

# A detailed performance analysis of UDP/IP, TCP/IP, and M-VIA network protocols using Linux/SimOS<sup>1</sup>

Chulho Won<sup>a</sup>, Ben Lee<sup>a</sup>, Chansu Yu<sup>b</sup>, Sangman Moh<sup>c</sup>, Kyoung Park<sup>d</sup> and Myung-Joon Kim<sup>d</sup>

<sup>a</sup>*School of Electrical Engineering and Computer Science, Oregon State University*

*E-mail: {chulho, benl}@eecs.orst.edu*

<sup>b</sup>*Department of Electrical and Computer Engineering, Cleveland State University*

*E-mail: c.yu91@csuohio.edu*

<sup>c</sup>*School of Internet Engineering, Chosun University, Gwangju, Korea*

*E-mail: smmoh@chosun.ac.kr*

<sup>d</sup>*Internet Server Group, Digital Home Research Division, Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea*

*E-mail: {kyoung, joonkim}@etri.re.kr*

**Abstract.** This paper presents a performance study of UDP/IP, TCP/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 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, TCP/IP, 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.

**Keywords:** SimOS, complete system simulation, Linux, instruction set simulators, UDP/IP, TCP/IP, M-VIA

## 1. Introduction

The growing demand for high-performance communication on 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 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 the processor speed but also on the communication protocols and their 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.

---

<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.

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 [2,3] 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, TCP/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 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, TCP/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. The discussion also includes the 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, TCP/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 [1–4,6,13,14]. The 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 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

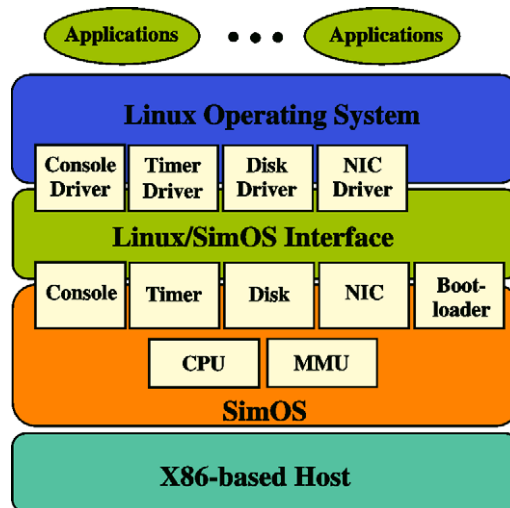


Fig. 1. The structure of Linux/SimOS.

R10000. The CPU models support the execution of the MIPS instruction set [12]. SimOS also models a memory management unit (MMU), including the related exceptions. Therefore, the virtual memory translations occur 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 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 timer. These devices provide the interface between the simulator and the real world. The console model allows a user to read

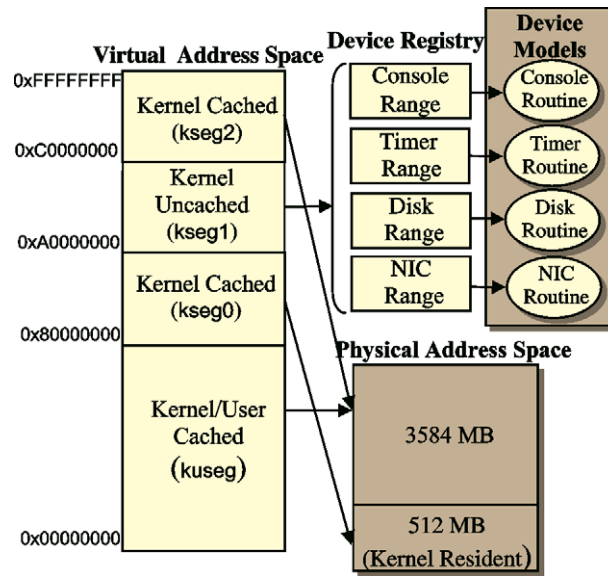


Fig. 2. Address mapping mechanism in SimOS.

