

An Implementation and Analysis of the Virtual Interface Architecture

Philip Buonadonna

440 Soda Hall

University of California, Berkeley

Berkeley CA 94720 USA

philipb@cs.berkeley.edu

Andrew Geweke

440 Soda Hall

University of California, Berkeley

Berkeley CA 94720 USA

geweke@cs.berkeley.edu

David Culler

627 Soda Hall

University of California, Berkeley

Berkeley CA 94720 USA

culler@cs.berkeley.edu

Abstract:

Rapid developments in networking technology and a rise in clustered computing have driven research studies in high performance communication architectures. In an effort to standardize the work in this area, industry leaders have developed the Virtual Interface Architecture (VIA) specification. This architecture seeks to provide an operating system-independent infrastructure for high-performance user-level networking in a generic environment. This paper evaluates the inherent costs and performance potential of the Virtual Interface Architecture through a prototype implementation over Myrinet. The VIA prototype is compared against established research user-level networks using simple communication benchmarks on the same hardware. We consider extensions to the VI Architecture that improve its performance for certain types of communication traffic and outline further research areas in the VIA design space that merit investigation.

Keywords:

Virtual Interface Architecture Cluster System-Area Network Interconnect User-Level

Introduction

Networking software has not kept pace with the explosive improvements of physical-layer hardware. Typical protocol overheads can be several times the actual transport latency [2, 3]. While this problem is substantial even for bulk transfers, it becomes especially acute for applications that

frequently send small packets of data, such as fine-grained parallel programs and distributed coherence protocols. In high-performance communication environments, overheads of dozens of microseconds to send a single packet are unacceptable. The need to minimize overhead in the face of high data-link rates is illustrated by Figure 1, which plots the maximum overhead at which half the link bandwidth can be successfully used for several average packet sizes.

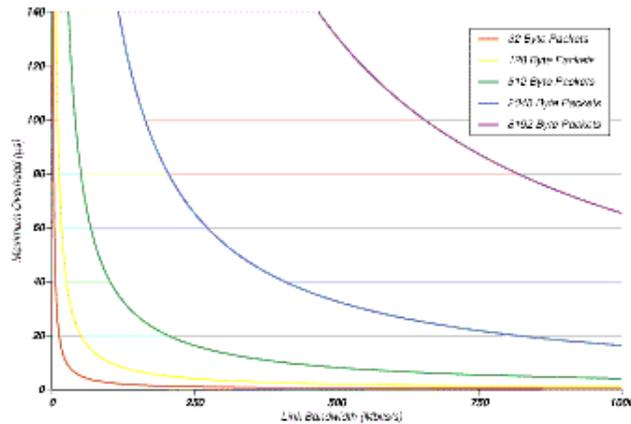


Figure 1: Maximum Allowable Overhead to Achieve a Throughput of One-Half the Link Rate For a Range of Average Message Sizes

There have been many efforts to reduce the overhead of traditional system networking stacks, including optimizations to reduce the frequency of TLB or cache misses and tight integration across protocol layers in the stack (*e.g.*, [2, 3, 4, 16]). Still, in current implementations, overhead remains large when compared to transmission times for typical packet sizes on a gigabit-level interconnect. The alternative solution to this problem is to remove the operating system from the critical path of communication entirely, providing direct user-level networking (ULN). The operating system is used to set up the data structures and mappings required to provide the user process with direct, protected access to the network interface. The process can then send and receive packets without further operating system involvement, thus greatly reducing the communication overhead. The network interface must have a certain degree of intelligence to enforce the protection boundaries established by the operating system. Several academic research efforts in high-performance communications have demonstrated successful ULN solutions [5, 6, 7, 8, 9]. By standardizing the interface between the application and the networking hardware, the Virtual Interface Architecture [1] may bring fast ULN into the mainstream and enable its use by a wide variety of important, demanding applications. Already, some initial proof-of-concept studies have been performed which demonstrate the architecture's performance and ability to support overlying applications [11, 20].

The goal of this paper is to provide a preliminary critical analysis of the VI Architecture, its implementation requirements, and its performance as compared to established academic ULNs. Section 2 gives an overview of the goals and fundamental mechanics of the VI architecture, and Section 3 compares it to established ULN efforts. Section 4 describes our prototype VIA implementation, which is used as a basis of the analysis. Section 5 examines the communication performance obtained with VIA and compares this to other ULNs on the same high-speed network hardware. Section 6 contains an overall evaluation of VIA along with an investigation into possible improvements.

The VI Architecture

This section provides background on VIA, including its design goals, basic structure, primitive operations and assumptions about its underlying hardware and software. More extensive background is provided by the specification [1].

Goals of VIA

The Virtual Interface Architecture is targeted as a communication infrastructure for System Area Networks (SANs). It seeks to enable a wave of new cluster applications that are able to harness the communication performance of these emerging networks. The core of the specification is a set of principles outlining user-level networking -- where the operating system is involved only in infrequent setup operations -- which establish the structures for hardware protection checks and direct access to the network. For this goal to be realized on a broad scale, there must be a specification against which network interface hardware can be designed and an interface against which applications can be programmed. Providing such an interface is challenging, because it needs to be abstract enough to allow for innovation and a range of implementations, yet be concrete enough to build against. A clear design goal in VIA was to preserve the experience with traditional network interfaces and to allow for low-cost implementations. The specification describes a communication library interface (called "VIPL") as well as a descriptor-processing model, similar in structure to many Ethernet adapters. Both of these are examples for reference, rather than hard requirements; by providing both layers, quite a bit of room is left for evolution as experience is gained with this approach.

The specification reveals only portions of the underlying transport mechanisms to applications. Most importantly, the format of a *descriptor* -- a data structure that describes the location and size of a data buffer -- is revealed to applications and thus cannot change. However, queue structures that tie together collections of descriptors and the "doorbell" mechanism that indicates events are hidden from the application by the "VIPL"/user-level library.

