

# Linux/SimOS: A Complete System Simulation Environment for Evaluating High-Speed Communication Systems<sup>1</sup>

Chulho Won<sup>†</sup>, Ben Lee<sup>†</sup>, Chansu Yu<sup>\*</sup>, Sangman Moh<sup>‡</sup>, Kyoung Park<sup>‡</sup>, and Myung-Joon Kim<sup>‡</sup>

<sup>†</sup>Electrical and Computer Engineering Department  
Oregon State University  
{chulho, benl}@ece.orst.edu

<sup>\*</sup>Department of Electrical and Computer Engineering  
Cleveland State University  
c.yu91@csuohio.edu

<sup>‡</sup>Computer and Software Laboratory  
Electronics and Telecommunications Research Institute (ETRI)  
Daejon, Korea  
{simmoh, kyoung, joonkim}@etri.re.kr

## Abstract

This paper presents a performance study of UDP/IP and M-VIA using Linux/SimOS. Linux/SimOS is a Linux operating system port to a complete machine simulator SimOS. A complete machine simulator includes all the system components, such as CPU, memory, I/O devices, etc., and models them in sufficient detail to run an operating system. Therefore, a real program execution environment can be set up on the simulator to perform detailed system evaluation in a non-intrusive manner. The motivation for Linux/SimOS is to alleviate the limitations of SimOS (and its variants), which only support proprietary operating systems. Therefore, the availability of the popular Linux operating system for a complete machine simulator will make it an extremely effective and flexible open-source simulation environment for studying all aspects of computer system performance, especially evaluating communication protocols and network interfaces. The contributions made in this paper are two-fold: First, the major modifications that were necessary to run Linux on SimOS are described. These modifications are specific to SimOS I/O device models and thus any future operating system porting efforts to SimOS will experience similar challenges. Second, a detailed analysis of the UDP/IP protocol and M-VIA is performed to demonstrate the capabilities of Linux/SimOS. The simulation study shows that Linux/SimOS is capable of capturing all aspects communication performance, including the effects of the kernel, device driver, and network interface.

*Key words:* SimOS, complete system simulation, Linux, instruction set simulators, UDP/IP, M-VIA.

---

<sup>1</sup> A shorter version of this paper appears in *The 2002 International Conference on Parallel Processing (ICPP-02)*, August 18-21, Vancouver, BC, 2002.

# 1 Introduction

The growing demand for high-performance communication for system area networks (SANs) has led to significant research efforts towards low-latency communication protocols, such as Virtual Interface Architecture (VIA) [10] and InfiniBand Architecture (IBA) [11]. Before these protocols can become established, they need to be accurately evaluated to understand how they perform and to identify key bottlenecks. However, detailed performance analysis of network protocols is often difficult due to their complexity. This is because communication performance is dependent not only on processor speed but also on the communication protocol and its interaction with the kernel, device driver, and network interface. Therefore, these interactions must be properly captured to evaluate the protocols and to improve on them.

The evaluation of communication performance has traditionally been done using *instrumentation* [3], where data collection codes are inserted to a target program to measure the execution time. However, instrumentation has three major disadvantages. First, data collection is limited to the hardware and software components that are visible to the instrumentation code, potentially excluding detailed hardware information or operating system behavior. Second, instrumentation codes interfere with the dynamic system behavior. That is, event occurrences in a communication system are often time-dependent, and the intrusive nature of instrumentation can perturb the system being studied. Third, instrumentation cannot be used to evaluate new features or a system component that does not yet exist.

The alternative to instrumentation is to perform simulations [1, 4, 6, 13, 14]. At the core of these simulation tools is an *instruction set simulator* capable of tracing the cycle-level interactions between hardware and software. However, they are suitable for evaluating general application programs whose performance depends only on processor speed, not communication speed. That is, these simulators only simulate portions of the system hardware and thus are unable to capture the complete behavior of a communication system.

On the other hand, a *complete machine simulation* environment [3, 2] removes these deficiencies. A complete machine simulator includes all the system components, such as CPU, memory, I/O devices,

etc., and models them in sufficient detail to run an operating system. Therefore, a real program execution environment can be set up on the simulator to perform system evaluation in a non-intrusive manner. Another advantage of a complete system simulation is that system evaluations do not depend on the availability of the actual hardware. For example, a new network interface can be prototyped by replacing the existing model with the new model.

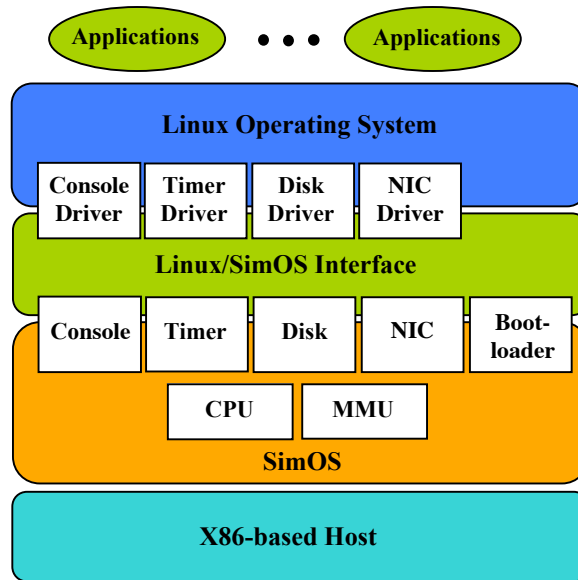
Based on the aforementioned discussion, this paper presents a performance analysis of network protocols UDP/IP and M-VIA using Linux/SimOS. *Linux/SimOS* is a Linux operating system port to a complete machine simulator SimOS [3]. The development of Linux/SimOS was motivated by the fact that the current version of SimOS only supports the proprietary SGI IRIX operating system. Therefore, the availability of popular Linux operating system for a complete machine simulator will make it an extremely effective and flexible open-source simulation environment for studying all aspects of computer system performance, especially evaluating communication protocols and network interfaces. The contributions made in this paper are two-fold: First, the major modifications that were necessary to run Linux on SimOS are described. These modifications are specific to SimOS I/O device models and thus any future operating system porting efforts to SimOS will experience similar challenges. Second, a detailed analysis of UDP/IP and M-VIA protocols is performed, which clearly show the advantage of using Linux/SimOS. Linux/SimOS is capable of capturing all aspects communication performance that includes the effects of the kernel, device driver, and network interface. These results help understand how the protocols work, identify key areas of interests, and suggest possible opportunities for improvement not only in the protocol stack but also in terms of hardware support.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 discusses the Linux/SimOS environment and the major modifications that were necessary to port Linux to SimOS, as well as changes required to run M-VIA, which is an implementation of the Virtual Interface Architecture for Linux, on Linux/SimOS. Section 4 presents the simulation study of UDP/IP and M-VIA. Section 5 concludes the paper and discusses some future work.

