Design Choices in the SHRIMP System: An Empirical Study

Matthias A. Blumrich[*], Richard D. Alpert[†], Yuqun Chen[*], Douglas W. Clark[*], Stefanos N. Damianakis[*], Cezary Dubnicki[*], Edward W. Felten[*], Liviu Iftode[‡], Kai Li[*], Margaret Martonosi[§], and Robert A. Shillner[*]

Abstract

The SHRIMP cluster-computing system has progressed to a point of relative maturity; a variety of applications are running on a 16-node system. We have enough experience to understand what we did right and wrong in designing and building the system. In this paper we discuss some of the lessons we learned about computer architecture, and about the challenges involved in building a significant working system in an academic research environment. We evaluate significant design choices by modifying the network interface firmware and the system software in order to empirically compare our design to other approaches.

1 Introduction

Multicomputer and multiprocessor architectures appear to be converging due to technological and economic forces. A typical architecture is now a commodity network connecting a set of compute nodes where each node consists of one or more microprocessors, caches, DRAMs, and a network interface. The node architectures of different systems are not only similar to one another, but are often commodity high-volume uniprocessor or symmetric multiprocessor systems. This approach can track technology well and achieve low cost/performance ratios. In such architectures, the network interface becomes arguably the key component that determines the functionality and performance of communication.

1.1 Network Interface Design Challenges

An ideal network interface should have a simple design and yet deliver communication performance close to the hardware limit imposed by the nodes and the routing network. It should support low-level communication mechanisms upon which message-passing and shared-memory systems, and applications perform well. It should also provide protection in a multiprogrammed, client/server environment. This is challenging for several reasons. First, the network interface device sees only physical memory whereas applications use virtual memory. Second, the network interface is a single, physical device shared among multiple untrusting processes, whereas the application processes would like a private communication mechanism to guarantee performance, reliability and protection.

Traditional network interface designs often impose large software overhead (thousands of CPU cycles) to send and receive a message because they rely on the operating system kernel to obtain exclusive access, check for protection, translate between virtual and physical addresses, perform buffer management, create packets, and set up DMA transfers.

The SHRIMP project studies how to design network interfaces to satisfy the design challenges. Our approach is to use a virtual memory-mapped communication model [12, 21], and implement it with some hardware support at the network interface level to minimize software overhead. Several other projects and commercial products have used similar memory-mapped communication models, including HP's Hamlyn project [15], Digital's MemoryChannel [23] and Dolphin's network interface. Although these efforts all proved that the memory-mapped communication paradigm can indeed achieve high-performance communication with minimal software overhead, many network interface design issues for virtual memory-mapped communication are not well understood.

1.2 Lessons Learned

This paper reports our experimental results on a 16-node SHRIMP multicomputer. In addition to measuring the behavior of the system as it is, we reprogrammed the network interface and low-level communication software to answer "what if" questions about the design. We measured

[*]Department of Computer Science, Princeton University, Princeton, NJ 08544

[†]NEC Research Institute, Princeton, NJ 08540

[‡]Rutgers University, Department of Computer Science, Piscataway, NJ 08855

[§]Department of Electrical Engineering, Princeton University, Princeton, NJ 08544

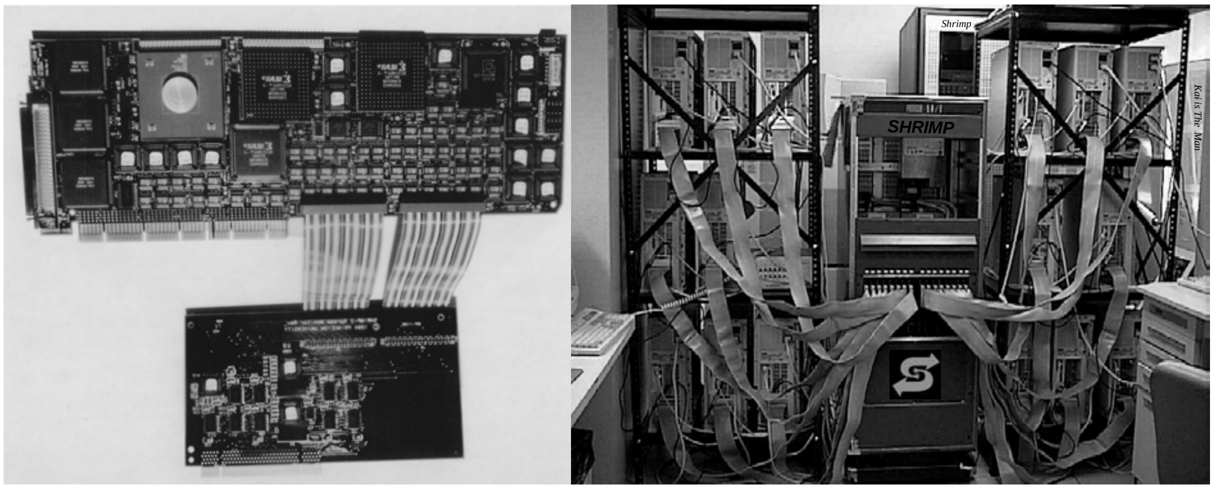

Figure 1. Photographs of the network interface (left) and the 16-node SHRIMP system (right)

performance with applications from four categories: using the virtual memory-mapped communication mechanism directly, using an NX-compatible message-passing library, using a stream-sockets-compatible library, and using shared virtual memory systems of several types.