VI Architecture Mechanics

The challenge in all ULNs is to provide protected access to shared communication resources. In traditional networks, the user process makes a system call to perform communication. The networking stack multiplexes the traffic from many processes onto the network through a kernel device driver, which manipulates the physical network interface card (NIC). It also de-multiplexes traffic received from the network to the appropriate process. While it is not difficult to map a NIC into a user address space and thus avoid the kernel stack, such an approach either limits communication to a single user or allows one process to compromise the communications of other processes. ULNs allow a user process to access an object in the user address space that represents an *endpoint* of the network. Writing to and reading from this endpoint causes messages to be transmitted and received. The multiplexing and de-multiplexing of traffic between these virtual endpoints and the physical network is performed by the network interface hardware and its control firmware, rather than the kernel.

In VIA, the endpoint is a data structure that closely resembles the DMA descriptor queues of a typical network interface device and is called a *virtual interface*, or VI. The operating system provides creation and destruction of VIs by mapping a portion of the NIC's resources into the address space of the associated user process. This way, an application cannot access the VIs of other

processes.

VIA is a point-to-point, connection-oriented protocol, allowing the operating system to be involved in connecting and disconnecting VIs and providing for further operating system control over the use of the network by controlling the quantity and specifics (*e.g.*, port numbers) of the connections. For an overview of the VI Architecture and its components, we refer the reader to Figure 2.

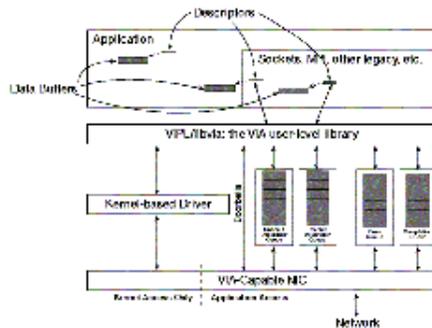


Figure 2. Overview of the VI Architecture

VI setup, connection, and management are accomplished by interacting with a kernel driver via the user-level library (the leftmost path in Figure 2); this kernel driver interacts with the NIC in an implementation-defined manner. The kernel driver typically has the ability to access some portion of the NIC's resources that notifies the NIC of VI creation, connection, and so forth, while the user-level process does not.

Each VI comprises a pair of work queues -- send and receive -- as shown in the lower right portion of Figure 2, and a pair of *doorbells*, the center path in Figure 2. None of these structures are directly visible to an application and are accessed only using the user-level library. Data buffers and the work queues themselves are allocated in an area of the host's memory to which the network card has access. These memory regions are pinned in physical memory and are called *registered memory* in VIA terminology. The application's intent to send or receive packets is communicated by constructing an appropriate descriptor and, via the user-level library, ringing the appropriate doorbell on the NIC. The address mappings involved are set up by the operating system, so correct protection and virtualization is maintained while the operating system remains completely out of the critical send-receive path. Legacy communication layers, such as MPI, Berkeley Sockets, or Microsoft's WinSock can be built on top of this library. Data copies are required only where the higher-level data areas cannot be mapped to VI buffers.

VIA Operations

This section describes the basic operations that an application may perform on a VI and how they would typically be implemented. We provide the names of the calls into the VIA API as defined in the specification [1] to clarify which operations are being performed.

Set-Up and Tear-Down. To set up a VI endpoint, the application calls the user-level library function `VipCreateVi`, which passes the information through to the kernel-level driver; the driver, in turn, passes the creation information to the NIC. It is here that the OS can deny the application access to the network interface or to a particular endpoint.

Register and De-Register Memory. All data buffers and descriptors must reside within a region of "registered memory"; registered memory regions are obtained by calling `VipRegisterMemory`, which interacts with the kernel driver to register a specified region of memory. This registration will usually pin the specified pages into physical memory, thus allowing DMA I/O from the NIC, and communicate the addresses of the registered memory regions to the NIC.

Connect and Disconnect. VIA is a connection-oriented protocol. Before packets can be sent, the endpoint must be connected to a remote endpoint (using `VipConnectRequest`). These calls are passed into the kernel-level driver, which constructs the internal state and protection keys of the VI. The NIC is then notified of the endpoint connection. VIA does not define an addressing scheme, so existing addressing schemes can be reused.

Transmit Data. Three key data structures are used to send data in VIA. First, the data buffer that the application wishes to transmit from is allocated by the application itself within a registered memory region. Typically, one registered memory region is used for many messages. For each message, the application moves data into the buffer and builds a *descriptor*, in registered memory, for the buffer. The descriptor contains the address and length of the data, plus protection and status information. The application, via the user-level library, posts a *doorbell* to indicate that the descriptor is ready for transmission by writing the address of the descriptor to a register on the NIC.

The NIC maintains a queue of doorbells; when the doorbell reaches the front of the queue, the NIC then traverses this double indirection to transmit the data. Finally, the NIC updates the status field of the descriptor, used by the application to determine transmit completion or error.

Receive Data. To receive a message, the application allocates an empty data buffer in registered memory and builds a descriptor for it. It then posts a doorbell to the NIC to indicate that space is available for an incoming packet; both of these steps must be completed before a packet arrives. If a packet arrives and the NIC has no previously posted receive doorbells, the packet is discarded.

When a packet arrives, the NIC uses the doorbell as a starting point and traverses the double indirection, writing the received packet into the allocated buffer. The descriptor is then updated with status information to indicate that the buffer has newly-received data in it.

Semantics. All send and receive operations in VIA are non-blocking, and return as soon as the doorbell has been posted. This allows greater overlap of communication and computations than blocking semantics, and routines are provided to poll or block for completion of a descriptor.

Details. The doorbell mechanism in VIA is implementation-dependent; future VIA-compatible NICs are likely to provide hardware support, while existing ones may resort to *ad hoc* solutions or -- as for legacy Ethernet devices [11] -- use a highly-optimized system call.

The double indirection used in VIA allows a minimal amount of state to be kept on the card; a queue of 32- or 64-bit descriptors is all that is required. This minimal state does, however, exact a performance penalty, as we will see later.

Assumptions of VIA

VIA assumes that the underlying network is highly reliable, if not perfectly so. VIA has several

reliability modes into which an endpoint can be switched. In the least-reliable mode ("unreliable service"), VIA simply attempts to transmit packets. Only if something completely prevents it from placing the packet onto the wire is an error signaled. In reliable delivery mode, VIA signals an error if the packet did not make it onto the wire successfully. Finally, for reliable reception, an error is signaled if the packet is not received *into the destination host's memory* successfully. In all of these schemes, the failure is catastrophic -- all following transactions on that endpoint are discarded and the endpoint is placed into an error state.