## 2 Related Work

There exist a number of simulation tools that contain detailed models of today's high-performance microprocessors [1, 2, 3, 4, 6, 13, 14]. SimpleScalar tool set includes a number of instruction-set simulators of varying accuracy/speed to allow the exploration of microarchitecture design space [6]. It was developed to evaluate the performance of general-purpose application programs that depend on the processor speed. RSIM is an execution-driven simulator developed for studying shared-memory multiprocessors (SMPs) and non-uniform memory architectures (NUMAs) [1]. RSIM was developed to evaluate parallel application programs whose performance depends on the processor speed as well as the interconnection network. However, neither simulators support system-level simulation because their focus is on the microarchitecture and/or interconnection network. Instead, system calls are supported through a proxy mechanism. Moreover, they do not model system components, such as I/O devices and interrupt mechanism that are needed to run the system software, such as the operating system kernel and hardware drivers. Therefore, these simulators are not appropriate for studying communication performance.

SimOS was developed to facilitate computer architecture research and experimental operating system development [3]. It is the most complete simulator for studying computer system performance. There are several modifications being made to SimOS. SimOS-PPC is being developed at IBM Austin Research Laboratory, which models a variety of PowerPC-based systems and microarchitectures [16]. There is also a SimOS interface to SimpleScalar/PowerPC being developed at UT Austin [17]. However, these systems only support AIX as the target operating system. Therefore, it is difficult to perform detailed evaluations without knowing the internals of the kernel. Virtutech's SimICS [2] was developed with the same purpose in mind as SimOS and supports a number of commercial as well as Linux operating systems. The major advantage of SimICS over SimOS is improved simulation speed using highly optimized codes for fast event handling and a simple processor pipeline. However, SimICS is proprietary and thus the internal details of the simulator are not available to the public. This makes it difficult to add or modify new hardware features. The motivation for Linux/SimOS is to alleviate these restrictions by



**Figure 1. The Structure of Linux/SimOS.**

developing an effective simulation environment for studying all aspects of computer system performance using SimOS with the flexibility and availability of the Linux operating system.

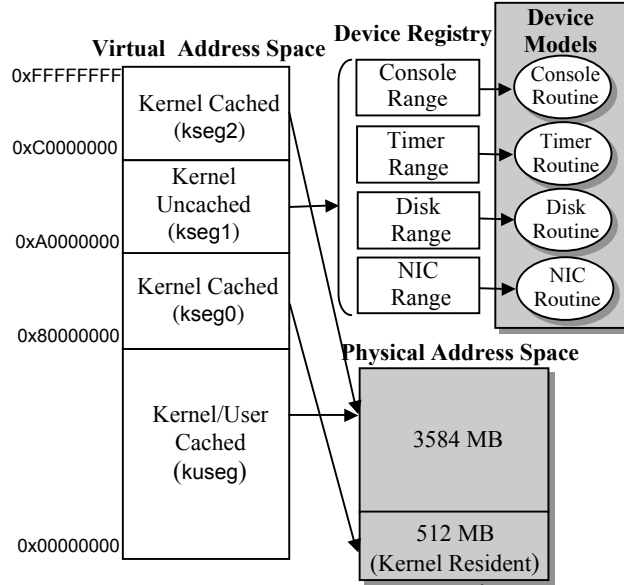
### 3 Overview of Linux/SimOS

Figure 1 shows the structure of Linux/SimOS. An x86-based Linux machine serves as the host for running the simulation environment. SimOS runs as a target machine on the host, which consists of simulated models of CPU, memory, timer, and various I/O devices (such as Disk, Console, and Ethernet NIC). On top of the target machine, Linux kernel version 2.3 for MIPS runs as the target operating system.

#### 3.1 SimOS Machine Simulator

This subsection briefly describes the functionality of SimOS, and the memory and I/O device address mapping. For a detail description of SimOS, please refer to [3].

SimOS supports two execution-driven, cycle-accurate CPU models: Mipsy and MSX. Mipsy models a simple pipeline similar to MIPS R4000, while MSX models a superscalar, dynamically scheduled pipeline similar to MIPS R10000. The CPU models support the execution of the MIPS instruction



**Figure 2. Address mapping mechanism in SimOS**

set [12]. SimOS also models a memory management unit (MMU), including the related exceptions. Therefore, the virtual memory translation occurs as in a real machine. SimOS also models the behavior of I/O devices by performing DMA operations to/from the memory and interrupting the CPU when I/O requests complete. It also supports the simulation of a multiprocessor system with a bus-based cache-coherent memory system or a Cache-Coherent Non-uniform Memory Architecture (CC-NUMA) system.

Figure 2 represents the SimOS memory and I/O device address mapping. The virtual address space is subdivided into four segments. Segments kseg0 through kseg2 can only be accessed in the kernel mode, while segment kuseg can be accessed either in user or kernel mode. The kernel executable code is contained in kseg0 and mapped directly to the lower 512 Mbytes of the physical memory. The segments kuseg and kseg2, which contain user process and per process kernel data structures, respectively, are mapped to the remaining address space in the physical memory. Therefore, communication between CPU and main memory involves simply reading and writing to the allocated memory. On the other hand, I/O device addresses are mapped to the uncached kseg1 segment, and a hash table called the *device registry* controls its access. The function of the device registry is to translate an I/O device register access to the appropriate I/O device simulation routine. Therefore, each I/O device has to first register its