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

messages from and type in commands to the simulated machine. 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.

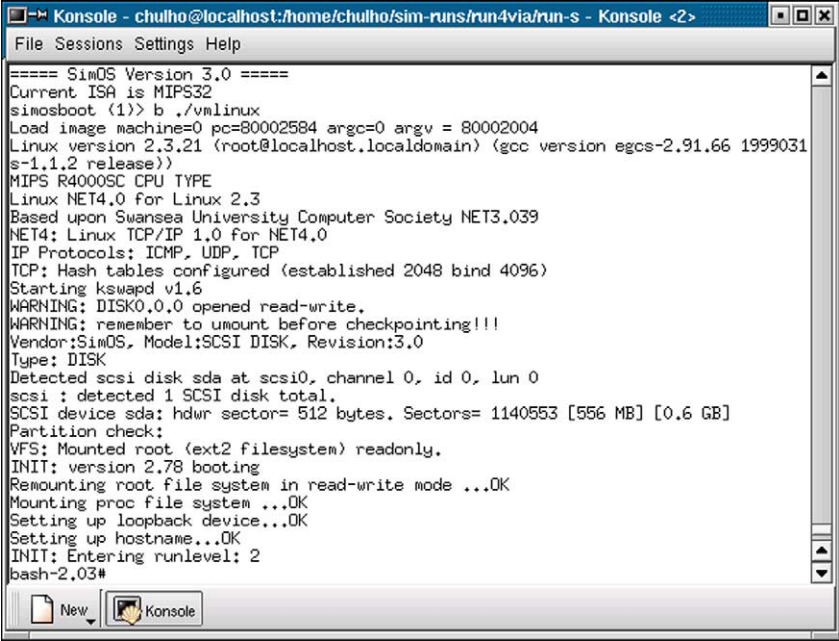
### 3.2. Linux/SimOS interface

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.

#### 3.2.1. Timer and console

SimOS implements a simple real-time *clock* that indicates the current time as the number of seconds that have elapsed 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 Fig. 3.



```

Konsole - chulho@localhost:/home/chulho/sim-runs/run4via/run-s - Konsole <2>
File Sessions Settings Help
===== 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#
  
```

Fig. 3. Linux/SimOS console output with Linux boot message.

### 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 the 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 with 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.

### 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 Fig. 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, and 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.

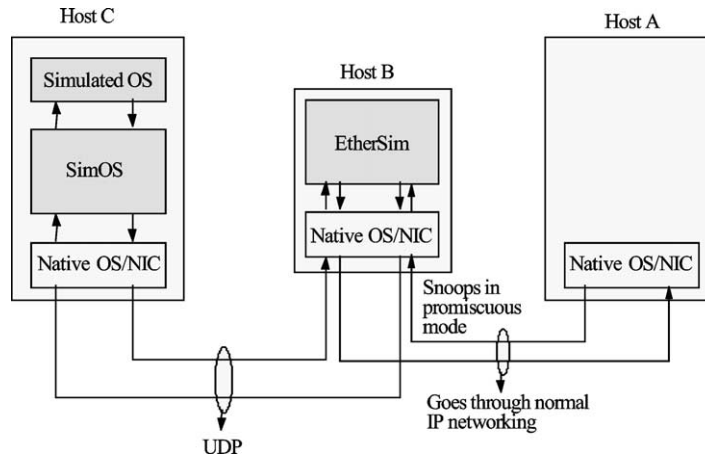


Fig. 4. Communication between a simulated host and a real host using EtherSim.