The hardware on top of which a VIA implementation is built can be quite arbitrary. The descriptor model allows a degree of compatibility with familiar network technologies. However, some degree of special support for VIA, while not required, is extremely beneficial. Previous implementations at Intel [11] were forced to resort to calling the kernel to post a doorbell due to a lack of flexibility in hardware. In this paper we will show that special support for the doorbell mechanism should be incorporated into VIA-capable hardware to ensure good performance.

Comparison with Existing User-Level Networks

With this basic understanding of VIA, we may compare it to important precursor ULNs and contemporaries that will be used for performance comparisons below. VIA is described as building upon Active Messages, U-Net, Fast Messages, and Virtual Memory-Mapped Communication [9]. We also consider BIP, because it represents close to the minimal message layer for the hardware used in our performance comparison.

Active Messages [5] was the first widely used ULN. Originally developed for parallel supercomputers, it was implemented on a wide variety of network interface hardware, including several fast cluster networks [13]; it was intended as a kind of assembly language for communication. Rather than expose any particular underlying queue structure, it provided a very simple RPC-like programming interface. Small messages correspond essentially to the arguments passed in registers across a procedure call. Large messages are treated as memory-to-memory transfers. The handler fires implicitly when the message is available at the destination with the message data as arguments and executes in the remote user address space. The implementation brings data out of the queues into handler arguments. To avoid deadlock, handlers may execute whenever attempting to send a message. In addition, they may execute either through interrupts or polling the network. The Generic Active Message (GAM) specification [19] was the basis for several of the following ULNs. It associated the user level network implicitly with a process. The later Active Messages, version 2, specification [18] made virtual network endpoints explicit, allowing a user process to possess many endpoints, and integrated the execution model with threads. Endpoints contain simple send and receive queues and may be paged to and from the NIC, but their internal structure is hidden.

U-Net [7] sought explicitly to virtualize the descriptor queue and buffer structure seen by device drivers at user level. It is, of all the ULNs, the most similar to VIA, although it differs in the details of the queue structure and how address resolution for buffer areas is performed. TCP and GAM were built on top of U-Net, with the user process handling error detection, timeout and retry. It was the first solid demonstration of fast user level communication over conventional Ethernet and ATM LANs and advanced the state of application-specific protocols.

Fast Messages [6] used the GAM interface, rather than exposing an underlying queue structure, but eliminated the implicit execution of handlers. Instead, the user process provides buffer storage in

advance for incoming messages (like VIA, but with a simpler structure) and manages the flow control to avoid deadlock. It was very widely used on cluster interconnects and advanced the protocols for resolving endpoint contention with bulk transfers.

Virtual Memory-Mapped Communication (VMMC) [9] is a model for communication quite different from the above systems. Here, a "reflective" memory region is set up across a network, and writes from one side are automatically propagated into the other. This model maintains correct virtualization and multiplexing, as the operating system is involved in creating the required mappings. VMMC, however, exposes no queues and has very basic primitive operations (some of which may be a hardware load or store, in fact). User applications typically build message queues with the communication regions.

BIP [8], the Basic Interface for Parallelism, is an extremely simple ULN, developed for Myrinet, with a single send queue and receive queue for one process on each node. The basic `bip_send()` and `bip_recv()` calls are passed direct pointers to data buffers. While queues are used, they are not directly exposed to the user-level process. We use it for comparison as it is very close to the fastest possible on the Myrinet hardware.

The Berkeley VIA Implementation

We have implemented a subset of VIA and have used this implementation to analyze basic performance characteristics of the architecture. The host platforms for our prototype, outlined in Table 1, provide a good horizontal cross section of widely used computers and operating systems. For the network, we used Myricom's Myrinet M2F [10] with the LANai 4.x-based network interface card. The programmable nature of the Myrinet NICs allowed us to emulate VIA-capable hardware in a flexible, instrumentable system, thus providing insightful analysis. As we demonstrate later, a critical difference between our two platforms is the type of I/O bus: the Solaris platform uses the 25 MHz, virtually-addressed S-Bus, while the Windows NT platform uses the 33 MHz, physically-addressed PCI bus with two to three times the available bandwidth.

Platform	Operating System	CPU/Speed	Bus Type	Installed Memory
Sun Ultra 1	Sun Solaris 2.6	Sun UltraSPARC I, 167 MHz	S-Bus	128 MBytes
Dell Dimension XPS	Microsoft Windows NT	Intel Pentium II, 300 MHz	PCI	128 MBytes

Table 1: Host Configurations

Because of the vague nature of the VIA specification, it is worthwhile to discuss some major decisions we have made in our implementation. For several reasons, we chose to follow the suggested reference implementation contained in the appendices of the VIA specification. Principal among them is our belief that this suggested implementation most closely approximates the specifiers' vision for real-world VIA implementations. Due to the lack of rigor in the specification, implementing something different would have essentially meant creating a new ULN scheme and

dubbing it VIA.

Another decision of our implementation was to keep as little information in the NIC's memory as possible. Although our NICs had 512KBytes or 1MByte of RAM available, we typically used less than 40 KBytes of data space -- all of the VI endpoint descriptor and data buffers exist in host memory, with the associated queue structures abstracted from the NIC. This requires fairly sophisticated logic to be present in the NIC, but little memory; we believe this to be consistent with the specifiers' likely desire to build low-cost VIA NICs using ASICs. The only storage required on the NIC, in fact, is a queue of 4-byte doorbells and staging areas for descriptors and data buffers as well as the firmware program.

Implementation Specifics

To provide protected creation and connection of VIs, a queue for protected commands is created on the NIC at system startup. This queue is mapped only into the kernel driver's memory space on the host, ensuring that only the kernel driver can perform those operations.

Upon creation of a VI, a single page of the NIC's memory is mapped into the requesting process's address space to provide a doorbell mechanism. Doorbells are posted by writing into one of the first two words of this region, which are constantly polled by the NIC's firmware. Additionally, a fixed amount of kernel memory is allocated, pinned into physical memory and returned to the user. This host memory allocation mechanism is substituted for user-registered memory regions to simplify the initial implementation without hindering accurate performance analysis. The appropriate address for the host memory regions is then passed to the card. Destroying a VI is simply the reverse of this process.

