can be used to implement special functionality [8].

The capabilities of SAN network interfaces can be taken advantage of in several ways:

1. To improve the performance of traditional send and receive operations.

2. To provide support for general-purpose data movement and synchronization mechanisms that can be utilized by protocols to alleviate key bottlenecks.

3. To perform protocol processing itself in the network interface.

The first type of support has been exploited in many SVM projects and is also used in our base system, HLRC-SMP. At the other end of the spectrum, the network interface can be used to perform full-blown protocol processing, including diff creation and application and the management of time-stamps and write notices. This approach requires that (i) the NI be fast enough relative to the main processor at performing protocol operations, (ii) the NI be able to access the application data and the protocol data quickly and safely, (iii) the main processor and the NI have a consistent view of data mappings and protections. Typically, this approach requires a fast communication co-processor with similar access to main memory as the main processor (e.g. on the memory bus instead of in the network interface). The Intel Paragon provided such a communication architecture, and this approach was taken for SVM in [32]; however, the performance benefit in that case was found to be small. SANs that include a programmable processor in the NI itself usually employ much slower processors (in terms of both clock rate and the architectural features) than the host processor and do not have good fine-grained access to the host's main memory. Moreover, in cases where a fast, symmetric co-processor is available on the node, the cost of protocol processing does not justify devoting a whole fast processor to it, and that processor is likely to be better used as an additional processor for application computation within the node.

Our focus here is on the intermediate approach: taking advantage of general purpose mechanisms for data movement and synchronization in the network interface to reduce key costs and reduce the involvement of the main processors. Such mechanisms include support for:

- automatic transfer of application and protocol data based on memory loads and stores

- reducing the cost of explicit communication used in protocols and processor involvement in protocol handling

- providing protocol-independent synchronization functionality in the network interface.

Previous work [17, 22, 3] has partially addressed the first of these possibilities by taking advantage of write propagation support in the network interface to propagate shared or protocol data to remote memories without involving the remote processors. AURC [17] is a protocol designed for the SHRIMP multicomputer, which uses the automatic write propagation provided by the specialized network interface to propagate fine-grained writes transparently to the home of each page. Similarly, the Cashmere protocol [22] takes advantage of the remote write capability of the DEC Memory Channel interconnect to propagate updates somewhat differently in the context of a different protocol. [3] proposes the use of a hardware diff engine to propagate updates to shared data. A discussion on how remote write access capabilities of VM-based networks can be used in SVM systems is provided in [15]. In all the above cases, any necessary protocol and data movement activity is handled through either interrupts or code instrumentation.

The type of system we are interested in, and the system we use, is not well suited to a similar approach. The automatic write propagation used by AURC requires either that second-level caches be write-through or that substantial intrusive hardware support be provided to ensure that the appropriate modifications from a write-back cache are indeed propagated according to the consistency model [5]. Modern nodes do not provide either type of support. The Cashmere system uses the DEC Memory Channel network and interface, which has a memory-mapped I/O address space. Writes that need to be propagated have to be instrumented so that they are *doubled* to this address space as well, whereupon are automatically propagated by the NI. Myrinet and many other general interfaces do not provide this support. Moreover, a study that simulated the Myrinet interface and the required support for both write-through and write-back caches, found that the gain from providing this support and hence avoiding diffs was not large [5]. We do not use this approach of data propagation based on ordinary or instrumented stores (or even loads) in this paper. Rather, we focus on the second and third possibilities listed above: Internode communication is triggered by explicit data or protocol messages issued by a processor or by explicit synchronization operations, not directly by load and store operations to memory, and the NI is used to support these operations better and to reduce the involvement of compute processors. Neither of these cases has been explored before to our knowledge, at least in the context of such a system.

3 Network Interface and Protocol extensions

In this section we describe the extensions we made to the communication layer (network interface firmware and software) and the key protocol costs to which each mechanism is targeted.

Interrupt cost has been identified as a key bottleneck for SVM systems [6]. Reducing the number of interrupts is therefore one of the primary goals of this work. One way to avoid interrupts is to use polling on the main processor to handle messages instead. This approach has been investigated in Cashmere-2L [22] where the main processor polls the network queues for incoming messages on program back-edges. This requires that the code be instrumented to add polling instructions, and affects the time when messages are handled. A similar polling method was used and compared with interrupts in [33]. The tradeoffs between interrupts and polling are not very clear (the performance differences are found to be small for the platform used in [33]), and the instrumentation needed for x86 platforms is not easily available.