#### 4. Simulation study of UDP/IP, TCP/IP, and M-VIA

This section presents the performance measurements of UDP/IP, TCP/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 three protocols, test programs were run on Linux/SimOS that accepts command-line options specifying send/receive, a message size, and address. UDP and TCP/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 evaluate the performance of M-VIA. vpingpong employs a number of library functions provided by VIP Provider Library to initiate message transfers on M-VIA. The program has two modes of operation, 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 1200 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 performance of UDP and TCP/IP 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. Some modifications were necessary to run M-VIA on Linux/SimOS. First, because M-VIA was released only for x86-based Linux hosts, some of the source codes had to be modified to run them on Linux/SimOS. In particular, the code for fast traps (vipk\_core/vipk\_ftp.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.

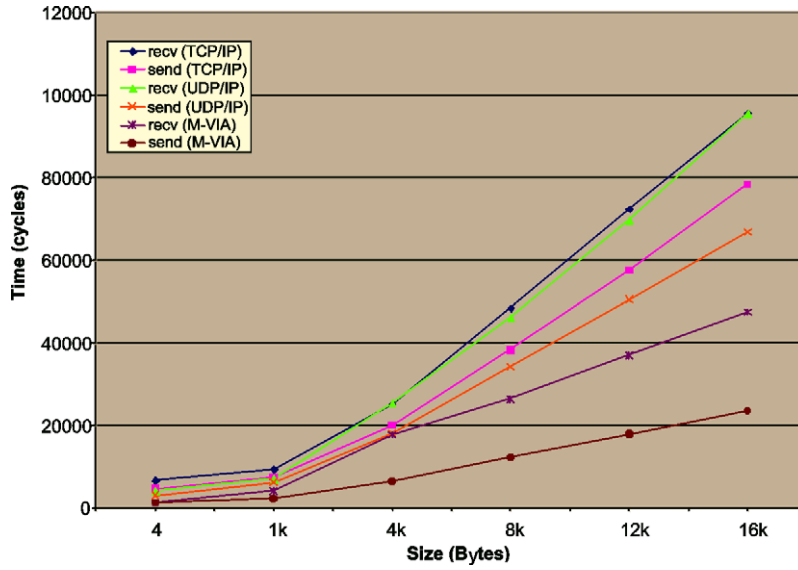


Fig. 5. Message send/receive latencies.

#### 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 Fig. 5. The send results are based on the number of clock cycles required to perform `sendto()` for UDP, `send()` for TCP, or `VipPostSend()` for M-VIA. The receive results are based on the time between the arrival of a message and when `recvfrom()` for UDP, `recv()` for TCP, 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 and TCP, and memory registration for M-VIA. These results also do not include the effects of MAC and physical layer operations.

The results in Fig. 5 clearly show the advantage of using low-latency, user-level messaging, especially for small messages. For message sizes less than MTU, the improvement factors for M-VIA send/receive latencies over UDP and TCP/IP are 2.2~2.6/1.6~3.1 and 3.1~3.3/2.3~4.9, respectively. For message sizes greater than MTU, the improvement factors for M-VIA send/receive latencies over UDP and TCP/IP are 2.8/1.4~2 and 3.1~3.3/1.4~2, respectively.

#### 4.3. Layer-level performance

The latencies for UDP/IP, TCP/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 on each layer and how each layer contributes to the final result. The latencies for UDP and TCP/IP were broken into layers associated with APPL, UDP/TCP, IP, DEV, and DMA. APPL includes the time required to perform socket operations `sendto()/recvfrom()` and `send()/recv()` for UDP and TCP, respectively. UDP/TCP and IP are the times for executing UDP/TCP 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()` (and



