

Realizing the Performance Potential of the Virtual Interface Architecture

Evan Speight, Hazim Abdel-Shafi, and John K. Bennett

Rice University

Department of Electrical and Computer Engineering, MS 366

Houston, Texas 77005

+1 (713) 737-5672

{espeight,shafi,jkb}@rice.edu

ABSTRACT

The Virtual Interface (VI) Architecture provides protected user-level communication with high delivered bandwidth and low per-message latency, particularly for small messages. The VI Architecture attempts to reduce latency by eliminating user/kernel transitions on routine data transfers and by allowing direct use of user memory for network buffering. This results in significantly lower latencies than those achieved by network protocols such as TCP/IP and UDP. In this paper we examine the low-level performance of two VI implementations, one implemented in hardware, the other implemented in device driver software. Using a set of low-level benchmarks, we measure bandwidth, latency, and processor utilization as a function of message size for the GigaNet cLAN and Tandem ServerNet VI implementations. We report that both VI implementations offer significant performance advantage relative to the corresponding UDP implementation on the same hardware. We also investigate the problems associated with delivering this performance to distributed applications running on clustered multiprocessor workstations. Using an existing MPI library implemented using UDP as a baseline, we explore several performance and implementation issues that arise when adapting this library to use VI instead of UDP. Among these issues are memory registration costs, polling vs. blocking, reliability of delivery, and memory-to-memory copying. By eliminating explicit acknowledgements, reducing memory-to-memory copying, and choosing the most appropriate synchronization primitives, we reduced the message latency seen by user applications by an average of 55% across all message sizes. We also identify several areas that offer the potential for further performance enhancement.

1. INTRODUCTION

Over the past decade the peak bandwidth of networks used to connect workstation-class machines has increased by two orders

of magnitude, while the actual message latencies experienced by applications using these networks have at best seen only modest decrease. The primary reason for this discrepancy is the continued high software overhead associated with each message. The high cost of network access exists primarily because network resources have traditionally been managed exclusively by the kernel. The host operating system multiplexes accesses to the network hardware across process-specific endpoints set up through a standard set of network APIs. While providing a simple interface for network communication, the drawback to this approach is that nearly every network access requires one or more traps into the kernel, significantly increasing the amount of overhead associated with network API calls. Another source of poor performance is the copying of data at two main levels in the critical network path: copying into and out of kernel data buffers, and copying data between user buffers at the application level and in standard message passing APIs commonly used to write distributed applications.

In order to address these problems, researchers have focused on moving communication code out of the kernel and into user space. Building on ideas in academic research on user-level communication (among them U-Net [13], VMMC [12], Active Messages [8], and application device channels [11]), Compaq, Intel, and Microsoft have jointly developed the Virtual Interface Architecture (VI) Specification [9]. The VI Specification describes a network architecture that provides user-level data transfer. Several vendors have recently announced VI implementations, including GigaNet's hardware implementation of the VI Architecture on their proprietary cLAN network interface card, and three software emulations of the VI Architecture on existing hardware: ServerNet (Tandem), Myrinet (Myricom), and Ethernet (Intel). Hardware implementations of the VI Architecture provide hardware support for VI context management, queue management, and memory mapping functions in addition to standard network communication functions. Software emulations provide this functionality in NIC firmware (Myrinet), in special purpose device drivers (ServerNet), or in an intermediate driver layered on top of the standard network driver (Gigabit Ethernet).

In this paper we first quantify the low-level network performance of two VI implementations: GigaNet and ServerNet. We compare the performance of VI to UDP on the same hardware in order to gain an understanding of the potential performance impact of moving to a user-level network layer. We found that in both cases the VI implementation substantially outperformed UDP on the

same hardware. For small messages (<1K), latency improved by an average of 72% and 40% for GigaNet and ServerNet, respectively. Large message (1K to 64K) performance improved by 60% and 25%.

In the second portion of this paper, we examine how to translate this improved performance into reduced application execution time. We discuss memory registration costs, data copying, reliability of delivery, and polling vs. blocking receives in the context of using VI to implement a standard message-passing interface such as MPI [1]. We use the implementation of `MPI_Recv()` and `MPI_Send()` from the MPI library to guide our discussion of how to address these issues.

The remainder of the paper is organized as follows: Section 2 presents an overview of the Virtual Interface Architecture. Section 3 describes the methodology used to measure low-level performance. In Section 4 we present and discuss the results of our low-level experiments. Section 5 discusses how to deliver the low-level VI performance to application programs. Related work is described in Section 6, and our conclusions are presented in Section 7.

2. VI ARCHITECTURE OVERVIEW

The Virtual Interface Architecture represents a significant deviation from traditional OS-network interfaces, placing more direct access to the network in the user space while attempting to provide the same protection as that provided by operating system-controlled protocol stacks. The VI Architecture provides hardware support for direct user access to a set of virtual network interfaces, typically without any need for kernel intervention.

The VI Architecture is designed to address three important problems associated with the high cost of network access:

- Low bandwidth – Network-related software overhead limits the usable bandwidth of the network. In many instances only a fraction of the possible network bandwidth can be utilized.
- Small message latency – Because processes in a distributed system must synchronize using network messages, high latency for typically small synchronization messages can greatly reduce overall performance.
- Processing requirements – The overhead of message processing can significantly reduce processing time that the CPU can dedicate to application code.

