Linux/SimOS - A Simulation Environment for Evaluating High-speed Communication Systems[1]

Chulho Won[†], Ben Lee[†], Chansu Yu[*], Sangman Moh[‡], Yong-Youn Kim[‡], Kyoung Park[‡]

[†]Electrical and Computer Engineering Department
Oregon State University
{chulho, benl}@ece.orst.edu

*Department of Electrical and Computer Engineering
Cleveland State University
c.yu91@csuohio.edu

‡Computer and Software Laboratory
Electronics and Telecommunications Research Institute (ETRI)
Taejon, Korea
{yykim, smmoh, kyoung}@etri.re.kr

Abstract

This paper presents Linux/SimOS, a Linux operating system port to SimOS, which is a complete machine simulator from Stanford. 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.

---

1 Introduction

The growing demand for high-performance communication for system area networks (SANs) has led to much research efforts towards low-latency communication protocols, such as Virtual Interface Architecture (VIA) [10] and InfiniBand Architecture (IBA) [11]. As these new protocols emerge, they need to be evaluated to understand how they perform and to identify key bottlenecks. Communication performance is dependent on not only 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, the 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 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 interaction between the hardware and the software at cycle-level. 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 the system evaluation does 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 *Linux/SimOS*, a Linux operating system port to SimOS, which is a complete machine simulator from Stanford [3]. Our work is motivated by the fact that the current version of SimOS only supports the proprietary SGI IRIX operating system. 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 in a non-intrusive manner, including the effects of the kernel, device driver, and network interface.

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. Section 4 presents the simulation study of UDP/IP and M-VIA, an implementation of the Virtual Interface Architecture for Linux. 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 simulators 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 uni- and multiprocessor 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 for users 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.
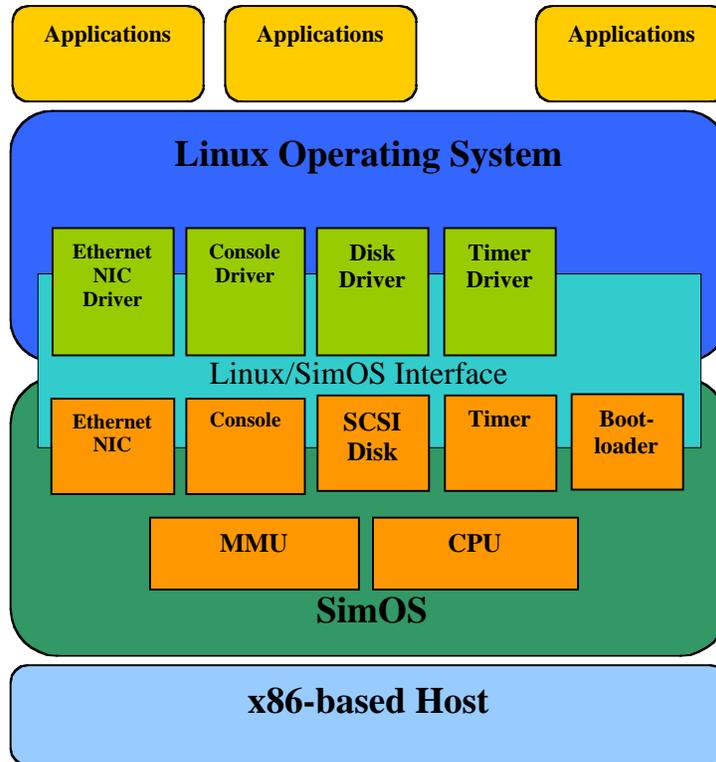
Figure 1: The structure of Linux/SimOS

3 Overview of Linux/SimOS

Figure 1 shows the structure of Linux/SimOS. A x86-based Linux machine[2] 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/MIPS kernel version 2.3 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].