Our experiments evaluate the consequences of our architectural approach, as well as answer questions about hardware parameters. Among the questions we consider are the following:

- Did it make sense to build custom hardware, or could we have gotten comparable results by using off-the-shelf hardware and clever software?
- Was the automatic update mechanism in SHRIMP useful, or would a simple block transfer mechanism have given nearly the same performance?
- Was user-level initiation of outgoing DMA transfers necessary, or could we have gotten nearly the same performance with a simple system-call-based approach and clever software?
- How important was our emphasis on avoiding receiver-side interrupts?

In addition to answering these questions, we discuss other lessons learned, including some things that consumed much of our design time, yet turned out not to matter.

2 The SHRIMP System

The architecture of the SHRIMP system has been described in several previous publications [10, 11, 12, 22]—notably [9]—and will only be described in as much detail as necessary here. Specific details of the architecture and implementation will be described more thoroughly throughout this paper.

2.1 Architecture

The SHRIMP system consists of sixteen PC nodes connected by an Intel routing backplane, which is the same as that used for the Paragon multicomputer [27]. The backplane is organized as a two-dimensional mesh, and supports oblivious, wormhole routing with a maximum link bandwidth of 200 Mbytes/second [43]. The right-hand photograph in Figure 1 shows the basic interconnection between the nodes and the backplane. The backplane is actually relatively small but, for convenience, we power it with the standard Paragon cabinet which is capable of housing a complete 64-node system.

The custom hardware components in the system consist of the SHRIMP network interfaces (one per node), and simple transceiver boards (not shown) to connect each network interface to a router on the backplane. The transceiver boards are necessary because the PCs and the backplane are on separate power supplies, requiring differential signaling between them.

The SHRIMP network interface (Figure 1) consists of two boards because it connects to both the Xpress memory bus [28] and the EISA I/O bus [6]. The memory-bus board simply snoops all main-memory writes, passing address and data pairs to the EISA-bus board. The EISA-bus board contains the bulk of the hardware, and connects to the routing backplane. Figure 2 shows the principal datapaths of the network interface.

Three important aspects of the DEC 560ST PCs used to construct the SHRIMP system bear mentioning. First, the 60 MHz Pentium processor has a two-level cache hierarchy that snoops the memory bus and remains consistent with

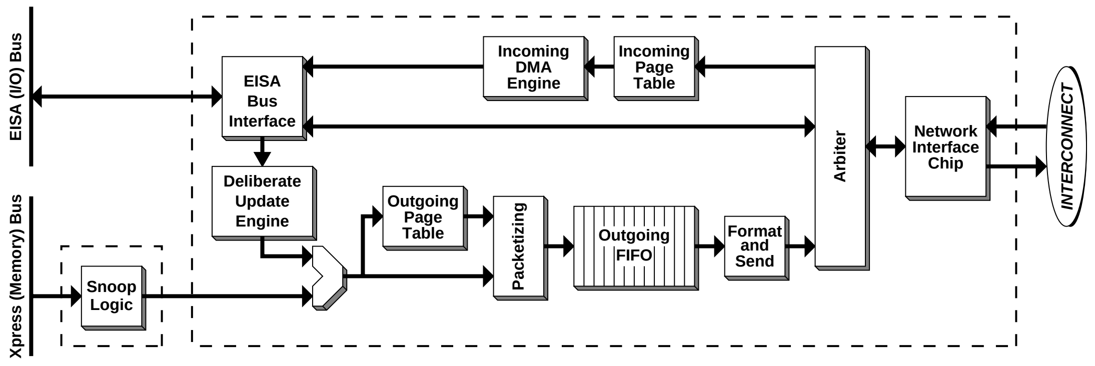

Figure 2. Basic architecture of the SHRIMP network interface

all main memory transactions, including those from the network interface. Second, the caches can be specified to operate in write-back, write-through, or no-caching mode on a per-page basis. Third, the memory bus does not cycle-share between the CPU and any other main memory master.

The SHRIMP network interface was designed to work in concert with the communication programming model, called Virtual Memory-Mapped Communication (VMMC), in order to provide an efficient, high-performance communication subsystem. The following description takes a top-down approach, beginning with VMMC.

2.2 Communication Model

Buffers and Import/Export The basic VMMC model supports direct data transfer to *receive buffers*, which are variable-sized regions of contiguous virtual memory. In order to receive data to a receive buffer, a process *exports* the buffer together with a set of permissions. Any other process with proper permission can *import* the receive buffer to a *proxy receive buffer*, which is a local representation of the remote receive buffer.

Deliberate Update In order to transfer data, a process specifies a virtual address in its memory, a virtual address within a proxy receive buffer, and a transfer size. This causes the communication subsystem to transfer a contiguous data block of the specified size starting at the specified memory address to the remote receive buffer indicated by the specified proxy address (subject to buffer size restrictions). Such a transfer is called *deliberate update* because the transfer is initiated explicitly.

Automatic Update Alternatively, a portion of virtual memory can be *bound* to an imported receive buffer (or portion thereof) such that all writes to the bound memory are automatically transferred to the remote receive buffer as a side-effect of the local memory write. This mechanism is called *automatic update* because no explicit transfer initiation is required. Due to implementation restrictions, automatic update bindings (also called mappings) must be page-aligned on both the sender and receiver.

Notifications VMMC allows a process to *enable notifications* for an exported receive buffer. This causes a control transfer to a specified user-level handler whenever a message is received for that buffer. Notification control transfers are similar in semantics to Unix signals. The system provides no guarantee as to when the notification is delivered to user level, and it does not prevent the received data from being over-written. However, it does provide queueing of multiple notifications. Exporting processes can optionally block and un-block notifications, but not for individual receive buffers.