The VI Architecture is designed to reduce the amount of processing overhead of traditional network protocols by removing the necessity of a process taking a kernel trap on every network call. Instead, consumer processes are provided a direct, protected interface to the network that does not require kernel operations to send or receive messages. Each such virtual interface is analogous to the socket endpoint of a traditional TCP connection: each VI is bi-directional and supports point-to-point data transfer. Support for these virtual interfaces is intended to be implemented in hardware on the network interface card (NIC). The network adapter performs the endpoint virtualization directly and performs the tasks of multiplexing, de-multiplexing, and data transfer scheduling normally performed by an OS kernel and device driver. At the time of writing, the majority of VI implementations do not support these functions in hardware. Instead, standard NIC's are used, and the necessary VI support is emulated by device driver software.

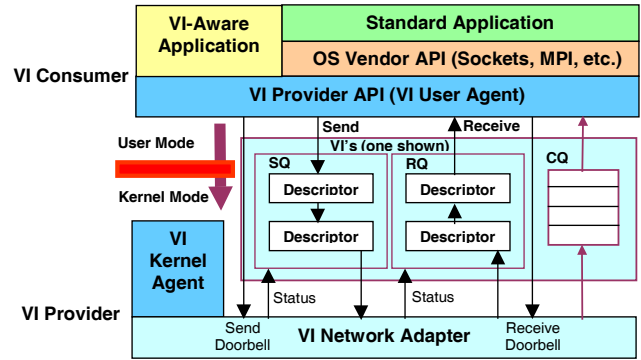


Figure 1. Diagram of the Virtual Interface Architecture

Figure 1 depicts the organization of the Virtual Interface Architecture. The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The VI Provider consists of the VI network adapter and a Kernel Agent device driver. The VI Consumer is generally composed of an application program and an operating system communication facility such as MPI or sockets, although some “VI-aware” applications communicate directly with the VI Provider API. It is the development of this OS vendor API that is the subject of the discussion in Section 5. After connection setup by the Kernel Agent, all network actions occur without kernel intervention, resulting in significantly lower latencies than network protocols such as TCP/IP. Traps into kernel mode are only required for the creation and destruction of VI's, VI connection setup and teardown, interrupt processing, registration of system memory used by the VI NIC, and error handling. VI Consumers access the Kernel Agent using standard operating system mechanisms.

A VI consists of a Send Queue and a Receive Queue. VI Consumers post requests (Descriptors) on these queues to send or receive data. Descriptors contain all of the information that the VI Provider needs to process the request, including pointers to data buffers. VI Providers asynchronously process the posted Descriptors and mark them when completed. VI Consumers remove completed Descriptors from the Send and Receive Queues and reuse them for subsequent requests. Both the Send and Receive Queues have an associated “Doorbell” that is used to notify the VI network adapter that a new Descriptor has been posted to either the Send or Receive Queue. In a hardware VI Architecture implementation, the Doorbell is directly implemented on the VI Network Adapter and no kernel intervention is required to perform this signaling. The Completion Queue allows the VI Consumer to combine the notification of Descriptor completions of multiple VI's without requiring an interrupt or kernel call.

2.1 Connection Procedures

The VI Architecture provides a connection-oriented networking protocol similar to TCP. Programmers make operating system calls to the Kernel Agent to create a VI on the local system and to connect it to a VI on a remote system. Once a connection is established, the application send and receive requests are posted directly to the local VI. A process may open multiple VI's between itself and other processes, with each connection transferring information between the hosts. The VI architecture provides both reliable and unreliable delivery connections. The

expense of setting up and tearing down VI's means that connections are typically made at the beginning of program execution.

2.2 Memory Registration

The VI architecture requires the VI Consumer to register all send and receive memory buffers with the VI Provider in order to eliminate the copying between kernel and user buffers. This copying typically accounts for a large portion of the overhead associated with traditional network protocol stacks. The registration process locks the appropriate pages in memory, allowing for direct DMA operations into user memory by the VI hardware without the possibility of an intervening page fault. After locking the buffer memory pages in physical memory, the virtual to physical mapping and an opaque handle for each memory region registered are provided to the VI Adapter. Memory registration allows the VI Consumer to reuse registered memory buffers and avoid duplicate locking and translation operations. Memory registration also takes page-locking overhead out of the performance-critical data transfer path. Since memory registration is a relatively expensive VI event, it is usually performed once at the beginning of execution for each buffer region.

2.3 Data Transfer Modes

The VI Architecture provides two different modes of data transfer: traditional send/receive semantics, and direct reads from and writes to the memory of remote machines. Remote data reads and writes provide a mechanism for a process to send data to another node or retrieve data from another node, without any action on the part of the remote process (other than VI connection). The send/receive model of the VI Architecture follows the common approach to transferring data between two endpoints, except that all send and receive operations complete asynchronously. Data transfer completion can be discovered through one of two mechanisms: polling, in which a process continually checks the head of the Descriptor Queue for a completed message; or blocking, in which a user process is signaled that a completed message is available using an operating system synchronization object.

3. METHODOLOGY

3.1 Hardware

The results presented in Sections 4 and 5 were obtained on a network of Compaq ProLiant 5500 Server systems. Each server has four 450 MHz Xeon processors with 1Mbyte of L2 cache and 512 Mbytes of interleaved EDO DRAM. Each system has two 33 MHz, 32-bit PCI busses, and all run Windows NT 4.0, Service Pack 4. Our baseline case for each network is the performance of the Winsock 2 UDP stack that ships with Windows NT. The relevant characteristics for each network used in this paper are given in Table 1.

