AN ABSTARCT OF THE DISSERTATION OF

Chulho Won for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on June 11, 2004.
Title: Eager Data Transfer Mechanism for User-Level Network Protocol.

Abstract approved: Redacted for Privacy

This dissertation investigates the use of a hardware mechanism called Eager Data Transfer (EDT) for achieving the reduction of communication latency for user-level network protocol. To reach the goal, the dissertation addresses the following research issues.

First, the development of a communication system performance evaluation tool called *Linux/SimOS* is presented. Linux/SimOS provides a full system profiling capability to allow measurement at various level including hardware, operating system, and application.

Second, the performance analysis of network protocols is presented. For the assessment of overhead related to network protocol operation, *Linux/SimOS* was used to perform the detailed latency measurements for TCP/IP, UDP/IP, and M-VIA network protocols.

Finally, EDT is proposed for reducing communication latency. Since the data transfer time constitutes a significant portion of overall communication latency, the reduction of data transfer time leads to low communication latency. EDT is based on cache coherence interface hardware for reducing data transfer overhead during network protocol operation. Our simulation result shows that EDT is very effective in attaining low communication latency compared to the DMA-based approaches.

EAGER DATA TRANSFER MECHANISM FOR USER-LEVEL NETWORK PROTOCOL

by
Chulho Won

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 11, 2004
Commencement June 2005

Doctor of Philosophy dissertation of Chulho Won presented on June 11, 2004.

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Director, School of Electrical Engineering and Computer Science

Redacted for Privacy

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for privacy

Chulho Won, Author

ACKNOWLEDGMENTS

It is a great pleasure to acknowledge the support and help I have received throughout my academic years from professors, friends, colleagues, and families.

I would like to express gratitude to my advisor, Dr. Ben Lee for his guidance and support. Also, I wish to thank my program committees, Dr. Alexander F. Tenca, Dr. Molly Shor, Dr. Huaping Liu, and Dr. Harold Parks for providing valuable time and effort.

My campus friends and colleagues at Oregon State University have always offered a relaxing and friendly atmosphere. In particular, I would like to thank Dr. Jong-Hoon Youn, Jeong-Min Lee, Dae-Hyun Yoon, Gil-Cho Ahn, Dr. Daniel Arroyo-Ortiz, Vinu Pattery, and many others for their friendship.

Many colleagues at ETRI have provided valuable advice and support during my study. I like to thank Dr. Woo-Jong Han at Intel, Dr. Sang-Man Moh, and Kyoung Park at ETRI for generous research funding, Won-Se Sim at AMD, Dr. Woo-Jong Yoo, Yoon-Ok Park, and In-Cheol Jeong, Dr. Dongho Song, and many others for their advice and support.

My special thanks go to my families for their endless love and dedication. I would like to thank my parents and mother-in-law for their dedication and encouragement. Finally, I like to thank my wife Hyunsook and son Sungyeon for their love and patience. The completion of my study was possible with their dedication and love.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF TABLES

EAGER DATA TRANSFER MECHANISM FOR USER LEVEL NETWORK PROTOCOL

1 Introduction

For last decades, rapid development of VLSI (Very Large Scale Integration) technology brought a revolutionary speed-up of processor. The advancement of VLSI technology enables more and faster circuits in the same die area. Since 1970, commercial microprocessors increase their clock rates by 30% per year and increase their logic density (number of transistors in a chip) by 40% per year [15]. Due to the use of high-density chip, microprocessor designs aggressively integrated the advanced computer architectures such as *superscalar* and *multithreading*. As a result, the performance of microprocessors has been increasing at a much greater rate than clock frequency.

Network technology was also developed rapidly. Today's high-speed LAN (Local Area Network) or SAN (System Area Network) networks reached to several gigabit per second data rates. There are two main driving forces in the technology. The advent of high-speed transmission medium such as fiber greatly contributed to the development of high-performance networks in speed and reliability. Today's high-performance SAN networks transmit and receive at several gigabits rate and with extremely low error rate. Switching technology is also quickly catching up the speed of the transmission medium [41]. Due to the rapid development of the network technology, high-speed interconnection for computer is readily available.

Cluster computers, which consist of high-performance COTS (cheap off-the-shelf) PCs (Personal Computers) connected with high-speed SAN become the most

cost-effective computing platform for parallel applications [12]. The advanced microprocessors are closing the performance gap between dedicated parallel computers and cluster computers for parallel applications. The success of cluster computers attracts parallel applications. As more parallel applications are developed for cluster computers, there is an increasing demand for high-performance communication.

The performance of challenging applications such as parallel scientific application is greatly dependent on message passing facility (message send and receive). Since parallel applications use mainly small messages [16, 24, 26, 29], the performance of the applications depends on small message performance. However, traditional network protocols such as TCP/IP and UDP/IP are not suitable for small message communication because those protocols cause a high overhead in message passing. They are more suitable for large messages. Large messages demand high bandwidth because latency and per-message overhead are amortized over the time of the transfer. Thus, latency is the secondary issue for the bulk message transfer. The support for small messages is different from that for large messages. For small message, network transmission uses relatively large packet header. Moreover, small message suffers from inefficient use of network bandwidth because control overheads such as data transfers, packet formation and link control are relatively big compared to packet payload. Therefore, the reduction of the control overhead is critical for the performance of small message communication.

Considering the emerging applications, the low latency communication becomes a main design issue for cluster network protocols and network interface (NI).

User-level network protocols such as *Virtual Interface Architecture* (VIA) and *InfiniBand Architecture* (IBA) significantly reduce user-to-user communication latency compared with traditional network protocols [17, 21]. The traditional network protocols are executed inside of the operating system and user applications get communication services through kernel system calls. Kernel system call causes a significant increase in user-to-user communication latency because it involves the context switching from user mode to kernel mode. During the kernel mode operation, the user data should be copied into the kernel space for kernel mode operation. Since the kernel mode operation and the extra data coping operations increases the likelihood of flushing the application code and data out of the processor cache memory [35, 45, 46]. Therefore, the in-kernel network protocols not only increase communication latency but also degrade the user application performance. In contrast, user-level network protocols such as VIA (Virtual Interface Architecture) and IBA (InfiniBand Architecture) execute time-critical operations such as message send and receive without the kernel involvement. User applications get communication service by directly calling the user-level network protocols. User-level network protocol uses programmable NI (Network Interface), which includes a set of user interface and processor to execute a message passing primitives (message send and receive) for user-level communication services. User application accesses to the NI-specific user interface (i.e. the doorbell registers in VIA [17]) for calling the message passing primitives on the NI. Therefore, it does not cause the expensive context switch. Additionally, the user-level network protocols avoid data copying overhead by zero-copy data transfer.

Although the user-level protocols have been a success in reducing communication latency, the demand for low latency communication is increasing much higher than the current level. This is due to two factors: technology advancement and application demand. The rapid technology advancement produces high-speed network links and fast processors. As the trend is continued, there will be a significant speed discrepancy developed rapidly in the future. Consequently, the latency caused in NI and network protocol will become a main bottleneck for communication performance. To avoid the problem, the reduction of communication latency is needed to keep up with the rapid development of network and processor technology. Additionally, emerging parallel applications (i.e. scientific grand challenge applications such as global environmental modeling and protein dynamics modeling) need much lower latency communication. For the future success of cluster computer as parallel computing platform, it is necessary to meet the demand for the future applications by providing low latency communication facility.

The context of the dissertation is low latency communication using user-level network protocols for challenging parallel applications. The importance of the research area is clearly understood based on the following considerations.

- Low latency communication is the key issue for the future success of cluster computers as parallel computing platform competing against dedicated parallel machines [13]. High-performance processors and networks have been the main driving force for the past success of the cluster computers. With the rapid development of processor and network technology, the most of the communication overhead occurs in network interface and network protocol. Low

latency communication support in network interface and network protocol should be one of the most important system design issues.

- Low latency communication benefits not only emerging challenge applications but also common applications. The network traffic analysis on a departmental network usage showed the major traffic using legacy network TCP/IP and UDP/IP protocols use small message size less than 200 bytes [24]. A benchmark study on a commercial database found that the messages are less than 200 bytes [26]. A workload analysis using representative scientific parallel application showed that the messages have size less than a kilobyte [16]. Regardless of application type, small messages compose major network traffic on LAN and SAN environment. Low latency communication greatly enhances the performance of communication applications.

The main research goal was to attain low-latency communication using user-level network protocols. To accomplish the goal, the research was conducted on three main areas.

- Communication system evaluation and design tool.

  As the communication research is required to perform the extensive exploration of vast design space covering network interface hardware and network protocol, an efficient evaluation tool is necessary to understand how the protocols perform and to identify key performance factors. Network 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.

Additionally, an efficient evaluation tool can used to assist new designs of network interface and network protocol. Detailed evaluation helps to have a better design. Therefore, supporting of design and evaluation in a single tool greatly enables the fast turn around time for the design and evaluation cycle.

To meet the requirements, the *Linux/SimOS* was developed as an efficient communication system evaluation and design tool [60]. It was designed based on the execution-driven simulator developed at Stanford University [44] and popular Linux operating system. The best advantage of using Linux/SimOS is the full-system evaluation capability, which allows system evaluation at different levels including hardware, operating system, and applications.

- Detailed performance analysis

  The assessment of communication overhead can be achieved through detailed performance analysis. The main focus was on the main overhead locations in network protocol processing. *Linux/SimOS* was used for detailed performance analysis of the network protocols including TCP/IP, UDP/IP and M-VIA. Due to the full system performance profiling capability of the *Linux/SimOS,* all aspects of network protocol processing were captured including application layer, operating system, network protocol, device driver, and network interface hardware.

- Low latency communication support

  The detailed performance study with M-VIA user-level network protocol showed that the data transfer time between the host processor and NI constitutes a significant portion of the overall communication latency [16]. Thus the reduction of data transfer time significantly improves communication performance. Therefore, the research focus was on removing the data transfer overhead from the fast-path of network protocol processing.

  A data transfer mechanism called Eager Data Transfer (EDT) was proposed to exploit an opportunity of removing the data transfer overhead from the fast-path of network protocol processing. The EDT uses cache coherence hardware to detect the availability of user data and move the data immediately.

The rest of dissertation is organized as follows. Chapter 2 describes the Communication system evaluation and design tool, *Linux/SimOS*. Chapter 3 presents the detailed performance analysis of TCP/IP, UDP/IP, and M-VIA network protocols. Chapter 4, as the main part of the dissertation, presents the design of Eager Data Transfer (EDT) Mechanism. It also presents the simulation environment and results. For the simulation results, EDT is compared with DMA-based data transfer mechanisms. Chapter 5 discusses the conclusion.

2 Linux/SimOS

2.1 Introduction

Demand for high-performance network protocols is growing to support emerging applications for cluster computers such as parallel computing and IO networking. The examples are the new network protocol standards such as Virtual Interface Architecture (VIA) [17] and InfiniBand Architecture (IBA) [21]. VIA network protocol was designed to support low-latency communication applications. Since the network protocol executes the protocol stack outside of the kernel, the communication overhead is extremely low compared with the legacy network protocols. IBA has a similar architecture as VIA. As new protocols emerge, accurate evaluation method is needed to understand how the protocols perform and to identify key bottlenecks. Since network protocol operations are closely coupled with system components such as operating system, device driver, and network interface, effective evaluation tool is needed to capture the interactions between components.

The evaluation of communication performance has traditionally been done using instrumentation, 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 use processor simulators [37, 39, 11, 57, 49]. 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. That is, these simulators are unable to capture the complete behavior of a communication system.

On the other hand, a complete machine simulation environment [44, 28] does not have the limitation. A complete machine simulator includes all the system components, such as processor, memory, I/O (Input/Output) 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 modeled by building a simulation model for the network interface.