---

[2] SimOS was originally developed to run on SGI IRIX/MIPS platform. Since then it has been ported to run on several platforms, including Sun SPARC, IBM PowerPC and DEC Alpha. In our case, x86 Linux machine was chosen because of its popularity and availability.
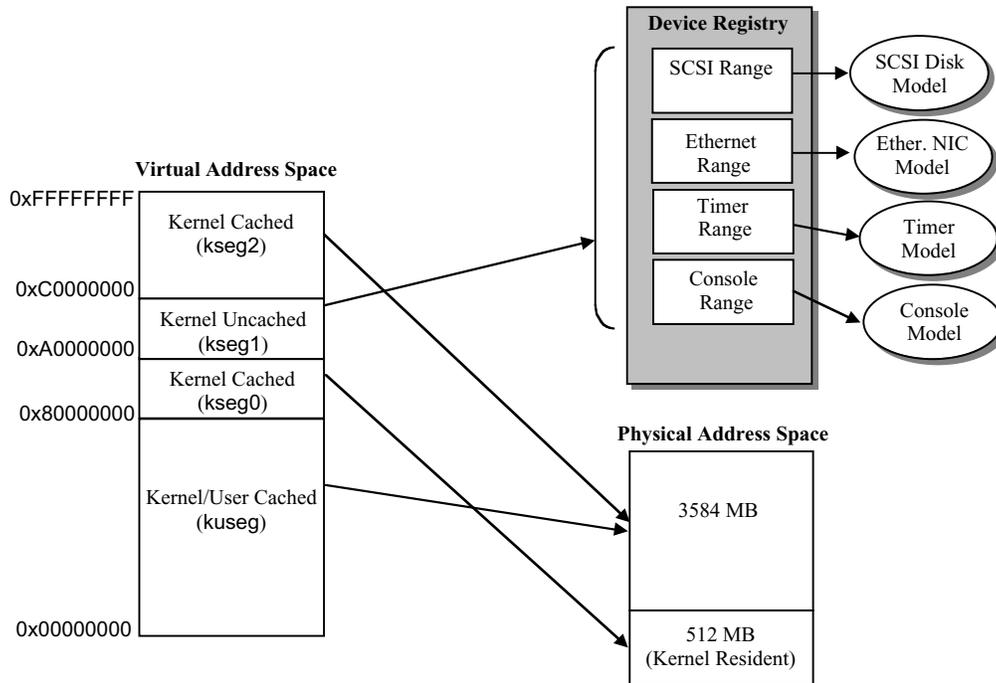
Figure 2: SimOS Memory and I/O Device Address Mapping

SimOS supports two execution-driven, cycle-accurate CPU models: Mipsy and MSX. Mipsy models a simple pipeline similar to the MIPS R4000, while MSX models a superscalar, dynamically scheduled pipeline similar to MIPS R10000. The CPU models support the execution of the MIPS instruction set [12], including privileged instructions. 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 bused-based cache-coherent memory system or a Cache-Coherent Non-uniform Memory Architecture (CC-NUMA) memory 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 lower 512 MB in 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 accesses. 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 first has to 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.

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

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 provides the functionality of allowing 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 NFS client 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 is 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 the Linux hardware drivers and SimOS I/O device modules.

Timer and Console

SimOS provides 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 as shown in 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.

The *console model* in SimOS consists of two registers: a control/status register and a data register. In particular, the data register is implemented as a single entry FIFO queue. However, real serial controllers, such as UART, have multiple-entry FIFO queue for faster serial I/O. Therefore, the Linux console driver was modified to support only a single character transfer over the single entry FIFO.

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 the 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.

Kernel Bootloader

When the kernel and the device drivers are prepared and compiled, a kernel executable is generated in ELF binary format[3] [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 the executable file and looks for headers in the 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.

Ethernet NIC

The SimOS *Ethernet NIC model* supports connectivity to simulated hosts as well as to real hosts. This is achieved using UDP packets of the local host. The Ethernet NIC model encapsulates its simulated Ethernet frames in UDP packets and sends them through NIC of the local host to a network simulator

---

[3] GNU C compiler can also be configured to generate the ECOFF binary format, which the SimOS bootloader also accepts. However, in the process of generating the ECOFF binaries, we encountered a non-trivial problem and opted to instead modify the bootloader to properly handle ELF format.
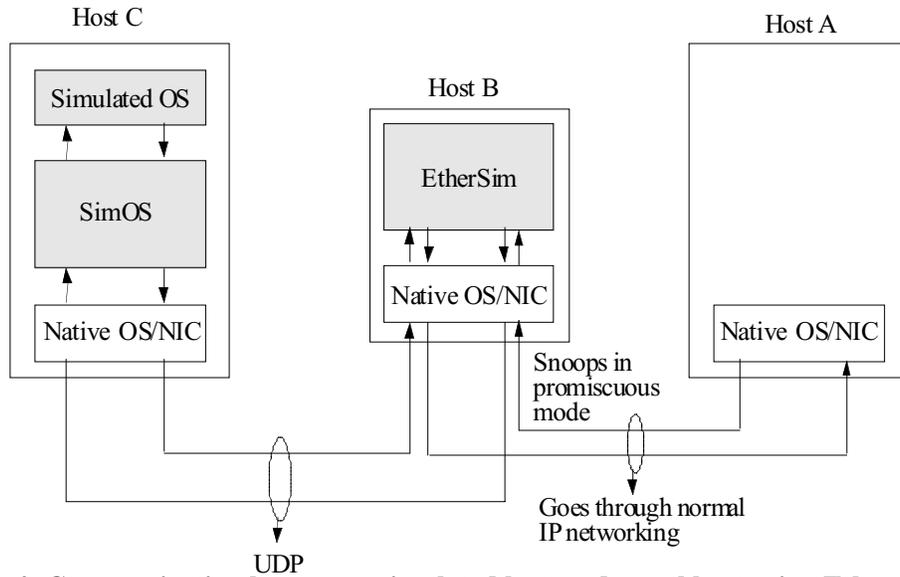
Figure 4: Communication between a simulated host and a real host using Ethersim.