3.2 VI Implementations

Two VI implementations are used in this study: GigaNet cLAN and Tandem ServerNet. The GigaNet cLAN NIC implements VI in hardware as described in Version 1.0 of the VI Specification. The cLAN implementation uses a proprietary network architecture that follows the ATM 8.03 standard for communication. Tandem has announced plans to provide VI hardware support on the next generation of ServerNet NIC's, but current VI functionality is

achieved through software emulation. This emulation consists of a small kernel agent loaded with the software VI transport protocol stack. We used the Version 1.0.10 Kernel Agent provided by Tandem. Although the performance of the VI software emulation is substantially lower than what can be expected with hardware support, the observed network latencies were still lower than the latencies associated with the corresponding UDP network protocol stack. Our intent is not to compare the performance of the software VI implementation with the hardware VI implementation, but rather to examine how effective each is in providing improved performance over UDP on their respective hardware. We have also included performance measurements for the commercially available Packet Engines Gigabit Ethernet UDP implementation as a baseline case for comparison purposes.

<i>NIC</i>	<i>Peak Bandwidth</i>	<i>Switch Latency</i>	<i>Configuration Used</i>
cLAN (GigaNet)	125 MB/sec	480 ns	GNN 1000 NIC GNX 5000 Switch
ServerNet (Tandem)	50 MB/sec	300 ns	ServerNet PCI NIC ServerNet Switch (X fabric only)
Gigabit Ethernet (Packet Engines)	125 MB/sec	500 ns	GNIC II NIC FDR – 12 Hub

Table 1. Network parameters.

3.3 Evaluation Benchmarks

We performed the following measurements to gauge low-level network performance:

- Bandwidth and latency – We measure the bandwidth and latency between two processes sending equal amounts of data back and forth repeatedly. We vary the message size from 1 byte to 64K bytes and report the observed average bandwidth used and the average one-way latency. Additionally, we report the percentage of the advertised peak bandwidth each configuration utilizes in an attempt to provide a common basis of comparison between the different architectures.
- Polling vs. blocking – As mentioned in Section 2.3, VI supports both polling and blocking mechanisms for determining the completion status of a send or receive operation. We examine the performance difference between these two completion notification mechanisms. The results presented in Section 4.1 show bandwidth and latency measurements for blocking receives. The effect of polling is examined in Section 4.2.
- Processor utilization – We report the processor utilization measured during the bandwidth and latency test. Because the low-level tests do nothing but communicate, this measurement shows the processor overhead incurred in performing network-related tasks.

4. Low Level Benchmark Results

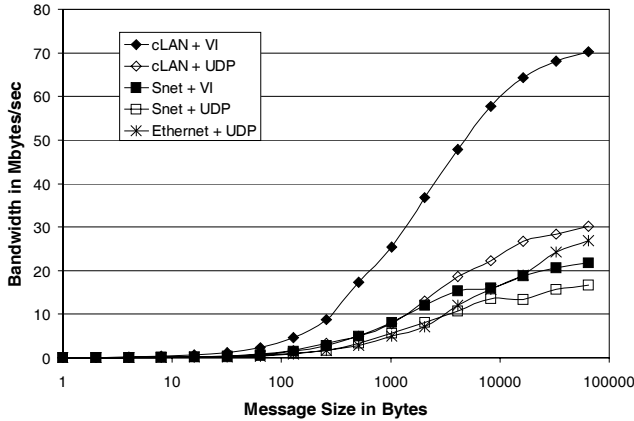


Figure 2. Observed Bandwidth

4.1 Observed Bandwidth and Latency

Figure 2 shows the observed bandwidth in Mbytes/sec for each network protocol configuration for powers of two message sizes between 1 byte and 64 Kbytes. Because the network configurations have different peak bandwidth capabilities, meaningful comparisons can only be made between the two different protocols on the same network architecture. For each network configuration, we see that the VI implementation achieves a higher peak bandwidth than its corresponding UDP counterpart, due in large part to the reduced software overhead associated with the VI Architecture. For the cLAN network, the presence of hardware support for VI improves performance over that of UDP on cLAN by an average of 62% over all message sizes. VI emulation provides an average improvement of 27% for ServerNet.

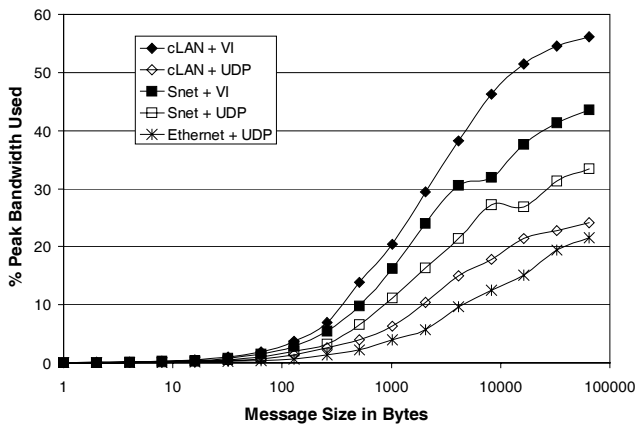


Figure 3. Percentage of Peak Bandwidth Used

Figure 3 shows the observed bandwidth of each network configuration normalized to the peak achievable bandwidth advertised by the manufacturer. Figure 3 therefore gives an indication as to how efficiently each implementation makes use of the available network resources. We found that the cLAN and Ethernet UDP implementations use a smaller percentage of the available bandwidth than ServerNet with UDP. However, since the maximum achievable bandwidth of ServerNet is 2.5 times lower than either the Ethernet or cLAN network, it is easier to

realize a larger percentage of the peak performance due to the fixed costs associated with network calls, and the lower bandwidth demands required from the system memory and I/O buses. For the VI implementations, the hardware-assisted cLAN network is able to utilize a higher percentage of the peak bandwidth than the emulated VI currently used in ServerNet due to moving descriptor manipulation into hardware.