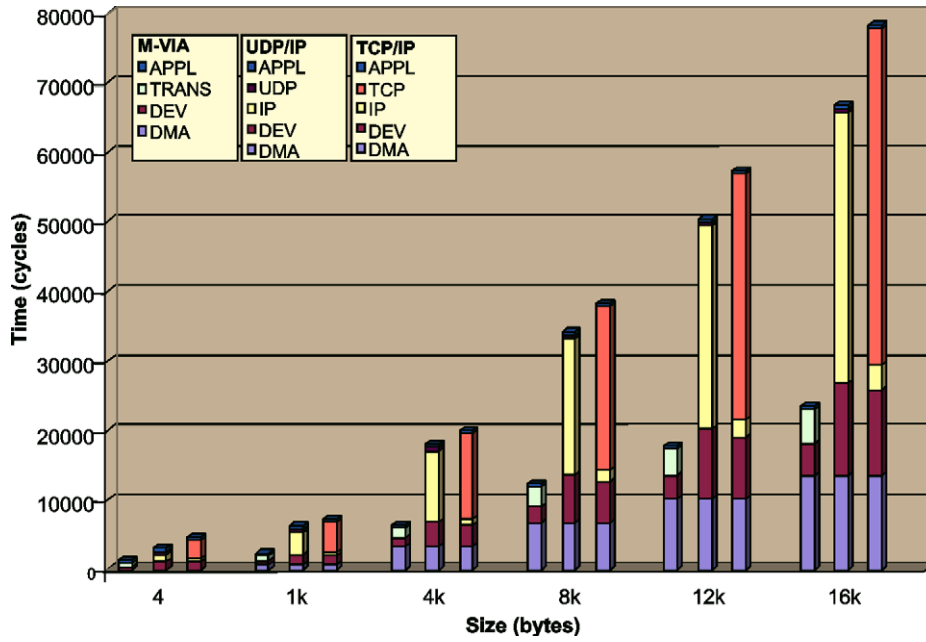


Fig. 6. Send latency.

VipRecvWait()). 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 because the 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 Figs 6 and 7, respectively, where each message size has three bar graphs for M-VIA (left), UDP/IP (middle), and TCP/IP (right). The results are also presented in a tabular form in Table 2. The maximum message size in Table 2 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 Figs 6 and 7 since they would overshadow the other results.

Figure 6 shows the send latencies. For UDP/IP, APPL and UDP are small and remain relatively constant. However, IP and DEV dominate as the message size grows. In particular, for message size greater than MTU, IP increases significantly as a function of messages size. This is because IP handles packet fragmentation, data copying from user space to socket buffer, and checksumming. In addition, DMA also takes a significant portion of the latency for message sizes over 4 Kbytes. For TCP/IP send, TCP constitutes the largest portion of the overall execution time. This is because the TCP layer performs the most of time-consuming operations, which include packet de-fragmentation, data copying from socket to user buffers, checksumming, ACK reception, and flow control. Note that there is no congestion control since packet losses were not simulated. Moreover, TCP dominates while IP represents only a small portion of the overall execution time. This is because, unlike UDP, packet fragmentation and data copying from user space are performed at the TCP layer. Thus, IP has minimal effect on the overall performance of TCP/IP, while it is the direct opposite for UDP/IP. 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 MTU, all the UDP/IP layers, except APPL, increase rapidly. Among them, UDP and IP increases are most noticeable. The increase in IP is caused by de-