The sequence of events occurring when a message is transferred is outlined in Figures 3 and 4. Figure 3 indicates the flow of data during a transmit or receive phase, while Figure 4 indicates the sequence and location of events; we indicate the relevant portions of both figures here.

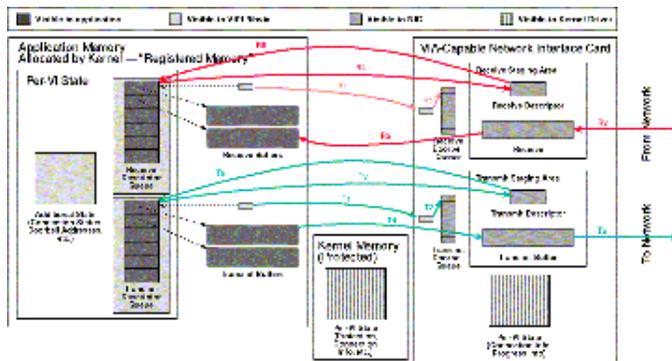


Figure 3. Location of Berkeley VIA Structures with Transmit and Receive Dataflow

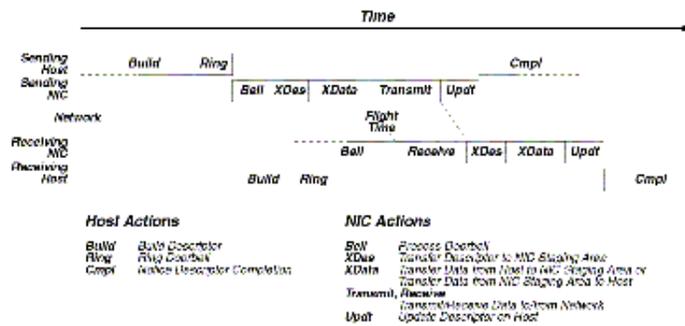


Figure 4. Sequence and Location of Events During VIA Transmit and Receive

To send a message, the application builds its message in a buffer within a registered-memory region and creates a descriptor, also within that region, for the message (*Build*). To notify the NIC of the descriptor's existence, the user-level library posts a doorbell to the NIC (*T1/Ring*). At this point, the application may proceed with computation; all further work is done by the NIC.

The NIC moves the received doorbell into a queue of unprocessed doorbells (*T2/Bell*). When the doorbell reaches the front of the queue, the NIC uses the address contained in the doorbell to locate and transfer the descriptor across the I/O bus to a staging area (*T3/XDes*). Using the information in the descriptor, it transfers the data itself across the I/O bus to a data staging area (*T4/XData*). When the data is available, the NIC places it onto the network using the network address contained in the descriptor (*T5/Transmit*). Once the transfer onto the wire is complete, the NIC updates the status field of the descriptor to signal completion (*T6/Updt*). At this point, the application can detect completion of the message transmission (*Cmpl*) by reading the descriptor's status field.

To receive a message, the application allocates space for data and creates a descriptor for that buffer (*Build*). The user-level library notifies the NIC of this buffer's existence by posting a doorbell to the NIC (*R1/Ring*). The NIC immediately moves the doorbell into a queue (*R2/Bell*). The application may now proceed with computation; the NIC will keep the doorbell in its queue until the associated buffer is required for inbound data.

When inbound data does arrive, the NIC immediately receives the packet into its staging area (*R3/Receive*). Only after examining the header of the packet does it know for which VI the packet is destined. The NIC then uses the next doorbell in that VI's queue to locate and transfer the associated descriptor to the NIC's descriptor staging area (*R4/XDes*). The NIC transfers the inbound message from its staging area to the data buffer indicated in the descriptor (*R5/XData*). Finally, the NIC updates the status field of the descriptor on the host to signal receive completion (*R6/Updt*). Once this is complete, the application can detect receipt of a message (*Cmpl*) by reading the descriptor's status field.

Implementation Limitations

Our implementation suffers from significant costs imposed by the doorbell mechanism. As the Myrinet NIC provides no direct hardware support, doorbells are implemented as single memory locations on the NIC and polled by the firmware, requiring 1.5us per pair. This penalty, significant yet small, would lead to unacceptable performance when scaled to even a moderate number of VIs.

The NIC firmware was written with simplicity and instrumentability in mind, not maximum bandwidth. The NIC firmware completes each transaction in strict sequence, making no attempt to pipeline the available data-transfer engines. This enables easy measurements and breakdown of transaction costs, but limits achievable bandwidth -- we estimate a loss of up to 33%.

We chose to implement a subset of VIA rather than the entire standard, guided by a desire to implement VIA quickly while maintaining accurate results. Because of this, we have left unimplemented scatter/gather capability, reliability modes, error and completion queues, and the RDMA facilities. These facets of VIA are important and warrant further investigation, but are outside of the scope of this paper.

Readers desiring additional details on the implementation are referred to the distribution, including source code, available at <http://www.millennium.berkeley.edu>

Performance Analysis

To evaluate the performance characteristics of our VIA implementation, we chose three principal benchmarks. These included application-to-application latency, streaming bandwidth, and LogP [12] performance breakdown. Where applicable, performance results for similar analysis on U-Net, Active Messages 2, and BIP, all using Myrinet network hardware, are presented. Comparisons with these other architectures are made to illustrate strengths/weaknesses with VIA and suggest various bounds on performance.

Application-to-Application Latency

The first performance aspect analyzed is application-to-application latency, which measures the time for a packet of arbitrary size to be sent from a node across the network to a remote node. This benchmark measures the total time to send a message to a remote node and then for that node to reflect the same message back to the source. The timing interval specifically measures posting receive and send descriptors on the originating node, transmitting the message, receiving and re-transmitting the message on the reflecting node, and re-receipt at the origin. The average of the round-trip measurements is divided in half to obtain one-way latency. Because the different I/O buses have substantially different performance characteristics, we have presented the results separately in Figures 5 and 6.