**Table 1. I/O device address mapping.**

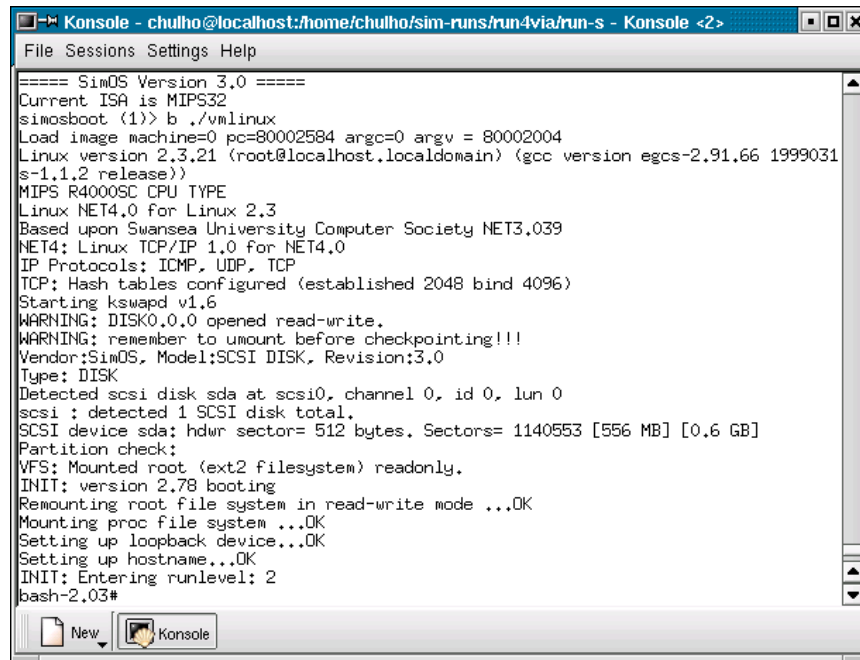
Device	Start address	Size in bytes
Timer	0xA0E00000	4
Console	0xA0E01000	8
Ethernet NIC	0xA0E02000	2852
Disk	0xA0E10000	542208

device registers with the device registry, which maps an appropriate device simulator routine at a location in the I/O address space. This is shown in Table 1. In response to device driver requests, I/O device models provide I/O services and interrupt the CPU as appropriate.

SimOS provides several I/O device models, which includes console, SCSI disk, Ethernet NIC, and a timer. These devices provide the interface between the simulator and the real world. The console model allows a user to read messages from and type in commands into the simulated machine's console. The SimOS NIC model enables a simulated machine to communicate with other simulated machines or real machines through the Ethernet. By allocating an IP address for the simulated machine, it can act as an Internet node, such as a Web browser or a Web server. SimOS uses the host machine's file system to provide the functionality of a hard disk, maintaining the disk's contents in a file on the host machine. Reads and writes to the simulated disk become reads and writes to this file, and DMA transfers require simply copying data from the file into the portion of the simulator's address space representing the target machine's main memory.

### **3.2 Linux/SimOS Interface**

In this subsection, the major modifications that were necessary to port Linux to SimOS are discussed, i.e., *Linux/SimOS interface*. Most of the major modifications were done on the I/O device drivers for Linux. Therefore, the description will focus on the interfacing requirements between Linux hardware drivers and SimOS I/O device modules.

The image shows a terminal window titled "Konsole - chulho@localhost:/home/chulho/sim-runs/run4via/run-s - Konsole <2>". The terminal output displays the following text:

```
==== SimOS Version 3.0 ====
Current ISA is MIPS32
simosboot (1)> b ./vmlinux
Load image machine=0 pc=80002584 argc=0 argv = 80002004
Linux version 2.3.21 (root@localhost.localdomain) (gcc version egcs-2.91.66 1999031
s-1.1.2 release)
MIPS R4000SC CPU TYPE
Linux NET4.0 for Linux 2.3
Based upon Swansea University Computer Society NET3.039
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
TCP: Hash tables configured (established 2048 bind 4096)
Starting kswapd v1.6
WARNING: DISK0.0.0 opened read-write.
WARNING: remember to umount before checkpointing!!!
Vendor:SimOS, Model:SCSI DISK, Revision:3.0
Type: DISK
Detected scsi disk sda at scsi0, channel 0, id 0, lun 0
scsi : detected 1 SCSI disk total.
SCSI device sda: hdwr sector= 512 bytes. Sectors= 1140553 [556 MB] [0.6 GB]
Partition check:
VFS: Mounted root (ext2 filesystem) readonly.
INIT: version 2.78 booting
Remounting root file system in read-write mode ...OK
Mounting proc file system ...OK
Setting up loopback device...OK
Setting up hostname...OK
INIT: Entering runlevel: 2
bash-2.03#
```

**Figure 3. Linux/SimOS console output with Linux boot message.**

### 3.2.1 Timer and Console

SimOS implements a simple real-time *clock* that indicates the current time in seconds past since January 1, 1970. The real-time clock keeps the time value in a 32-bit register located at address 0xA0E00000 (see Table 1). A user program reads the current time using the `gettimeofday()` system call. The Linux timer driver was modified to reflect the simplicity of the SimOS timer model. The SimOS real-time clock has a single register, while a timer chip in a real system has tens of registers that are accessed by the driver. Also, the Linux timer driver periodically adjusts the real-time clock to prevent it from drifting due to temperature or system power fluctuation. Since these problems are not present in a simulation environment, these features were removed to simplify debugging.

Console is used as a primary interface between the simulated machine and the external world. Linux commands are entered through the console, and the command execution results are printed on the console. A sample console output with Linux boot message is shown in Figure 3.

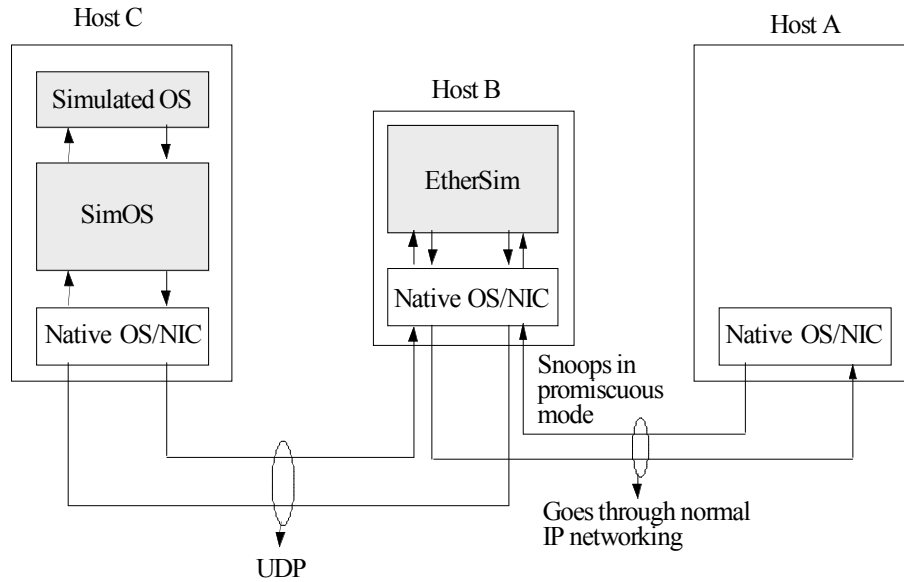
### 3.2.2 SCSI Disk

The SimOS *disk model* simulates a SCSI disk, which has the combined functionality of a SCSI adapter, a DMA, a disk controller, and a disk unit. Therefore, the registers in the SimOS disk model represent a combination of SCSI adapter registers, DMA descriptors, and disk status and control registers. This is different from a real SCSI disk, which implements them separately, and thus how the Linux disk driver views the disk. In particular, the problem arises when application programs make disk requests. These requests are made to the SCSI adapter with disk unit numbers, which are then translated by the disk driver to appropriate disk register addresses. But, the SimOS disk model performs the translation internally and thus the Linux disk driver is incompatible with the SimOS disk model. Therefore, the SimOS disk model had to be completely rewritten to reflect how the Linux disk driver communicates with the SCSI adapter and the disk unit.