Our approach of using network interface support to reduce processor involvement eliminates interrupts, for both data and synchronization transfers, and at the same time improves performance further by reducing other protocol costs. It is different from both interrupts and polling in that the processor is not informed of when messages come in or when requests are serviced. Message handling is thus asynchronous with regard to the main processors, which means that traditional protocols that rely on the processors being informed of protocol activity must also be altered to ensure correctness. Protocols are further altered for performance by taking advantage of the available features of the communication architecture.

As mentioned, our base protocol is HLRC-SMP [4, 28], which implements the well-known lazy release consistency (LRC) model [21] on a network of SMPs. Both HLRC 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). Traditional LRC schemes maintain distributed diffs at the writers, from where they must be fetched on a page fault [20]. In HLRC, the writer sends the diffs eagerly to a designated home node for the page, and the diffs are applied there to the home copy which is always kept up to date according to the consistency model. On a page fault, instead of fetching diffs from previous writers, the whole page is fetched from the home.

HLRC-SMP uses the communication layer only as a fast message passing system. The protocol uses four compute processes per four-processor node, and an extra, floating process to handle protocol requests. Each protocol request causes an interrupt that schedules the protocol process on one of the processors. The protocol requests that require interrupts in the base protocol are page fetches, lock acquisition, and diff application at the home. We now describe the basic mechanisms we implement in the network interface to handle incoming requests or data transfers and hence reduce interrupts and asynchronous protocol processing. We assume a communication layer that supports the memory mapped communication model, present in many state of the art communication systems [9, 14]. In each case, we describe how the mechanisms are used to alter the protocol itself.

Remote writes to protocol data structures: The memory mapped communication model we use allows for data transferred to a remote node to be deposited in specified destination virtual addresses in main memory without involving a remote compute processor. Many communication systems support this or similar type of operations. We use this remote write capability to directly update remote protocol data structures, e.g. page timestamps, invalidation lists etc.

Page fetches: We extend the communication system to support (in firmware) a remote fetch operation to fetch data from a remote node. The remote fetch operation can fetch data from exported portions of the remote memory to arbitrary addresses in the local memory. We use this operation to fetch page data from the home. When a remote page must be requested, the local processor first requests the timestamp or version number of the remote page and then requests the page itself. The request messages are asynchronous, so the request for the page does not have to wait for the timestamp to arrive. If the timestamp is determined to be incorrect, the requester retries.

Remote diff application: Another feature, present in many state of the art communication layers, that we use, is the ability to perform low overhead asynchronous send operations. We use this low overhead mechanism and direct writes to remotely apply the diff and update shared data at the home nodes. 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 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*. This saves 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, it may substantially increase the number of messages that are sent, since it introduces one message per contiguous run of modifications within a page rather than one message per page that has been modified. Using a scatter-gather operation to remotely apply the modifications can combine the best of both worlds. In this work however, we do not explore the scatter-gather operation.

We should note that direct diffs are best used when pages are fetched using the remote fetch mechanism described earlier and not via interrupts and protocol handlers. The reason is that, since direct diffs do not require a home processor to be interrupted, the home processor cannot know when it has the updated version of a page so that it can service page requests that arrived earlier and have been waiting for that version. When remote fetches are used, each processor retries whenever it fails to fetch the right version of a page, and does not depend at all on the remote processors.

Lock synchronization: The algorithm that implements locks in the base HLRC-SMP protocol is used in many other protocols as well. 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. The home of each lock is thus used to maintain a distributed list of locks. The last owner keeps the lock until another processor acquires the lock.

When the owner passes the lock along to another process it also sends along with the lock, the invalidations that the requester needs to have to maintain the release consistent view of the shared memory. Thus, since the base protocol uses lazy propagation of invalidations, the protocol handler services a remote acquire it sends to the requester both the lock (mutual exclusion part) and the page invalidations (coherence information).

To avoid the asynchronous protocol processing and the interrupts, we separate the mutual exclusion and the coherence information parts. For the coherence part, we propagate invalidations to other nodes in the system eagerly at releases, using the remote write capability of the communication system. Invalidations are still applied lazily at the next causally related acquire. Another way to decouple coherence information and mutual exclusion, without using interrupts, and still maintaining lazy invalidation propagation, is to use the remote fetch mechanism described above, to fetch write notices at acquires rather than propagate them eagerly at releases. However, there is a tradeoff here between acquire latency and potential extra messages, and we decide to stay with an eager propagation approach for write notices.