Based on the aforementioned discussion, *Linux/SimOS*, a Linux operating system port to SimOS, was developed as a complete system evaluation tool. The development of *Linux/SimOS* was motivated by the fact that the current version of SimOS only supports the proprietary commercial operating systems such as SGI IRIX [44], IBM AIX [51], and Compaq Digital UNIX [50]. Therefore, the availability of the open-source 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 advantage of the Linux/SimOS is the system-level profiling capability 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 chapter is organized as follows. Section 2.2 presents the related work. Section 2.3 discusses the Linux/SimOS architecture and the major modifications that were necessary to port Linux to SimOS. Section 2.4 discusses conclusions and future work.

2.2 Related Work

There exist a number of simulation tools that contain detailed models of today's high-performance microprocessors [44, 37, 39, 11, 57, 49, 28]. The SimpleScalar tool set includes a number of instruction-set simulators of varying accuracy/speed to allow the exploration of micro-architecture design space [11]. 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) [37]. 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 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 [44]. It is the most complete simulator for studying computer system performance. The original SimOS was developed for SGI IRIX operating systems. There are several versions based on the original development of SimOS. SimOS-Alpha was developed at Western Research Laboratory, which has developed a SimOS version to run unmodified Digital Unix operating system on it [50]. SimOS-PPC was developed at IBM Austin Research Laboratory [51], which has developed a SimOS version to run unmodified AIX operating system on it. There is also a SimOS interface to SimpleScalar/PowerPC being developed at UT Austin [52]. However, these systems only support proprietary operating systems. Therefore, it is difficult to perform detailed evaluations without knowing the internals of the kernel. Virtutech's SimICS [28] 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

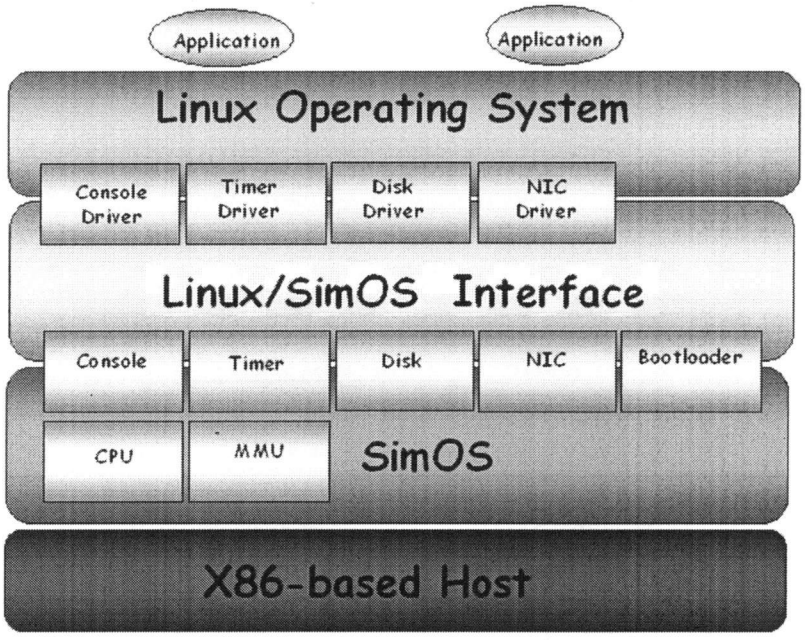


Figure 1: Linux/SimOS Architecture

computer system performance using SimOS with the flexibility and availability of the Linux operating system.

2.3 Architecture of Linux/SimOS

Figure 1 shows the architecture 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.

2.3.1 SimOS machine simulator

SimOS supports two execution-driven simulators: Mipsy and MSX. Mipsy models a simple pipeline similar to MIPS R4000, while MSX models a superscalar, dynamically scheduled pipeline similar to MIPS R10000. The CPU models support the execution of the MIPS instruction set [53]. SimOS also models a memory management unit (MMU), including the related exceptions. SimOS also models the behavior of I/O devices by performing DMA operations and interrupting the CPU when I/O requests complete.

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

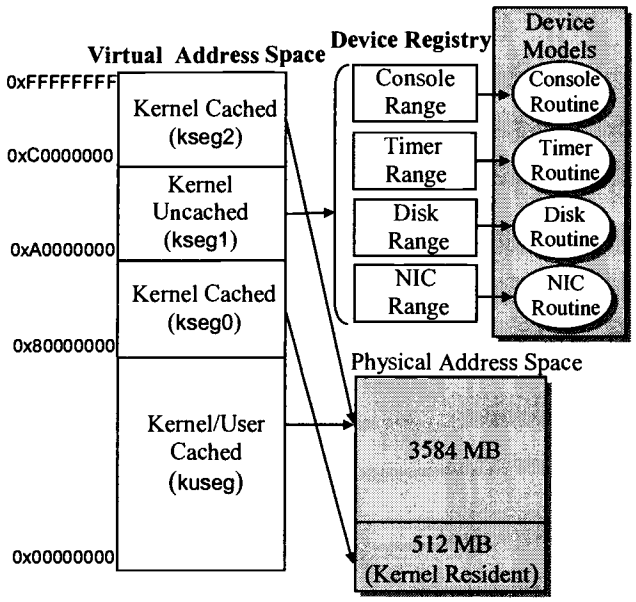


Figure 2: Address mapping in SimOS

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. The console model allows a user to read messages from and type in commands to the simulated machine. The 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.

2.3.2 Linux/SimOS Interface

This section describes *Linux/SimOS interface*, which consists of Linux operating system kernel, bootloader, and a set of IO device drivers. For the development of Linux/SimOS, the major effort was made at *Linux/SimOS interface*, specifically, I/O device drivers. Therefore, the description will focus on the interfacing requirements between Linux hardware drivers and SimOS I/O device modules.

- Kernel Bootloader

When the kernel and the device drivers are prepared and compiled, a kernel executable is generated in ELF binary format [53]. 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.

- 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

Table 1: Linux/SimOS I/O device address mapping.

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

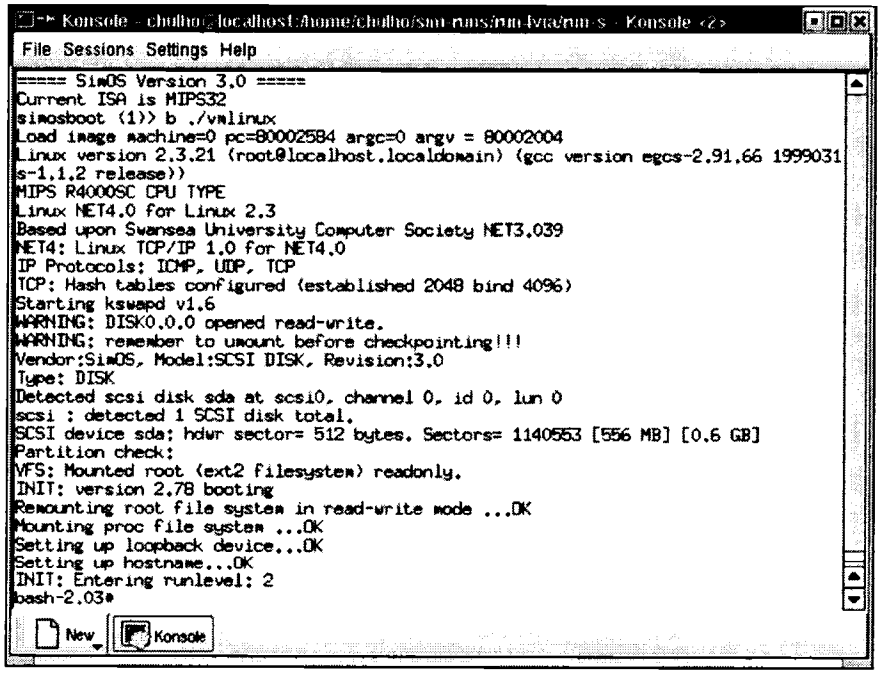


Figure 3: Linux/SimOS boot message.

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

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

- 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.

- Ethernet NIC

The SimOS *Ethernet NIC model* supports connectivity to simulated 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

Network simulation in SimOS can be performed using a separate simulator called *EtherSim* [44]. 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

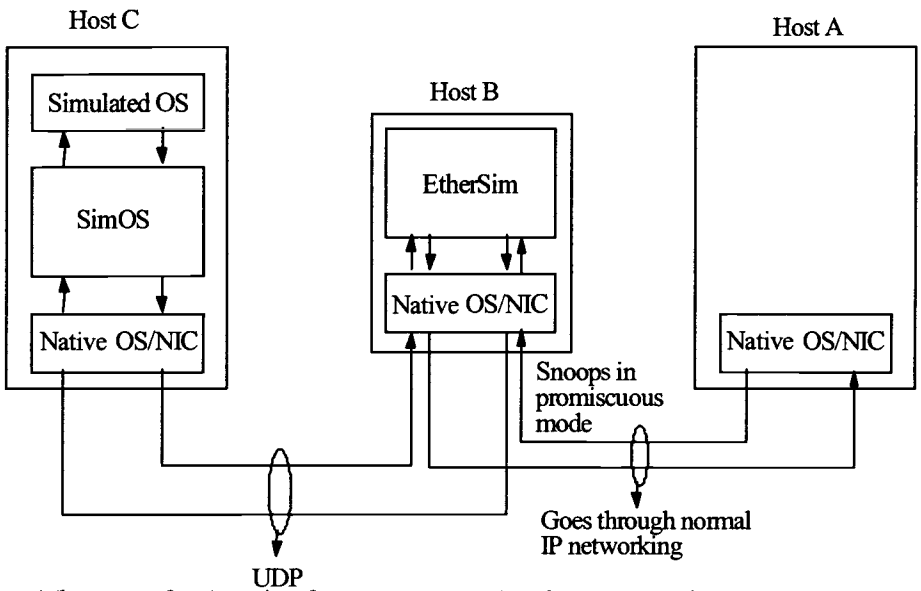


Figure 4: The use of EtherSim for communication between a simulated host and a real system

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

Some modifications were necessary to run EtherSim on a host running Linux operating system. The modifications made were mainly to improve portability. The original EtherSim that comes with the SimOS source distribution was written for Sun Solaris operating system, and could not be ported directly to Linux. Therefore, several of the Solaris operating system specific network library calls were replaced with libpcap [54] and libnet [27], 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.

2.4 Conclusion and Future Work

This chapter discussed our efforts to port Linux operating system to SimOS. 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.

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 Architecture, whose design is 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 using Linux/SimOS over hardware/emulation-based methods is that both hardware and software optimization can be performed using a single tool. This prototyping can provide some insight on how the next generation of HCA should be designed for the InfiniBand Architecture.

3 Detailed Performance Analysis of Network Protocols

3.1 Introduction

The growing demand for high-performance communication on System Area Networks (SANs) has led to significant research efforts towards low latency communication network protocols, such as Virtual Interface Architecture (VIA) [17] and InfiniBand Architecture (IBA) [21]. 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 network protocols and their interaction with the rest of the system such as operating system kernel, device driver, and network interface. Therefore, these interactions must be properly captured to evaluate the protocols and to improve on them.

This chapter presents a performance analysis of network protocols UDP/IP, TCP/IP, and M-VIA using Linux/SimOS. The detailed analysis of UDP/IP, TCP/IP, and M-VIA protocols is performed to help deep understanding of network protocol processing and communication performance. 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 network protocol stack but also in network interface.

The rest of the chapter is organized as follows. Section 3.2 presents the related work. Section 3.3 presents the detailed performance analysis of UDP/IP, TCP/IP, and M-VIA. Section 3.4 concludes the paper and discusses some future work.

3.2 Related Work

Traditionally, research works on performance analysis of network protocols rely on the instrumentation. Since the traditional approach uses instrumentation codes, it limits the profiling scope and it also interrupts the normal flow of execution. Due to the limitation on profiling scope, it is used for measuring application performance, not for system performance. Therefore, the instrumentation based performance analysis does not provide full system profiling capability.

Makineni *et al.* investigates the TCP/IP performance using windows OS and server applications [30]. They used Pentium-M™ processor built-in performance counters and private performance tracking software to access the performance counters. Various TCP operations modes were executed to perform the architectural characterization related to the execution. They focused on investigation the processing requirements for the future increased network bandwidth.