### 3.2.3 Kernel Bootloader

When the kernel and the device drivers are prepared and compiled, a kernel executable is generated in ELF binary format [15]. It is then responsibility of the SimOS *bootloader* to load the kernel executable into the main memory of the simulated machine.

When the bootloader starts, it reads and looks for headers in the executable file. An ELF executable contains three different type headers: a file name header, program headers, and section headers. Each program header is associated a program segment, which holds a portion of the kernel code. Each program segment has a number of sections, and a section header defines how these sections are loaded into memory. Therefore, the bootloader has to use both program and section headers to properly load the program segment. Unfortunately, the bootloader that came with the SimOS distribution was incomplete and thus did not properly handle the ELF format. That is, it did not use both program and section headers to load the program. Therefore, the bootloader was modified to correct this problem.



**Figure 4. Communication between a simulated host and a real host using EtherSim.**

### 3.2.4 Ethernet NIC

The SimOS *Ethernet NIC model* supports connectivity to simulated hosts as well as to real hosts. The Ethernet NIC model is controlled by a set of registers mapped into the memory region starting at 0xA0E02000 (see Table 1). The data transfer between the simulated main memory and NIC occurs via DMA operations using descriptors pointing to DMA buffers. Typically, the Linux NIC driver allocates DMA buffers in the uncached `kseg1` segment. Since the device registry controls this memory region in SimOS, two modifications were necessary to differentiate between I/O device accesses and uncached memory accesses. First, the Linux Ethernet driver was changed to allocate DMA buffers using the device registry. Second, the device registry was modified to handle the allocated DMA buffer space as an uncached memory space.

Network simulation in SimOS can be performed using a separate simulator called *EtherSim* [3]. The main function of EtherSim is to forward the received packets to the destination host. Although EtherSim is not directly related to the Linux/SimOS interface, its functionality and the modifications that were made to facilitate network simulation with Linux/SimOS are briefly discussed.

EtherSim basically takes care of the activities of sending simulated Ethernet frames and receiving

IP packets on behalf of SimOS (i.e., a simulated host). EtherSim can be configured to have any number of real and simulated hosts. A simulated host communicating with another host via EtherSim is shown in Figure 4. EtherSim maintains the address information of the simulated host(s), which includes the IP and Ethernet addresses as well the socket address of the simulated NIC. A simulated host sends a simulated Ethernet frame to EtherSim using UDP. EtherSim then extracts the IP packet from the simulated Ethernet frame and forwards it to the destination host. In the case of a receive, EtherSim captures IP packets destined for one of the simulated hosts by running its host's Ethernet interface in promiscuous mode. It then forms a UDP packet from the captured packet and forwards it to the simulate host.

Some modifications were necessary to run EtherSim on a host running Linux operating system. The modifications made were mainly to improve portability. The original EtherSim that comes with the SimOS source distribution was written for Sun Solaris operating system, which could not be ported directly to Linux. Therefore, several of the Solaris operating system specific network library calls were replaced with libpcap [9] and libnet [8], which are standard libraries related to network packet capturing. As a result, the modified version of EtherSim can run on any Linux host, even on the same host running Linux/SimOS.

## **4 Simulation Study of UDP/IP and M-VIA**

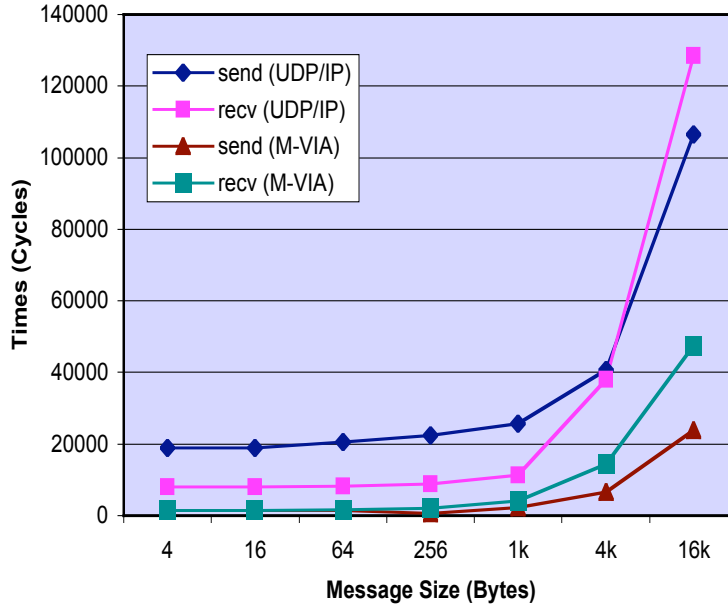
This section presents the performance measurements of UDP/IP and M-VIA [12] to demonstrate the capabilities of Linux/SimOS. The measurements were performed in a simulation set-up, where Linux/SimOS is used to model two host machines connected through a network. To evaluate the performance of these two protocols, test programs were run on Linux/SimOS that accepts command-line options specifying send/receive, a message size, and address. UDP/IP performance was measured by having a client program send/receive a message to/from a server program. On the other hand, `vpingpong` was used to evaluating M-VIA. `vpingpong` employs a number of library functions provided by VIP Provider Library to initiate message transfer over M-VIA network protocol. The program has two modes of op-

eration, send and receive which are selectable with a command-line option. When `vpingpong` starts, a given number of messages are exchanged between a sender and a receiver. The `vpingpong` program is one of the test programs included in the M-VIA 1.2b2 source code distribution [12].