For the mutual exclusion part, we extend the communication layer to provide the programmer with network locks. Lock acquisition and release for mutual exclusion become communication system, rather than HLRC protocol operations. The implementation of the locks, in the network interface firmware, is similar to the algorithm described earlier. However, no coherence information is involved and instead of using host processors in the nodes to maintain the distributed lists for locks using interrupts, the network interface processors do so without host processor involvement.

Summary: The final version of the protocol (SVM-NI) that takes advantage of all these extensions does not use interrupts and asynchronous protocol handlers at all, either for protocol data or for application data or for synchronization. The new model is that, instead of getting access to remote resources through interrupts handlers, processor do so by using standard remote operations implemented in the network interface. These operations extend the functionality of the network interface, and their implementation does not require receive processor intervention.

4 Experimental Testbed

We implement the network interface extensions on a cluster of Intel PentiumPro SMPs connected with Myrinet. The nodes in the system are 4-way PentiumPro SMPs at 200 MHz. The PentiumPro processor has an 8 KByte data and an 8 KByte instruction cache. The processors are equipped with a 512 KBytes unified 4-way set associative L2 cache and 256 MBytes of main memory per node.

Myrinet [8] is a high-speed system-area network. A Myrinet network is 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 MByte/s peak bandwidth each. Actual node-to-network bandwidth is usually constrained by the 133 MBytes/s I/O bus on which the NI sits. All four nodes are connected directly to an 8-way switch.

The communication layer that we use on top of the Myrinet network is VMMC [9]. VMMC provides protected, reliable, low-latency, high-bandwidth user-level communication. The one-way latency is about 19μs and the maximum bandwidth about 95 MBytes/s. The basic feature of VMMC that we use in this work is the remote write capability. The sender can directly write data to exported regions of the receiver's memory, without receiving side (process or processor) intervention. We extend VMMC with remote fetch and lock support as discussed earlier. The time for a one page remote fetch operation is about 110μs (about 40μs for one word). We expect these times to be better on future network interfaces.

The communication layer of the system includes a performance monitoring tool [25] that gathers network-level data. The core of the monitor is firmware run by the NI processor on the Myrinet NI card. Working at the NI firmware level gives us access to crucial determinants of application performance not available in higher-level software. Several pieces of the tool's functionality would be infeasible or costly to implement in higher-level software, such as a global synchronized clock and the ability to measure network-level latencies.

5 Performance Results

In this section we evaluate the new protocols that result from our extensions to the network interface. In the presentation of the results we evaluate the features and resulting protocols cumulatively in the order in which they were discussed. Figure 2 shows the speedups for each protocol and Figure 3 the average execution time breakdowns. Breakdowns are averaged over all processors for space reasons. However, in our analysis we use per-processor breakdowns to capture the effects of imbalances across processors. Table 1 presents various statistics for the applications we use. The protocols we evaluate are:

Base: This is the base protocol HLRC-SMP, where VMMC is used only as a fast communication layer.

Application	Problem Size	Overall	Data Time	Lock Time	# Messages	Average size (Bytes)
FFT	4M points	52.50	45.37 (44.92)	0.00	3096.24	2041.57
LU (contiguous)	4096x4096 matrix	4.63	13.46 (11.20)	0.00	6979.24	1826.53
Ocean (rowwise)	514x514 ocean	18.40	21.76 (19.26)	9.21	17602.93	391.34
Water (nsquared)	4096 molecules	21.00	15.26 (46.17)	62.76	60752.12	54.48
Water (spatial)	15625 molecules	6.80	41.60 (41.80)	9.69	9385.63	1801.54
Radix (local)	4M keys	90.79	26.76 (27.00)	53.21	2481.31	956.19
Volrend (stealing)	256x256x256 cst head	45.30	43.81 (42.44)	50.44	2884.88	316.88
Raytrace	car	39.89	2.52 (50.03)	59.01	2114.50	49.74
Barnes (rebuild)	32K particles	117.65	41.07 (68.25)	1.98	61763.32	180.94
Barnes (spatial)	128K particles	-20.16	40.99 (37.70)	33.84	3995.74	1478.20

Table 1: Application statistics. The second column shows the problem sizes we are using. The third column represents the overall improvement in each application between Base and Final. The fourth column is the improvement in data wait time between DW and DW+RF and the fifth column, the improvement for lock time between DW+RF+DD and Final. For remote fetch we also report the percentage improvement between DW and Final in parentheses. The last two columns show the average number of messages sent per processor for each application, and the average message size (in Bytes).