2.3 Implementation

The SHRIMP network interface (Figure 2) supports the basic communication mechanisms of the VMMC model. There is a thin user-level library layer that implements the actual application programming interface (API) of the model for high-level libraries and applications.

Buffers and Import/Export The export implementation pins virtual pages of the receive buffer to physical pages. The import implementation allocates an Outgoing Page Table (OPT) entry for each page of the proxy receive buffer, and configures the entries to point to the remote physical pages of the actual receive buffer.

Deliberate Update An application or user-level library initiates a deliberate update transfer by using the network interface's user-level DMA mechanism [10]. By executing a two-instruction load/store sequence to special I/O-mapped addresses, the application tells the SHRIMP DMA engine the source, destination, and size of the desired transfer.

Protection is guaranteed by a combination of page-mapping tricks and simple error checking in the network interface hardware.

Automatic Update To implement automatic update, the network interface maintains a one-to-one mapping between physical memory page numbers and OPT entries. This allows a write that is snooped off of the memory bus to address the OPT directly and obtain a remote physical page number. To implement an automatic update binding, the OPT entries corresponding to the bound memory pages are simply modified to point to the remote physical pages, and enabled for automatic update. Any writes to pages whose corresponding entries are not enabled for automatic update are snooped, but ignored. The network interface has a mechanism to combine consecutive automatic updates within a single page or during a specified number of cycles into a single packet.

Notifications To enable notifications, the interrupt bits are set in the Incoming Page Table (IPT) entries corresponding to the pages of an exported receive buffer. An arriving packet causes an interrupt when an interrupt bit in the packet's header (controlled by the sender) is set, and the interrupt bit in the destination page's IPT entry (controlled by the receiver) is also set. When an interrupt occurs, a single system-level handler is invoked to decide where to deliver the user-level notification. Note that the sender's interrupt request bit for an automatic update packet is stored in the OPT, while deliberate update allows the bit to be dynamically set as part of an explicit transfer initiation.

3 Applications and Experiments

We have implemented several high-level communication APIs and systems on the SHRIMP multicomputer, including the native VMMC library [21], an NX message-passing library [2], a BSP message-passing library [3], a Unix stream sockets compatible library [17], a Sun-RPC compatible library [7], a specialized RPC library [7], and Shared Virtual Memory (SVM) [25, 26]. Each API implementation takes advantage of the low-overhead, user-level communication mechanisms on the system and supports a few applications.

In this paper we selected applications based on four different APIs: VMMC, NX, Stream sockets, and SVM. The primary selection criterion is whether there is a noticeable amount of time spent on communication. Table 1 shows the selected applications and their characteristics. Each of these applications has two versions: one using automatic update and another using deliberate update. We have selected small problem sizes for our evaluation purposes. We use the following applications:

Application	Comm API	Problem Size	Seq Exec Time (sec)
Barnes-SVM	SVM	16K bodies	121.3
Ocean-SVM	SVM	130×130	12.8
Radix-SVM	SVM	2M keys, 3 iters	14.3
Radix-VMMC	VMMC	2M keys, 3 iters	10.9
Barnes-NX	NX	4K bodies, 20 iters	149.9
Ocean-NX	NX	514×514, 1×10^{-3}	69.2
DFS-sockets	Sockets	4 clients	6.9
Render-sockets	Sockets	$167\times63\times34$	13.8

Table 1. Characteristics of the applications used in our experiments. (Ocean-NX does not run on a uniprocessor; two-node running time is given)

Barnes-SVM This application is from the SPLASH-2 benchmark suite [46]. It uses the Barnes-Hut hierarchical N-body method to simulate the interactions among a system of particles over time. The computational domain is represented as an octree of space cells. The leaves of the octree contain particles, and the particles and space cells are distributed to processors based on their positions in space. At each time step of the simulation, the octree is rebuilt based on the current positions of the bodies and each processor computes the forces for the particles which have been assigned to it by partially traversing the tree.

Ocean-SVM This fluid dynamics application is also from the SPLASH-2 suite. It simulates large-scale ocean movements by solving partial differential equations at each time-step. Work is assigned to processors by statically splitting the grid and assigning a partition to each processor. Nearest-neighbor communication occurs between processors assigned to adjacent blocks of the grid. The matrix is partitioned in blocks of $n/$p whole, contiguous rows.

Radix-SVM This is another kernel from the SPLASH-2 suite. It sorts a series of integer keys into ascending order. The dominant phase of Radix is key permutation. In Radix a processor reads its locally-allocated n/p contiguous keys from a source array and writes them to a destination array using a highly scattered and irregular permutation. For a uniform distribution of key values, a processor writes contiguous sets of $\frac{n}{r*p}$ keys in the destination array (where r is the radix used); the r sets that a processor writes are themselves separated by $p-1$ other such sets, and a processor's writes to its different sets are temporally interleaved in an unpredictable way. This write pattern induces substantial false-sharing at page granularity.

Radix-VMMC A port of the SPLASH-2 integer radix sort kernel to the VMMC API. The versions for automatic update and deliberate update differ in the method by which sorted keys are distributed. In the automatic update version, each processor distributes its keys by placing them directly into arrays on remote processors using automatic update mappings. In the deliberate update version, the keys for each remote processor are gathered into large message transfers, and scattered by remote processors.

Ocean-NX A message-passing version of the algorithm described in Ocean-SVM.

Barnes-NX A message-passing version of the algorithm described in Barnes-SVM. Because this implementation uses an octree data structure, running on more than eight nodes introduces communication in what would otherwise be a compute-only phase, limiting speedup. This is also evident in the significant performance degradation caused by interrupts on each send (see Table 2).

