Linux/SimOS - A Simulation Environment for Evaluating High-Speed Communication Systems

Chulho Won and Ben Lee
Electrical and Computer Engineering Department
Oregon State University
{chulho, benl}@ece.orst.edu

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

Sangman Moh, Yong-Youn Kim, and Kyoung Park
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. The motivation for Linux/SimOS is to alleviate the limitations of SimOS, which only supports proprietary operating systems. The contributions made in this paper are two-fold: First, the major modifications that were necessary to run Linux on SimOS are described. 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 of communication performance, including the effects of the kernel, device drivers, and network interface.

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, accurate evaluation method is needed to understand how the protocols perform and to identify key bottlenecks. Communication protocols closely interact 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. Another advantage of a complete system simulation is that the system evaluation does not depend on the availability of the actual hardware. For example, a new network interface can be prototyped by building a simulation model for the network interface.

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 of 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 simulator for studying computer system performance. There are several variations of SimOS. SimOS-PPC is being developed at IBM Austin Research Laboratory, which models a variety of uni- and multiprocessor PowerPC-based systems and micro-architectures [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 SimIC [2] was developed with the same purpose as SimOS and supports a number of commercial operating systems including Linux. 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.

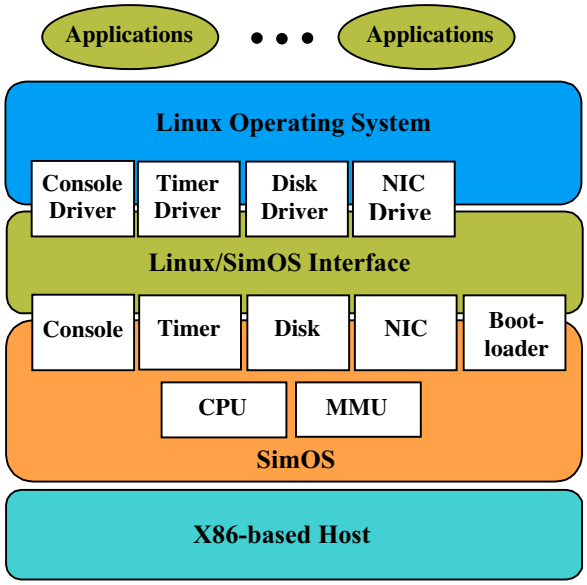

Figure 1. The structure of Linux/SimOS.

3. Overview of Linux/SimOS

Figure 1 shows the structure of Linux/SimOS. 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 I/O devices, such as disk, console, and Ethernet NIC. On top of the target machine, a 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].

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

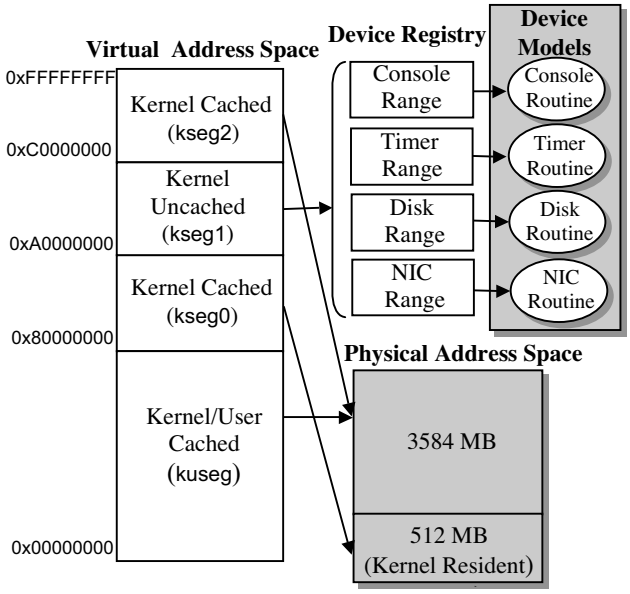

Figure 2. Address mapping mechanism in SimOS.

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.

SimOS provides several I/O device models for console, timer, SCSI disk, and NIC. These models provide not only the device functionality but also 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 an 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.

Table 1. I/O device address map.

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

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 device drivers and SimOS I/O device models.

3.2.1. 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 (see Table 1), and a user program reads the current time using the gettimeofday() system call.

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.

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 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.

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 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.

3.2.4. 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 called EtherSim [4], which runs on a different host. The main function of EtherSim is to forward the received packets to the destination host.

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.

4. Simulation Study of UDP/IP and M-VIA

This section presents the simulation results of UDP/IP and M-VIA [12] to demonstrate the capabilities of Linux/SimOS. To evaluate the performance of these two protocols, a test program was run on Linux/SimOS that accepts command-line options specifying send/receive, a message size, and an address. In order to aid the measurement of execution times of send/receive through the protocol layers, a simple program called Echo Server was written. The function of the Echo Server is to receive the network packets generated from a simulated host through the NIC, appropriately modify the headers of the received packets, and send back to the same simulated host. This allowed us to avoid the loop back mode and properly measure the performance of the driver and NIC.