called EtherSim, which runs on on a different host. The main function of EtherSim is to forward the received packets to the destination host. EtherSim will be explained in the next subsection.

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 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

Network simulation in SimOS can be performed using a separate simulator called *EtherSim* [3]. Although EtherSim is not directly related to the Linux/SimOS interface, we briefly discuss its functionality and the modifications that were made to facilitate network simulation with Linux/SimOS.

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 the 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, a 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 simulation results of UDP/IP and M-VIA [12]. To evaluate the performance of these two protocols, a simple test program was run on Linux/SimOS that accepts command-line options specifying send/receive, message size, and a UDP port number. The UDP/IP performance was evaluated by directly sending messages through the legacy protocol stack in Linux/SimOS. On the other hand, M-VIA[4], which is Virtual Interface Architecture implementation for Linux, requires three components: VI

---

[4] Because M-VIA was released only for x86-based Linux host, 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 basically rewritten
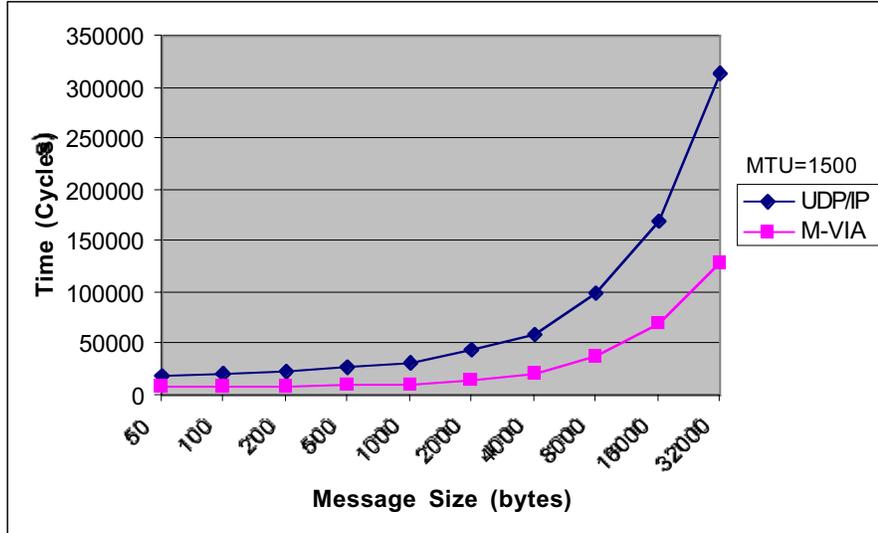
Figure 5: Total Execution Time vs. Message Size.

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, its driver had to be modified (as discussed in Subsection 3.2) to work with SimOS Ethernet NIC.

The CPU model employed was Mipsy with 32 KB L1 instruction and data caches with 1 cycle hit latency, and 1 MB L2 cache with 10 cycle hit latency. The main memory was configured to have 32 MB with hit latency of 100 cycles, and DMA on the Ethernet NIC model was set to have a transfer time of 30 MB/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 perform in a non-intrusive manner.