DFS-sockets This application is a distributed cluster file system implemented on top of stream sockets. The file system uses the disks of all nodes to store data, and the memory of all nodes to cache data cooperatively. The file system uses the VMMC sockets library, which includes some non-standard extensions for block transfers. A synthetic workload is created by running client threads on half of the nodes; the client threads read large files. Caches are "warmed up" before the experiment begins, and the workload is chosen so that the working set of a client thread is larger than the memory of a single node, but the collective working sets of all clients will fit in the total memory of the nodes. Thus there are many node-to-node block transfers but no disk I/O in the experiments.

Render-sockets This is a Parallel Fault Tolerant Volume Renderer [4] which does dynamic load-balancing and runs in a distributed environment. Render-sockets is based on a traditional ray-casting algorithm for rendering volumetric data sets. It consists of a controller processor that implements a centralized task queue and a set of worker processors that remove tasks from the queue, process them and send the results back to the controller processor. The data set is replicated in all worker processors and is loaded at connection establishment.

Figure 3 shows up to 16-processor speedups for several applications we have run on the SHRIMP system. For each application, we measured both the automatic update and deliberate update implementations, and plotted the version with the better speedup.

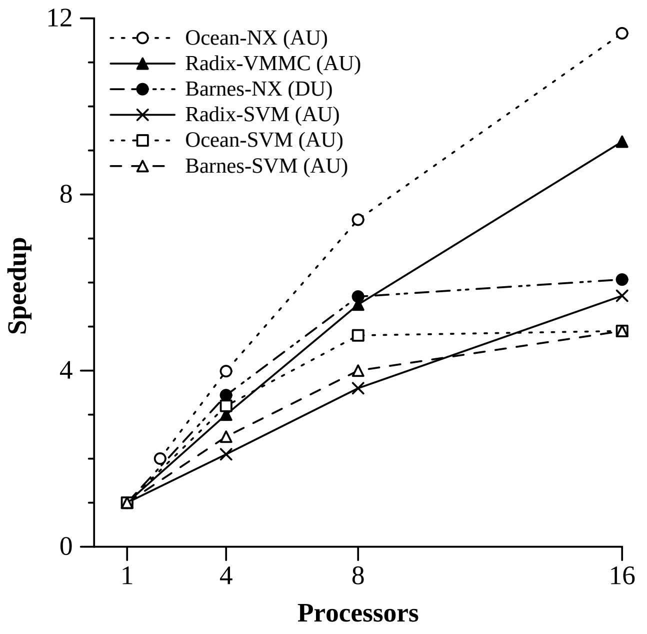

Figure 3. Speedup curves for a variety of applications running on SHRIMP

4 Experience and Design Issues

This section describes some key design issues, and what we have learned in building the SHRIMP network interface. When feasible, we evaluate our decisions via experimental evidence using the applications described in Section 3. In order to do the evaluation, we altered the network interface features by reprogramming its firmware and its low-level software libraries, to approximate the behavior of alternate designs.

4.1 Did It Make Sense to Build Hardware?

With nearly any major hardware project in a research environment, a central question is invariably "did it make sense to build hardware?"

In our case, the answer is "yes" for two main reasons. The first reason is performance. Our communication has better latency than several commercial network interfaces such as Myrinet [13], even though our nodes are old 60 MHz, EISA-bus based Pentium PCs and our network interface was designed in 1993. SHRIMP has a deliberate update latency of 6 μs, while the best latency achieved with 166 MHz, PCI-bus based Pentium PCs and Myrinet network interfaces running our optimized firmware for the same API [20] is slightly under 10 μs. Except for the automatic update mechanism, both systems implement the same VMMC API. The latency of SHRIMP is substantially better than that of the Myrinet system, even though the

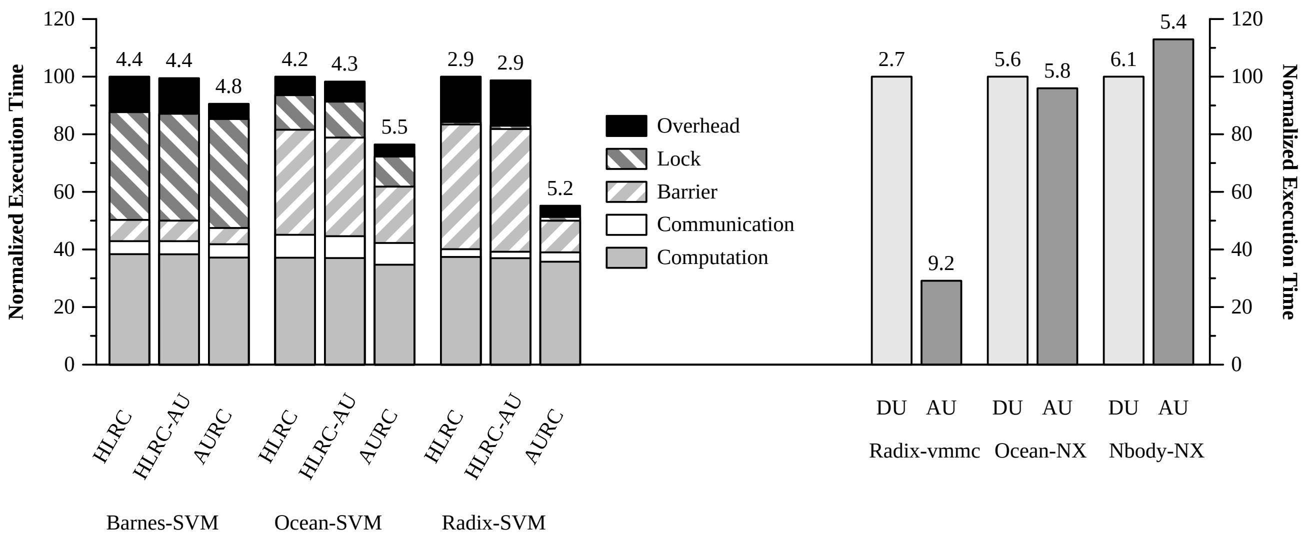

Figure 4. Comparing automatic update with deliberate update in three cases on a 16-node SHRIMP system: shared virtual memory, native VMMC, and NX message-passing library

nodes in the SHRIMP system are much slower than those in the Myrinet system.