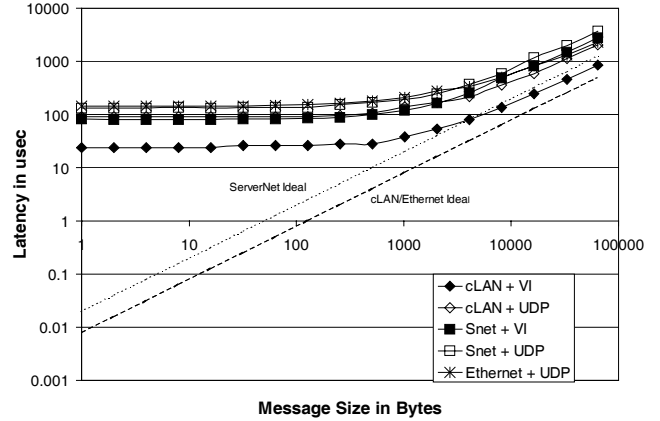


Figure 4. Observed Unidirectional Latency

Figure 4 depicts the observed unidirectional message latency, measured in microseconds, relative to the message size. The data in Figure 4 is plotted on a log-log scale to more distinctly show the performance at small message sizes. For small messages even the best VI implementation (hardware supported cLAN) delivers lackluster performance as compared to the ideal (wire time) latency provided by the network hardware. cLAN VI provides 24 μ sec latency for messages under 32 bytes, more than 73% better than that seen with UDP on cLAN. ServerNet VI also provides a substantial reduction in latency compared to UDP for small messages. Figure 4 indicates that as message sizes increase, software overhead diminishes in importance, and all network implementations approach their ideal latencies.

4.2 The Effect of Polling vs. Blocking Receives

The VI specification allows both polling and blocking (waiting) communication operations. To investigate the effect of the completion policy on performance, we ran the low-level benchmark both with polling and blocking receives for the two VI implementations. Polling always achieves the highest performance for both the cLAN and ServerNet VI implementations. This is mostly due to the high operating system overhead associated with waking up threads upon message receipt. Figure 5 shows the ratio of message latency between polling and blocking for various message sizes. For ServerNet VI, polling latency is an average of 0.83 times that of blocking latency for small messages. This ratio gradually approaches one as message sizes increase and operating system signaling overheads become less of a factor in the overall communication time. cLAN VI achieves a larger reduction in latency with polling, ranging from 0.37 times that of blocking receives with small messages and approaching unity with large messages.

Although these results suggest that polling is a superior alternative to blocking, some caution is warranted. As a practical matter in a system designed to execute distributed applications, it is usually too costly to dedicate an entire processor to polling for

incoming messages because this potentially wastes processor cycles that could be otherwise used for useful computation.

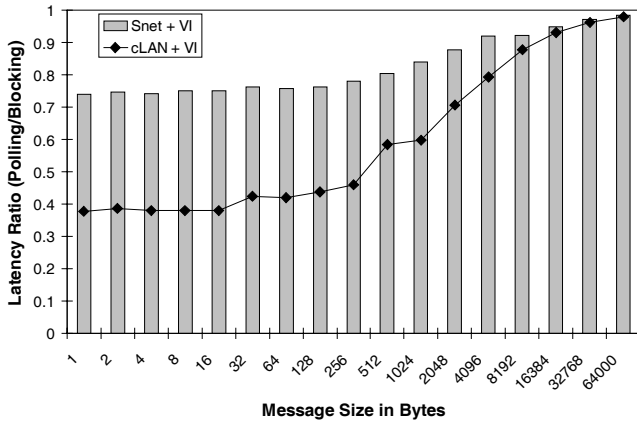


Figure 5. Ratio of Latency for Polling vs. Blocking

Additionally, if polling is implemented in hardware, it will place a heavier load on the I/O bus because each poll results in an I/O bus transfer, slowing down network transfers to the memory bus. Our experiments indicate that the strict use of polling actually achieves lower overall performance on distributed applications. A compromise solution that uses polling for a certain number of iterations before eventually blocking appears to be most appropriate for multi-threaded applications.

4.3 Processor Utilization Measurements

Figure 6 depicts the load that each networking protocol and architecture places on the host processor for both polling and blocking receives. The results in Figure 6 were obtained by utilizing the Xeon’s hardware performance counter to sample the processor utilization every 1 msec. Because of the perturbation resulting from this sampling, the results vary from run to run and explain the uneven nature of the curves.

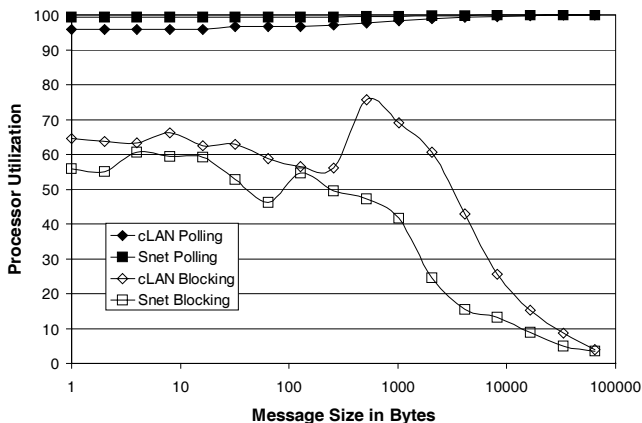


Figure 6. Processor Utilization with VI

When blocking is utilized, both implementations exhibit over 50% processor utilization for small messages in order to process incoming messages and set up for response messages. Because of the reduced latency in the hardware enabled VI implementation, the processor utilization for cLAN VI is higher than the software

implementation on ServerNet because the same number of messages must be sent in a smaller amount of time. As the message size increases, the load placed on the processor decreases due to the DMA operations used to actually place the data “on the wire”, significantly reducing the requirements placed on the processor.