The simulation study focused on the execution time (in cycles) required to perform a send using UDP/IP and M-VIA. Simulations were run with a fixed MTU (Maximum Transmission Unit) size of 1,500 bytes and varying message sizes. The total execution time required to perform the respective sends for different message sizes is shown in Figure 5. The results are based on the number of cycles executed by the CPU that begins with the socket call sendto() for UDP/IP and VipPostSend() for M-VIA and ends

---

because the MIPS system supports a different system call convention than x86-based systems.
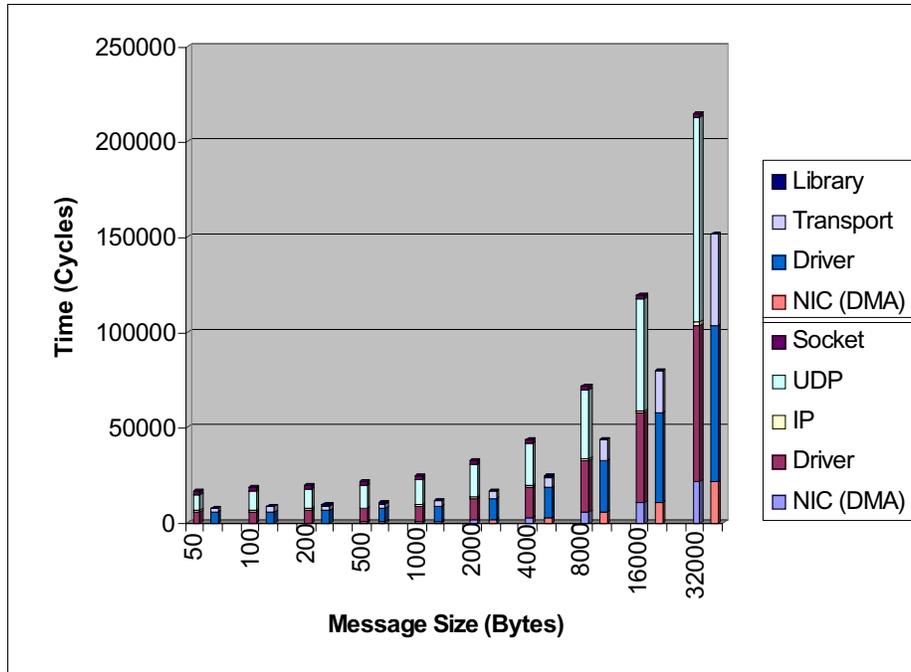
Figure 6: Execution times for each layer vs. Message Size.

with the completion of DMA operations in the NIC. These results do not include the time needed to set up the socket communication for UDP/IP and memory region registration for M-VIA. It also does not include the effects of MAC and physical layer operations. These results clearly show the advantage of using low-latency, user-level messaging. The improvement factor ranges from 2.3 to 2.9 over UDP/IP

The cycle times for UDP and M-VIA send were then divided based on the various layers available for each protocol. This allows us to see how much time is spent at each layer of the protocol and how the data size affects the number of cycles required to perform a send. For UDP/IP, the layers are Socket, UDP, IP, Driver, and DMA operations for NIC. For M-VIA, the layers are VI Library calls, Transport layer, Driver, and NIC. These results are shown in Figure 6, where each message size has a pair of bar graphs representing the execution times of UDP/IP (left) and M-VIA (right). The amount of cycle time spent on the Socket layer stays relatively constant for all message sizes and represents only a small portion of the total execution time. In contrast, the amount of time spent on the UDP layer varies with the message size. This is due to the fact that data copying from user space to sk_buff buffers and checksumming operation (which is optional) are done at the UDP layer. The effect of the IP layer was

minimal compared to the rest of the layers and remains constant for message size smaller that MTU size. There was a slight increase in execution time for message sizes larger than the MTU size due IP fragmentation, but again, this effect was minimal. For M-VIA send (i.e., VipPostSend), the Library layer creates a descriptor in registered memory, adds the descriptor to the send queue, and rings the door bell. The Transport layer then performs virtual-to-physical translation and transfers the control to the Driver layer. As can be seen in Figure 6, the Library layers have a negligible effect on the overall performance, especially when message size is large. However, virtual-to-physical translation in the Transport layer constitutes a significant portion of the total execution time.

For Driver and NIC layers, both protocols show similar results. This is because M-VIA uses similar type of drivers to communicate with the Ethernet NIC model. The only major difference between the two drivers is that M-VIA driver supports a fast trap. The primary function of the Driver layer is to set up the NIC's DMA, copy data to DMA buffers, and receive interrupts when the DMA completes the transferring of data from the DMA buffers in the main memory to the FIFO buffers in the NIC. As can be seen, the execution time of the Driver layer varies as a function of the message size and represents a significant portion of the total execution time. For M-VIA, it is the dominant factor in the total execution time. Finally, the DMA transfer in the NIC layer is shown, which also varies linearly with the message size. This is consistent since DMA setup and interrupt processing are already reflected in the Driver layer; therefore, DMA transfer time is dependent only on the message size.