Another reason for building hardware is that it allowed us to experiment with automatic update and compare it with deliberate update. Our network interface is the only one we know of that implements a virtual memory-mapped automatic update mechanism. By having such a feature, we could experiment on a real system with real applications to understand whether it is a good idea, what the performance implications are, and what design decisions make sense.

In short, we feel that building this system was useful, since it allowed us to experiment and learn things that would have remained unknown otherwise.

4.2 Was Automatic Update a Good Idea?

There are two principal advantages to automatic update communication. First, it has extremely low latency. The end-to-end latency is just 3.71 μs for a single-word transfer between two user-level processes [9]. Second, it can eliminate the need to gather and scatter data. In particular, large data structures that are written sparsely can be exported in their entirety, and mapped remotely for automatic update.

We built three implementations to evaluate the impact of using automatic update support to improve the performance of shared virtual memory [34]. The first implements the HLRC protocol [47] which uses only deliberate update communication. The second is similar to the first except that it uses automatic update to propagate the diffs transparently as they are produced, instead of buffering them and sending them explicitly using deliberate update messages. We call this approach HLRC-AU. The third approach implements the Automatic Update Release Consistency (AURC) protocol [25]. This implementation eliminates diffs entirely and uses automatic update mappings to propagate updates eagerly to home pages.

The left-hand side of Figure 4 compares the three SVM implementations using three different applications on the 16-node SHRIMP system. The number on top of each bar indicates the speedup relative to a sequential run. The benefit of omitting diffs and relying on the automatic update mechanism (as in AURC) is quite large (9.1%, 30.2%, and 79.3%). AURC outperforms HLRC for applications that exhibit a large degree of write-write false sharing. These applications pay a significant amount of overhead on diffing in the HLRC case, whereas the AURC implementation does not have a noticeable increase in computation time or network contention due to write-through mapping and automatic update traffic. Moreover, by eliminating the diff computation at synchronization events, the AURC approach reduces the synchronization waiting time, compared to HLRC. This overhead reduction further accounts for the observed performance improvement. On the other hand, using the automatic update mechanism to simply propagate diffs in the HLRC-AU case has very little benefit compared with HLRC. In fact, in some cases it it can slightly hurt the performance.

Another case showing the benefit of the automatic update mechanism is Radix-VMMC (radix sort using the native VMMC API). As illustrated on the right-hand side of Figure 4, the automatic update version improves the speedup of deliberate update by a factor of 3.4.

Application	System Call Cost
Barnes-SVM	23.2%
Ocean-SVM	17.7%
Radix-SVM	2.3%
Radix-VMMC	5.9%
Barnes-NX	52.2%
Ocean-NX	10.1%
Render-sockets	6.8%

Table 2. Execution time increase on 16 nodes due to requiring a system call for every message sent

Application	Notifications	Total Messages	%
Barnes-SVM	779,136	2,394,690	33%
Ocean-SVM	35,624	473,003	8%
Radix-SVM	161,627	386,671	42%
Radix-VMMC	0	2,160	0%
Barnes-NX	10,623	1,024,124	1%
Ocean-NX	11,380	1,007,342	1%
DFS-sockets	0	3,931,894	0%
Render-sockets	0	65,015	0%

Table 3. Per-application characterization of notification, and notifications as a percentage of total messages (16 nodes)

There are two principal drawbacks to our automatic update implementation which limit its usefulness for supporting higher-level APIs other than shared virtual memory. First, the send and receive buffers must have the same alignment with respect to page boundaries, and second, the hardware does not guarantee consecutive ordering when a deliberate update transfer initiation is followed by an automatic update transfer. In this case, ordering is determined by memory bus sharing between the CPU and the network interface's Deliberate Update Engine (Figure 2).

Our experiences to date have shown that automatic update is not so helpful for applications using high-level message-passing libraries such as NX and stream sockets. These applications tend to do large message sends in which the latency of the data movement itself is much more significant than the message initiation overhead. We have written versions of these libraries that use automatic update instead of deliberate update as the bulk data transfer mechanism. Comparing the performance of these two versions, we found that although automatic update delivers lower latency, this effect is often overridden by the DMA performance of deliberate update.

4.3 Was User-Level DMA Necessary?

Designing and building the User-Level DMA (UDMA) mechanism was a major focus of the SHRIMP effort. The primary goal of this was to reduce send-side overhead. We were able to reduce the send overhead to less than 2 μs for 60 MHz Pentium PC nodes that use the EISA bus.

In this section, we evaluate the benefit of UDMA over kernel-level approaches. To isolate the effects of kernel-level vs. user-level implementations, we wrote a kernel-level driver that simulates what the software would do in a SHRIMP-like architecture that lacked UDMA. We then modified the SHRIMP software library to call this kernel driver before each message send. With this kernel-level implementation, we measured application performance and compared it to performance of the actual SHRIMP system.

Table 2 summarizes the results of this experiment. It shows that the additional system call increases the execution time by 2% to 52%, depending on the application.