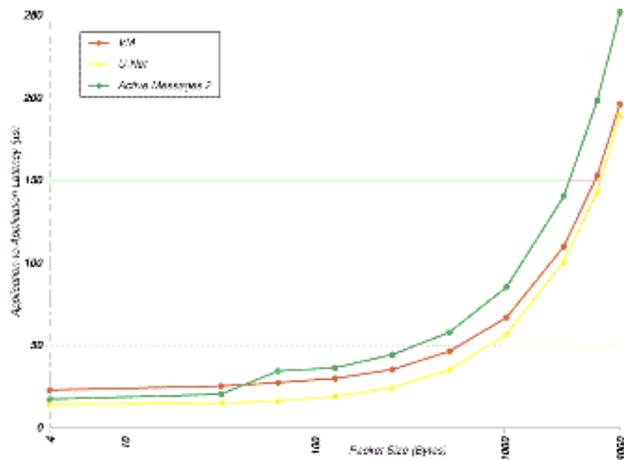


Figure 5: Application-to-Application Latency, S-Bus

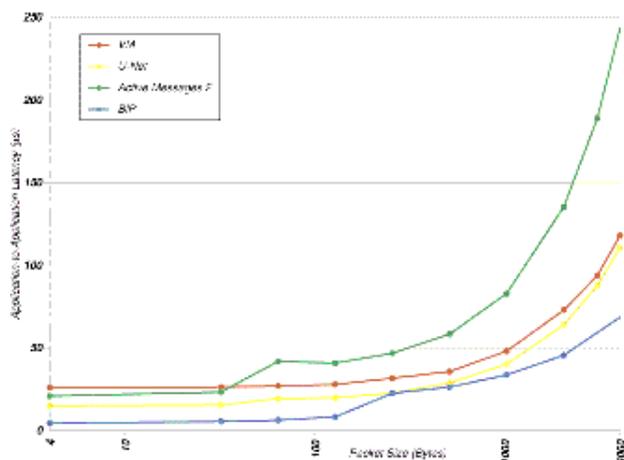


Figure 6: Application-to-Application Latency, PCI

VIA has a minimum transfer time of 23 μ s and transfers additional data at about 185 Mbit/s using an S-Bus-based platform. Using a PCI-based platform, the minimum transfer time is 26 μ s and additional data proceeds at about 360 Mbit/s. The greater minimum transfer time over PCI is attributable to the slower clock rate of the PCI NIC as compared to the S-Bus NIC (33 MHz vs. 37.5 MHz); the greater bandwidth is attributable to the inherent bandwidth of the I/O bus (approximately 1000 Mbit/s vs. approximately 500 Mbit/s).

The overhead incurred by VIA's double-indirection scheme -- 23 μ s S-Bus/26 μ s PCI -- is at least as great as the overhead of any other ULN. Active Messages 2 exhibits a somewhat lower start-up overhead of 17.5 μ s/21 μ s, while U-Net's single indirection and BIP's complete lack of indirection lead to much lower overheads of 14.3 μ s/15.2 μ s and 4.7 μ s (PCI only), respectively.

For larger message sizes, overhead becomes less important. Active Messages 2 requires acknowledgements for reliability and flow control; on the S-Bus platform, this limits it to 140 Mbit/s for larger messages. U-Net and VIA both are limited by the S-Bus's bandwidth; large messages proceed at 185 Mbit/s for both implementations. (Figures do not approach the 500 Mbit/s maximum of the S-Bus because this benchmark does not "stream" packets across the link.)

On the PCI platform, performance on large messages varies much more among implementations. VIA is very comparable, at 360 Mbit/s, to U-Net, which transmits large messages at 340 Mbit/s; this is expected, as, once the extra layer of indirection is removed, U-Net transmits packets in a very similar manner to VIA. AM transmits large messages at 145 Mbit/s; this figure is low due to the extra round-trip incurred by AM's acknowledgements. BIP, by contrast, achieves a very high figure of 510 Mbit/s. This is because BIP fragments packets into 256-byte chunks and "streams" them across the connection, making this single-packet test, at high packet sizes, nearly resemble the following bandwidth test.

Bandwidth Performance

The bandwidth analysis measures the maximum sustainable data transfer rate a stream of messages of various sizes. The test measures the time required to send a stream of messages in rapid succession from one node to another. Bandwidth is then calculated by dividing the total payload size (payload size per message, multiplied by the number of messages) by the elapsed time to send the stream. The results of this analysis are shown in Figures 7 and 8.

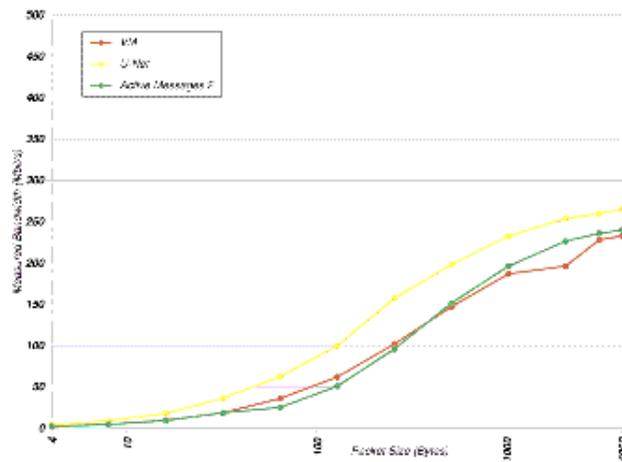


Figure 7: Measured Bandwidth, S-Bus

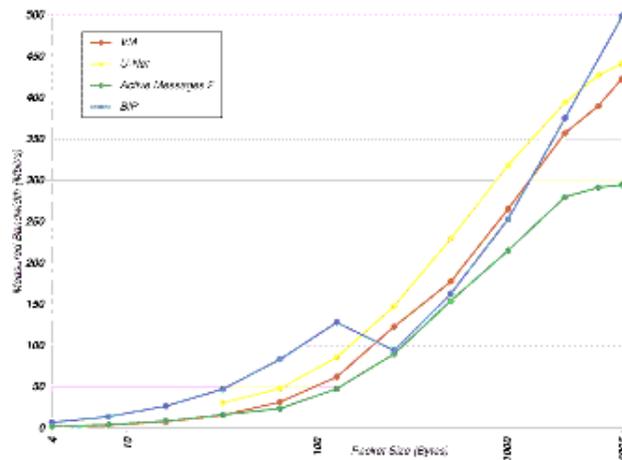


Figure 8: Measured Bandwidth, PCI

VIA scales smoothly to a bandwidth of 235 Mbit/s on S-Bus and 425 Mbit/s on PCI with packets approximately 4KBytes long.