## 4.1 Simulation Environment

The CPU model employed was Mipsy with 32 Kbyte L1 instruction and data caches with 1 cycle hit latency, and 1 Mbyte L2 cache with 10 cycle hit latency. The main memory was configured to have 32 Mbyte with hit latency of 100 cycles, and DMA on the Ethernet NIC model was set to have a transfer rate of 240 Mbytes/sec. The results were obtained using SimOS's data collection mechanism, which uses a set of annotation routines written in Tcl [18]. These annotations are attached to specific events of interest, and when an event occurs the associated Tcl code is executed. Annotation codes have access to the entire state of the simulated system, and more importantly, data collection is performed in a non-intrusive manner.

The UDP/IP performance was evaluated by directly sending messages through the legacy protocol stack in Linux/SimOS. On the other hand, M-VIA consists of three components: VI provider library (`vipl`) is a collection of library calls to obtain VI services; M-VIA kernel module (`vipk_core`) contains a set of modularized kernel functions implemented in user-level; and M-VIA device drivers (`vipk_dev`) provide an interface to NIC. In order to run M-VIA on Linux/SimOS, some modifications were necessary. First, because M-VIA was released only for x86-based Linux hosts, some of the source codes had to be modified to run it on Linux/SimOS. In particular, the code for fast traps (`vipk_core/vipk_ftrap.S`) had to be rewritten because the MIPS system supports a different system call convention than x86-based systems. Second, the driver for M-VIA had to be modified (similar to the discussion in Subsection 3.2) to work with SimOS Ethernet NIC.



**Figure 5. Message Send/Receive Latencies.**

## 4.2 Overall Performance

The performance study focused on the latency (in cycles) to perform send/receive. These simulations were run with a fixed MTU (Maximum Transmission Unit) size of 1,500 bytes with varying message sizes. The total cycle times required to perform send/receive as a function of message size are shown in Figure 5. The send results are based on the number of clock cycles required to perform the socket call `sendto()` for UDP/IP and `VipPostSend()` for M-VIA. The receive results are based on the time between the arrival of a message and either when the socket call `recvfrom()` for UDP/IP or `VipPostRecv()` for M-VIA returns. These results represent only the latency measurement of major operations directly related to sending and receiving messages and do not include the time needed to set up socket communication for UDP/IP and memory region registration for M-VIA. These results also do not include the effects of MAC and physical layer operations.

The results in Figure 5 clearly show the advantage of using low-latency, user-level messaging, especially for small messages. For message sizes less than the MTU size, the improvement factors for M-VIA send and receive latencies over UDP/IP are 11~14.1 and 2.6~5.6, respectively. For message sizes

greater than the MTU size, the improvement factors for M-VIA send and receive latencies over UDP/IP are 4.2~6.3 and 2.7~2.8, respectively.

### 4.3 Layer-Level Performance

The latencies for UDP/IP and M-VIA send/receive were then divided based on the various layers available for each protocol. This allows us to observe how much time is spent at each layer of the protocol and how each layer contributes to the final result. The latencies for UDP/IP were broken into layers associated with APPL, UDP, IP, DEV, and DMA. APPL includes the time required to initiate `sendto()` or `recvfrom()` and perform socket operations. UDP and IP are times for executing UDP and IP protocols, respectively. DEV represents the device driver and includes all the operations between IP and host-side DMA, including DMA interrupt handling. Finally, DMA represents the time to DMA data between host memory and NIC buffers.

Similarly, the latencies for M-VIA were broken into layers associated with APPL, TRANS, DEV, and DMA. APPL represents the time required to initiate VI provider library functions, `VipPostSend()` and `VipPostRecv()`. This involves creating a descriptor in the registered memory and then adding the descriptor to the send/receive queue. The transport layer then performs virtual-to-physical/physical-to-virtual address translation and fragmentation/de-fragmentation. Therefore, TRANS represents the time spent on the transport layer, but also includes part of the device driver, mainly DMA setup. This is due to the fact that M-VIA implementation does not separate the two layers for optimization purposes. Thus, DEV includes only the DMA interrupt handling time. Again, DMA represents the time to DMA data between host memory and NIC buffers.

The results of send and receive latencies are summarized in Figures 6 and 7, respectively, where each message size has a pair of bar graphs for M-VIA (left) and UDP/IP (right). The results are also presented in a tabular form in Table 3. The maximum message size in Table 3 is 32 Kbytes due to the fact that M-VIA's data buffer size was limited to 32 Kbytes. Also, 32-Kbyte results were not included in Fig-

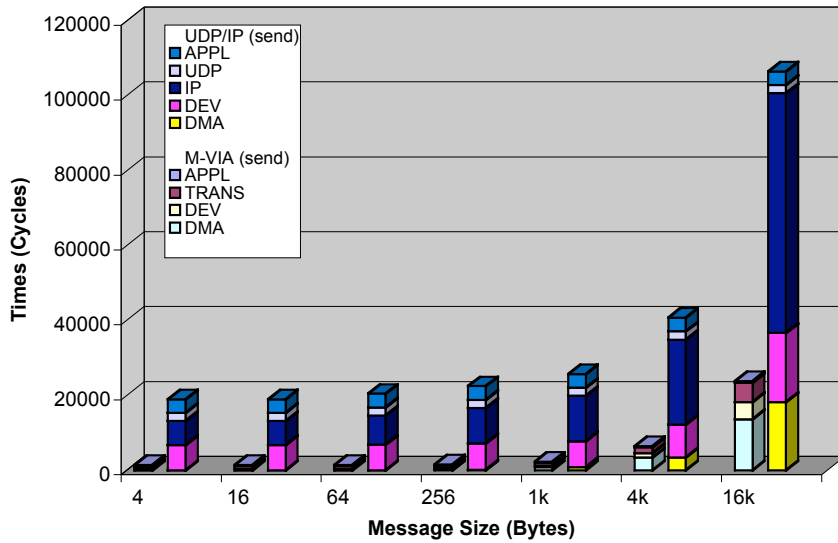


Figure 6. Send Latency

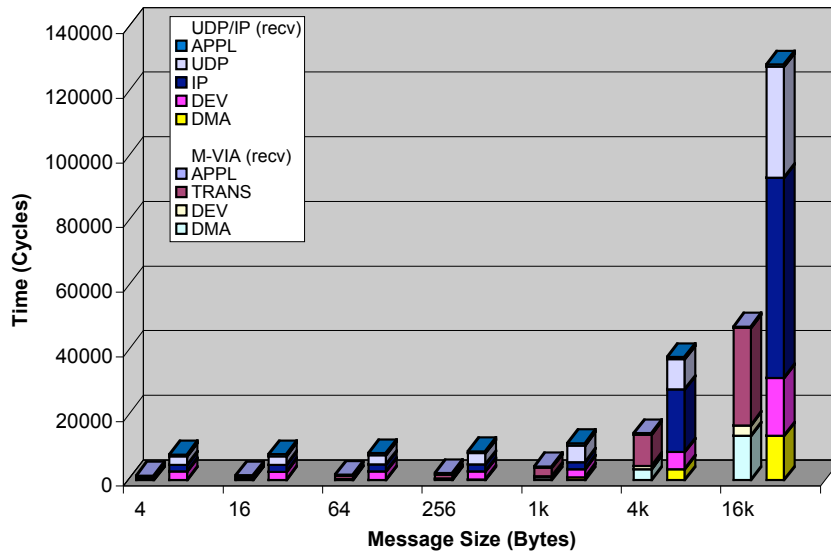


Figure 7. Receive Latency

ures 6 and 7 since they would overshadow the other results.