The UDP/IP performance was evaluated by directly sending messages through the legacy protocol stack in Linux/SimOS. On the other hand, M-VIA, which is Virtual Interface Architecture implementation for Linux, requires 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 (as discussed in Subsection 3.2) to work with SimOS Ethernet NIC.

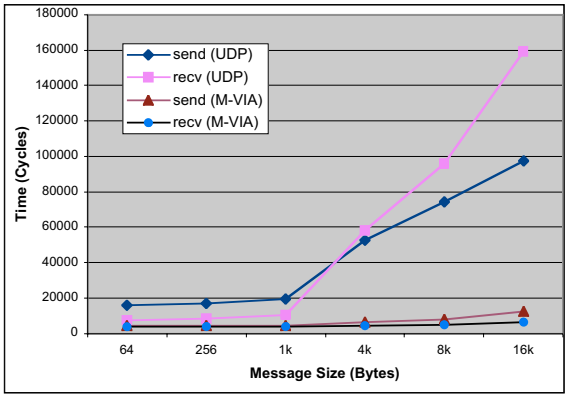

Figure 3. Total execution time vs. message size.

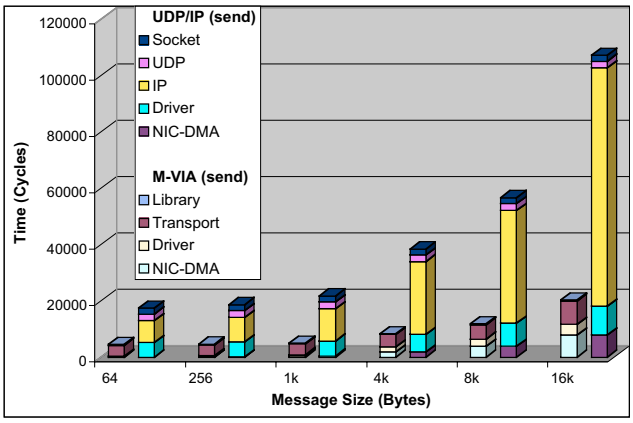

Figure 4. Send execution times for each layer vs. message size.

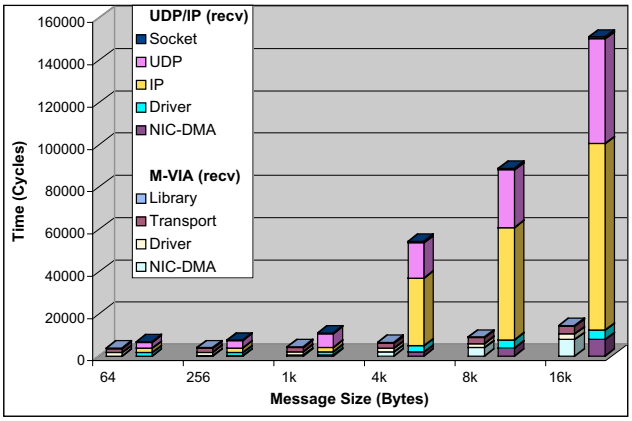

Figure 5. Receive execution times for each layer vs. message size.

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 240 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 performed in a non-intrusive manner.

The simulation study focused on the execution time (in cycles) required to perform send/receive using UDP/IP and M-VIA. These simulations were run with a fixed MTU (Maximum Transmission Unit) size of 1,500 bytes and varying message sizes. The total execution times required to perform the respective send/receive for different message sizes are shown in Figure 3. The send results are based on the number of 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 when the socket call recvfrom() for UDP/IP and VipPostRecv() for M-VIA return. 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. The results in Figure 3 clearly show the advantage of using low-latency, user-level messaging. The improvement factors for send and receive range from 3.5 to 9.3 and 2 to 24 over UDP/IP, respectively.

The cycle times for UDP and M-VIA send/receive 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/receive. 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 DMA operations for NIC. These results are shown in Figures 4 and 5, where each message size has a pair of bar graphs representing the execution times of M-VIA (left) and UDP/IP (right).

For UDP/IP send/receive, 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 IP layer increases as the message size increases. This is due to the fact that in Linux, in addition to IP fragmentation, data copying from user space to sk_buff buffer during a send is done at the IP layer. In contrast, data copying from sk_buff buffer to user space during a receive is done at the UDP layer. For receive, the time spent on the IP layer increases for message sizes larger than MTU mainly due to defragmentation.