Direct Writes (DW): This protocol uses the remote write capability of VMMC to directly update only remote protocol data structures. Shared data are propagated to the home of each page with the traditional diff mechanism. Non-contiguous portions of protocol data structures are sent directly to remote data structures with separate messages and are not packed into bigger messages. Since the overhead of the asynchronous send operation in VMMC is relatively small (about 1.5μs), this approach results in more pipelining: Instead of having the host processor pack data in one big message before the NI can take over, the host's posting of small messages using asynchronous sends is pipelined with the NIs preparation and sending of them. Invalidations are thus propagated eagerly, and synchronization is separated from coherency information.

We see that all applications with the exception of Water (nsquared) perform either comparably or better with this DW than with the base protocol. This is primarily due to the removal of packet processing for protocol messages that now use direct writes. However, the DW protocol does send more messages, both because it uses eager propagation and because it uses small messages (Table 1). 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. These messages occupy the queues in the NIs, increasing the time it takes for a lock message to get through and get a reply. It is worth noting here that in the Final version of the protocol the effect of this traffic problem is much less apparent and performance of Water (nsquared) improves by 21% compared to Base. This happens because lock messages need not be delivered to host memory but are handled completely in the network interface, indicating that the bottleneck due to the traffic in the previous case was between NI and host.

One way to fix this problem is to propagate invalidations as before at acquires using a scatter-gather operation to both perform direct writes and reduce the number of messages. This approach requires that interrupts are used to inform the last releaser when it needs to send the invalidations. Another approach is to implement lazy invalidation propagation in the protocol by taking advantage of the remote fetch operation and to change the protocol data structures such that one read can fetch the invalidation for multiple intervals (thus having fewer, larger messages). Both of these methods, however, may hurt the performance of other applications. It is not clear if there is one best method, and some of these tradeoffs are worth exploring.

Remote Fetch (RF): In this protocol we introduce the remote fetch operation to fetch application and protocol data without interrupting the remote processor. The remote fetch operation is mainly used in this protocol to fetch pages and page timestamps. Other uses are also possible depending on the protocol version. We see 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 and Barnes (rebuild) see a high improvement in performance. The data wait time is reduced up to 45%, with most applications seeing improvements in data wait time of more than 20%. It is interesting to note that the improvement in data wait time in FFT (45%) comes exclusively from eliminating the interrupts and the related scheduling effects within an SMP. The use of the remote fetch operation does not reduce in this case the contention in the network.

Direct Diffs (DD): This protocol uses the remote write operation to update home pages and eliminates the need for interrupts in diff application. 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, Barnes (rebuild), Raytrace, Volrend and Water (nsquared). The benefits in performance come from eliminating the interrupts (and related scheduling effects in the SMP nodes) and from better load balancing of protocol costs, and they come despite the fact the direct diffs make messages even smaller (one per contiguous run of modifications in a page).

We also note that Barnes (spatial) performs worse with DD. This is because the number of messages in the network increases by more than a factor of 30 due to the highly scat-

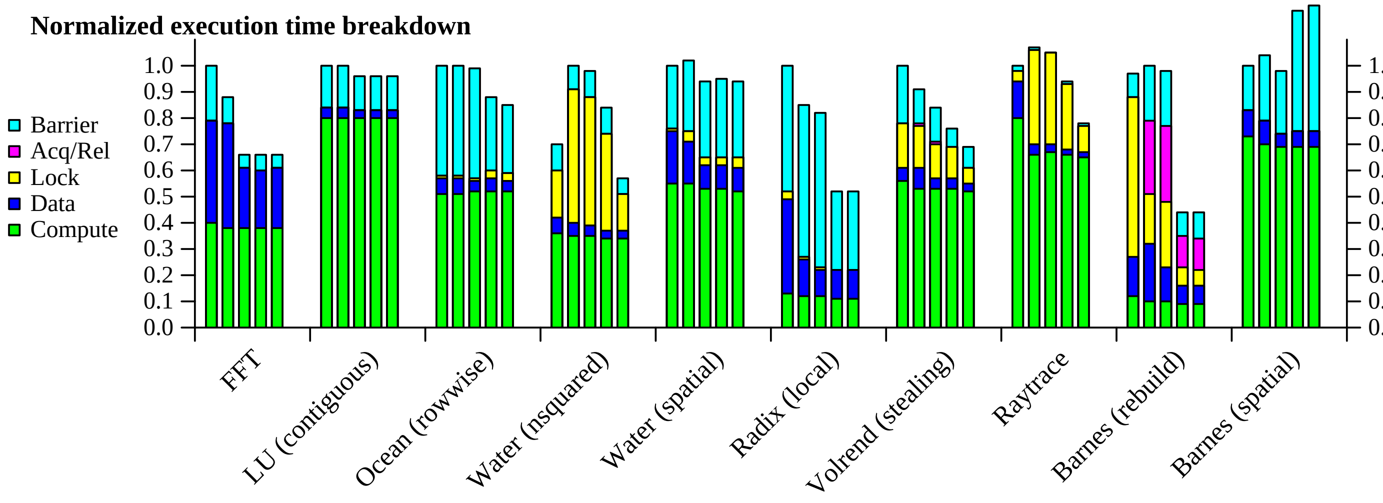

Figure 3: Normalized execution time breakdowns. From left to right the bars for each application are (i) HLRC-SMP (Base), (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