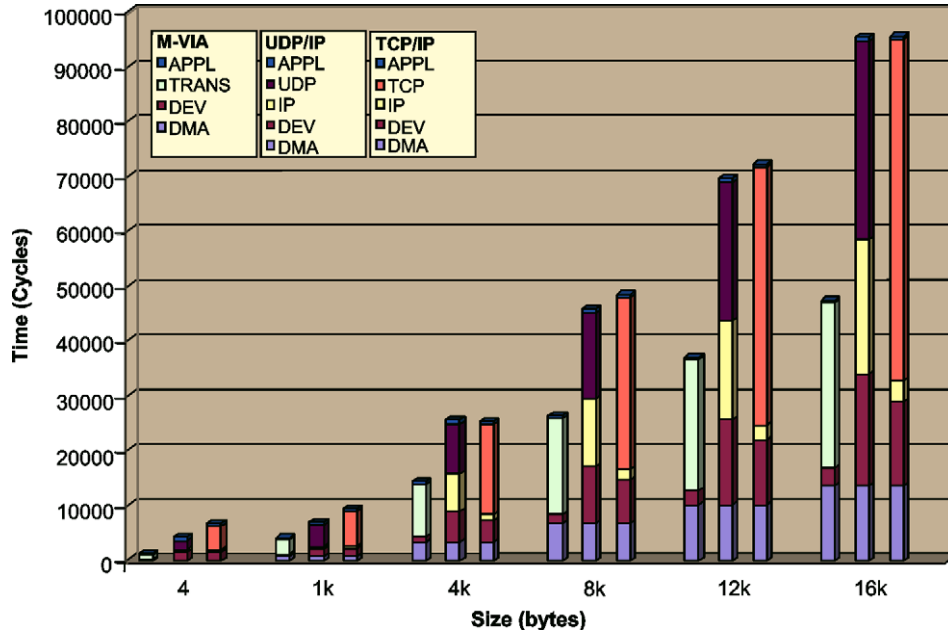


Fig. 7. Receive latency.

fragmentation, while the increase in UDP is due to copying data from socket buffer to user space and checksumming. This is because in Linux, user data copying in UDP/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 TCP/IP, TCP also represents the largest portion of the overall execution time. For M-VIA, TRANS has the most noticeable increase due to extra data copying required from DMA buffer to user space. In addition, de-fragmentation contributes significantly to TRANS for messages larger than MTU.

The latencies for DEV are similar for UDP and TCP/IP, but are larger than M-VIA. There are several reasons for this. First, TCP and UDP/IP have a device queue that acts as an interface between IP and DEV layers for packet send and receive. In contrast, M-VIA does not have an equivalent interface for performance reasons. Second, the legacy protocols have a layered structure, and thus UDP and TCP/IP include the device driver operations in DEV. Since M-VIA integrates the device driver in its transport layer, the resulting latency is included in TRANS, rather than DEV. Third, UDP and TCP/IP have a much larger protocol stack, which increases the likelihood of cache conflicts. For DMA, the latencies increase linearly with the message size. This is consistent since DMA initiation and interrupt handling are already reflected in DEV; therefore, DMA transfer time is dependent only on the message size.

#### 4.4. Function-level performance

The pie charts shown in Figs 8 and 9 give a more detailed picture about what contributes to the amount of time spent on each layer for UDP/IP and M-VIA for message sizes 256 bytes and 4 Kbytes, respectively. For UDP/IP, APPL was further subdivided into APPL\_LIB and APPL\_SOCKET. APPL\_LIB represents the latency between a system call (i.e., `sendto()` or `recvfrom()`) and the start of socket operation (i.e., `sock_sendmsg()` or `sock_recvmsg()`), while APPL\_SOCKET includes the time for socket layer operations. For UDP/IP send, IP was subdivided into IP\_COPY and IP\_ETC. IP\_COPY represents the latency related to copying user data into socket buffer and checksum operations, while IP\_ETC represents the rest of IP overhead. For UDP/IP receive, UDP was subdivided into UDP\_COPY