On the S-Bus platform, all three ULNs achieve bandwidths within 10% of each other at a 4-Kbyte packet size: 235 Mbit/s for VIA, 240 Mbit/s for AM2, and 265 Mbit/s for U-Net. This suggests that each implementation is being limited by the bandwidth of the S-Bus.

On the PCI platform, results differ more. VIA’s maximum bandwidth of 425 Mbit/s is comparable to that of U-Net (440 Mbit/s), and the similar curves indicate that U-Net’s lower overhead is the primary reason for its slightly greater bandwidth. Active Messages 2 reaches only 295 Mbit/s with a 4-Kbyte packet size, again due to its requirement for acknowledgements. With 512-byte or smaller packets, however, its performance is almost identical to that of VIA, due to its lack of a need for a descriptor or other indirection with small messages.

BIP, by contrast, reaches a bandwidth of about 500 Mbit/s at a 4-Kbyte packet size, and its curve is significantly steeper than that of either VIA or U-Net. This high bandwidth is achieved because BIP fragments packets at or above 256 bytes. In essence, this streams 256-byte packets across the connection *without* requiring host involvement on each one, thus eliminating a significant source of overhead. However, at a packet size of exactly 256 bytes, BIP’s bandwidth dips momentarily as its fragmentation scheme becomes active without achieving much advantage.

LogP Metrics

To help understand the performance results presented above, we look at a simple cross section of the latency for sending a single packet. The time measures are derived from the LogP [12] conceptual model of the VIA architecture. Additionally, we apply the simple extensions to the LogP model as suggested by Culler *et al.* [13]. A summary of the parameters and our results for a 4-byte packet are given in Table 2. We determined the values of the four parameters using micro-benchmarks developed for VIA; we assume a virtual interface has already been established and connected. The shaded values are those we believe to be inherent to VIA, while the non-shaded values may be substantially improved with better software and/or hardware.

Name	Description	S-Bus	PCI
Latency	An upper bound on the time to transmit a message from source to destination.	17 us	18 us
overhead	The time period during which the processor is engaged in sending or receiving a message. This is extended into receive overhead (o_r) and send overhead (o_s). In our implementation, o_r and o_s are identical.	4 us	5 us
gap	The minimum time interval between consecutive message transmissions or consecutive message receptions at a processor.	11 us	16 us
Processors	The number of processors.	1	1

Table 2: LogP Parameters and measured results.

Latency is the time for the sending host to build descriptors and post a doorbell ($Build + Ring$), the sending NIC to process the doorbell and transfer the descriptor and data ($Bell + XDes + XData$), the data to move across the network ($Transmit + Flight\ Time$), and the receiving NIC to transfer the descriptor and data, and then update the descriptor's status ($XDes + XData + Updt$). Thus, in all,

$$L = Build + Ring + Bell + 2 \cdot XDes + 2 \cdot XData + Transmit + Flight\ Time + Updt$$

In addition, due to our implementation's need to poll to recognize doorbells, a factor of up to 3 us must be added to the above term: if a doorbell is posted just as the firmware polls it, the term is zero; if it is posted just after the firmware polls it, the factor is 3 us. Adding more VIs to our NIC would increase this term by 1.5 us per VI on a S-Bus-based platform, 2 us per VI on a PCI-based platform.

The latency we observed could be substantially improved by better hardware support for VIA and a more-highly-optimized firmware program. Hardware support for doorbells would improve $Bell$ and eliminate the variable polling factor, and decreasing the relatively high DMA-transfer overhead of our NIC would improve $XDes$, $XData$, and $Updt$. The $Build$ and $Ring$ components, however, cannot be improved above those advances offered by general improvements in processors and I/O bus technology.

Overhead is the time to build descriptors and post a doorbell to the NIC. Using the events defined in Figure 4,

$$O = Build + Ring$$

The overhead observed is clearly fundamental to VIA. The host must always build a descriptor and post a doorbell to the NIC for each transfer. No improvement in the software or hardware available to us would have decreased this parameter.

In VIA, o_r is not as critical to application performance as o_s , because the time that must be spent to receive a packet can be spent at a time chosen by the application -- no time is spent when the packet is actually received.

Gap is the time for the host to build a descriptor and post a doorbell ($Build + Ring$), the NIC to process the doorbell and transfer the descriptor and data ($Bell + XDes + XData$), the data to move across the network ($Transmit$), and the NIC to update the descriptor's status ($Updt$). Thus,

$$G = Build + Ring + Bell + XDes + XData + Transmit + Updt$$

Like latency, the gap here could be substantially improved by better hardware support for VIA and better firmware. All except the $Build$ and $Ring$ components would be improved by the same methods used to improve latency.

Evaluation

In this last section, we evaluate VIA for its potential as a lightweight networking infrastructure. From our realization on Myrinet hardware, we conclude that VIA is a viable interconnect architecture with performance comparable to established ULN implementations. It exhibits performance comparable to its slightly more mature predecessors and can be straightforward to implement. The following

sections offer a retrospective on lessons learned from our studies and investigate some extensions to specification that improve performance for certain messages.

Retrospectives

Simplicity/Complexity. VIA offers a relatively simple conceptual model to the application: messages are sent and received in a stream of packets, just like a traditional UDP/IP implementation. This provides a more widely understood interface than that of, say, Active Messages. However, the internal mechanisms of VIA can be rather complex when compared to existing research ULNs such as U-Net or AM 2, even on simple implementations and message types. VIA's scatter/gather-capable descriptors, when combined with reliability modes, RDMA transactions, and completion/error queues all make for a very complex system, and one in which the complexity is a hindrance to performance.

Standardization. A distinct advantage of VIA is its attempt to define an industry standard for user level networking. Although several fast communication protocols are being researched and developed, none are compatible with each other. Since VIA is the product of three industry leaders, it reflects, to some degree, the combined interest of commercial computing. If adopted, VIA would allow for cross compatibility between cluster vendors both at the application and the network layers.