tered nature of diffs within each page. This problem can be addressed either by changing the layout of data structures in the application or by adding a scatter-gather operation in the NI. The second approach combines the best of both direct diffs and the traditional diff propagation mechanism since it minimizes both the number of messages and the total size of the data sent. This effect for Barnes (spatial) disappears in all protocols when direct diffs are not used. Thus, using any combination of the network interface extensions without direct diffs, does not decrease performance.

Network Interface Locks (NIL): This protocol uses the locking support of the network interface. Since we explore the effect of the extensions cumulatively, this version includes all NI extensions and is the Final version of the protocol (SVM-NI). Compared to the previous version, the Final version substantially improves the performance of applications that use locks frequently. Table 1 shows that lock time is reduced up to 59%. If we take into account Water (nsquared) as well, the improvements are even higher. These improvements come from the fact that no interrupts are necessary any more, 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.

It is interesting to note that in Volrend, with this Final protocol we manage to take advantage of task stealing and improve load balancing. In previous studies [18], it was found that task stealing is not effective in Volrend because of the cost of locks.

Interactions of NI extensions: Figure 3 shows that the improvements in certain protocol costs do not always depend only on the extension that attacks the specific cost. For example, the introduction of direct diffs improves the performance of both lock and data wait times: lock time is reduced in Raytrace and data wait time is reduced in Barnes (original). This happens because each change affects the behavior of the network interface, the pattern and the number of messages in the network, the queuing and contention, and (through asynchronous requests) the behavior of the protocol. Thus, it is not always straightforward to predict or quantify the improvement introduced by each extension, or determine the exact reasons for the results. To partially address this issue, in the next section we give a more general characterization of what happens in the network for the base and the final protocols, using the performance monitor.

Summary: We see that support for remote write and remote fetch operations in the NI have an effect on both the design and performance of SVM systems. They significantly enhance the ability of the network interface to support SVM effectively. They allow much more flexibility in the protocol design, since they provide to protocols a wider choice of solutions for accessing remote resources and allow them to better balance system costs. This often results in very substantial performance improvements. Table 1 shows that just by using the remote fetch operation the data wait time improves up to 45%.

Support for locking in the NI is also very useful when locking is frequent, often because it frees lock messages from having to travel all the way to the host processor and get stuck in queues However, locking support requires more intrusive changes to the NI, and it may be more difficult to provide in network interfaces that do not have a programmable processor. Providing locking support in the NI improves application lock times up to 60% in our experiments.

Although we have not yet evaluated this, we find that support for scatter-gather in the network interface can be beneficial for SVM protocols, for updating both protocol and shared data as has been discussed at various points in the paper.

Overall, we see from Figure 2 that the Final version (SVM-NI) that leverages all our general-purpose NI exten-