4.4 How Important is Interrupt Avoidance?

Another major system design goal was to minimize the number of receive-side interrupts. In many cases, no interrupts are required. Some communication models, however, rely on receive-side interrupts as part of every message arrival, so interrupts cannot be eliminated. For these cases, we provide the ability to attach an optional "notification" to each message.

How often are notifications used? The SVM implementation relies on the notification mechanism. As a result, we see in Table 3, a significant fraction of the messages invoke notifications. In contrast, the sockets and VMMC applications do not use notifications at all. Instead, they rely on polling to detect the arrival of data.

How much do we save by avoiding receive-side per-message interrupts? To answer this next question, we modified VMMC so that every arriving message causes an interrupt, which triggers a null kernel-level handler. Table 4 gives the extra cost imposed by these extra interrupts. The slowdown varies between roughly negligible and 25%, depending on the application. Note that a real system would exhibit higher overhead than this since it would have to do some work in the interrupt handler. If interrupts are necessary on each *packet* rather than each message, overheads will be even higher in some cases.

Application	Slowdown
Barnes-SVM	18.1%
Ocean-SVM	25.1%
Radix-SVM	1.1%
Radix-VMMC	0.3%
Barnes-NX	6.3%
Ocean-NX	15.7%
DFS-sockets	18.3%
Render-sockets	8.5%

Table 4. Execution time increase due to requiring an interrupt for every message arrival. All data is for 16 nodes except for Barnes-NX (8 nodes)

4.5 Other Design Issues

We saw above that some of the areas where we focused our effort were fruitful, leading to very significant benefits in practice. On the other hand, there were some issues on which we spent considerable time that ended up having a minimal impact on performance. This subsection considers some of these issues.

4.5.1 Automatic Update Combining

As discussed in the previous section, the two main advantages of automatic update are low latency and implicit scatter/gather of data. In order to achieve the lowest latency, the basic automatic update mechanism creates a packet for every individual store, and launches it immediately. As a result, large automatic update transfers suffer a loss of bandwidth because each packet generates an individual bus transaction at the receiver.

However, when automatic update is used to send a large amount of data, the focus is not necessarily on achieving the lowest possible latency. In this case, the network interface hardware can automatically combine a sequence of consecutive stores into a single packet to improve the bandwidth. Although large packets have a higher latency, they make efficient use of data streaming on the backplane and burst DMA at the receiver.

Automatic update combining is specified on a per-page basis in the outgoing page table when a binding is created. The basic combining mechanism accumulates consecutive stores into a single packet until either a non-consecutive store is performed, a page boundary is crossed, a specified sub-page boundary is crossed, or a timer expires.

We ran Radix-VMMC and several AURC SVM applications using automatic update with and without combining. In all cases, enabling combining had less than a 1% effect on overall performance. This is because these applications write sparsely, so very little combining takes place. Additionally, the lazy character of the SVM protocol makes combining even less effective.

In the absence of deliberate update, however, combining is very helpful for applications that would otherwise use deliberate update. These applications send large messages to contiguous addresses—an ideal situation for combining. For example, DFS-sockets runs about a factor of two slower when forced to use automatic update without combining.

4.5.2 Outgoing FIFO Capacity

The Outgoing FIFO (Figure 2) was included in the design of the network interface in order to provide flow control for automatic update. The Xpress memory bus connector that the network interface uses does not allow a memory write to be stalled, so some sort of buffer for automatic update packets is required. Furthermore, we need some mechanism to keep this buffer from overflowing.

To prevent overflow, the network interface generates an interrupt when the amount of data in the FIFO exceeds a programmable threshold. The system software is then responsible for de-scheduling all processes that perform automatic update until the FIFO drains sufficiently.

The lower bound on Outgoing FIFO capacity is the memory write bandwidth multiplied by the time it takes the CPU to recognize the threshold interrupt. On our system, the lower bound is roughly 1K bytes, so a large FIFO is not required.

However, the software flow control is costly, so it is desirable to choose a FIFO capacity that minimizes its occurrence. The FIFO drains faster than it fills, so the only way it can overflow is if it is unable to drain. There are two ways this can happen. First, incoming packets have top priority for access to the NIC, so the FIFO cannot drain when an incoming packet is arriving. Second, the FIFO may be unable to drain if there is network contention.

The first scenario is unlikely to occur on the SHRIMP system because the memory bus cannot share cycles between the CPU and the network interface. Therefore, incoming packets effectively block the CPU from performing automatic update writes to memory. The second scenario is likely to occur under conditions of high communication volume, especially when there is a many-to-one communication pattern. In this case, there is a tradeoff between FIFO capacity and threshold interrupt frequency, and that tradeoff is application dependent.

When designing the SHRIMP network interface, we decided to use 4K-byte-deep, 1-byte-wide FIFO chips because they represented the knee in the price/capacity curve at that time. The Outgoing FIFO is actually 8 bytes wide in order to keep up with the memory bus burst bandwidth, so its total capacity is 32K bytes.

We ran our applications with the FIFO size set artificially to 1K bytes, and there was no detectable difference in performance compared to the normal-sized FIFO. This occurred because our applications have relatively low communication requirements.

4.5.3 Deliberate Update Queueing

The SHRIMP deliberate update mechanism operates by performing user-level DMA [10] transfers from main memory to the network interface. Transfers of up to a page (4K bytes) are specified with two user-level memory references to proxy memory, which is mapped to the network interface. Protection of local and remote memory is provided through proxy memory mappings.

A significant drawback of this protection scheme is that deliberate update transfers cannot cross local or remote page boundaries, since protection is enforced by the ability to reference proxy pages. Therefore, large data transfers must be performed as multiple, individual deliberate update transfers.