The pie charts shown in Figures 7 and 8 give further details about what was contributing to the amount of time spent on each layer. The UDP layer has the following operations: UDP header operations, data copy, sk_buff structure operations, checksumming, and error processing. Since data copy and checksum operations dominate as message size grows, this was categorized as UDP Copy+Cksum and all others were summed up into a single category called UDP Ops. The Driver layer was broadly categorized as DMA setup, consisting of data copy from sk_buff to DMA buffers and setting up DMA, and Interrupt processing. Socket and IP layers were not subdivided since their effect on the overall
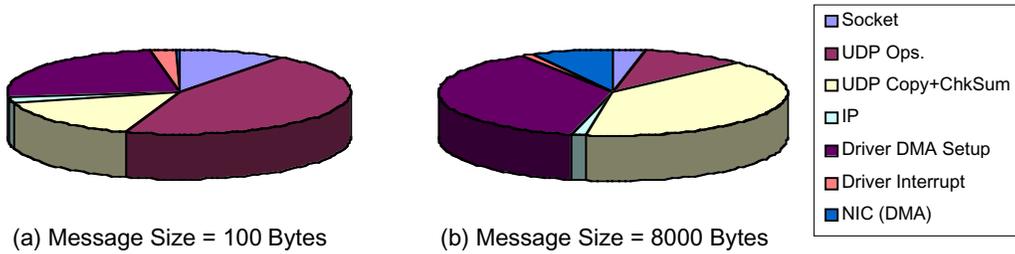
14

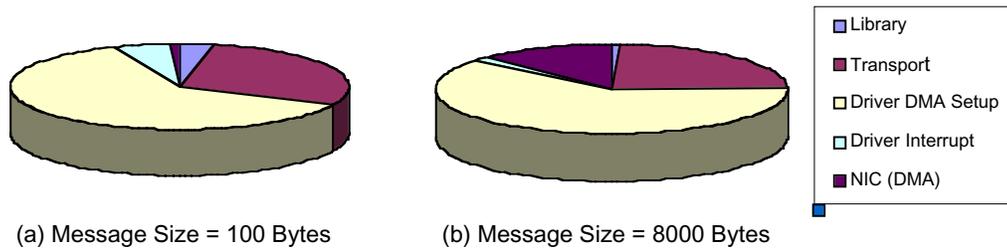**Figure 7: Send time breakdown for UDP/IP.**

Figure 8: Send time breakdown for M-VIA.

execution time was negligible. This was also the case for the NIC layer since MAC and the physical layers were not modeled. As shown in Figure 7, for small messages (e.g., 100 bytes), the largest latency is the UDP Ops. followed by Driver DMA Setup. But as message size grows (e.g., 8000 bytes), UDP Copy+ChkSum and DMA Setup overheads dominate due to data copy operations in both categories.

For M-VIA, the Driver layer was further subdivided into Driver DMA Setup, consisting of data copy from user space to DMA buffers and setting up DMA, and Driver Interrupt. The data copy from user space to DMA buffers are required because Ethernet NIC model is not VIA-aware. Thus, the results in Figure 8 clearly show VIA-aware NIC would eliminates this overhead and thereby significantly improve the performance.

5 Conclusion and Future Work

This paper discussed our effort of porting Linux operating system to SimOS. Moreover, the capability of Linux/SimOS was demonstrated by performing detailed simulation study of UDP/IP and M-VIA. The

results confirm that Linux/SimOS is an excellent tool for studying communication performance by showing the 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[5], it is a powerful and flexible simulation environment for studying all aspects of computer system performance.

There are endless 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 and evaluate new network interfaces. One such example is the Host Channel Adapter (HCA) [11] for InfiniBand, 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 the InfiniBand Architecture.

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/sun4m: A VIRTUAL WORKSTATION," *Proc. the USENIX Annual Technical Conference (NO 98), New Orleans*, Lousiana, June 1997.

[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,* 34(6):12-15, 1991.

[6]   D. Burger *et al*., "The SimpleScalar Tool Set, Version 2.0," *U. Wisc. CS Dept. TR#1342*, June 1997.

---

[5]   We are planning to make Linux/SimOS available to the public in the near future.

[7]   M. Beck, *et al.*, *LINUX Kernel Internals,* Second 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 at
      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. Available at
      http://www.cs.wisc.edu/~larus/warts.html.

[14] SIMCA, the Simulator for the Superthreaded Architecture. Availanle at http://www-
      mount.ee.umn.edu/~lilja/SIMCA/index.html

[15] Dominic 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, *34th 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," To appear in *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), Jan 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, 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.