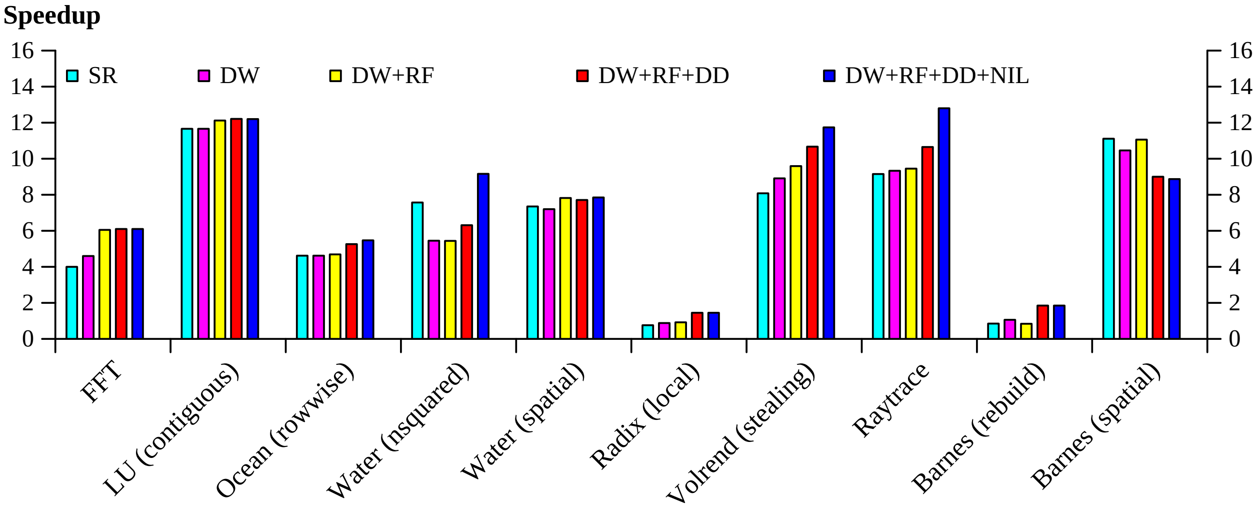

Figure 4: Application speedups. From left to right the bars for each application are (i) HLRC-SMP (Base), (ii) direct writes (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

sions improves application performance up to 50% for reasonably well performing applications and up to 120% for applications that do not perform very well under SVM. These are large improvements, and make a big difference to the end user, making SVM more competitive.

6 Further Analysis

We have seen that using the NI in the manner we have done helps improve SVM performance substantially. However, Figures 5 together show that the resulting performance is still not quite where we would like it to be to compete with what efficient hardware coherence can achieve. In this section we use the performance monitoring tool in the communication layer [25] to answer two further questions. First, are there any performance problems caused by the changes we have made in the communication or protocol layers, which are simply overcome by the improvements, or have our changes only made improvements. We do this by examining the network interface activity in detail for the Base and Final protocol versions. Second, despite all these changes, how far is performance from what we might achieve with hardware coherence, and what are the main remaining sources of bottlenecks that we should go after to get the next leap in SVM performance.

6.1 Network Interface Activity and Protocol Tradeoffs

Two key potential differences in traffic in the Final protocol from the Base protocol are the use of eagerness and the related use of smaller protocol messages. The small messages achieve pipelining between host and NI as discussed earlier and the eagerness reduces acquire latency. However, traffic in the system is increased and this may result in increased contention. Since messages are asynchronous, the contention may or may not be hidden from the processors. Table 2 tries to quantify the amount of contention for certain applications in the Base and Final protocols, using the NI performance monitor. Each column represents one stage of the path from the sender to the receiver:

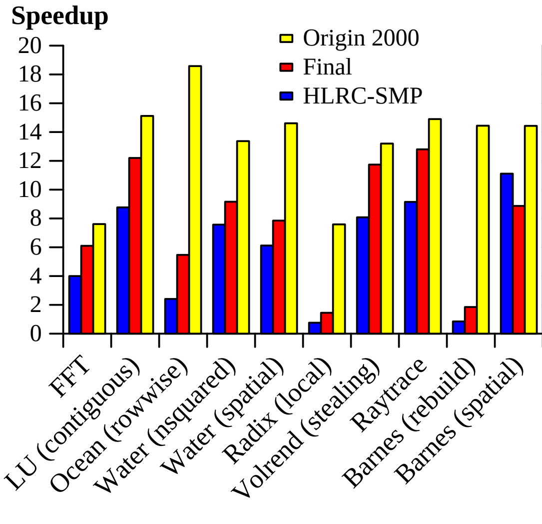

Figure 5: Application speedups for a hardware DSM machine (Origin-2000), the Base and the Final protocols.

Post time is the time from the time the host posts a request, until the data for this request are in the NI. This stage may involve DMAing of data from host memory to the NI for big messages.

Send time is the time it takes the NI to send each packet to the network from the time it first appears in the NI. This stage includes queuing in the NI before a packet is sent to the network.

Receive time is the time it takes each packet from when the first byte leaves the sending NI until the last byte is in the receive NI.

Deliver time is the time it takes the receiving NI to deliver the packet to user memory from the moment it arrives.

In all cases queuing and contention is included in the measurements. Note that in VMMC there is no explicit receive operation and thus there is no receive stage in the message transfer pipeline itself where the host processor is involved. Packets are delivered directly from the NI to the host memory. Destination processor involvement for message processing even in the base protocol is after the message data have been placed somewhere in memory.