The polling results indicate a very different trend, with both implementations requiring the processor to spend virtually the entire time polling for incoming messages. As observed in Section 4.2, this will cause the polling processor to be unable to perform computation, and may result in worse overall performance on distributed applications.

5. MPI PERFORMANCE USING VI

In Section 4, we observed that for low-level benchmarks the Virtual Interface Architecture offered significant performance advantages relative to UDP. The challenge to the system designer is delivering this performance to distributed applications. Large- or medium-scale distributed applications are rarely written to directly use the underlying network protocol API (i.e., the VI Programming Layer (VIPL) for VI or Winsock for UDP/TCP). Instead, software layers such as PVM, MPI, or software DSM are used to provide ease of use, portability, and standardization across a wide variety of platforms. These improvements in usability come at the cost of increased software overhead. It is therefore this software layer that must be optimized for the potential performance provided by VI networks to be fully realized by user applications.

In this section, we discuss the issues that must be addressed in order to deliver this performance. We describe a preliminary implementation of the `MPI_Send()` and `MPI_Recv()` functions [1] on the cLAN VI, and then describe several ways in which the VI implementation of these functions can be optimized by taking advantage of differences between the VI networking layer and the UDP socket-based networking layer.

Figure 7 shows the results of the low-level latency benchmark described in Section 4 (labeled *VI - Native*), and the results from the same test run using MPI calls instead of VIPL calls (labeled *MPI - VI - Baseline*). These results were obtained by simply replacing the existing UDP sends and receives in the Brazos [23] MPI implementation with the corresponding VI calls, without regard to optimizing the resulting MPI implementation in any way to use VI. For comparison purposes, the results obtained using the cLAN / UDP MPI implementation are also shown (labeled *MPI - UDP*). The dramatic difference in latency between the *VI - Native* and the *MPI - VI - Baseline* curves demonstrates the high cost associated with using the MPI programming environment. It is this difference that must be reduced in order to deliver the high performance potential of VI to end-user applications. In the next three sections, we describe three mechanisms for accomplishing this goal. The performance impact of these optimizations, in isolation and in aggregate, is shown in Figure 7. An expanded view for message sizes less than 1K is shown in Figure 8.

5.1 Eliminating Explicit Acknowledgements

Many messaging layers utilize an unreliable protocol such as UDP in order to provide high performance on networks of workstations. This means that the messaging layer itself must provide the reliability guarantees expected by the user in the form

of explicit acknowledgements and retries. These acknowledgements adversely impact application performance.

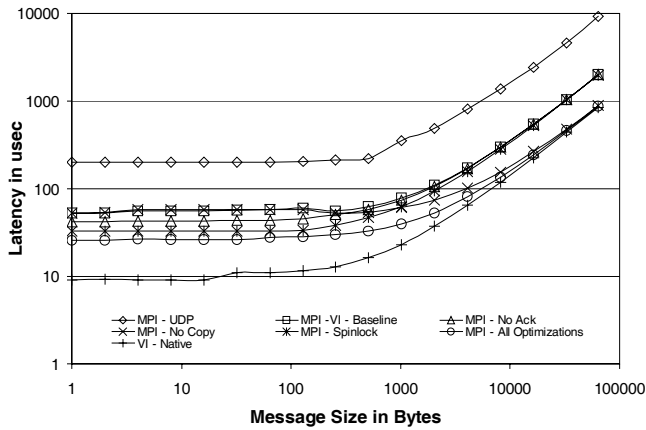


Figure 7. Improving the Performance of MPI on VI

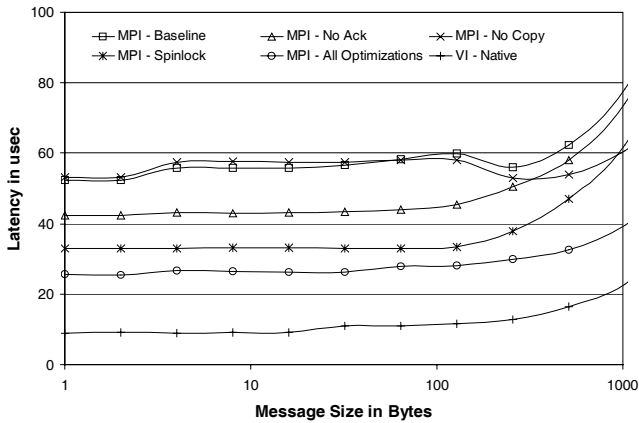


Figure 8. Expanded View (Note: Y axis is not log scale)

The Virtual Interface Architecture provides a reliable-delivery connection, which can be used to remove the need for user-level retries and explicit acknowledgements. Because much of the associated VI functionality is expected to be provided in hardware, this capability should be a less expensive alternative to providing reliable end-user communication in the messaging layer. The curve in Figure 7 labeled *MPI - No Ack* shows the performance advantage obtained by switching from an unreliable to a reliable VI connection, thereby eliminating the explicit acknowledgements used in the baseline MPI messaging layer. This optimization provides an average 19.5% performance improvement over the *MPI - VI - Baseline* case for messages less than 1K. The improvements observed result from the elimination of setting up and sending an extra reply message and the resulting wait time at the original sender. As message sizes increase, the time to send the acknowledgement decreases relative to the time to send the original message, decreasing the performance benefits of this optimization.

5.2 Eliminating Unnecessary Copying

The *MPI - UDP* and *MPI - VI - Baseline* MPI implementations use a single set of receive buffers that all incoming messages are received into and sent from. These buffers are allocated and

registered with the VI NIC at the beginning of program execution, and incoming messages are copied out of these common buffers into posted user buffers before the computation thread is allowed to proceed. By allowing the user to register an arbitrary buffer (this capability is not part of the MPI Protocol Specification), we can avoid this copying. In this case, messages are sent and received directly from the user-supplied buffer. The curve labeled *MPI - No Copy* in Figure 7 shows that eliminating this copying reduces the latency an average of approximately 46% across all message sizes from the *MPI - VI - Baseline* case, with the performance advantage being higher with large messages because these take longer to copy. Memory constraints may still make large buffers problematic, thus requiring aggressive buffer management by the programmer in spite of this optimization.