For M-VIA send/receive (i.e., VipPostSend/VipPostRecv), the Library layer creates a descriptor in the registered memory, adds the descriptor to the send

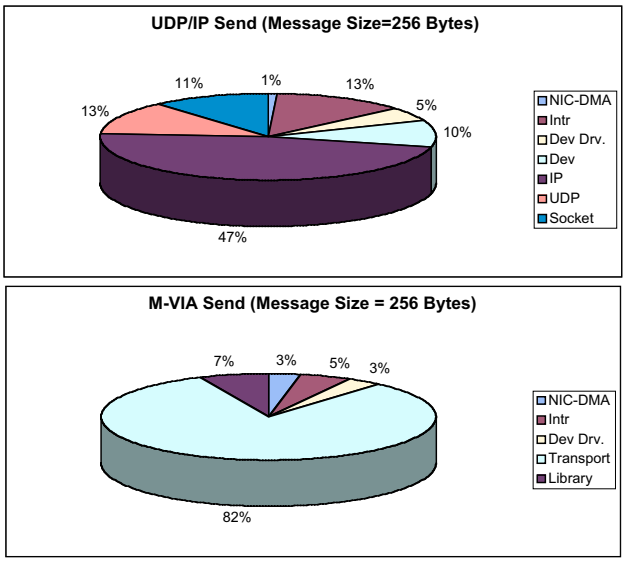

Figure 6. Send time breakdown for UDP/IP and M-VIA for message size 256 bytes.

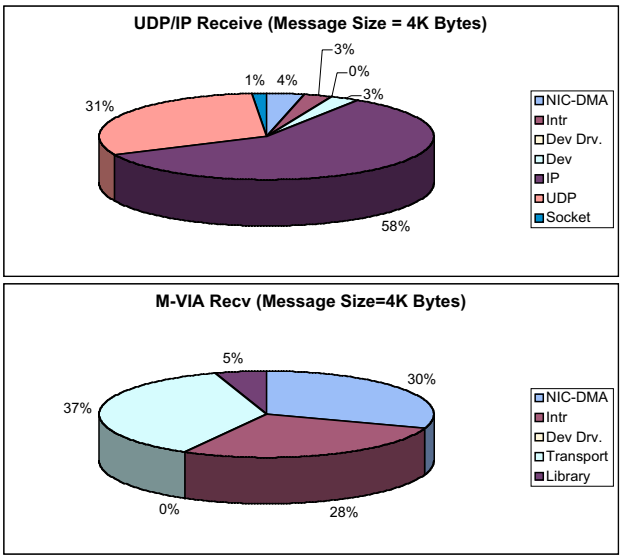

Figure 7. Receive time breakdown for UDP/IP and M-VIA for message size 4K bytes.

queue, and rings the door bell. The Transport layer then performs virtual-to-physical address translation and transfers the control to the Driver layer. As can be seen from the figures, the Library layer has a negligible effect on the overall performance. However, virtual-to-physical address translation and fragmentation/defragmentation in the Transport layer constitute a significant portion of the total execution time.

For Driver and NIC layers, both protocols show similar results. This is because M-VIA uses a similar type of driver to communicate with the Ethernet NIC model. The primary function of the Driver layer is to set up the NIC's DMA and receive interrupts from NIC. As can be seen, the execution time of the Driver layer varies as a function of the message size and represents only a small portion of the total execution time. The DMA transfer in the NIC 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 6 and 7 give further details about what contributes to the amount of time spent on each layer. The UDP layer has the following operations: UDP header operations, data copy (for receive), sk_buff structure operations, and error processing. For send, data copying and fragmentation occurs at the IP layer and it becomes dominant as message size grows. As a result it constitutes a large portion of the overall send time as shown in Figure 6. In contrast, Figure 7 shows that data copying for a receive operation occurs at the UDP layer and thus represents a large portion of the overall execution time compared to a send operation. However, due to its large message size (4 Kbytes) IP layer still dominates because of defragmentation.

For UDP/IP, the Driver layer was further subdivided into interrupt handling (Intr), device specific driver functions (Dev Drv.), and general device functions (Dev). Dev Drv. controls NIC hardware functions, such DMA setup, while Dev provides an interface between the IP layer and Dev Drv, such as packet multiplexing/demultiplexing. Thus, for a send operation, the IP layer initiates a DMA operation using Dev Drv. via Dev and Intr is notified of the completion of the DMA transfer. On the other hand, a receive operation is interrupt initiated and thus handled only by Intr and Dev. As shown in Figures 6, all three portions of the Driver layer represent a significant portion of the overall execution time for sending a small message. However, for a large message as shown in Figure 7, the Driver layer becomes relatively insignificant.

For M-VIA, the Driver layer was further subdivided into interrupt handling (Intr) and device specific driver functions (Dev Drv.). There is no Dev in M-VIA since it is a low-latency, user-level messaging. As can be seen in Figure 6, the combined effect of Intr and Dev Drv. is minimal for small messages. However, for a large message as shown in Figure 7, Intr portion is significant but Dev Drv. has no effect. This is because receive operations are completely handled by Intr.

4. Conclusion and Future Work

This paper discussed our efforts to port 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, 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 evaluate a new network interface to be implemented. 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 the 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*, February 2002, Vol. 35, No. 2, 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," 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. 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, *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," *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.