For UDP/IP send shown in Figure 6, APPL and UDP remain relatively constant. However, IP and DEV dominate as the message size grows. In particular, for 16-Kbyte and 32-Kbyte (see Table 3) messages, IP layer increases significantly as a function of messages size. This is because IP handles both packet fragmentation and data copying from user space to socket buffer. Therefore, IP portion increases

**Table 3. Message send/receive latency vs. message size.**

Protocol	Layers	Message Size (bytes)							
		4	16	64	256	1k	4k	16k	32k
M-VIA Send	APPL	347	347	347	347	347	347	347	394
	TRANS	622	622	622	622	696	1495	4996	8290
	DEV	437	437	437	437	437	1187	4570	6678
	DMA	3	12	49	213	853	3413	13653	27306
	<b>Total</b>	<b>1409</b>	<b>1418</b>	<b>1455</b>	<b>1619</b>	<b>2333</b>	<b>6442</b>	<b>23566</b>	<b>42668</b>
M-VIA Receive	APPL	349	349	349	359	359	359	369	383
	TRANS	690	714	884	1223	2648	9506	29327	59569
	DEV	350	350	350	350	350	875	3465	8050
	DMA	3	12	49	213	853	3413	13653	27306
	<b>Total</b>	<b>1392</b>	<b>1425</b>	<b>1632</b>	<b>2145</b>	<b>4210</b>	<b>14153</b>	<b>46814</b>	<b>95308</b>
UDP/IP Send	APPL	3648	3648	3648	3648	3648	3658	3658	3698
	UDP	2165	2165	2165	2165	2165	2165	2175	2205
	IP	6402	6448	7812	9404	12077	22774	64014	116095
	DEV	6694	6684	6785	7031	6928	8754	18343	29760
	DMA	3	12	49	213	853	3413	18243	27306
	<b>Total</b>	<b>18912</b>	<b>18957</b>	<b>20459</b>	<b>22461</b>	<b>25671</b>	<b>40764</b>	<b>106433</b>	<b>179064</b>
UDP/IP Receive	APPL	566	566	576	576	616	656	716	716
	UDP	2680	2718	2954	3533	5218	9393	34297	80940
	IP	2045	2163	2163	2183	2143	19370	61921	122273
	DEV	2554	2374	2661	2294	2274	5143	17840	31790
	DMA	3	12	49	213	853	3413	13653	27306
	<b>Total</b>	<b>7848</b>	<b>7833</b>	<b>8403</b>	<b>8799</b>	<b>11104</b>	<b>37975</b>	<b>128427</b>	<b>263025</b>

substantially for messages greater than the MTU size. In addition, DMA also takes a significant portion of the latency for message size over 4 Kbytes. For M-VIA send, latencies are relatively evenly spread among APPL, TRANS, and DEV for message size up to 1 Kbyte. However, as message size increases beyond 1 Kbytes, DMA takes up most of the latency and increases rapidly. TRANS and DEV also increase significantly for message sizes larger than 1 Kbytes due to fragmentation and interrupt handling, respectively.

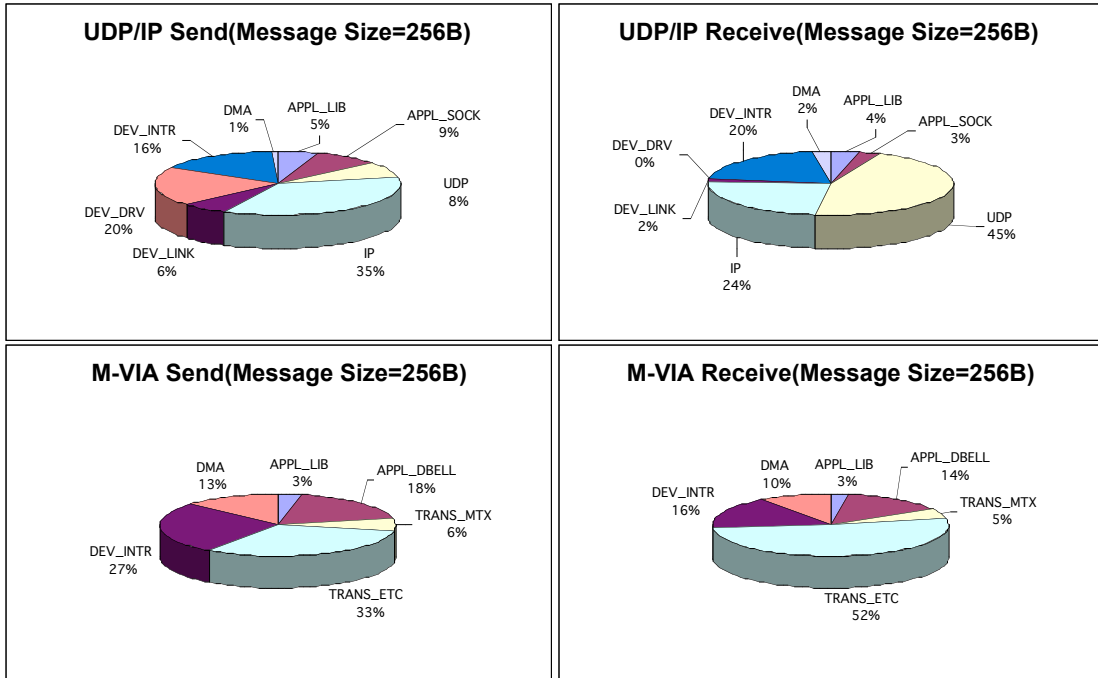
Figure 7 shows the receive latencies. When messages are larger than the MTU size, all the UDP/IP layers, except APPL, increase rapidly. Among them, UDP and IP increase are the most noticeable. Again, IP increase is caused by de-fragmentation, but UDP increase is due to copying data from socket buffer to user space. This is because in Linux, user data copying in UPD/IP occurs in two different layers for send and receive. In the case of a send, the IP layer handles both packet fragmentation and data copying from user space to socket buffer. On the other hand, for receive, packet de-fragmentation occurs at the IP layer while data copying from socket buffer to user space occurs at the UDP layer [26]. For M-

VIA, TRANS has the most noticeable increase due to extra data copying required from DMA buffer to user space. In addition, for messages larger than the MTU size, de-fragmentation contributes significantly to the TRANS layer.