Small Message Handling. One of VIA's biggest problems is its poor efficiency at handling small messages. With the endpoint residing in host memory, the architecture's descriptor processing model transfers the actual data payload only after two additional memory-to-NIC transfers (the doorbell and the descriptor). A descriptor itself may be tens of bytes long (45 bytes minimum in the suggested reference implementation) while many messages may only be a few bytes. This incurs a high overhead for short messages. In the Berkeley VIA implementation, we alleviate this overhead to some extent by transferring between the host and the NIC only those elements of the descriptor necessary to complete communication; still, small messages are quite slow compared to existing research ULNs.

Doorbell Resource Requirements. According to the specification, each doorbell pair for a particular VI is a resource mapped from the VI NIC into the process's virtual address space. This mapping is done on a unique page to ensure security and to prevent the use of system calls. This consumes a great deal of available address space, while most of the consumed space is unused. In our implementation, an additional problem arises since a memory-mapped page on the host corresponds to a full page of SRAM on the Myrinet NIC. This constitutes a significant demand on the NIC's limited memory space. To minimize this high resource demand, the specification suggests that multiple VIs for a single process have doorbells mapped to the same page.

Another issue concerns the process of monitoring doorbells. Using a simple polling loop, it takes approximately 3 us on the S-Bus NIC (4 us on the PCI NIC) to poll the two sets of doorbells available. This value increases linearly with the number of VIs active and could substantially increase the gap and latency times, reducing overall performance. While some of this overhead may be hidden during DMA transfers, a more efficient system would be to have a doorbell resource that automatically handles doorbell tokens. This suggests that some sort of specialized hardware support for doorbells is necessary for a VIA compliant NIC to perform reasonably well.

Virtualization Effects. The ability to virtualize network resources at a user level without operating system assistance is one of the benefits of VIA. However, this produces some inherent characteristics

that are problematic. The de-multiplexing necessary during receive operations requires that the interface momentarily buffer or block the incoming message to retrieve the destination receive descriptor. While some caching could be done, a cache of receive descriptors would not necessarily scale well with the number of VIs and would not ultimately improve performance for a stream of incoming messages with the same destination. The extensions discussed in the next section deal only with optimizations on the sending side; while the same optimizations might be implemented on the receiver, it is not clear that this would be worthwhile, as a large loss of generality might be required.

Scalability. A third issue with VIA centers around concerns of scalability in the face of a large number of nodes. Presently, a VI on a given node can connect with only one VI on a remote node. Thus, a message sent to M different nodes would require M separate VIs, each with their own set of separate resources and set-up/teardown overhead. This methodology does not scale well when the number of recipients becomes large and may place high demands on the source NIC as well as on the network itself. This defeats the original purpose of VIA: to minimize overhead and maintain high performance.

Spanning Multiple Interfaces. It is possible in many existing systems to gain additional bandwidth by adding multiple physical interfaces to a system; the VI Architecture, however, makes distribution of a single VI across multiple NICs difficult or impossible. Support for such a "distributed VI" would be a compelling addition.

Ambiguity. The VIA specification itself leaves many details missing. One of the principal ambiguities deals with the reliability modes of VIA: the reliable reception and delivery modes guarantee that a message will be delivered *exactly* once. Some studies suggest that this is not possible in a finite amount of time [14]. Further, VIA supports functions to block until a descriptor's status changes from "pending"; it is not clear how this would be implemented without invoking the operating system and incurring associated penalties.

VIA Extensions

In light of the lessons learned from our implementation of VIA, we suggest some possible extensions to improve the overall architecture. Table 3 indicates the results of some prototype extension implementations. As these modifications are primarily targeted at reducing overhead, we show figures for four-byte packets.

Extension	Description	4-Byte RTT/2, S-Bus	4-Byte RTT/2, PCI
None	--	25 us	26.5 us
Descriptorless messages	Messages transmitted directly from NIC with doorbell as signal; eliminates two DMA transfers	17 us	19 us
Merged descriptors	Descriptors merged with message; eliminates one DMA transfer	23 us	26 us

Table 3: VIA Extensions and Results

Short Messages. To improve the support for small messages, we propose the concept of a *descriptorless message*. Here, the doorbell itself provides all the necessary information to perform the network transaction, thus eliminating the need for a bulky descriptor. A special doorbell would be associated with a memory region specially designated for small messages; a token written to the doorbell would indicate the size of a particular data message, rather than the address of a descriptor. When the NIC receives one of these doorbells, it immediately begins transfer of the message. Completion of the transfer is marked by the NIC clearing the doorbell register. Multiple doorbells could be used to enable the host to enqueue several small messages without waiting for the NIC to complete transmission of any of them.

We implemented a prototype descriptorless message system and observed its performance over a range of message sizes. The data memory region was allocated from unused space in the NIC's memory and was filled using programmed I/O from the host. Table 3 illustrates the results. In general, we believe that this approach is likely to be very successful for small messages, as evidenced by the substantial savings in message latency, especially considering that message receipt proceeded in the standard fashion in our tests. (An equivalent approach for receipt of small messages is an open area of research.)

Merged Descriptors. An alternative to the descriptorless message approach, a merged descriptor is a combination of both a standard VIA descriptor and data. Merged descriptors could be located in any registered memory region and stored along with regular VIA descriptors; they contain both the descriptor itself and, immediately following, message data. An extra word in the doorbell token is used to identify the length of the descriptor/data combination. This eliminates one of the bus transactions required in the standard implementation. This approach differs from the immediate data field suggested in the specification in that the data can be of variable size and larger than 32 bits.

We have implemented a prototype merged descriptor system; possible performance gains using this method are shown in Table 3. Gains here are relatively small, suggesting that eliminating a single extra DMA from the critical path is not terribly advantageous.

Because of the significant modifications required to the NIC firmware to support merged descriptors and descriptorless messages, latencies here do not exactly correspond with those given earlier; all times are higher by approximately 2 us. The additional logic and storage required on the NIC slows down even the standard descriptor model. We believe that with a hardware implementation or additional optimization, these problems could be greatly reduced or eliminated entirely.