5.3 Choosing a Synchronization Primitive

The Brazos parallel programming environment utilizes a dedicated messaging thread that receives all messages destined for user-level threads on a specific node. User-level threads send requests and then wait for the messaging thread to signal them when their request has completed and the reply has been received. This signaling mechanism can be either an operating system event, or simply a shared variable that is used as a spin lock. For example, when using the Windows NT operating system, the former mechanism corresponds to waiting using the `WaitForSingleObject()` call; the latter to a call that is simply a wrapper to an atomic swap command found in the x86 instruction set (`InterlockedExchange()`). There are significant differences in the performance of these two mechanisms. The curve in Figure 7 labeled *MPI - Spinlock* shows the reduction in latency obtained for the simple latency test using spin locks instead of operating system primitives for inter-thread synchronization in the MPI implementation. The use of spin locks substantially reduces the latency for messages smaller than 1K, resulting in an average reduction of 38%. As discussed in Section 4.2, this result also shows the significance that operating system provided synchronization mechanisms can add to the performance-critical network access path.

5.4 Aggregate Performance Improvement

Figure 9 shows the relative contributions of each performance improvement to the baseline MPI performance on VI. The curve marked *MPI - All Optimizations* shows that the three improvements together provide a nearly constant performance advantage of 51-55% over the *MPI - VI Baseline* case shown in Figure 7. When messages are small, eliminating user-level copying does not benefit performance because of the small overhead associated with copying small amounts of data. As message sizes increase, this copying becomes more important until nearly the entire aggregate performance benefit is due to the elimination of copying. Acknowledgements and OS signaling are more detrimental to smaller message latency because they are essentially fixed costs. Therefore, reducing these two effects yields performance gains for small messages. Larger message sizes reduce the importance of fixed costs and increase the importance of costs directly relating to the size of the messages. By utilizing all three optimizations, performance benefits are obtainable for all message sizes.

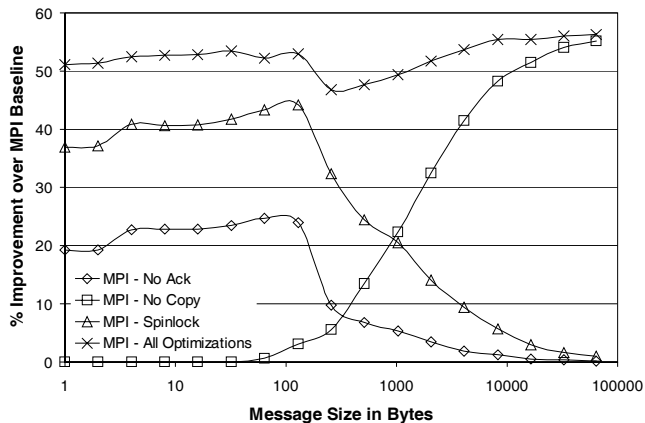


Figure 9. Sources of Performance Improvement

5.5 Other Issues

Although the aggregate result of applying these optimizations is a significant reduction in overall message latency, for messages less than 1K the latency of the *MPI - All Optimizations* curve is still an average of 2.5 times that of the *VI - Native* curve. In this section we discuss two remaining performance bottlenecks, as well as two problems that, although not a factor in our simple benchmark, may prove problematic for more complex applications.

Asynchronous Messaging Thread

The DSM system on which the MPI layer is built was designed such that all messages are received by a dedicated runtime system thread, which is then responsible for signaling the correct application thread upon receiving a message. This works well for software DSM, in which requests tend to be asynchronous, and dedicating a thread to messaging prevents the interruption of computational threads in a clustered SMP environment. It is relatively easy to implement MPI send and receive calls using the same mechanism. However, this introduces extra latency by requiring the signaling of the application thread by the runtime system messaging thread, as discussed in Section 5.3. Allowing user threads to receive messages directly could further improve performance.

Receive Buffer Manipulation

In order to allow the quick retirement of `MPI_Send()` calls, it is essential to provide temporary buffer storage on the receiving side if no corresponding application buffer has yet been posted. The time required to manipulate the data structures needed to provide this mechanism also adds to the latency seen by the sending process.

Point-to-Point vs. Multicast Connections

Connection procedures employed by the message passing layer are similar for VI and TCP. Both protocols are connection-oriented, and require the connections to remain in place throughout the program's execution. Connection procedures are usually implemented completely transparently to the user, typically in an initialization call such as `MPI_Init()`. However, for performance reasons, most software messaging layers such as MPI are written using connection-less protocols like UDP that do not provide reliable communication or flow control. The difference between establishing endpoints in a connection-less

protocol versus a connection-oriented protocol is not in and of itself critical to the performance of a distributed application. However, one of the performance disadvantages of a connection-oriented protocol such as VI or TCP is the lack of support for group communication such as multicast or broadcast. We have previously shown the performance advantages of utilizing multicast in distributed applications [24]. Using VI implies that group communication calls (such as `MPI_Bcast()`) will not be able to take advantage of available multicast/broadcast hardware in order to reduce overall message counts, which can seriously degrade performance.