Kant *et al.* investigates the TCP/IP performance using SimpleScalar and SPEC benchmarks (SPEC-WEB) [25]. Network traffic, which was generated from the execution of SPEC benchmarks was fed into SimpelScalar simulator. Due to the memory copy operation of receive operation, they focused on the receive operation of TCP receive side. They investigate cache pollution effects by TCP/IP payloads.

Because the network traffic generation and the performance analysis were performed on different platforms, the accuracy is questionable. And also the other source of inaccuracy is in that the simulator they used does not have a capability of simulating operating system and hardware system.

Compared to the existing method, performance analysis of network protocols using the Linux/SimOS provides much more detailed information such as application program characterization, operating system overhead, network protocol layers, device driver overhead, and network interface hardware operations.

3.3 Performance analysis of UDP/IP, TCP/IP, and M-VIA

This section presents the performance measurements of UDP/IP, TCP/IP, and M-VIA [58] 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 written. For testing UDP/IP and TCP/IP, client/server programs were written using the socket library. For M-VIA, a VI client/server program was used. UDP/IP and TCP/IP performance was measured by sending messages from client program to server program. On the other hand, a program called *vpingpong* was used to evaluate the performance of M-VIA. The micro-benchmark program uses a number of library functions provided by VIP Provider Library to initiate message transfers on M-VIA. It has two modes of operation, send and receive, which are selectable with a command-line option. The vpingpong program is one of

the test programs included in the M-VIA 1.2b2 source code distribution [34].

3.3.1 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. 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.

3.3.2 Overall Performance

The performance measurement focused on the latency (in cycles) to perform send/receive. These simulations were run with a fixed MTU (Maximum Transmission Unit) size of 1,500 bytes with varying message sizes. The total cycle times required to perform send/receive as a function of message size are shown in Figure 5. The send results are based on the number of clock cycles measured from application's send library call (i.e., sendto for UDP, send for TCP, or VipPostSend for M-VIA) to network send. The receive results are based on the time measured from network receive to the return of application's receive library call (i.e., recvfrom for UDP, recv

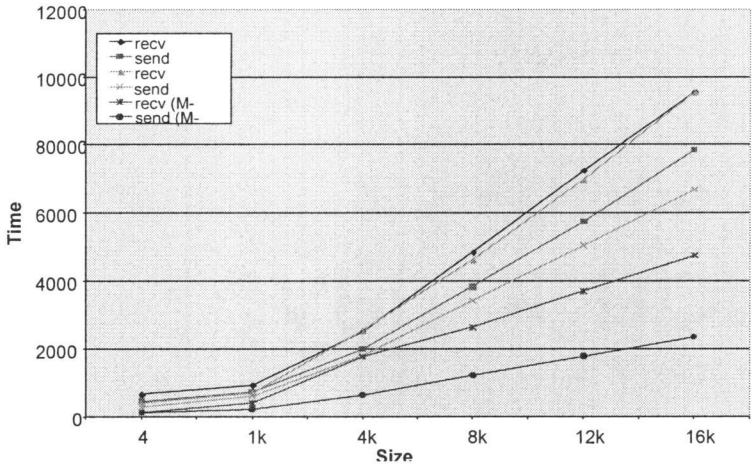


Figure 5: Message Send/Receive Latency

Table 2: Message 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
	Total	**1425**	**1427**	**2395**	**6504**	**12343**	**17882**	**19469**	**42668**
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
	Total	**1402**	**1937**	**4205**	**14452**	**26454**	**37020**	**47455**	**95395**
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
	Total	**3166**	**3669**	**6356**	**18136**	**34291**	**50498**	**66836**	**134276**
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
	Total	**4278**	**4238**	**7099**	**25633**	**45912**	**69708**	**95504**	**192219**
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
	Total	**4715**	**4712**	**7436**	**20104**	**38347**	**57426**	**78390**	**394691**
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
	Total	**6836**	**6699**	**7495**	**25378**	**48521**	**72404**	**95629**	**189046**

for TCP, or VipRecvWait for M-VIA). 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, TCP, and memory registration for M-VIA. These results also do not include the effects of MAC and physical layer operations.

The results in Figure 5 compare the latency of the network protocols. For message sizes less than the 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 the 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.

3.3.3 Layer-Level Performance

The send and receive latencies for UDP/IP, TCP/IP, and M-VIA were then subdivided 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 subdivided into layers associated with *APPL*, *UDP/TCP*, *IP*, *DEV*, and *DMA*. *APPL* includes the time required to perform socket operations **sendto** and **recvfrom** for UDP, and **send** and **recv** for TCP. *UDP/TCP* and *IP* are the times for executing UDP/TCP protocol layer and IP protocol layer, 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 subdivided into layers associated with *APPL*, *TRANS*, *DEV*, and *DMA*. *APPL* represents the time required to initiate VI provider library functions VipPostSend and 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 Figures 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 Figures 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 the MTU, IP increases significantly as a function of messages size. This is because IP handles packet fragmentation, data

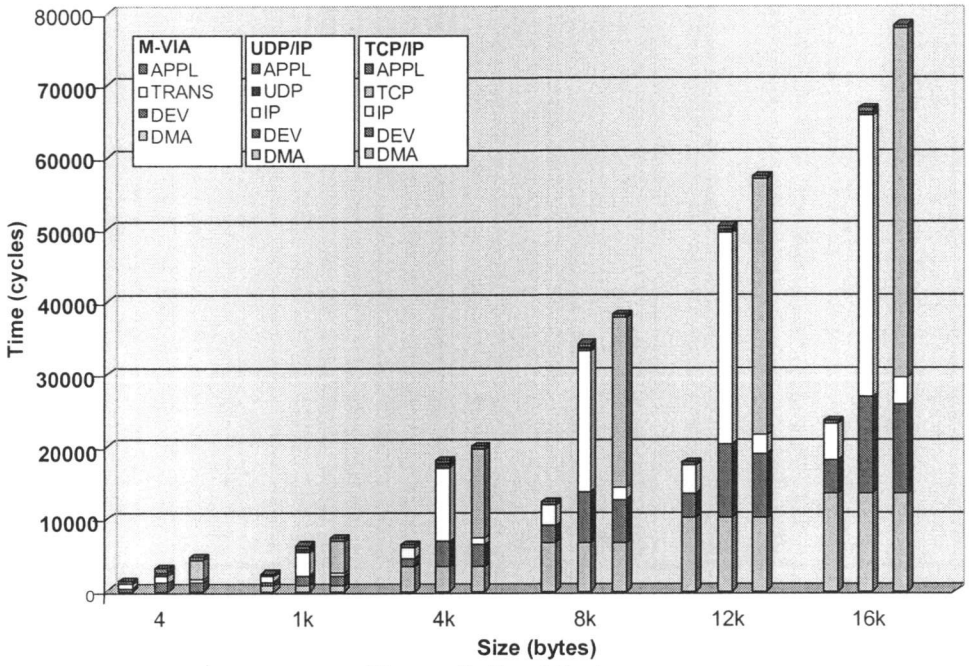


Figure 6: Send Latency.

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. Moreover, TCP dominates while IP represents only a small portion of the overall execution time. This is because 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. 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-fragmentation, while the increase in UDP is due to copying data from socket buffer to user space and check-summing. This is because in Linux, user data copying in UPD/IP occurs in two different layers for send and receive. In the case of a send, the IP layer handles both packet fragmentation and data copying from user space to socket buffer. On the other hand, for receive, packet de-fragmentation occurs at the IP layer while data copying from socket buffer to user space occurs at the UDP layer. For TCP/IP, *TCP* also represents the largest portion of the overall execution time. For M-VIA, *TRANS* has

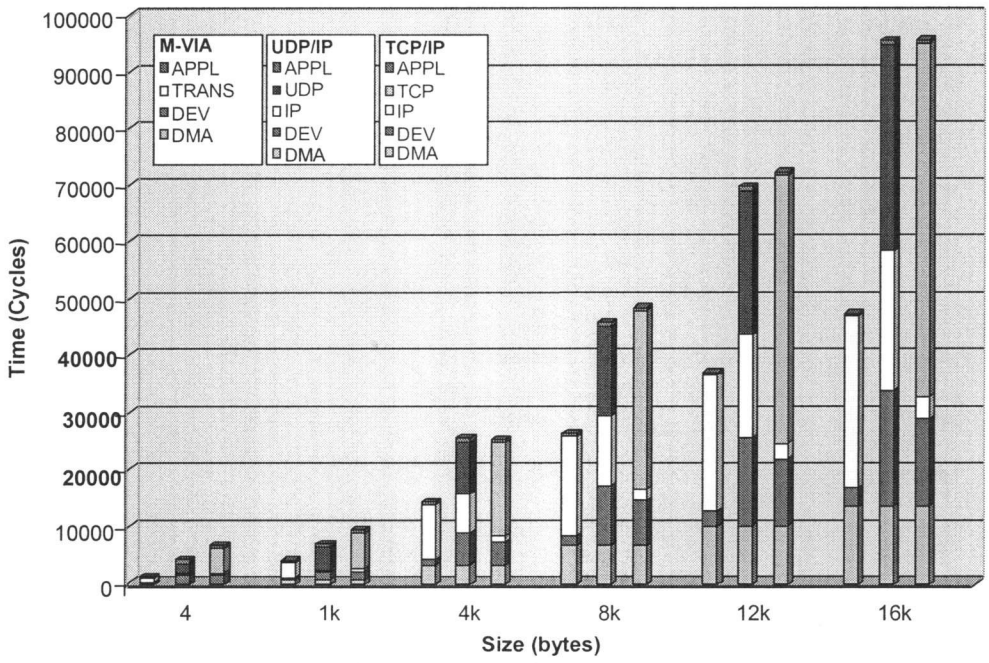


Figure 7: Receive Latency.

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 reason. Second, the legacy protocols have a layered structure, and thus UDP/IP 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.

3.3.4 Function-Level Performance

The pie charts shown in Figures 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_SOCK*. *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_SOCK* includes the time for

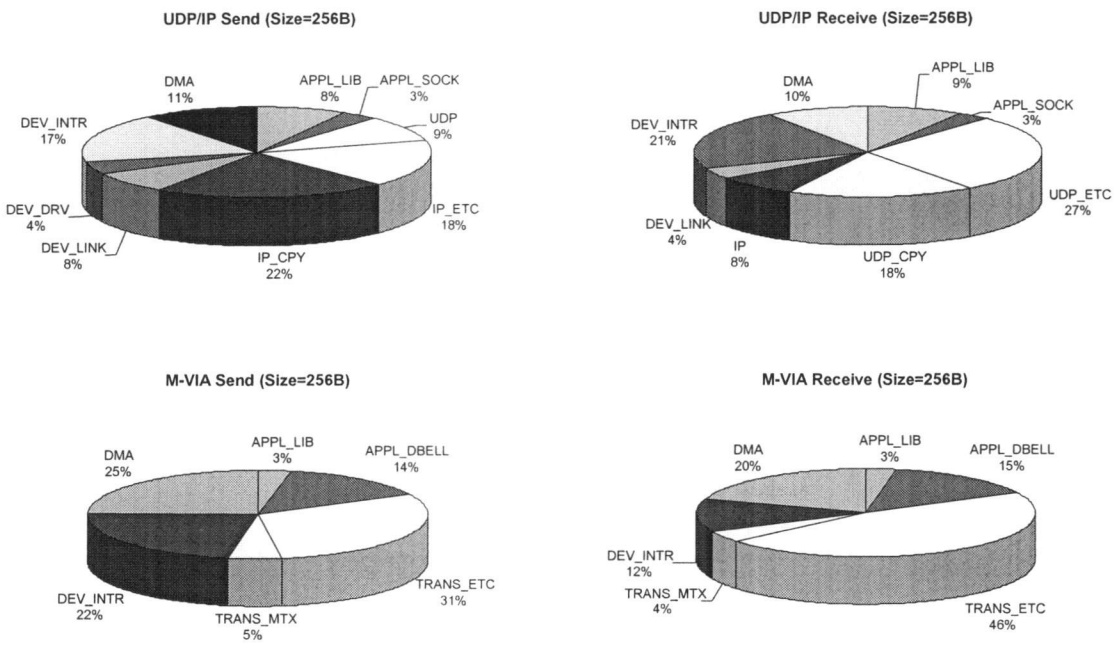


Figure 8: Latency breakdown for 256-byte message size.

socket layer operations. For UDP/IP send, *IP* was subdivided into *IP_CPY* and *IP_ETC*. *IP_CPY* 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_CPY* 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.