Conclusion

The focus of the Berkeley VIA project was to build a reference implementation of the Virtual Interface Architecture based on the recently released specification. Developed on the Sun Ultra 1 and Intel x86-based PCs using a Myrinet network, we were able to implement a VIA-capable network interface and a fundamental subset of the VIA specification. Traps to the operating system are completely eliminated during actual communication transactions, allowing significant performance gains over traditional network architectures. The results of the project demonstrate that VIA has good potential as a lightweight networking infrastructure for clusters. One-way packet latencies as low as 24 us have been achieved, and bandwidths up to 425 Mbit/s have been observed. Overall

performance is competitive with presently available fast communication architectures. From the experience gained during this project we have identified the principal strengths and weaknesses of VIA. We have proposed a small set of extensions to the existing specification to that may improve VIA while maintaining the fundamental concepts of the underlying architecture.

Future intentions are to continue work on the Berkeley VIA to develop it into a worthwhile networking option for the Berkeley Millennium Project. Plans include porting to different network hardware (Gigabit Ethernet, Tandem's ServerNet, and Fujitsu's Synfinity-1). We also intend to probe other areas of the architecture that might provide performance improvement. We intend to investigate the performance aspects of building other network protocols (TCP/IP, AM) on top of VIA; doing so will help to build the application base that may run with VIA and minimize compatibility issues when VIA is installed into a cluster.

Acknowledgements

This project would not have been possible without the invaluable assistance of Shahaf Gal, who helped extensively with the Windows NT build and the benchmarking tools. Thanks also go to members of the Berkeley NOW project including Alan Mainwaring, Rich Martin, and Steve Lumetta, as well as to Matt Welsh of the U-Net project, all of whom provided a great deal of experience and insight for this paper. Additional support was provided by Microsoft, especially Jim Gray and Joe Barrera, and by Intel's Technology 2000 grant program. As well, support was provided by the National Science Foundation Infrastructure Grant CDA 94-01156 and the Defense Advanced Projects Research Administration Grant F30602-95-C-0014.

References

- [1] **Virtual Interface Architecture Specification. Version 1.0**, Compaq, Intel and Microsoft Corporations, Dec 16, 1997, available at <http://www.viarch.org>
- [2] D.D. Clark, V. Jacobson, J. Romkey, H. Salwen, **An analysis of TCP processing overhead.** *IEEE Communications Magazine*, vol.27, (no.6), June 1989. pp.23-29.
- [3] M. Abbot and L. Peterson, **Increasing Network Throughput by Integrating Protocol Layers.**, *IEEE/ACM Transactions on Networking*. Vol. 1, (no.5), Oct. 1993, pp 600-610,
- [4] T. Braun, C. Diot. **Protocol implementation using integrated layer processing.** *Computer Communication Review*, vol.25, (no.4), (ACM SIGCOMM '95, Cambridge, MA, USA, 28 Aug.-1 Sept. 1995.) ACM, Oct. 1995. p.151-61.
- [5] T. von Eicken, D. E. Culler, S. C. Goldstein, K.E. Schauer. **Active messages: a Mechanism for Integrated Communication and Computation.** *Computer Architecture News*, vol.20, (no.2), (19th Annual International Symposium on Computer Architecture, Gold Coast, Qld., Australia, May 1992. pp.256-266
- [6] S. Pakin, V. Karamcheti, A. A. Chien. **Fast messages: efficient, portable communication for workstation clusters and MPPs.** *IEEE Concurrency*, vol.5, (no.2), April-June 1997. p.60-72.

- [7] T. von Eicken, Anindya Basu, Vineet Buch and Werner Vogels, **U-Net: A User-level Network Interface of parallel and Distributed Computing**, *Proc. of the 15th ACM Symposium of Operating Systems Principles*, vol. 29, (no.5), Dec 1995, pp. 40-53
- [8] L. Prylli and B. Tourancheau. **BIP: a New Protocol Designed for High Performance Networking on Myrinet.**, *Workshop PC-NOW, IPPS/SPDP98*, Orlando, USA, 1998.
- [9] C. Dubnicki, L. Iftode, E. W. Felten, Kai Li. **Software support for virtual memory-mapped communication**, *Proceedings of IPPS '96, the 10th International Parallel Processing Symposium, Proceedings of International Conference on Parallel Processing*. Honolulu, HI, USA, 15-19 April 1996. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, p.372-381.
- [10] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and Wen-King Su, **Myrinet: A Gigabit-per-Second Local Area Network**. *IEEE Micro*, vol. 15, (no. 1), Feb 1995, pp. 29-36
- [11] D. Dunning et al., **The Virtual Interface Architecture**, *IEEE Micro*, vol. 18, (no. 2), Mar/Apr 1998, pp. 66-75
- [12] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian and T. von Eicken, **LogP: Towards a Realistic Model of Parallel Computation**. *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1993. pp.1-12.
- [13] D. E. Culler, L. Tin Liu, R. P. Martin and C. Yoshikawa, **Assessing Fast Network Interfaces**, *IEEE Micro*, vol. 16, (no. 1), Feb 1996, pp. 35-43
- [14] L. Lamport, R. Shostak, M. Pease **The Byzantine generals problem.**, *ACM Transactions on Programming Languages and Systems*, vol. 4, (no. 3), July 1982, pp. 382-401.
- [15] R. Gusella, **A Measurement Study of Diskless Workstation Traffic on an Ethernet**, *IEEE Trans. Communications*, vol. 38, (no. 9), Sep. 1990, pp. 1557-1568
- [16] J. Kay and J. Pasquale, **The Importance of Non-Data Touching Processing Overheads in TCP/IP**, *Computer Communication Review*, vol. 23, (no. 4), Oct. 1993, pp. 259-268
- [17] B. N. Chun, A. M. Mainwaring, D. E. Culler. **Virtual network transport protocols for Myrinet**, *IEEE Micro*, vol. 18, (no. 1), Jan.-Feb. 1998., pp. 53-63
- [18] A. Mainwaring and D. Culler. **Active Message Applications Programming Interface and Communication Subsystem Organization**, University of California, Berkeley, Dec. 1995. Available at <http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps>.
- [19] D. Culler, K. Keeton, L. Krumbein, L.T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. **The Generic Active Message Interface Specification**, University of California, Berkeley, Feb. 1995. Available at <http://now.cs.berkeley.edu/Papers/Papers/gam-spec.ps>.
- [20] R.S. Madukkarumukumana, C. Pu and H.V. Shah, **Harnessing User-Level Networking**

Architectures for Distributed Object Computing over High-Speed Networks, Proc. of the 2nd USENIX Windows NT Symposium, Aug. 1998, pp. 127-135.