Table 2  
Message send/receive latency vs. message size

Protocol	Layers	Message size (bytes)							
		4	256	1k	4k	8k	12k	16k	32k
M-VIA send	APPL	357	357	357	382	385	377	387	394
	TRANS	610	612	722	1512	2762	3850	4956	8290
	DEV	455	455	468	1197	2370	3415	4573	6678
	DMA	3	3	853	3413	6826	10240	13653	27306
	<b>Total</b>	<b>1425</b>	<b>1427</b>	<b>2395</b>	<b>6504</b>	<b>12343</b>	<b>17882</b>	<b>19469</b>	<b>42668</b>
M-VIA receive	APPL	439	439	439	473	475	453	459	470
	TRANS	666	1153	2622	9706	17473	23791	30023	59569
	DEV	294	296	291	860	1680	2536	3320	8050
	DMA	3	49	853	3413	6826	10240	13653	27306
	<b>Total</b>	<b>1402</b>	<b>1937</b>	<b>4205</b>	<b>14452</b>	<b>26454</b>	<b>37020</b>	<b>47455</b>	<b>95395</b>
UDP/IP send	APPL	525	525	527	527	543	533	549	600
	UDP	415	419	417	441	443	485	475	500
	IP	992	1336	3199	10218	19636	29090	38854	79180
	DEV	1221	1340	1360	3537	6843	10150	13305	26690
	DMA	3	49	853	3413	6826	10240	13653	27306
	<b>Total</b>	<b>3166</b>	<b>3669</b>	<b>6356</b>	<b>18136</b>	<b>34291</b>	<b>50498</b>	<b>66836</b>	<b>134276</b>
UPD/IP receive	APPL	608	608	608	720	732	830	830	880
	UDP	1678	1896	3937	9051	15693	25069	36065	73040
	IP	395	399	397	6954	12311	18146	24551	50188
	DEV	1594	1286	1304	5495	10350	15423	20405	40805
	DMA	3	49	853	3413	6826	10240	13653	27306
	<b>Total</b>	<b>4278</b>	<b>4238</b>	<b>7099</b>	<b>25633</b>	<b>45912</b>	<b>69708</b>	<b>95504</b>	<b>192219</b>
TCP/IP send	APPL	450	448	450	470	502	500	514	680
	TCP	2604	2585	4499	12265	23469	35247	48393	336555
	IP	374	372	370	840	1650	2564	3596	6632
	DEV	1284	1258	1264	3116	5900	8875	12234	23518
	DMA	3	49	853	3413	6826	10240	13653	27306
	<b>Total</b>	<b>4715</b>	<b>4712</b>	<b>7436</b>	<b>20104</b>	<b>38347</b>	<b>57426</b>	<b>78390</b>	<b>394691</b>
TCD/IP receive	APPL	518	520	520	522	520	540	536	630
	TCP	4266	4347	4293	16490	31395	47115	62362	123956
	IP	399	395	399	981	1890	2773	3682	7169
	DEV	1650	1388	1430	3972	7888	11736	15396	29985
	DMA	3	49	853	3413	6826	10240	13653	27306
	<b>Total</b>	<b>6836</b>	<b>6699</b>	<b>7495</b>	<b>25378</b>	<b>48521</b>	<b>72404</b>	<b>95629</b>	<b>189046</b>

and UDP\_ETC. UDP\_CPY represents the portion of the UDP overhead related to copying packet payloads and checksumming, while UDP\_ETC represents the rest of the UDP latency. DEV 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 represents the time for interrupt handling when DMA transfers complete. For UDP/IP receive, DEV\_DRV is always zero because DMA is initiated by NIC when packets arrive.

The M-VIA layers were also subdivided to focus on the effects of doorbell mechanism, interrupt handling, and memory translation table lookup operations, while APPL was further subdivided into APPL\_LIB and APPL\_DBELL. APPL\_LIB represents the latency between a VIA library call (i.e., `VipPostSend()` or `VipPosrRecv()`) and start of