For DEV and DMA layers, both UDP/IP and M-VIA protocols show similar results. This is because M-VIA uses a same type of device driver to communicate with the Ethernet NIC model. The primary function of the DEV layer is to set up NIC's DMA and receive interrupts from NIC. As can be seen, the latency of DEV remains relatively constant for message size up to 1 K bytes, but increases significantly when the messages are larger than the MTU size. DMA also varies linearly with the message size. This is consistent since DMA initiation and interrupt handling are already reflected in the DEV layer; therefore, DMA transfer time is dependent only on the message size.

#### **4.4 Function-Level Performance**

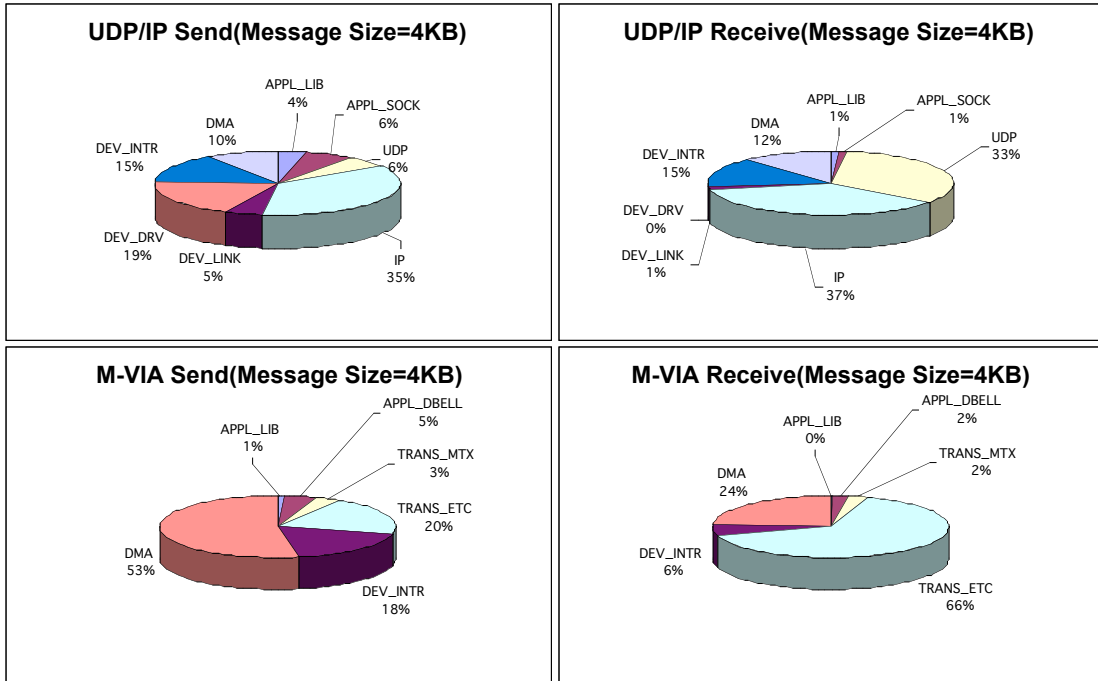
The pie charts shown in Figures 8 (send) and 9 (receive) give a more detailed picture about what contributes to the amount of time spent on each layer. For UDP/IP, APPL layer was further subdivided into APPL\_LIB and APPL SOCK. APPL\_LIB represents the latency between a system call (i.e., `sendto` and `recvfrom`) and the start of socket operation (i.e., `sock_sendmsg` and `sock_recvmsg`). APPL SOCK includes the time for socket layer operations. DEV layer was also subdivided into DEV\_LINK, DEV\_INTR, and DEV\_DRV. DEV\_LINK is for a device-independent interface between IP and network devices, which forwards packets to a specific device depending on the packet type. DEV\_DRV includes the time for initializing the host-side DMA. DEV\_INTR includes time for interrupt handling for the completion of a DMA transfer. For UDP/IP receive, DEV\_DRV is always zero because DMA is initiated by NIC when packets arrive.



**Figure 8. Latency breakdown for 256-byte message.**

M-VIA layers were also subdivided to focus on the effects of doorbell mechanism, interrupt handling, and memory translation table lookup operations. APPL was further subdivided into APPL\_LIB and APPL\_DBELL. APPL\_LIB represents the latency between a VIA library call (i.e., VipPostSend or VipPostRecv) and start of the doorbell operation. APPL\_DBELL is the time to execute a doorbell operation. The doorbell is a mechanism to start the NIC for servicing VIA library calls. These library calls eventually lead to the execution of the device driver. However, the library calls cannot directly call the device driver. Instead, system call `ioctl()` is used to start the device driver. The indirect invocation of the device driver is needed because the NIC is assumed to be a traditional NIC, rather than a VIA-aware NIC. Therefore, the latency for indirect invocation of the device driver is included in APPL\_DBELL. TRANS was also subdivided into TRANS\_MTX and TRANS\_ETC. TRANS\_MTX is the overhead for memory translation table lookups, and TRANS\_ETC represents the rest of the transport layer operations.

As can be seen from the figures, APPL\_DBELL for M-VIA represents a significant portion of the overall send and receive latency. For example, an `ioctl()` system call for a 256-byte message send requires



**Figure 9. Latency breakdown for 4-Kbyte message**

around 300 cycles and represents 86% of the APPL layer. This suggests that a hardware doorbell mechanism will be very effective for small messages. Using VIA-aware NIC that uses a control register for doorbell support would virtually eliminate the APPL\_DBELL overhead. For 256-byte message, DEV\_INTR constitutes 27% and 16% of send and receive latencies, respectively. As message size increases to 4 Kbytes, DEV\_INTR still represents 18% and 6% of send and receive latencies, respectively. DEV\_INTR increases slightly as message size increases because every fragmented packet generates an interrupt for both send and receive. One solution for reducing DEV\_INTR is to provide interrupt coalescing feature on the NIC. Using this solution, DEV\_INTR can be kept constant regardless of the message size. Finally, TRANS\_MTX represents only a small portion of the transport layer for both 256 bytes and 4 Kbytes messages, thus memory translation table lookup has minimal effect on latency. However, TRANS\_ETC for M-VIA receive dominates for 4 Kbyte message size, indicating VIA-aware NIC capable of DMAing data directly from NIC buffer to user space would significantly reduce receive latencies.