For the M-VIA layers were subdivided to examine the effects of doorbell mechanism, interrupt handling, and memory translation table lookup operations. *APPL* was further subdivided into *APPL_LIB* and *APPL_DBELL*. *APPL_LIB* represents the latency between a VIA library call (i.e., VipPostSend for send or VipRecvWait for receive) and start of 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

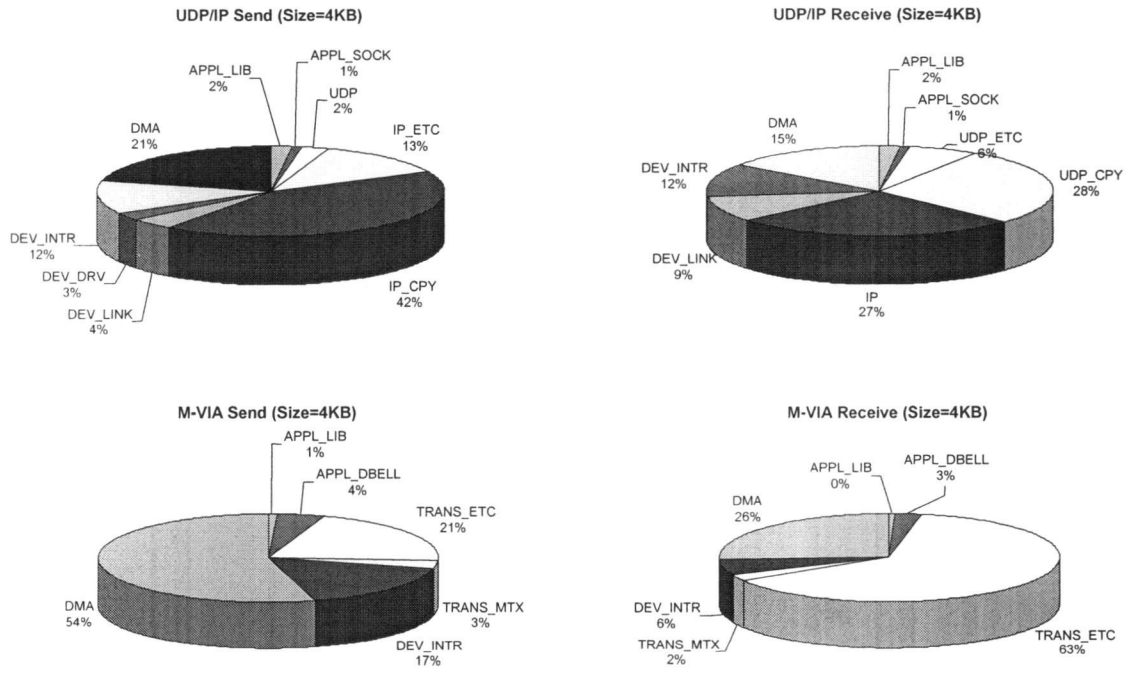


Figure 9: Latency breakdown for 4-Kbyte message size.

than a VIA-aware NIC. Therefore, the latency for indirect invocation of the device driver is included in *APPL_DBELL*. *TRANS* was 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 reduced significantly for large 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-ware NIC capable of performing DMA transfer directly from NIC buffer to user space would significantly reduce receive latencies.

3.4 Conclusion and Future Work

This chapter presented a detailed performance analysis of UDP/IP, TCP/IP, and M-VIA using Linux/SimOS.

Our analysis found that the legacy network protocols cause most of overhead in the network transport layer such as TCP and UDP. Compared with the legacy network protocols with much bigger size of protocol stack, the user-level network protocol shows that it has much smaller size of stack. Therefore, the advantage of using user-level network protocols is in low latency communication. This is due to two factors: no operating system calling overhead and thin network transport layer. Finally, the major finding from the detailed performance analysis is that the user-level network protocol can reduce its communication overhead by reducing its data transfer time in its protocol processing.

For the future study, there are numerous possible uses for Linux/SimOS. For example, one can study the characterization of new network protocols such as Scheduled Transfer Protocol (STP) [47]. STP is a new ANSI specified data transfer protocol. The new network protocol is designed for applications where high bandwidth or low latency is required. It has smaller protocol state information than the legacy protocols TCP, which makes STP suitable for offloading protocol processing on network interface (NI). Utilizing programmable NI for offloading protocol processing helps STP to achieve high bandwidth and reduce host processor utilization. The use of Linux/SimOS greatly helps the process of characterizing STP for offloading protocol and evaluating new NI architecture for better performance. Another possibility is storage networking protocols over IP networks such as Internet

SCSI (iSCSI) or Fibre Channel over IP (FCIP). The IP-based storage network protocols are used to facilitate data transfers over intranets and to manage storage over long distances. Because of the ubiquity of IP networks, the IP-based storage networking can be used to connect storages over LANs, WANs, or the Internet and can enable location-independent data storage and retrieval. Again, the use of Linux/SimOS captures the interaction among operating system and storage interface (i.e., SCSI commands) for performance assessment. And it also helps to achieve detailed characterization of emerging storage networking protocols.

4 Eager Data Transfer

4.1 Introduction

Due to the fast evolving network and processor technology, cluster computer systems have become the most cost-effective platform for parallel and distributed computing. Advances in technology are closing the performance gap between dedicated parallel computers and cluster computers. As the popularity of cluster computing grows, there is an increasing demand for low-latency network protocol and intelligent network interface hardware. Since the performance of parallel and distributed applications is greatly dependent on message passing facility, low-latency message processing becomes a main design issue for cluster network protocols and network interface (NI).

User-level network protocols such as *Virtual Interface Architecture* (VIA) [17] and *InfiniBand Architecture* (IBA) [21] significantly reduce user-to-user communication latency compared with traditional network protocols (e.g., TCP/UDP/IP). User-level protocols execute time-critical operations, such as message send and receive, without the kernel involvement, and implement zero-copy data transfer to avoid data copy overhead. Although the user-level protocols have been successful in lowering communication latency, the demand for even lower communication latency remains high. Therefore, this paper exploits an opportunity of further reducing communication latency using special network interface hardware.

Our prior study on the communication performance of VIA showed that the data transfer time between the host and NI constitutes the largest portion (about 45

%) of the overall communication latency [58]. Therefore, the reduction of user data transfer time significantly improves communication performance.

There are several alternatives for transferring data between the host and NI: Programmed IO (PIO), Direct Memory Access (DMA), Cache-Coherent DMA (CC-DMA), and Coherent Network Interface (CNI). *PIO* is a traditional way to access device registers on an NI residing on the I/O bus using uncached loads/stores. Uncached accesses transfer one to eight bytes at a time, which typically results in more bus transactions than using DMA [29, 8]. *DMA* moves data over the memory bus in block transfer mode, which efficiently utilizes the memory bus bandwidth. CC-DMA is an advanced form of DMA that does not require the cache to be explicitly flushed before the DMA operation. This is done by using a special logic to detect accesses to memory locations for which there are dirty cache lines and either allowing data to be accessed directly from cache or implicitly flushing cache lines before the data can be accessed from the main memory [8, 2]. *CNI* allows data transfer between NI devices registers and cache memory by relying on the underlying cache coherence protocol [32, 33]. It uses the bus bandwidth efficiently by transferring data in cache-block units and cache invalidation signal is used as an efficient event-notification mechanism.

Despite these existing data transfer mechanisms, there is an opportunity to further reduce the data transfer time between the host and NI, especially in the context of low-latency, user-level network protocols such as VIA and IBA. In order to understand the opportunity for improvement, consider a message send in VIA. The user first prepares a message in the host memory and notifies the NI of a message send request. The NI copies the user message from host memory to NI buffer and then

starts the network transport protocol to inject the message into the network. As can be seen by these steps, a large time gap exists between when the user message is prepared and when it is copied into the NI buffer. Therefore, performing the user message preparation and the copying of user message simultaneously can significantly reduce the overall latency.

This chapter describes a hardware-based speculative approach called *Eager Data Transfer* (EDT) to reduce the data transfer time. EDT employs cache-coherence interface hardware to efficiently transfer data between the host and NI. Since EDT relies on underlying cache-coherence mechanism, it is assumed that the EDT-based NI is located on host memory bus to observe bus transactions (i.e. cache coherence-related bus transfers). The two advantages of EDT are (1) efficient use of memory bus bandwidth since data transfers are done in cache block units, and (2) no software overhead since the data transfer process is controlled completely in hardware. In order to evaluate the effectiveness of EDT, an EDT-based NI was modeled and simulated on Linux/SimOS.

The rest of the chpater is organized as follows. Section 4.2 discusses the related work. Section 4.3 overviews VIA and its basic data transfer mechanism. Section 4.4 presents the proposed EDT mechanism. Section 4.5 describes a VIA implementation for EDT-based NI. Section 4.6 presents the simulation results. Finally, Section 4.7 discusses conclusion and future work.

4.2 Related Work

Banikazemi showed that PIO is faster than DMA for transferring small size data (16 bytes or less) [5]. Bhoedjang also presented a comparison of data-transfer performance between DMA and PIO [8]. Their results show that PIO using Pentium-Pro™ write combining buffers, which combines multiple write commands over the I/O bus into a single bus transaction, moves data faster than DMA for data size up to 1,024 bytes because of the DMA start-up cost. However, DMA is more efficient than PIO for large data because data transfers can be performed in bursts and without disturbing the processor.

Since DMA transfers data to and from host memory, both cache and memory must be coherent before a DMA transfer can start. For systems that do not support cache coherent DMA, software is used to explicitly flush the dirty lines from cache to memory using cache flush instructions, such as *cache* (MIPS) [31], *dcbf* (IBM PowerPC) [43], and *wbinvd* (Intel Pentium) [42]. For systems that support cache coherent DMA, there are two methods for maintaining coherency between cache and memory using hardware. The first option is to suspend the current DMA transaction until the cache sends the dirty cache line to the host memory [55]. This method, referred to as cache coherent DMA with retry, forces DMA bus request to be retried whenever the requested data is not in the host memory. When the dirty cache line is written back to the host memory, the DMA request is tried again and the data is read from the host

memory. The second option, called cache coherent DMA with intervention, is based on using a cache coherent bus [2]. The idea is to have the cache snoop the cache coherent bus and whenever there is a request for a dirty cache line, the cache supplies the requested data. The main advantage of the second one is that the requested data can be accessed by the DMA without having to wait until the cache line is flushed to the host memory.

The work closest to ours is CNI, which was proposed by *Mukherjee et al.* [29, 32, 33]. CNI allows a coherent, cacheable memory block implemented as a Cacheable Device Register (CDR) to be shared between the host processor and NI. CNI reduces the unnecessary bus bandwidth by transparently transferring data between the host processor and NI in cache blocks rather than words. Cacheable Queue is an extension of CDR to represent a contiguous region of memory blocks and is managed by a pair of head and tail pointers. For example, the host processor sends a data by simply writing it to the next free queue location and incrementing the tail pointer. The cache coherence protocol invalidates the copy of the tail pointer in the NI, which causes the NI to initiate a read request for the block. Mukherjee *et al.* [29] compared regular PIO (i.e., without write combining buffers) and CNI, and CNI showed better performance than PIO by 17-53%.