This drawback can be overcome by adding a queue on the network interface to store deliberate update transfer requests. This adds some complexity to the design, since it requires an associative memory to allow the host operating system to check whether a particular page is involved in a transfer request. To avoid incorrect data transfers, the operating system must avoid replacing any page that is involved in a pending transfer request.

To evaluate queueing, we implemented a 2-deep queue on the SHRIMP network interface. We tested several SVM applications because we expected them to benefit the most (due to their small transfer sizes). To expose the effect of queueing we used asynchronous sends, i.e. the send operation returned without waiting until the data was sent to the network.

The impact of queueing on performance was very small—within 1% of the total execution time. We suspect this is because the memory bus in our PCs cannot be cycle-shared between the CPU and I/O. As a result, even if the CPU wants to initiate multiple message transfers with queueing, it must compete for the memory bus with the ongoing DMA.

5 Related Work

Spanning the areas of communications research, parallel system design, and parallel software, the research in this paper relates to several large bodies of prior work. Here we discuss a selection of closely-related papers.

A key contribution of this paper is an empirical design retrospective based on a working 16-node SHRIMP system. In that sense, this paper can be categorized along with previous design evaluations of research machines such as the DASH multiprocessor [32], the Illinois Cedar machine [31], the MIT Alewife multiprocessor [1], and the J-machine multicomputer [37]. SHRIMP has leveraged commodity components to a much greater degree than J-machine, Cedar, Alewife or even DASH, thus this paper focuses primarily on evaluating its custom hardware support for communication.

In terms of networking fabric, the Intel Paragon backplane used in SHRIMP is admittedly not "commodity" hardware, but to first-order it resembles (both in design and performance) current commodity networks such as Tandem's ServerNet [40] and Myrinet [13].

At the network interface, SHRIMP uses its automatic and deliberate update mechanisms to support particular parallel programming models and constructs. This work relates to several prior efforts. Spector [39] proposed a remote memory reference model to perform communication over a local area network and the implementation is programmed in a processor's microcode. This model has been revived by Thekkath et al. [41] using fast traps. Druschel et al. [19] proposed the concept of application device channels which provide protected user-level access to a network interface. U-Net [5] uses a similar abstraction to support high-level protocols such as TCP/IP.

The automatic update mechanism in SHRIMP is derived from the Pipelined RAM network interface [35], but is able to perform virtual memory-mapped communication and map DRAM memory instead of dedicated memory on the network interface board. SHRIMP's automatic update is also similar to MemoryChannel (developed independently and concurrently at Digital), in which memory updates are automatically reflected to other nodes [23]. Page-based automatic-update approaches were also used in Memnet [18], Merlin [36], SESAME [45], Plus [8] and Galactica Net [29]. These prior systems did not, however, provide for both automatic and deliberate update.

This paper also quantifies the relationship between particular low-level hardware primitives and the performance of the higher-level software they support. As with active messages [44], SHRIMP's mechanisms provide low-level support for fast communication and for effective overlap of communication with computation. The "sender-based" communication in Hamlyn also supports user-level message passing, but places more burden on application programs by requiring them to construct their own message headers [15].

Some previous machines have worked to streamline the hardware-software interface by mapping network interface FIFOs into processor registers [14, 24, 37]. Such approaches go against SHRIMP's goal of using commodity CPUs. A slightly less integrated approach—mapping FIFOs to memory rather than registers—was employed in the CM-5 [42]. CM-5 implementation restrictions limited the degree of multiprogramming, however, and applications were still required to construct their own message headers.

Finally, at the applications level, our software evaluations draw on prior work on several programming models. The shared virtual memory used here relates to a significant body of prior SVM research [16, 30, 33, 47]. We also leverage off of the NX model for message passing programs [38].

6 Conclusions

We constructed a 16-node prototype SHRIMP system and experimented with applications using various high-level APIs. We found that the SHRIMP multicomputer performs quite well for applications that do not perform very well with traditional network interfaces.

Using applications built on four different communication APIs, we evaluated several of our design choices. We learned several lessons, many of which we would not have learned without building the real system.

- The virtual memory-mapped communication model allows applications to avoid taking receive-side interrupts and to avoid using explicit receive calls. This improves application performance significantly.

- Building custom hardware was difficult, but it allowed us to achieve significantly better performance than was possible with off-the-shelf hardware, and it allowed us to evaluate issues like automatic update vs. deliberate update, which would not have been possible otherwise.

- The automatic update mechanism is quite useful for applications using the native VMMC API and for shared virtual memory applications, but it does not help message-passing applications, which perform better when using a high-performance deliberate update communication mechanism.

- The user-level DMA mechanism can significantly reduce the overhead of sending a message, and leads to significantly better performance than even an aggressive kernel-based implementation. On the other hand, many of our ideas about how to design an aggressive kernel-based implementation came from our study of SHRIMP.

- The automatic update combining mechanism can significantly reduce the network traffic by combining consecutive updates into a single packet. Combining is most useful when the automatic update mechanism is used to replace the deliberate update mechanism for bulk data transfers. But combining provides little performance benefit for SVM applications and our Radix-sort application which uses the native VMMC API. This result surprised us.

- We see no application performance improvements with a hardware mechanism to queue multiple asynchronous deliberate update requests because the memory bus in our PCs cannot be cycle-shared between the CPU and I/O.

- We learned that a small outgoing FIFO is adequate for our network interface due to both the existence of FIFOs in the network interface chip and the constrained bus arbitration strategy of the PC nodes in our system.