## 5. Conclusion and Future Work

This paper presented a detailed performance analysis of UDP/IP and M-VIA using Linux/SimOS. Our study confirms that Linux/SimOS is an excellent tool for studying communication performance, and is able to provide details of the various layers of the communication protocols, in particular the effects of the kernel, device driver, and NIC. Moreover, since Linux/SimOS open-source, it is a powerful and flexible simulation environment for studying all aspects of computer system performance.

There are numerous possible uses for Linux/SimOS. For example, one can study the performance of Linux/SimOS acting as a server. This can be done by running server applications (e.g., web server) on Linux/SimOS connected to the rest of the network via EtherSim. Another possibility is to prototype a new network interface. One such example is the Host Channel Adapter (HCA) for InfiniBand [11], which is in part based on Virtual Interface Architecture. Since the primary motivation for InfiniBand technology is to remove I/O processing from the host CPU, a considerable amount of the processing requirement must be supported by the HCA. These include support for message queuing, memory translation and protection, remote DMA (RDMA), and switch fabric protocol processing. The major advantage of Linux/SimOS over hardware/emulation-based methods used in [19, 24] is that both hardware and software optimization can be performed. This prototyping can provide some insight on how the next generation of HCA should be designed for InfiniBand Architecture.

## Acknowledgement

This research was supported in part by Electronics and Telecommunications Research Institute (ETRI) and Tektronix Inc.

## 6. References

- [1] V. S. Pai *et al.*, “RSIM Reference Manual, Version 1.0,” ECE TR 9705, Rice Univ., 1997.
- [2] P. S. Magnusson *et al.*, “Simics: A Full System Simulation Platform,” *IEEE Computer*, Vol. 35,

No. 2, February 2002, pp. 50-58.

- [3] S. Harrod, "Using Complete Machine Simulation to Understand Computer System Behavior," Ph.D. Thesis, Stanford University, February 1998.
- [4] D. K. Panda *et al.*, "Simulation of Modern Parallel Systems: A CSIM-Based Approach," *Proc. of the 1997 Winter Simulation Conference*, 1997.
- [5] N. Leavitt, "Linux: At a Turning Point?," *IEEE Computer*, Vol. 34, No. 6, 1991.
- [6] D. Burger *et al.*, "The SimpleScalar Tool Set, Version 2.0," *U. Wisc. CS Dept. TR#1342*, June 1997.
- [7] M. Beck, *et al.*, *LINUX Kernel Internals, 2nd Edition*, Addison-Wesley, 1997.
- [8] Libnet, Packet Assembly System. Available at <http://www.packetfactory.net/libnet>.
- [9] Tcpdump/libpcap. Available at <http://www.tcpdump.org>.
- [10] D. Dunning, *et al.*, "The Virtual Interface Architecture," *IEEE Micro*, March/April, 1998.
- [11] Infiniband™ Architecture Specification Volume 1, Release 1.0.a. Available <http://www.infinibandta.org>.
- [12] LBNL PC UER, "M-VIA: Virtual Interface Architecture for Linux," see <http://www.extremelinux.org/activities/usenix99/docs/mvia>.
- [13] WARTS, Wisconsin Architectural Research Tool Set. <http://www.cs.wisc.edu/~larus/warts.html>.
- [14] SIMCA, the Simulator for the Superthreaded Architecture. Linux/SimOS acting as a server. <http://www.mount.ee.umn.edu/~lilja/SIMCA/index.html>.
- [15] D. Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers, Inc., 1999.
- [16] SimOS-PPC, see <http://ww.cs.utexas/users/cart/simOS>.
- [17] SimpleScalar Version 4.0 Tutorial, *34<sup>th</sup> Annual International Symposium on Microarchitecture*, Austin, Texas, December, 2001.
- [18] M. Rosenblum *et al.*, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 1, January 1997, pp. 78-103.
- [19] J. Wu *et al.* "Design of An InfiniBand Emulation over Myrinet: Challenges, Implementation, and Performance Evaluation," Technical Report OUS-CISRC-2/01\_TR-03, Dept. of Computer and Information Science, Ohio State University, 2001.
- [20] M. Banikazemi, B. Abali, L. Herger, and D. K. Panda, "Design Alternatives for Virtual Interface Architecture (VIA) and an Implementation on IBM Netfinity NT Cluster," *Journal of Parallel and Distributed Computing*, Special Issue on Clusters, 2002.
- [21] M. Banikaze, B. Abali, and D. K. Panda, "Comparison and Evaluation of Design Choices for Implementing the Virtual Interface Architecture (VIA)," *Fourth Int'l Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '00)*, January

2000.

- [22] S. Nagar *et al.*, "Issues in designing and Implementing A Scalable Virtual Interface Architecture," *2000 International Conference on Parallel Processing*, 2000.
- [23] A. Begel, "An Analysis of VI Architecture Primitives in Support of Parallel Distributed Communication," to appear in *Concurrency and Computation: Practice and Experience*, 2002.
- [24] P. Buonadonna, A. Geweke, and D.E. Culler, "An Implementation and Analysis of the Virtual Interface Architecture," *Proceedings of SC '98*, Orlando, FL, Nov. 7-13, 1998.
- [25] A. Rubini, *Linux Device Driver*, 1st Ed., O'Reilly, 1998.
- [26] J. Crowcroft, *et al.*, *TCP/IP and Linux Protocol Implementation: System Code for Linux Internet*, Wiley, 2001, pp. 482-483.
- [27] P. A. Farrell and H. Ong, "VIA Communication Performance over a Gigabit Ethernet Network," *19th IEEE International Performance, Computing, Communications Conference (IPCCC 2000)*, 2000, pp. 181-189.
- [28] M. Baker, P. A. Farrell, H. Ong, and S. L. Scott, "Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network," *Proceedings of 4th Annual Linux Showcase & Conference*, Atlanta, GA, 2000, pp. 353-362.
- [29] F. Seifert, D. Balkanski, and W. Rehm, "Comparing MPI Performance of SCI and VIA," *Proceedings of SCI-EUROPE 2000*, Munich, Germany, Aug. 2000.