Compared to the aforementioned data transfer mechanisms, the proposed EDT has the following advantages. DMA has overhead for initiating data transfer such as writing a DMA descriptor and explicitly or implicitly flushing the cache. For PIO, the host processor cycles are used for the data transfer operation. Therefore, the host processor utilization is reduced. Thus, the processor cycles are wasted instead of

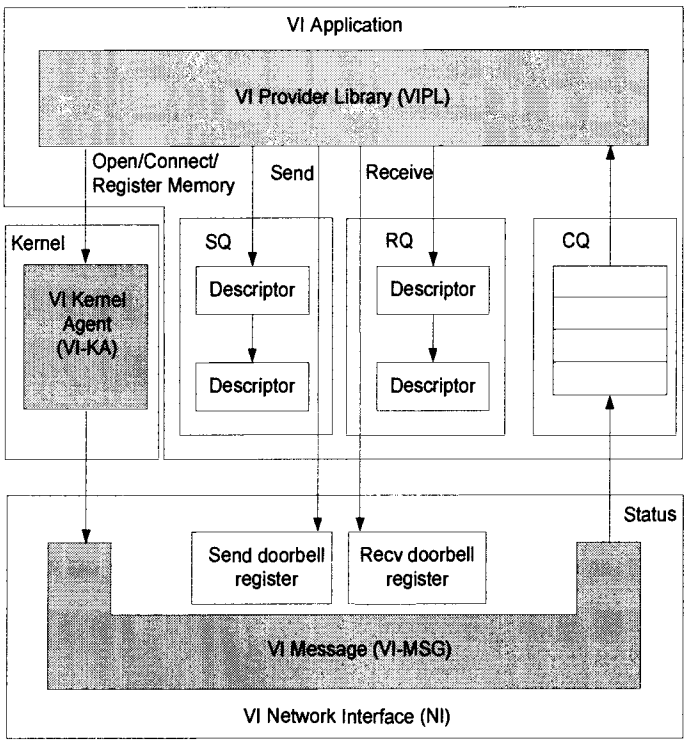


Figure 10: VI Architecture

being used for executing application programs. In CNI, the host processor gives explicit notification to NI by writing to head and tail pointers residing on cache lines. This requires the host processor to keep updating the cache variable to keep track on the progress of the data transfer, which introduces additional overhead. Moreover, CNIs, specifically cacheable queues, require the construction of arrays to be performed in strict sequential order thereby restricting the programming paradigm.

4.3 Overview of VI Architecture

Virtual Interface Architecture (VIA) is an industry standard developed by Compaq, Intel, and Microsoft [17, 56]. VIA was designed to provide low-latency, user-level communication over a System Area Network (SAN). Unlike the legacy network protocols of TCP/IP and UDP/IP, which were developed to operate in Wide Area Network or Internet, VIA is a light-weight protocol that avoids the kernel involvement for time-critical communication services, such as message send/receive. This section briefly describes an implementation of VIA and how data transfers are performed between the host and NI.

Figure 10 shows the basic components of VIA: Send Queues (SQ) and Receive Queues (RQ), Completion Queues (CQ), VI Network Interface (NI), VI Kernel Agent, VI Provider Library (VIPL), and VI Application [56]. A work queue pair SQ and RQ compose a Virtual Interface (VI), which is the communication end point that allows an application to submit message requests directly to the communication facility running on the NI hardware. A user application posts requests on the queues in the

form of descriptors. A descriptor is a data structure that contains all the information needed to process the user request. Each descriptor contains one control segment followed by an optional address segment and zero or more data segments. Each data segment contains the virtual address of the user buffer. The address segment contains the virtual address of the user buffer at the destination node.

Each VI is associated with send and receive doorbell registers. A descriptor posting is followed by writing a token to the doorbell register, which notifies the NI to process the descriptor. When a user request completes, the associated descriptor in the work queue is updated with a status value, and a notification is inserted into the Completion Queue (CQ). Applications can check the completion status of their message request via the descriptor or CQ. A CQ merges the completion status of multiple work queues.

The VIA specification requires that a user application registers the virtual memory regions that are used to hold VI descriptors, user communication buffers, and CQs. The purpose of the memory registration is to have the VIPL pin down the user's virtual memory in physical memory so that the NI can directly access the user buffers. This eliminates the need for copying of data between user buffers and intermediate kernel buffers typically required in traditional network protocols. VIA specifies two types of data communications: the send/receive messaging model and remote direct memory access (RDMA) model. However, this study is focused only on the send/receive messaging model.

The detailed operations for VIA message send and receive are also shown in Figure 11, where it is assumed that a DMA engine is used to transfer user data between

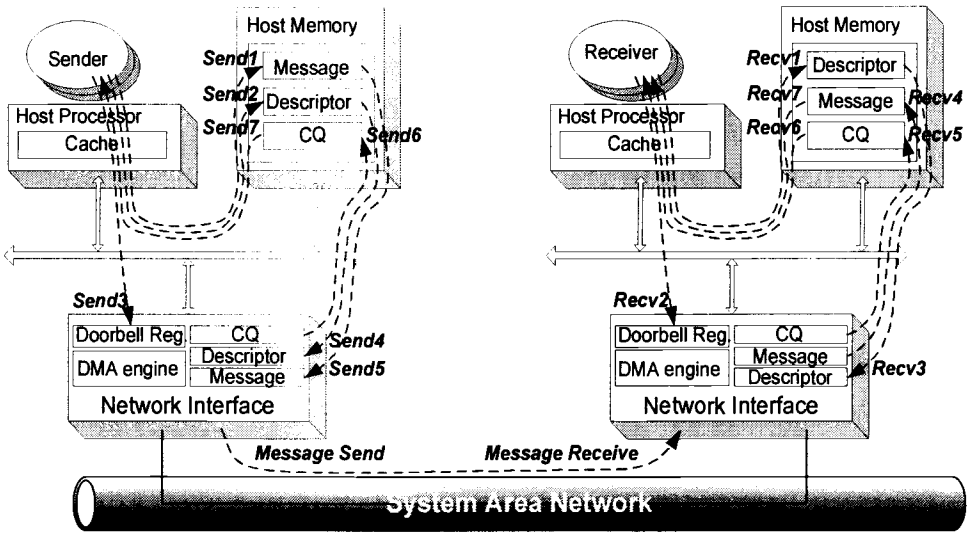
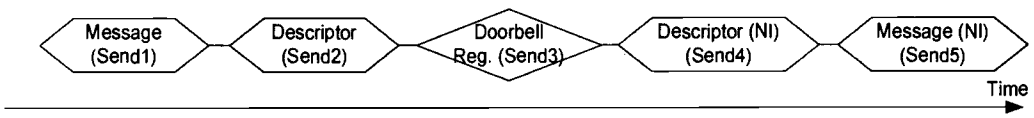


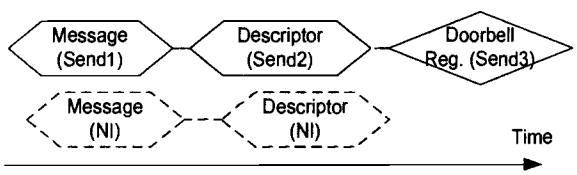
Figure 11: Message Send and Receive Operations

the host memory regions and NI. In order to initiate a message send, the sender builds a message (Send1) and a descriptor (Send2) in the registered memory regions. The descriptor includes the address of the message buffer, message size, type of operation, and a status field. Then, a token is written to the send doorbell register to notify the NI of a message send operation (Send3). The doorbell token includes the address of the descriptor, which in turn holds the address of the user message. Since the address of the user message is contained in the descriptor, the NI executes a DMA transfer for the descriptor (Send4), followed by another DMA transfer for the message (Send5). Finally, after the message is sent out to the network (Message Send), the completion status is set in the CQ (Send6).

Message receive is performed in two separate sequences: posting a descriptor and receiving a message. A descriptor posting follows the same steps as in the send case. The receiver builds a descriptor (Recv1) and writes a token (Recv2) to the receive doorbell register, which is followed by DMA transfer of the descriptor (Recv3). The remaining steps are executed when a message arrives (Message Receive). The NI moves the message into the registered memory region, which is pointed to by the address held in the descriptor (Recv4), and sets the completion status in the CQ (Recv5). The receiver polls the CQ to detect a new message arrival (Recv6) and reads the message from the registered memory region (Recv7).



(a) Message send operation using DMA



(b) Message send operation using EDT

Figure 12: The EDT vs. DMA-based Data Transfer

4.4 Eager Data Transfer (EDT)

Figure 12a shows the timing of the DMA-based approach for a message send operation. As can be seen in the figure, there is a time interval between when the user process writes data to the message buffer (Send1) and when data is DMA from the message buffer to NI buffer (Send5). Since this time accounts for a significant portion of the overall latency, the primary motivation of EDT is to overlap the execution of the user data writes (Send1) and the DMA transfer of data to NI buffer (Send5) to reduce the overall latency for message send/receive. Figure 12b shows the advantage of the EDT mechanism, where data transfers are performed in cache-line units as the user data is generated by the application. Therefore, as the application builds the message in the user buffer, the entire message is copied into the NI buffer. Thus, right after a token is written into the doorbell register (Send3), the NI can immediately proceed with sending of the message from the NI buffer.

To support the eager data transfer, the EDT-based NI employs a simple cache coherence hardware, which includes a set of tags to hold memory addresses for registered memory regions. The tags are used for monitoring the host memory bus to detect host processor's memory accesses to the registered regions. The bus traffic monitoring and the associated cache coherence control are implemented by observing the subset of the underlying cache coherence protocol. The tags are also used to associate a registered host memory region with a NI memory region. In addition, the NI buffer uses status bit per each cache block to indicate the modification of the data. Using the memory association information, memory updates on a host memory region

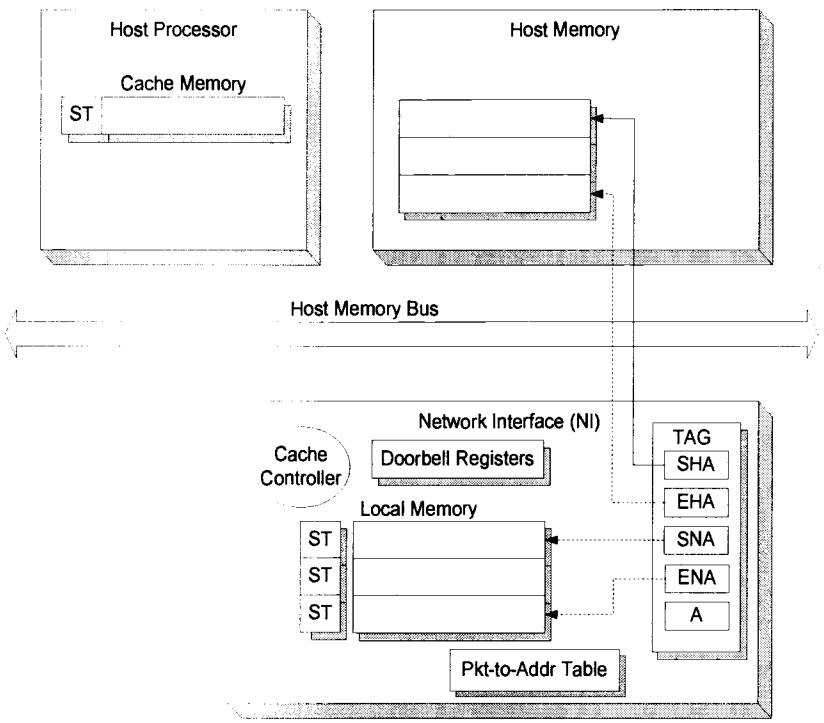


Figure 13: EDT Architecture

are reflected on the associated NI buffer. The proposed EDT mechanism does not depend on a specific cache coherence protocol. However, the MESI (*Modified, Exclusive, Shared,* and *Invalid*) cache coherence protocol is assumed for the explanation [23].