Memory Registration

User-level communication protocols usually require a registration procedure to be performed for any memory space that will be used for network communication (i.e., as discussed in [13] and [11]). Because user-level protocols typically make use of DMA to transfer data into and out of user buffers, the data must not be allowed to be paged-out by the operating system. Memory registration under VI consists of locking the pages into the host memory and providing an opaque memory handle that is associated with the region of memory. Protocols such as TCP or UDP do not require such operations because the operating system takes care of copying data between user and kernel space buffers. The drawback to the TCP / UDP approach is the higher overhead of network operations. However, there are also disadvantages to requiring a program to register buffer space, as is done with VI:

1. Memory registration is expensive – The time required to register memory in VI can be substantial (see Figure 10), especially if the memory region is large. This prohibits high-performance applications from simply registering and de-registering memory before network accesses, which would introduce unacceptably high software overheads to the network communication path. Registration is also expensive in terms of TLB use, non-paged memory consumed by the process, and the complexity of application code.
2. Buffer space is limited to some amount less than the physical memory on the machine – Many distributed applications send large data arrays between processing nodes. If these arrays are large enough, there may not be any way to pre-register all of the needed space before computation begins. Alternately, applications may only use a small portion of a large array at any particular point in the program, but through the course of the execution the same process may manipulate many different parts of the array. The VI Consumer is then faced with two alternatives: register and de-register memory on-the-fly during computation, or always receive into a single buffer and copy from this buffer into application buffer space during computation. Neither of these alternatives is optimal since they both add latency to the path the application sees during network accesses.

Figure 10 shows that the cost of buffer registration in the cLAN architecture rises sharply for buffer sizes larger than a single x86 memory segment (64K), whereas the cost remains constant at about 15 μ sec for buffer sizes below 64K. By contrast, simply calling the Win32 `VirtualLock()` and `VirtualUnLock()` procedures to alternately pin and unpin pages in memory requires significantly less time for large message sizes.

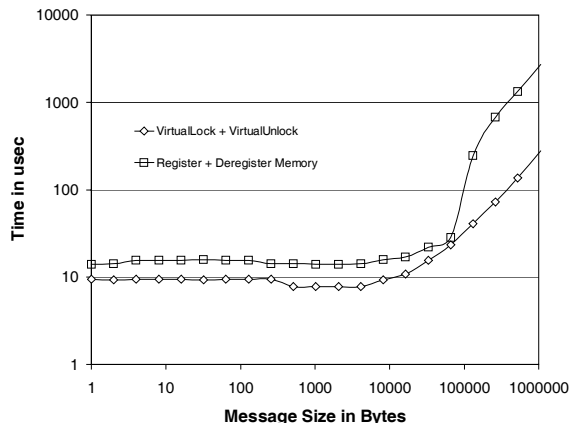


Figure 10. Cost of Memory Registration

6. RELATED WORK

The VI architecture derives from a large body of related work in user-level communication. The VI specification borrows its basic operation from U-Net [13], virtual interfaces to the network from application device channels [11], and remote memory operations from the Virtual Memory Mapped Communication (VMMC) [12] model and from Active Messages (AM) [14]. Fast Sockets [22] offer increased communication performance by collapsing protocol layers, using simple buffer management strategies, and by using “receive posting” to bypass data copying. Thekkath et al. proposed separating network control and data flow, and employed unused processor opcodes to implement remote memory operations [26]. Fast Messages [19] allow direct user-level access to the network interface, but do not support simultaneous use by multiple applications. The HP Hamlyn network implements user-level sends and receives in hardware [7]. ParaStation [27] provides unprotected user-level access to the network interface. With Active Messages [14], each message contains the address of a user-level handler that is executed upon message arrival with the message body as an argument. This allows the programmer and compiler to overlap communication and computation, thereby hiding latency. Other work in the area of user-level communication includes PM [25], LFC [5], Trapeze [28], and BIP [20].

Several extant systems give the network interface access to virtual-to-physical address translation capability in order to facilitate user-level networking. In StarT [3], FLASH [17], and Typhoon [21], the network interface shares the TLB with the host processor. The Meiko CS-2 [15] and U-Net/MM network interfaces both incorporate a TLB for this purpose. Other systems have implemented this functionality in the operating system. Shrimp [6] allows any region in user virtual memory to be used as a network buffer. Like the VI Architecture, this region must be locked down in physical memory before use. The *fbufs* approach proposed by Druschel and Peterson also allows user virtual memory to be used for network buffering, but only pages from a limited range of virtual memory are allowed to be used for this purpose [10].

The effect of communication overhead on application performance has been studied previously. Martin et al. reported a

strong sensitivity of application performance to communication overhead [18]. Keeton et al. measured communication overhead and latency, and reported that increased overhead had a significant impact on the performance of applications with small message request-response behavior [16].

Berry et al. [4] developed prototype implementations of the VI Architecture on 100 Mbit/sec Ethernet, and on Myrinet. Both implementations used software emulations. In addition, they programmed the Lanai network interface controller on the Myrinet NIC to perform the VI emulation. The latter implementation achieved round-trip message latencies of about 55 microseconds for 32-byte messages and about 240 microseconds for 8 KByte messages.

7. CONCLUSIONS

In this paper, we have examined the low-level performance of the Virtual Interface Architecture on two network configurations: cLAN and ServerNet. The performance of VI is significantly better than UDP on the same network, with the hardware implementation achieving a 9 μ sec unidirectional latency for small messages, and 851 μ sec for large messages using native VI library calls. Using VI in an existing distributed programming environment without modifications to address concerns specific to VI resulted in a 6-fold increase in latency for small messages, indicating that much of the performance benefits provided by VI would be negated by a naive implementation of a messaging layer such as MPI. By applying three mechanisms, we were able to lower the latency of the MPI implementation from 53 μ sec to 28 μ sec for small messages, and from 1997 μ sec to 873 μ sec for large messages.