In Table 2 there are two numbers per stage for each application: one for the Base and one for the Final protocol. Each number is the ratio of the average time spent in the corresponding stage by small messages (up to 256 bytes) over the time spent in the same stage in uncontended transfers. Thus, each number is a ratio that shows how much contention we have in that stage. We separated out small messages from large messages, and found that large messages behave very similarly in the two protocols and the contention for them in the NIs is very small.

Interestingly, we see that the Final version of the protocol greatly increases contention at the NI or network for almost all applications for all stages except the last one (data delivery to host memory). Moreover, in the same version, applications that were found to perform poorly tend to have higher contention than others. The most apparent cases is Water (nsquared) and Barnes (rebuild).

Although the Final version incurs much higher contention in the network for small messages, the applications perform substantially better. This can be either because the improvements from other parts of the system, e.g. removing interrupts and scheduling problems from the processors, are higher, or that the Final system can tolerate contention because of its asynchronous communication. In the final protocol almost all operations are asynchronous and the host processor does not need to wait for the network operations to finish. The first reason is clearly a major factor, but we have not yet been able to determine the contribution of the latter. Nonetheless, the results suggest that we should indeed explore NI enhancements and protocol tradeoffs that reduce small messages. NI enhancements include scatter-gather for invalidations and diffs, and protocol tradeoffs include using lazier forms of propagation as discussed earlier. Scatter-gather almost certainly should help, while the impact of protocol tradeoffs depends on added latency and the actual impact of the contention we observe on performance.

Application	Post	Send	Receive	Deliver
Water	1.7/10.4	2.2/13.8	1.9/13.2	3.2/5.1
Barnes	-/8.6	-/12.6	-/12.4	-/6.0
Volrend	-/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	1.5/-	2.4/-	2.0/-	4.3/-
Water	1.8/4.6	3.2/6.9	3.4/6.2	4.7/4.6
Radix	1.8/4.3	3.5/1.3	3.4/5.3	4.6/7.1
Barnes	1.9/3.2	3.6/5.5	3.6/4.3	5.2/5.8

Table 2: Ratios of average times over the uncontended time for each network or NI stage in the path from the sender to the receiver, for small messages in the Base and Final versions of the system.

Application	MBC	BT	BWT
FFT	7.6	8.46	13%
LU (contiguous)	12.20	14.12	70%
Ocean (rowwise)	7.5	12.85	50%
Water (nsquared)	9.16	10.3	80%
Water (spatial)	7.15	12.56	63%
Radix (local)	1.65	4.52	6%
Volrend (stealing)	12.00	13.26	65%
Raytrace	13.00	14.82	80%
Barnes (rebuild)	2.03	2.69	81%
Barnes (spatial)	8.87	14.43	18%

Table 3: Application speedups. The second and third columns are the speedups for each application if we exclude (a) memory bus contention (MBC), and (b) barrier cost. The last column (BWT) shows the percentage of barrier time that is spent waiting. The rest of the time is spent in protocol processing.

6.2 Remaining Performance Bottlenecks for SVM

Finally, let us examine the key remaining bottlenecks in the Final protocol. The potential bottlenecks are memory bus contention and cache effects inside each SMP, lock time, barrier time and data wait time.

Figure 3 shows that for many applications the time spent in barrier operations is high. We divide barrier cost into two parts, the wait time at the barrier and the cost of protocol processing. This division is interesting, since it shows whether major improvements require eliminating (or improving) protocol processing at barriers or better load balancing of computation, communication and protocol costs. Protocol costs at barriers are mainly due to changing the protection of pages, updating page timestamps and sending messages. These costs can be reduced if a lightweight thread model is used instead of the heavierweight process model that we use for laziness [28]. The reason is that in the thread model invalidations and page timestamps need to be taken care of only once for all threads and this can be done in parallel. Of course, the thread model does reduce laziness of invalidations within an SMP, and there are other tradeoffs that must be explored [28]. Table 3 shows the effective speedups of the applications if we were to simply exclude the memory bus contention, the barrier wait time and the protocol wait time, respectively, from the parallel execution profile. The second column in Table 3 shows the speedup for each application if we exclude the increase in local memory access costs compared to the uniprocessor version. For four applications, FFT, Ocean, Radix and Barnes (rebuild), the aggregate compute time in the parallel execution increases when compared to the execution time of the sequential run. Actual compute time excluding local memory access shouldn't increase. For FFT, Ocean and Radix the increase is due to contention on the memory bus due to capacity misses from four rather than one processor competing for the bus within a node. This results in an increase in local compute time of 50%, 80% and 130% for each application respectively. This is specific to the design of the memory subsystem in the SMP nodes we use and may not be an issue in other architectures. For Barnes (rebuild), the reason appears to be, not capacity misses, but cache invalidations caused by transfers of shared data between the host memory to the network interface and vice versa.