Figure 13 shows the hardware architecture of the EDT-based NI, which includes doorbell registers, tags, local memory, Packet-to-Address table, and cache controller. During the memory registration of a host memory region, the corresponding NI buffer and tag entry are allocated. The allocated NI buffer is the same size as the message buffer in the host memory and is subdivided into memory blocks, where each block is equal to the cache line size. Each memory block has a Status bit (ST) indicating the state of each cache block in the local memory. Each tag entry consists of five fields: Start and End Host Addresses (SHA and EHA), Start and End NI Addresses (SNA and ENA), and Access bit (A). The host address pair, SHA and EHA, points to a registered memory region in the host memory. The NI address pair, SNA and ENA, points to the associated buffer in the NI local memory. A-bit is used to represent the validity of the tag. The Packet-to-Address Table (Pkt-to-Addr Table) is used during message receive operations to map incoming packets to the local memory.

The cache coherence protocol for EDT consists *Shared* (S) and *Invalid* (I) states and they share similar meaning with their MESI protocol counterparts. Once the Tag fields are properly set, the cache lines for the registered memory regions need to be set to either *Shared* (S) or *Invalid* (I) state. This step is necessary so that the EDT-based NI can monitor the host processor writes to the registered memory region. There are two

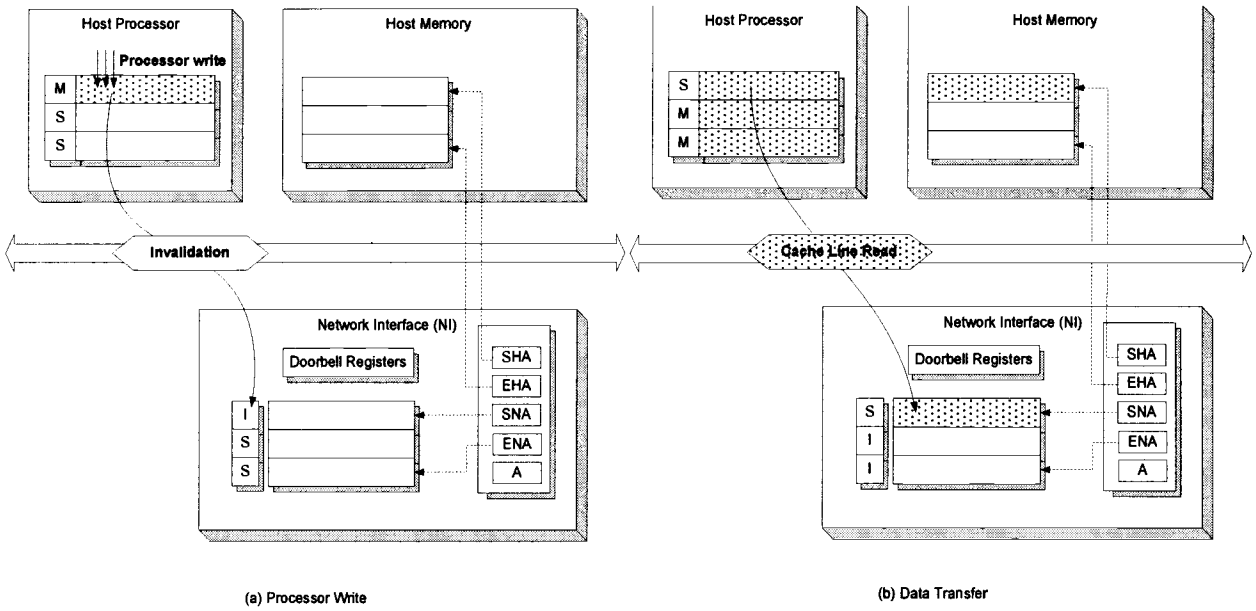


Figure 14: EDT for Message Send.

ways to accomplish this. First method is to invalidate the cache lines by flushing the cache. The second method is to have the EDT-based NI initiate read bus transactions for the register memory region. This is done by an initialization routine implemented in VI-MSG (Figure 10). The data is copied either from the host cache or from the host memory. If data is copied from the host cache, both the NI buffer blocks and the host cache lines transition to the *Share* (S) state. However, if the data is copied from the host memory, the data is held only in the NI buffer and the memory blocks transition to the *Share* (S) state (Note that unlike the MESI protocol, EDT does not need to differentiate between E and S states. Thus, for simplicity S state is used instead of E state.). In either case, the initialization overhead is a one-time cost incurred during memory registration and does not affect the communication latency send/receive operations.

Figure 14 shows the detailed operations for message send assuming both memory blocks of the NI buffer and the host cache lines in the *Shared* (S) state (similar steps would be performed if initialized to *Invalid* (I) state). When the host processor writes to the registered memory region (Send1 of Figure 2), there can be a cache hit or miss. If a processor write hits on the cache, an invalidation bus transaction is generated to gain exclusive ownership of the cache line. On the other hand, if a processor write misses on the cache, the cache line is first loaded from the host memory and then an invalidation bus transaction is generated. In either case, the cache line and the memory block in the NI buffer transition to the *Modified* (M) and *Invalid* (I) states, respectively. Once the EDT-based NI detects the invalidation bus transaction, the corresponding block in the NI buffer changes its state from S to I. At

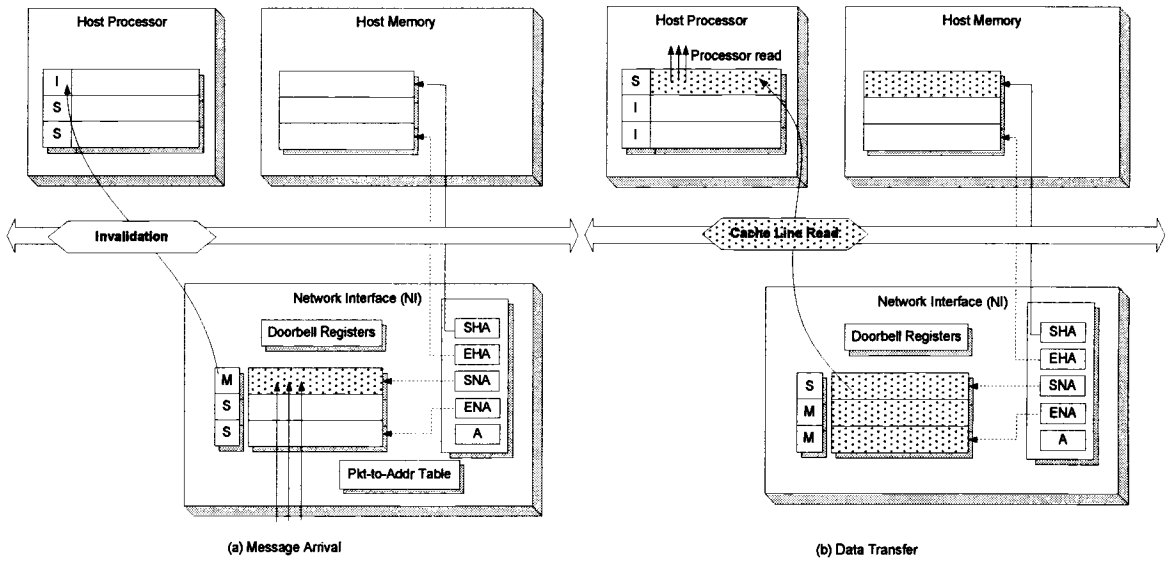


Figure 15: EDT for Message Receive

this point, the EDT-based NI places a bus read request to read in the cache block modified by the host processor. This causes the requested block to be transferred to the NI local memory, and both the cache blocks in the host processor and the local memory transition to the *Shared* (S) state. As the host processor continues to write to the registered memory region, memory blocks in the NI Local memory are invalidated and the updated cache lines are read into the NI buffer.

For the message receive case, it is important to note that the host processor read operations do not begin until the completion status is set in the CQ. Therefore, the data transfer phase cannot be overlapped with the host processor reads, as was the case for message send. However, the EDT mechanism can still avoid the use of DMA and thus eliminate DMA startup and interrupt processing overhead resulting in a performance gain. There are two possible EDT implementations for message receive. The first option is to rely on the cache coherence mechanism of EDT, which results in symmetrical behavior for message send and receive. The second method is to bypass the cache coherence and operate the EDT mechanism as a DMA engine. These two methods are described below.

Figure 15 shows the operations performed by EDT for message receive using cache coherence. This method requires an additional state, *Modified* (M), to indicate a message has been received from the network. As soon as a message is received, the network-side DMA (not shown in Figure 15) transfers it to the NI buffer. The Packet-to-Address Table is searched to determine the NI buffer address for the message. As the message is moved into the NI buffer, invalidation transactions are generated to invalidate the cache lines in the host cache (Figure 15a). The completion of the

invalidation transactions causes the NI buffer blocks to transition from the S state (initial state) to the M state. Once the message is moved to the NI buffer, VIA-MSG starts its processing and notifies the receiver process that a new message has arrived. After the notification, the user process attempts to read the message by placing read requests on the bus. This cause the blocks to be copied to the host processor's cache and both blocks in the host (I to S) and NI (M to S) transition to the S state. Figure 15 shows the operations during message receive when EDT operates as a DMA engine. Since cache coherence is not observed, simple *Valid*(V)/*Invalid*(I) states are used to indicate the status of the memory blocks. When a message is received, it is moved to the NI buffer and the memory blocks are set to Valid (V) state indicating they contain a new message. For each memory block set to the V state, the NI cache controller issues a bus write-back request to flush the memory block to the host memory.

The main difference between the two methods is that when the host processor is ready to read the message the first method reads it from the NI buffer while the second method reads it from the host memory. Therefore, both methods result in similar performance but the latter methods less complex since it does not rely on cache coherence.

Compared with the other approaches, the EDT uses bigger NI memory as the number of registered memory regions increases. During a memory registration, the EDT allocates a buffer in its local memory that is the same size as the host memory region. Therefore, the number of memory registrations is bounded by the size of the NI memory. The EDT was proposed for applications requiring low-latency communications, such as parallel applications. These applications use small message

sizes typically much smaller than 4 Kbytes [29]. For example, 4 Mbytes of NI memory can accommodate over a thousand of registrations with a 4 Kbytes buffer. Considering today's memory technology, hundreds Mbytes memory is affordable without too much cost. Practically there is no limit in providing a large NI memory for low-latency application programs. In case memory registrations cannot proceed because there is not enough free space in the NI memory, EDT-based NI can fall back to DMA-based transfer. Alternatively, the memory registration can be postponed until a free space is available.

4.5 VIA implementation for EDT-based NI

This section describes a VIA implementation for the EDT–based NI, specially focusing on VIA functions for sending and receiving messages. Our implementation, called SONIC-VIA, is based on M-VIA [36] and simulated on Linux/SimOS. The protocol layers of SONIC-VIA consist of VI Provider Library (VIPL), VI Kernel Agent (VI-KA), and VI Message (VI-MSG). The protocol layers are distributed over the host and NI; i.e., VIPL is compiled into the application, VI-KA is executed as a kernel module, and VI-MSG is executed on the NI as shown in Figure 10.

VIPL provides VI library functions for user applications. The VI library layer sends users requests to either VI-KA or VI-MSG. Since user requests are sent to two different layers, VIPL uses two different service callings; doorbell register and IOCTL system call. Doorbell registers are used to notify message send (VipPostSend) and receive (VipPostRecv) requests to VI-MSG running on the EDT-based NI. IOCTL

system calls are handled by VI-KA, which services calls to VI primitives other than message send and receive, such as VipOpenNI, VipConnectRequest, and VipRegisterMem.

The VI-MSG layer includes routines for message send and receive. When an application program writes a token to a doorbell register, VIA-MSG reads the token to determine the user (virtual) address for the descriptor. Then, the user address is mapped to an NI memory address through a two-level translation scheme: User (virtual) address to host memory (physical) address and then the host memory address to NI memory address. The second memory translation is performed by looking up tag entries (Figure 13). The descriptor is accessed from the NI memory address. As the descriptor holds the virtual address of the user data, VIA-MSG again goes through address translation and local memory access for user data. Thus, VI-MSG is responsible for maintaining address translation table, performing address translation, and accessing descriptor and data from the NI memory. The rest of VI-MSG operations involve processing network packet frames, performing fragmentation/de-fragmentation for user data whose size is over the MTU (Maximum Transfer Unit), and updating the status flag in the VI or CQ.