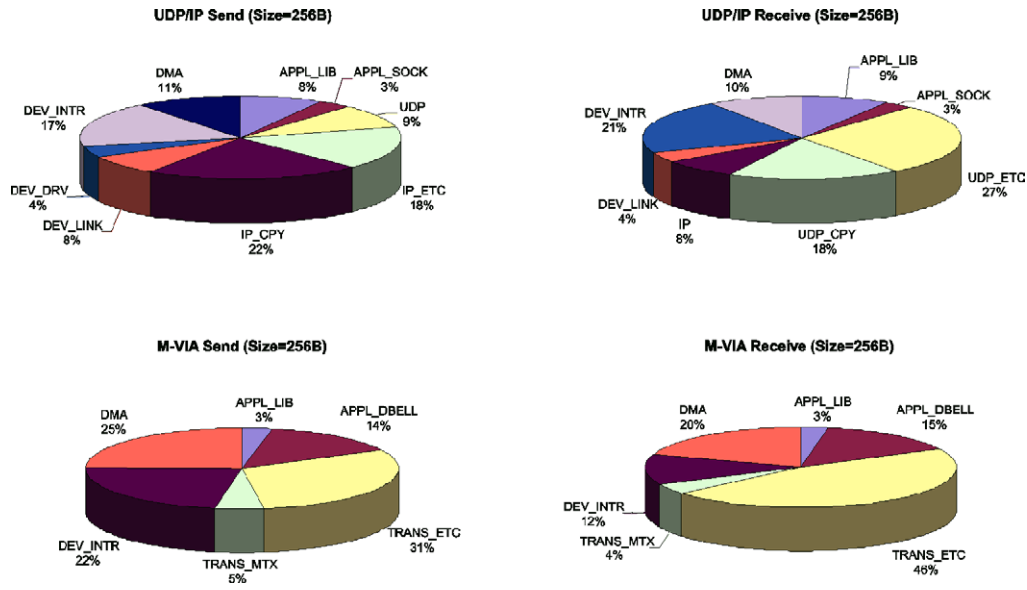


Fig. 8. Latency breakdown for 256-byte message size.

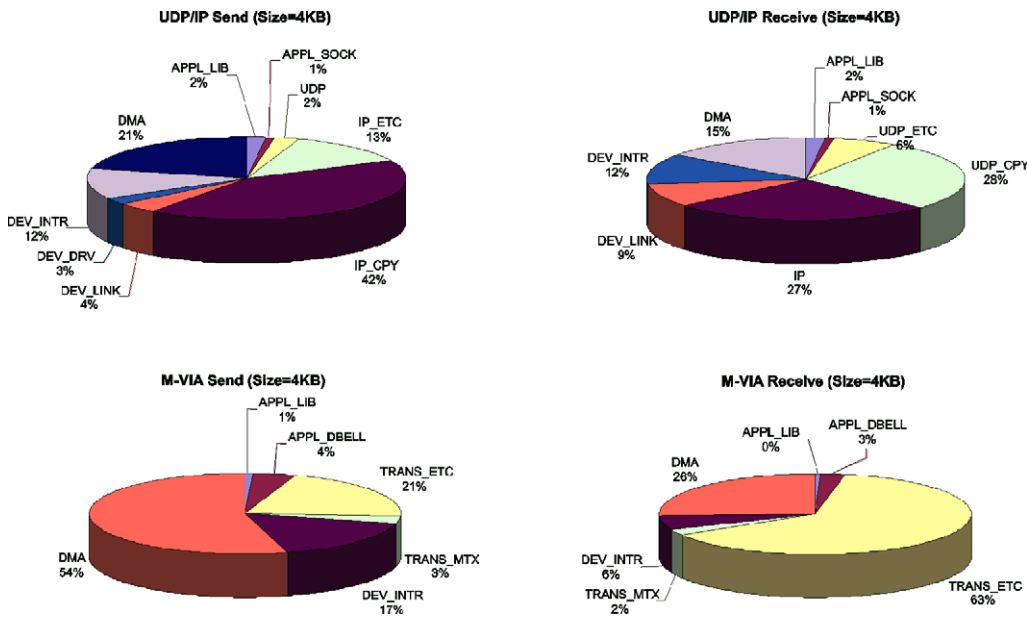


Fig. 9. Latency breakdown for 4-Kbyte message size.

the doorbell operation. APPL\_DBELL is the time to execute a doorbell operation. A doorbell is a mechanism to initiate the NIC to service 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 latencies. For example, an `ioctl()` system call for a 256-byte message send requires around 300 cycles and represents 84% 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 22% and 12% of send and receive latencies, respectively. As message size increases to 4 Kbytes, DEV\_INTR still represents 17% 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 performing DMA transfer 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, TCP/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 is based on an open-source environment, it is a powerful and flexible simulation platform 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 type of prototyping can provide some insight on how the next generation HCA should be designed for InfiniBand Architecture.