While many existing MPI applications can productively use available VI network performance, including streaming video delivery and other applications that utilize mostly large messages, there exists a large class of applications for which the performance promise of the VI Architecture remains unrealized. Further, it is unlikely that such applications will diminish in importance. In a study of NFS traffic [2], the vast majority of messages were found to be under 200 bytes in size, and these small messages accounted for almost half of the bits sent. Increasing use of object-oriented client-server programming models like CORBA and DCOM, and the increased presence of small databases in the workplace will only exacerbate this problem. In order to meet the challenges presented by these types of applications, intermediate software layers such as MPI must be optimized to expose the full benefit of user-level network access provided by architecture such as VI.

ACKNOWLEDGEMENTS

This research was supported in part by grants and donations from Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Tandem Corporation, Packet Engines, Inc., and by the Texas Advanced Technology Program under Grant No. 003604-022.

REFERENCES

- [1] *MPI: A Message-Passing Interface Standard*, Version 1.0 ed: Message Passing Interface Forum, 1994.
- [2] T. E. Anderson, D. E. Culler, D. A. Patterson, and T. N. Team, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, vol. 15, pp. 54-64, 1995.
- [3] B. S. Ang, D. Chiou, L. Rudolph, and Arvind, "Message Passing Support on StarT-Voyager," MIT Laboratory for Computer Science CSG-Memo-387, 1996.
- [4] F. Berry, E. Delegates, and A. M. Merritt, "The Virtual Interface Architecture Proof-of-Concept Performance Results," . Server Systems Technology, Intel Corporation, 1997.
- [5] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal, "Efficient multicast on Myrinet using link-level flow control," presented at Proceedings of the International Conference on Parallel Processing, pp. 381-390, 1998.
- [6] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," presented at Proceedings of the 21st Annual International Symposium on Computer Architecture, pp. 142-153, 1994.
- [7] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes, "An Implementation of the Hamlyn Sender-Managed Interface Architecture," presented at Proceedings of the Second Symposium on Operating System Design and Implementation, pp. 245-259, 1996.
- [8] B. Chun, A. Mainwaring, and D. Culler, "Virtual Network Transport Protocols for Myrinet," *IEEE Micro*, pp. 53-63, 1998.
- [9] Compaq Corporation, Intel Corporation, and Microsoft Corporation, "Virtual Interface Architecture Specification, Version 1.0," 1997.
- [10] P. Druschel and L. L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility.," presented at Proceedings of the 14th Annual Symposium on Operating System Principles, pp. 189-202, 1993.
- [11] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," presented at Proceedings of the Conference on Communications Architectures, Protocols, and Applications, pp. 2-13, 1994.
- [12] C. Dubnicki, A. Bilas, K. Li, and J. Philbin, "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet.," presented at Proceedings of the International Parallel Processing Symposium, pp. 388-396, 1997.
- [13] T. V. Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," presented at Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp. 40-53, 1995.
- [14] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer, "Active Messages: A Mechanism for Integrating Communication and Computation," presented at Proceedings of the 19th International Symposium on Computer Architecture, pp. 256-266, 1992.
- [15] M. Homewood and M. McLaren, "Meiko CS-2 Interconnect Elan-Elite Design," presented at Hot Interconnects, 1993.
- [16] K. Keeton, D. A. Patterson, and T. E. Anderson, "LogP Quantified: The Case for Low-Overhead Local Area Networks," presented at Hot Interconnects III, 1995.
- [17] J. Kuskin, D. Ofelt, M. Heinrich, *et al.*, "The Stanford FLASH Multiprocessor," presented at Proceedings of the 21st Annual International Symposium on Computer Architecture, pp. 302-313, 1994.
- [18] R. Martin, A. Vahdat, D. Culler, and T. Anderson, "Effects of Communication Latency, Overhead, and Bandwidth in a Clustered Architecture," presented at Proceedings of 24th Annual International Symposium on Computer Architecture, pp. 85-97, 1997.
- [19] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," presented at Proceedings of Supercomputing '95, 1995.
- [20] L. Prylli and B. Tourancheau, "Protocol Design for High Performance Networking: A Myrinet Experience," LIP-ENS, Lyons, France Tech. Report 97-22, 1997.
- [21] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory," presented at Proceedings of the 21st Annual International Symposium on Computer Architecture, pp. 325-337, 1994.
- [22] S. H. Rodrigues, T. E. Anderson, and D. E. Culler, "High-Performance Local Area Communication With Fast Sockets," presented at Usenix 1997 Conference, 1997.
- [23] E. Speight, "Efficient Runtime Support for Cluster-Based Distributed Shared Memory Multiprocessors," Ph.D. Thesis, Rice University, 1997.
- [24] E. Speight and J. K. Bennett, "Using Multicast and Multithreading to Reduce Communication in Software DSM Systems," presented at Proceedings of the Fourth Annual Symposium on High Performance Computer Architecture, pp. 312-321, 1998.
- [25] H. Tezuka, "Pin-down cache: A virtual memory management technique for zero-copy communication," presented at Proceedings of the International Parallel Processing Symposium, pp. 308-314, 1998.
- [26] C. A. Thekkath, H. M. Levy, and E. D. Lazowska, "Separating Data and Control Transfer in Distributed Operating Systems," presented at Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 2-11, 1994.
- [27] T. M. Warschko, J. M. Blum, and W. F. Tichy, "The ParaPC/ParaStation Project: Efficient Parallel Computing by Clustering Workstations," University of Karlsruhe, Department of Informatics Technical Report 13/96, 1996.
- [28] K. Yocum, "Cut-through delivery in Trapeze: An exercise in low-latency messaging," presented at Proceedings of the International Symposium on High Performance Distributed Computing, pp. 243-252, 1997.