Table 2: System Parameter

System Parameter		
Host Processor	Processor Speed:	1 GHz
	L1 Cache Size/Line Size/Assoc/Latency:	32KB/32B/2-way/1 cycle
	L2 Cache Size/Line Size/ Assoc/Latency:	1MB/ 128B/2-way/10 cycles
	Cache Coherence:	MESI, write-invalidation
Host Memory	Latency:	100 ns
	Width:	16 B
System Bus	Speed:	100 MHz
	Burst-mode BW:	1600 MB/sec
NI	NI Processor Speed:	100 MHz
	Local Memory Latency:	10 ns
	Host-side DMA (data transfer speed):	1600MB/sec

4.6 Performance Evaluation

4.6.1 Simulation Environment

The system simulation environment, Linux/SimOS was used for the evaluation of the VIA implementation for the EDT-based NI. To set up a full system evaluation environment, a VIA network protocol and benchmark programs were written and executed on top of the Linux/SimOS, which provides a real program execution environment to perform detailed system evaluation in a non-intrusive manner. Therefore, it is capable of capturing all aspects communication performance that includes the effects of application, network protocol, and network interface. The system configuration and parameters are summarized in Table 3.

The host processor model includes L1 and L2 caches and follows the MESI cache coherence protocol to maintain data consistency between cache and memory. Host memory latency was chosen based on the current DRAM speed and bus access time. The NI processor does not include private caches. This assumption was made based on the fact that most commercial NI processors, such as the Myrinet LANai processor, do not have caches [9]. The EDT-based NI has a data transfer speed comparable to the burst-mode memory bandwidth because it is connected to the host memory bus. Since our simulation study focuses only on network protocol processing within a node, our simulation results are based on a no-delay network model.

For performance evaluation, the EDT mechanism was compared against CC-DMA with retry ($CC\text{-}DMA_{retry}$) and CC-DMA with intervention ($CC\text{-}DMA_{intv}$). As

mentioned early, the difference between CC-DMA$_{retry}$ and CC-DMA$_{intv}$ is that the later scheme can read directly from the cache memory, and thus eliminates the additional bus traffic required to flush the cache lines to the host memory. A micro-benchmark was used to send and receive messages between two users on different hosts. A sender sends a fixed-size message to the receiver and then waits for a message arrive from the receiver. When the receiver receives a message, it sends a new message back to the original sender. Messages are sent back and forth between sender and receiver for a number of times. To show the impact of data transfer mechanism, message send/receive time was measured as a function of message size.

4.6.2 Simulation Results

The total execution times for message communication between two user applications are presented in Figure 16. These simulations were run with a fixed MTU size of 1,500 bytes. The execution time represents the number of host processor cycles between the library call VipPostSend from the sender and the return of the library call VipRecvWait by the receiver. These results do not include the effects of MAC, physical layer operations, and network transfer time. The message size was varied from 64 bytes to 4,096 bytes, which represent a typical range of message sizes for a SAN environment. Figure 16 shows that the proposed EDT mechanism results in much better performance compared to CC-DMA$_{retry}$ and CC-DMA$_{intv}$. The EDT-based NI attains 17% to 41% reduction in the user-to-user messaging latency

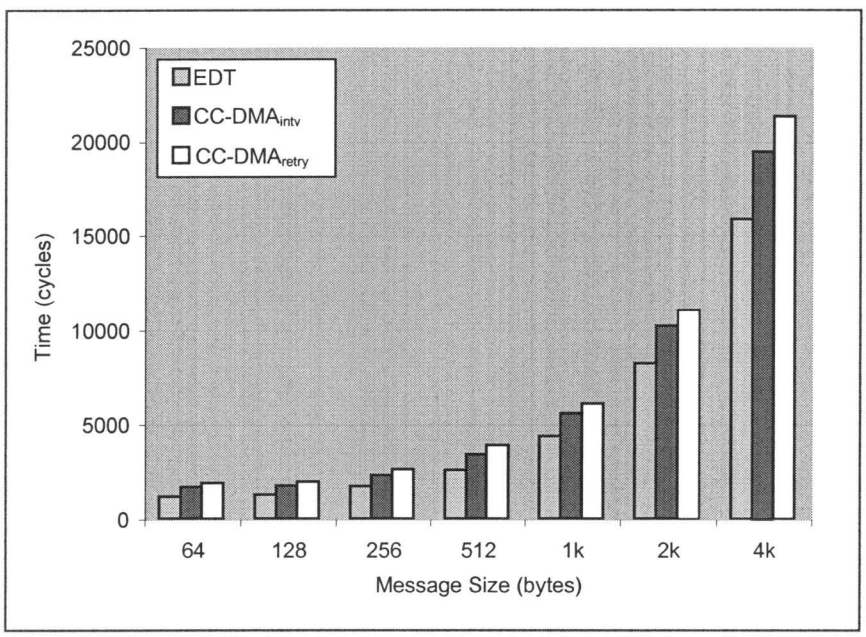


Figure 16: Total execution time

compared to CC-DMA$_{intv}$. More importantly, the performance improvement becomes more significant as message size increases.

In order to gain a better understanding of the performance improvement, Figure 17 shows the total execution times subdivided into three most significant operations in user-to-user communication: *User Data*, *Data Transfer*, and *Transport*. For each message size, there are three bar graphs representing the execution times (in cycles) of EDT (left), CC-DMA$_{intv}$ (middle), and CC-DMA$_{retry}$ (right). *User Data* is the time required for the user program to write a message to the user buffer (Send1 in Figure 11). *Data Transfer* includes the time to transfer data between host and NI. For CC-DMA$_{retry}$ and CC-DMA$_{intv}$, this includes the time for DMA setup, DMA operations, and handling interrupts after DMA operations complete. For EDT, this includes the time to perform bus write-back to flush the memory block to the host memory during message receive. *Transport* represents the time required to run the network protocol (VIA-MSG) to service the user send/receive requests.

The breakdown view of the total execution time in Figure 17 clearly shows how significantly each portion affects the overall latency and increases as data size grows. In particular, the amount of time spent on the *Data Transfer* portion depends on the underlying data transfer mechanism. The *Data Transfer* portions for CC-DMA$_{intv}$ are smaller than the ones for CC-DMA$_{retry}$. This is because the CC-DMA$_{intv}$ mechanism supports cache-to-cache transfer so that the user data and descriptor can be moved directly from the cache memory on host processor. In contrast, the CC-DMA$_{retry}$ mechanism requires two steps to move the user data from the cache memory:

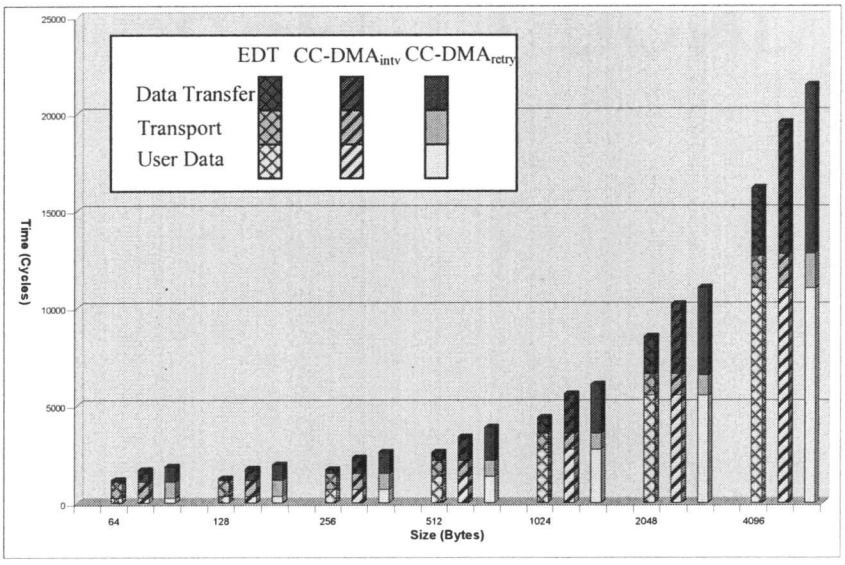


Figure 17: Breakdown of the total execution time

the cache memory: The user data in the cache memory has to be first flushed to the host memory and then moved to the NI buffer. The *Data Transfer* portion for EDT is the smallest because there are no DMA operations and the only cost is the bust write back operation for the receive side. The *User Data* portion increases with the message size, but are the same for all the data transfer mechanisms. As the message size increases beyond the MTU size, the *Transport* sections start to grow. This is due to the fact that the network protocol performs fragmentation and de-fragmentation. Again, the *Transport* sections do not vary with the underlying data transfer mechanism because all three methods were implemented on a common platform, i.e., SONIC-VIA.

The pie charts shown in Figure 18 give a more detail breakdown of the communication latency for message size of 256 bytes. *Transport* is further subdivided into *Send* and *Receive* portions. *Data Transfer* for CC-DMA$_{retry}$ and CC-DMA$_{intv}$ is subdivided into DMA initiation (*DMA Init*), execution (*DMA Exec*), and interrupt processing (*DMA Intr*). *DMA Init* is the time to set up the DMA engine with address and length. *DMA Exec* is the time for the DMA engine to move user data and descriptor between the host and NI buffers. *DMA Intr* is the time taken to process the interrupt signaling at the end of a DMA operation. Among the various portions, only the *DMA Exec* portion increases with increased message size, and the rest of the portions remain constant. As explained earlier, *DMA Exec* is bigger for CC-DMA$_{retry}$ than CC-DMA$_{intv}$ because the user data is directly transferred through host cache memory for CC-DMA$_{intv}$. These results clearly show that the EDT approach significantly reduce the communication latency virtually eliminating DMA operations.

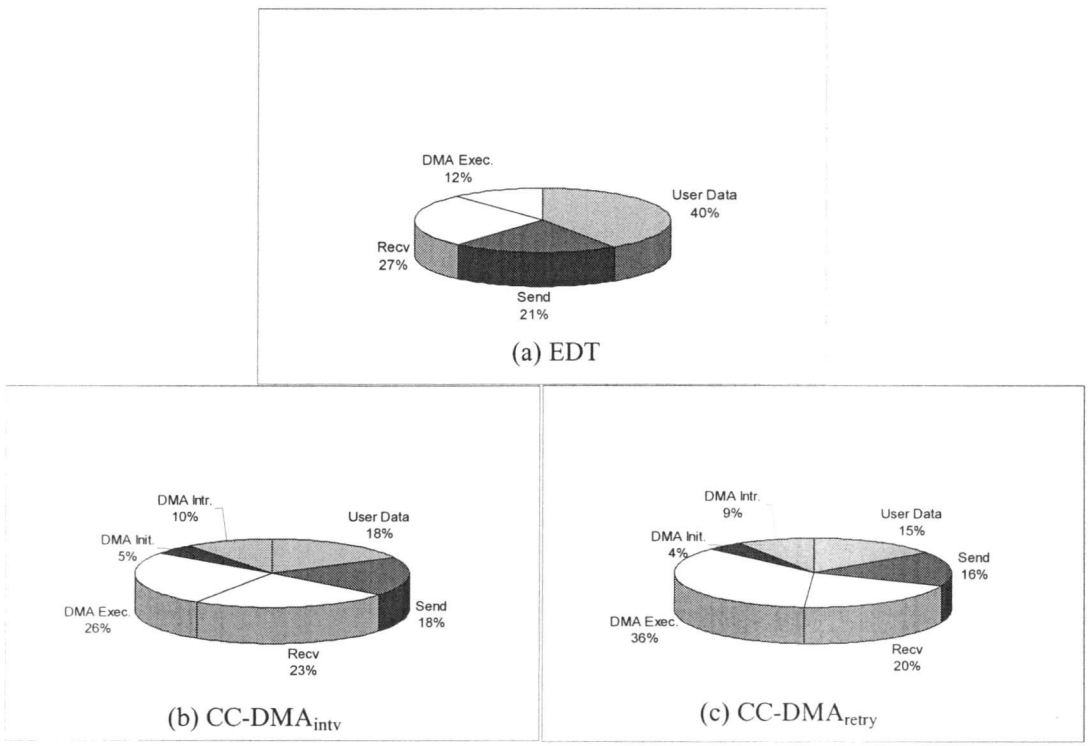


Figure 18: Detailed breakdown of total execution time

4.7 Conclusion and Future Work•

EDT mechanism was proposed to reduce communication latency for user-level network protocols. The mechanism reduces the time for data transfer between host memory and NI by overlapping writes to the user buffer with the actual transfer of user data from user buffer to NI buffer. Our detailed simulation study using Linux/SimOS showed that EDT reduces the message latency by 9% to 43% compared with the C-DMA based NI.