## Acknowledgements

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

## References

- [1] V.S. Pai, P. Ranganathan and S.V. Adve, RSIM Reference Manual, Version 1.0, Technical Report 9705, Dept. of Elec. and Comp. Eng., Rice University, August 1997.
- [2] P.S. Magnusson et al., Simics: A full system simulation platform, *IEEE Computer* **35**(2) (2002), 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, in: *Proc. of the 1997 Winter Simulation Conference*, 1997.
- [5] N. Leavitt, Linux: at a turning point?, *IEEE Computer* **34**(6) (1991).
- [6] D. Burger and T.M. Austin, The SimpleScalar tool set, Version 2.0, U. Wisc. CS Dept. TR#1342, June 1997.
- [7] M. Beck et al., *LINUX Kernel Internals*, 2nd edn, 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<sup>TM</sup> Architecture Specification Volume 1, Release 1.0.a. Available at <http://www.infinibandta.org>.
- [12] M-VIA: Virtual Interface Architecture for Linux, 2001. Available at <http://www.nersc.gov/research/FTG/via/>.
- [13] WARTS, Wisconsin Architectural Research Tool Set. Available at <http://www.cs.wisc.edu/~larus/warts.html>.
- [14] SIMCA, the Simulator for the Superthreaded Architecture. Available at <http://www.mount.ee.umn.edu/~lilja/SIMCA/index.html>.
- [15] D. Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers, Inc., 1999.
- [16] SimOS PowerPC. Available at <http://www.research.ibm.com/simos-ppc>.
- [17] SimpleScalar Version 4.0 Release Tutorial, in: *34th Annual International Symposium on Microarchitecture*, December 2001. Available at <http://www.simplescalar.com/tutorial.html>.
- [18] M. Rosenblum et al., Using the SimOS machine simulator to study complex computer systems, *ACM Transactions on Modeling and Computer Simulation* **7**(1) (1997), 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), in: *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, in: *International Conference on Parallel Processing*, August 2000.
- [23] A. Begel et al., An analysis of VI architecture primitives in support of parallel distributed communication, *Concurrency and Computation: Practice and Experience* **14**(1) (2002), 55–76.
- [24] P. Buonadonna, A. Geweke and D.E. Culler, An implementation and analysis of the virtual interface architecture, in: *Proc. of the SC98*, Nov. 1998.
- [25] A. Rubini and J. Corbet, *Linux Device Driver*, 1st edn, O'Reilly, 1998.
- [26] J. Crowcroft and I. Phillips, *TCP/IP and Linux Protocol Implementation: System Code for Linux Internet*, Wiley, 2001.
- [27] P.A. Farrell and H. Ong, VIA communication performance over a gigabit ethernet network, in: *19th IEEE International Performance, Computing, Communications Conference (IPCCC 2000)*, Feb. 2000, pp. 181–189.
- [28] M. Baker et al., Performance comparison of LAM/MPI, MPICH, and MVICH on a Linux cluster connected by a gigabit ethernet network, in: *Proc. of 4th Annual Linux Showcase & Conference*, Atlanta, Oct. 2000, pp. 353–362.
- [29] F. Seifert, D. Balkanski and W. Rehm, Comparing MPI performance of SCI and VIA, in: *3rd International Conference on SCI-based Technology and Research (SCI-Europe 200)*, August 2000.
- [30] C. Won et al., Linux/SimOS – A simulation environment for evaluating high-speed communication systems, in: *International Conference on Parallel Processing (ICPP-02)*, August 2002.