The third column in Table 3 shows that, for most applications, excluding barrier costs results in very good speedups. The exceptions to this are FFT, Water (nsquared), Radix and Barnes (original). The last column in Table 3 shows the percentage of barrier cost that is attributed to wait time (imbalances) as opposed to barrier processing cost. We see that for FFT, Radix, and Barnes (spatial) most of the barrier cost is protocol processing time. More specifically, most of this cost is attributed to modifying the access right of pages with `mprotect` and to updating page timestamps, so reducing this cost would be very valuable. Let us now look into the four applications for which barrier cost is not the key remaining bottleneck.

The remaining bottleneck in FFT is the high data wait time still, because of the low speed of the network interface and contention in the NI. If we compare the data wait time in FFT to what it would be with uncontended remote fetch operations, there is an increase of less than 30% in the cost. The rest of the cost is due to the uncontended cost of the remote fetch operation and can be addressed by faster networking hardware; protocol enhancements cannot do much about this. Even latency tolerance techniques might not work well due to the already existing contention and the fact that overall latency is dominated by end-points rather than transit which can be more easily overlapped.

Water (nsquared) is limited by the high lock time as discussed earlier. For Radix the remaining bottleneck is the high data wait time due to the very high communication to computation ratio and false sharing of pages. The cost of `mprotect`s is also very large here. Finally, in Barnes (rebuild) the problem is synchronization and data wait time. These costs are both high and relatively imbalanced and they result in high barrier time as well.

Summary: The above analysis suggests that the two major areas of future effort to further improve SVM performance at this scale, other than the protocol tradeoffs and scatter-gather discussed earlier, are improving barrier processing (including and especially `mprotect` time, and further improving data wait time by hardware enhancements in some applications.

7 Conclusions

We have investigated how the capabilities of modern commodity network interfaces can be exploited to make the performance of SVM systems more competitive with hardware coherence. We implemented a set of extensions in the programmable network interface of the Myrinet network, and evaluated their impact on application performance on a network of Intel PentiumPro SMPs. We chose to implement general-purpose data movement and synchronization mechanisms that are likely to be found (or can be implemented) in commodity network interfaces.

The software communication library that we use already provides us with a remote write interface, and we extend it by adding a remote fetch operation and by providing locking support. The resulting features in the communication layer (software and firmware) were used to alter the protocol layer accordingly. We presented an SVM protocol (SVM-NI) that uses these communication layer capabilities to completely avoid the need for asynchronous protocol processing by the host processors. In the final version of the protocol, whenever a processor wants to access some remote service, it does so through the general purpose extensions we add to the network interface. We also used a performance monitor, integrated in the communication layer, to understand the results and discover the sources of key remaining bottlenecks in the SVM system.

We find that although network contention increases in the final version of the system, due to the use of eager propagation and small messages, application performance improves substantially, up to 50% for well-performing applications and up to 120% for applications that still do not perform very well under SVM. Moreover, data wait time improves up to 45% and lock time up to 60%. The features we implemented in the NI also provide more flexibility in the choice of efficient protocol operations and management. Although locking support was also found to be very useful, it requires more intrusive support. Different applications benefited greatly from different NI features, indicating that all three should be supported if possible.

Our results and detailed analysis have led us to several areas of future work. First, a scatter-gather operation in the NI appears to be very useful in addressing certain key compromises. Second, while we used a set of protocol extensions that made sense given the communication enhancements and NI capabilities, other choices are possible too, introducing tradeoffs that need to be more fully explored. Support for other types of NI operations, e.g. remote atomic operations, can also be advantageous.

Finally, our results show that the next major features to attack to bring SVM closer in performance to efficient hardware-coherence, at least at this scale, are barrier processing time and especially the cost of `mprotect` operations to change page protections.

8 Acknowledgments

We would like to thank Dongming Jiang for providing us with the data for the Origin 2000. We are also grateful to the members of the PRISM group at Princeton, and in particular Liviu Iftode and Rudrajit Samanta, for the useful discussions during the course of this work.

References

[1] 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.

[2] 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.

[3] 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 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: analyzing the performance of clustered distributed virtual shared memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, October 1996.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[22] 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.

[23] 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.

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

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

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

[27] L. Prylli. Draft: BIP messages user manual for BIP 0.92. http://lhpca.univ-lyon1.fr/bip.html, July 1997.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Y. Zhou, L. Iftode, J. 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.