There are a number of ways EDT can be extended. First, a challenge is to extend EDT mechanism to the legacy network protocols such as TCP, IP, and UDP. Even though the user-level protocols have become imperative for SANs, reducing the latency of the legacy protocols will continue to be important. Another interesting application of the EDT is on embedded NI for System-On-Chip system (SOC). Because the EDT observes only a subset of cache coherence protocol, the design can be simplified for SOC applications. This will allow EDT-based embedded NI to achieve low-latency communication in less complexity and simple design.

5 Conclusions

This dissertation investigated the several aspects of attaining low latency communication using user-level network protocols. During the course of the dissertation research, three main research areas were identified and related research was made. The contributions made by this dissertation are summarized as follows.

- Communication system evaluation and design tool.

Linux/SimOS was developed as an efficient communication system evaluation and design tool [60, 40]. By combining the advantage of the open source simulator and Linux operating system, it provides the most comprehensive full-system evaluation capability, which allows system evaluation at different levels including hardware, operating system, and applications. It also serves as a system design tool because target hardware models and software can be developed easily on the tool.

- Detailed performance analysis

Detailed performance analysis of most representative network protocols was performed using *Linux/SimOS* [58]. The network protocols include TCP/IP, UDP/IP and M-VIA. Due to the full system performance profiling capability of *Linux/SimOS,* all aspects of network protocol processing were captured including application layer, operating system, network protocol, device driver, and network interface hardware.

- Low latency communication support

The detailed performance study with M-VIA user-level network protocol showed that the data transfer time between the host processor and NI constitutes a significant portion of the overall communication latency. *Eager Data Transfer* (*EDT*) mechanism was proposed for the reduction of data transfer time [59]. Through simulation, it was proved that *EDT* is the most effective mechanism for removing the data transfer overhead from the fast path of network protocol processing. As a result, it significantly improves the overall communication performance.

BIBLIOGRAPHY

[1] Alteon Networks, Inc., "Gigabit Ethernet/PCI Network Interface Card Host/NIC Software Interface Definition, Revision 12.3.11," June 1999.

[2] The Alchemy Au1100™ From AMD Internet Edge Processor Data Book, Available from http://www.sagitron.es/data_sheet/au1100.pdf.

[3] M. Baker *et al.*, "Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network," *Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta, pages 353-362, October 2000.

[4] M. Banikaze, B. Abali, and D. K. Panda, "Comparison and Evaluation of Design Choices for Implementing the Virtual Interface Architecture (VIA)," *Proceedings of Fourth International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing* (CANPC '00), January 2000.

[5] M. Banikazemi *et al.*, "Design Alternatives for Virtual Interface Architecture (VIA) and an Implementation on IBM Netfinity NT Clusters," *Proceedings of the International Parallel and Distributed Processing Symposium*, May 2000.

[6] M. Beck, *et al.*, *LINUX Kernel Internals, 2nd Edition*, Addison-Wesley, 1997.

[7] A. Begel *et al.*, "An Analysis of VI Architecture Primitives in Support of Parallel Distributed Communication," *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 1, pages 55-76, 2002.

[8] R.A.F. Bhoedjang, T. Ruhl, and H.E. Bal, "Design Issues for User-Level Network Interface Protocols on Myrinet," *IEEE Computer*, Vol. 31, No. 11, pages 53-60, November 1998.

[9] N.J. Boden *et al.*, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, Vol. 15, No. 1, pages 29-36, February 1995.

[10] P. Buonadonna, A. Geweke, and D.E. Culler, "An Implementation and Analysis of the Virtual Interface Architecture," *Proceedings of the SC98*, Nov. 1998.

[11] D. Burger *et al.*, "The SimpleScalar Tool Set, Version 2.0," U. Wisc. CS Dept. TR1342, June 1997.

[12] R. Buyya, *High-Performance Cluster Computer Architecture*, Prentice Hall, pages 3-8, 1999.

[13] R. Buyya, *High-Performance Cluster Computer Architecture*, Prentice Hall, pages 44-45, 1999.

[14] B. Chun *et al.*, "Virtual Network Transport Protocols for Myrinet," *Proceedings of Hot Interconnects'97*, April 1997.

[15] D.E. Culler and J.P. Singh, *Parallel Computer Architecture: A Hardware and Software Approach*, Morgan Kaufmann, pages 12-23, 1999.

[16] R. Cypher, *et al.*, "Architectural Requirements of Parallel Scientific Applications with Explicit Communication," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2-13, 1993.

[17] D. Dunning, *et al.*, "The Virtual Interface Architecture," *IEEE Micro*, March-April, 1998.

[18] S. Harrod, "Using Complete Machine Simulation to Understand Computer System Behavior," Ph.D. Thesis, Stanford University, February 1998.

[19] H. Hellwagner, "Exploring the Performance of VI Architecture Communication Features in the Giganet Cluster LAN," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000)*, 2000.

[20] IBM Corporation, "IBM InfiniBlue solutions," Available at http://www-3.ibm.com/chips/products/InfiniBand.

[21] Infiniband™ Architecture Specification Volume 1, Release 1.0.a. Available at http://www.infinibandta.org.

[22] Intel Compaq and Microsoft Corporations, "Virtual Interface Architecture Specification, Version 1.0," Available at http://www.viarch.org.

[23] Intel Corporation, "Pentium(R) Processor Family Developer's Manual," Available at http://developer.intel.com/design/ intarch/manuals/241428.htm.

[24] Jonathan Kay and Joesph Pasquale. "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *Proceedings of SIGCOMM93*, pages 259-268, 1993.

[25] K. Kant, V. Tewari, and R. Iyer, "GEIST – A Generator for E-commerce and Internet Server Traffic," *2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, October 2001.

[26] Kimberly A. Keeton, Thomas E. Anderson, and David A. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. *Proceedings of Hot Interconnects III,* 1995.

[27] Libnet, Packet Assembly System.
Available at http://www.packetfactory.net/libnet.

[28] P. S. Magnusson *et al.*, "Simics: A Full System Simulation Platform," *IEEE Computer*, Vol. 35, No. 2, pages 50-58, February 2002.

[29] S. Mukherjee *et al.*, "The impact of Data Transfer and Buffering Alternatives on Network Interface Design," *Proceedings of the fourth International Symposium on High-Performance Computer Architecture (HPCA)*, February. 1998.

[30] S. Makineni and R. Iyer, "Architectural Characterization of TCP/IP Packet Processingh on the Pentium® M Microprocessor," *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2004.

[31] MIPS R10000 Microprocessor User's Manual, Version 2.0,
Available from
http://techpubs.sgi.com/library/tpl/cgibin/getdoc.cgi/hdwr/bks/SGI_Developer/ books/R10K_UM/sgi_html/t5.Ver.2.0.book_396.html .

[32] S.S. Mukherjee and M.D. Hill, "Making Network Interfaces Less Peripheral," *IEEE Computer*, Vol. 31, No. 10, pages 70-76, October 1998.

[33] S.S. Mukherjee *et al.*, "Coherent network Interfaces for Fine-Grain Communication," *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996.

[34] M-VIA: Virtual Interface Architecture for Linux,
Available at http://www.nersc.gov/research/FTG/via.

[35] Erich Naum *et al.*, "Cache behavior of network protocols," *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1997.

[36] NERSC, "M-VIA: A High Performance Modular VIA for Linux,"
Available at http://www.nersc.gov/research/FTG/via.

[37] V. S. Pai *et al.*, "RSIM Reference Manual, Version 1.0," ECE TR 9705, Rice University, 1997.

[38] S. Pakin *et al.*, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proceedings of Supercomputing '95*, December 1995.

[39] D. K. Panda *et al.*, "Simulation of Modern Parallel Systems: A CSIM-Based Approach," *Proceedings of the 1997 Winter Simulation Conference*, 1997.

[40] V. Pattery *et al.*, "Analysis of the effectiveness of Multithreading for Interrupts on Communication Processors," *Proceedings of IASTED International Conference on Communications and Computer Networks (CCN 2002)*, November 2002.

[41] C. Patridge, *Gigabit Networking*, Addison Wesley, pages129-173, 1994.

[42] Pentium Processor Family Developer's Manual.
Available from http://developer.intel.com/design/pentium/manuals.

[43] The PowerPC Architecture: A Specification for a New Family of RISC Processors, Edited by C. May, D. Silha, R. Simpson, and H. Warren, Morgan Kaufmann Publishers, Inc., 1994.

[44] 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, pages 78-103, January 1997.

[45] J.D. Salehi, J.F. Kurose, and D. Towsley, "The Performance Impact of Scheduling for Cache Affinity in Parallel Nework Processing," *Proceedings of High-Performance Distributed Computing (HPDC-4)*, August 1995.

[46] J.D. Salehi, J.F. Kurose, and D. Towsley, "Further Results in Affinity-Based Scheduling of Parallel Networking," UM-CS-1995-046, May 1995.

[47] Scheduled Transfer Protocol on Linux (11.7.2001). SGI Inc.
Available at http://oss.sgi.com/projects/stp.

[48] F. Seifert, D. Balkanski, and W. Rehm, "Comparing MPI Performance of SCI and VIA," *Proceedings of the third International Conference on SCI-based Technology and Research (SCI-Europe 200)*, August 2000.

[49] SIMCA, the Simulator for the Superthreaded Architecture,
Available at http://wwwmount.ee.umn.edu/~lilja/SIMCA/index.html.

[50] SimOS-Alpha,
Available at http://research.compaq.com/wrl/projects/SimOS/SimOS.html.

[51] SimOS PowerPC,

Available at http://www.research.ibm.com/simos-ppc.

[52] SimpleScalar Version 4.0 Tutorial in conjunction with the *34th Annual International Symposium on Microarchitecture*,
Available at http://www.simplescalar.com/docs/ simple_tutorial_v4.pdf.

[53] D. Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers, Inc., 1999.

[54] Tcpdump/libpcap.
Available at http://www.tcpdump.org.

[55] J. R. Thorpe, "A Machine Independent DMA Framework for NetBSD," *Proceedings of USENIX 1998 Annual Technical Conference*, June 15-19, 1998.

[56] Virtual Interface Architecture Specification,
Available at http://www.viarch.org.

[57] WARTS, Wisconsin Architectural Research Tool Set,
Available at http://www.cs.wisc.edu/~larus/warts.html.

[58] C. Won *et al.*, "A Detailed Performance Analysis of UDP/IP, TCP/IP, and M-VIA Network Protocols Using Linux/SimOS," *Accepted to the Journal of Highspeed Network*, 2004.

[59] C. Won *et al.*, "Eager Data Transfer Mechasnim for User-Level Network Protocol," *In preparation,* 2004.

[60] C. Won *et al.*, "Linux/SimOS - A Simulation Environment for Evaluating High-Speed Communication Systems," *Proceedings of International Conference on Parallel Processing (ICPP-02)*, August 2002.

[61] 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.