Although some of these results were what we expected, others took us by surprise. Building and using the system gave us a much better understanding of the design tradeoffs in cluster architectures.

We look forward to greater insight as we continue to use the SHRIMP system.

7 Acknowledgements

This project is sponsored in part by ARPA under grant N00014-95-1-1144, by NSF under grant MIP-9420653, and by Intel Corporation. Edward Felten is supported by an NSF National Young Investigator Award and a Sloan Fellowship. Margaret Martonosi is supported in part by an NSF Career award.

We would like to thank Paul Close, George Cox and Justin Rattner for helping us access the routing network technology used in the Intel Paragon multicomputer, David Dunning and Roger Traylor for patiently helping us understand the details of the Intel iMRC and NIC, and Konrad Lai and Wen-Hann Wang for helping us understand the Pentium memory subsystem. Malena Mesarina designed an experimental version of the network interface, and David Oppenheimer contributed with system programming and testing. We also would like to thank David DeWitt, Richard Lipton, and Jeffrey Naughton for their help to start the research project, and Michael Carey for his contribution of the acronym SHRIMP.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Machenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, pages 2–13, May 1995.

[2] Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Kai Li. Design and Implementation of NX Message Passing Using SHRIMP Virtual Memory-Mapped Communication. In *Proceedings of the International Conference on Parallel Processing*, August 1996.

[3] Richard Alpert and James Philbin. cBSP: Zero-Cost Synchronization in a Modified BSP Model. Technical Report 97-054, NEC Research Institute, February 1997.

[4] J. Asplin and S. Mehus. On the Design and Performance of the PARFUM Parallel Fault Tolerant Volume Renderer. Technical Report 97-28, Univerity of Tromso, Norway, January 1997.

[5] A. Basu, Buch V, Vogels W, and von Eicken T. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 40–53, December 1995.

[6] BCPR Services Inc. *EISA Specification, Version 3.12*, 1992.

[7] Angelos Bilas and Edward W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *IEEE Transactions on Parallel and Distributed Computing*, February 1997.

[8] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 115–124, May 1990.

[9] Matthias A. Blumrich. *Network Interface for Protected, User-Level Communication*. PhD thesis, Department of Computer Science, Princeton University, June 1996. Available as Technical Report TR-522-96.

[10] Matthias A. Blumrich, Cezary Dubnick, Edward W. Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *IEEE 2nd International Symposium on High-Performance Computer Architecture*, pages 154–165, February 1996.

[11] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE MICRO*, 15(1):21–28, February 1995.

[12] Matthias A. Blumrich, Kai Li, Richard D. Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan S. Sandberg. Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[13] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabig-per-Second Local Area Network. *IEEE MICRO*, 15(1):29–36, February 1995.

[14] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing '88*, pages 330–339, 1988.

[15] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 245–260, October 1996.

[16] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[17] Stefanos N. Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. In *Proc. of 1st Intl. Workshop on Communication and Architectural Support for Network-Based Parallel Computing (Proceedings available as Lecture Notes in Computer Science 1199)*, February 1997.

[18] G. S. Delp, D. J. Farber, R. G. Minnich, J. M. Smith, and M. C. Tam. Memory as a Network Abstraction. *IEEE Network*, 5(4):34–41, July 1991.

[19] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a High-Speed Network Adapter: A Software Perspective. In *Proceedings of SIGCOMM '94*, pages 2–13, September 1994.

[20] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the IEEE 11th International Parallel Processing Symposium*, April 1997.

[21] Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the IEEE 8th International Parallel Processing Symposium*, April 1996.

[22] Edward W. Felten, Richard Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Liviu Iftode, and Kai Li. Early Experience with Message-Passing on the Shrimp Multicomputer. In *Proceedings of the 23nd Annual Symposium on Computer Architecture*, pages 296–307, May 1996.

[23] Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[24] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of 5th International Conference on Architectur al Support for Programming Languages and Operating Systems*, pages 111–122, October 1992.

[25] Liviu Iftode, Cezary Dubnicki, Edward Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of IEEE 2nd International Symposium on High-Performance Computer Architecture*, February 1996.

[26] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[27] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[28] Intel Corporation. *Express Platforms Technical Product Summary: System Overview*, April 1993.

[29] Andrew W. Wilson Jr. Richard P. LaRowe Jr. and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 246–255, May 1993.

[30] P. Keleher, A.L. 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.

[31] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U.M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoefinger, G. Jaxon, Z. Li, T. Murphy, J. Andrewes, and S. Turner. The Cedar System and an Initial Performance Study. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 213–223, May 1993.

[32] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The Stanford DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[33] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume II Software, pages 94–101, August 1988.

[34] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986. A revised version appeared in *ACM Transactions on Computer Systems, 7(4):321–359, November 1989*.

[35] Richard Lipton and Jonathan Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[36] Creve Maples. A High-Performance, Memory-Based Interconnection System For Multicomputer Environments. In *Proceedings of Supercomputing '90*, pages 295–304, November 1990.

[37] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation". In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 224–235, May 1993.

[38] Paul Pierce. The NX Message Passing Interface. *Parallel Computing*, 20(4), April 1994.

[39] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):260–273, April 1982.

[40] ServerNet Interconnect Technology. http://www.tandem.fi/product/snet1.htm, 1997.

[41] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, November 1994.

[42] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, November 1992.

[43] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

[44] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[45] Larry D. Wittie, Gudjon Hermannsson, and Ai Li. Eager Sharing for Efficient Massive Parallelism. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages 251–255, August 1992.

[46] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 24–37, Santa Margherita Ligure, Italy, June 1995.

[47] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, October 1996.