Anatomy of UDP and M-VIA for Cluster Communication

Xiao Zhang and Laxmi N. Bhuyan
Department of Computer Science and Engineering
University of California, Riverside, CA 92521
Email: {xzhang, bhuyan}@cs.ucr.edu

Wu-chun Feng
Los Alamos National Laboratory
Los Alamos, NM 87545
Email: feng@lanl.gov

Abstract

High interprocess communication latency is detrimental to parallel and grid computing. Over the years, the network bandwidth has increased rapidly while the end-to-end latency has not decreased much. This is because the latency is dominated by the protocol software execution time in the kernel instead of the raw transmission time over the link. In this paper, we perform an anatomical analysis of the complete communication path between a sender and a receiver through measurements. We present an in-depth evaluation of various components of the UDP protocol over Fast Ethernet. Virtual Interface Architecture (VIA) protocol has been recently proposed to overcome the software overhead of the TCP/UDP/IP protocol. We analyze M-VIA, a modular VIA implementation for Linux over Ethernet, and compare its performance with UDP. Our experiment shows that the minimum one-way latency of UDP over Fast Ethernet is about 50% higher than that of M-VIA for short messages. This is mainly due to the protocol processing reduction in M-VIA.

Index Terms

cluster communication, TCP/UDP/IP, Virtual Interface Architecture (VIA), measurement, performance evaluation.

I. Introduction

In cluster computing, high-bandwidth and low-latency end-to-end communication is critical to achieve high performance. High bandwidth is necessary because a large mount of data is transfered among cluster nodes. In addition, to enable quick synchronization and coordination

This research is supported by NSF grants CCR-0220096, ACI-0233858, and a grant from Los Alamos National Laboratory

among cluster nodes, a lot of small control messages must be delivered as quickly as possible. Hence, low latency for small messages is extremely important.

We have seen the rapid increase of the network bandwidth from 10Mb to 10Gb. However, the end-to-end latency did not decrease proportionally, especially in the case of transferring small packets. For example, we measured a 23 μs minimum end-to-end latency over 100Mb Ethernet with 400-MHz Celeron CPU and 33-MHz PCI, while 19 μs was reported using 10Gb Ethernet, dual 2.2-GHz Xeon CPUs and 133-MHz PCI-X [1]. This discrepancy between bandwidth and latency motivates us to investigate the underlying reason. While it is easy to get the end-to-end latency in a particular system, a detailed analysis at the lowest level of entire end-to-end communication path would be more insightful as it could lead to fundamental discoveries as to where software and hardware inefficiencies still exist. To the best of our knowledge, no such work exists for a send/receive operation in a parallel computing cluster environment.

End-to-end communication latency involves both hardware and software. The hardware includes all the network components, such as network interface cards (NICs) and switches etc., to transmit packets from one host to another. Although high-end clusters usually employ powerful nodes and advanced network, such as Myrinet [2], QsNet [3] and InfiniBand [4] [5], more and more clusters are built using PCs and Ethernet. With the advent of 10Gb Ethernet [6], this trend will continue.

The software, on the other hand, usually refers to message processing through network protocols by the end hosts. There are two kinds of network software architectures: the traditional kernel-based network architecture and the user-level network architecture.

In the traditional network architecture, the protocol software usually forms part of the kernel which is invoked by a networking API, such as socket. The most popular protocol is TCP/IP designed for transferring packets over Internet. For compatibility, it is also heavily used in clusters. TCP/IP actually includes two transport layer protocols: TCP [7] and UDP [8]. TCP is a connection-oriented protocol designed to provide reliable end-to-end transmission over an unreliable network. It uses acknowledgment, timeout, retransmission and flow control to ensure ordered delivery and avoid network congestion. TCP has been studied extensively in the literature. For example, Clark et al. conducted an early analysis of TCP processing overhead [9]. In [10], Kay et al. divided the TCP/IP processing overhead into 8 categories and plotted the overhead of each category. At application level, [11] provided a critical path analysis of TCP transactions. In

[12], Xie et al. also presented an architectural-level analysis of TCP/IP and proposed a frequent instruction pair optimization for network processors.

UDP, on the other hand, is connectionless and provides user applications the simplest way to send a packet to another application. It is usually considered as the basic protocol with minimum latency. Although UDP doesn't guarantee reliable message delivery, it is still widely used in the cluster environment because of the inherent reliability of the cluster hardware. Applications using UDP usually include some application level mechanisms to ensure reliability between the end nodes. An example of such a system using UDP is the Network File System (NFS) [13]. Parallel Virtual Machine (PVM) [14] and some implementations of Message Passing Interface (MPI), such as LAM/MPI [15] and MPICH-SCore [16], also use UDP for communications. Therefore, analyzing the behavior and timing of UDP helps understand the minimum communication latency of traditional network protocols.

Due to the deep protocol stack, TCP/UDP/IP imposes a large latency for every message sent over the network, degrading the communication performance especially in the cluster environment. The user-level network architecture has emerged to address this issue. By removing the operating system and its centralized networking stack from the critical communication path, the user-level network architecture provides user applications a user-level network interface, through which users can directly access their network interface in a protected fashion. The operating system is only involved in setting up the communication. A large number of user-level network software, such as GM [17], AM [18], BIP [19], VMMC [20], U-Net [21], have been proposed. A survey of these messaging software for Myrinet can be found in [22]. All of these efforts finally led to an industry standard called Virtual Interface Architecture (VIA) [23]–[25] by Intel, Compaq and Microsoft.

VIA is a connection-based protocol. It provides each user process with a protected, directly accessible interface to the network hardware - a Virtual Interface (VI). Each VI represents a communication endpoint. VI endpoint pairs can be logically connected to support bi-directional, point-to-point data transfer. VIA provides communication services by a user-level library called VI User Agent, through which a user application can establish a VI connection for sending and receiving messages. VIA has also been studied before. In [26], Buonadonna et al. implemented the Berkeley VIA using Myrinet and evaluate its performance with U-Net, AM and BIP. In [27], Banikazemi et al. evaluated and compared the performance of different implementations

of essential VIA components. They also proposed a micro-benchmark for evaluating different VIA design choices in [28].

A common misunderstanding is that VIA can only work with programmable NICs. In fact, user-level network software is just a concept. VIA, in part, can be regarded as another network protocol that speaks in a different language than TCP/UDP/IP. With proper NIC drivers and user interface, programs written in VIA can run on traditional Ethernet NICs. This effort has been accomplished by National Energy Research Scientific Computing Center (NERSC), which implemented a high performance modular VIA for Linux over Ethernet, called M-VIA [29].

In this paper, we perform an anatomical analysis of the UDP and M-VIA, and make a detailed comparison between the two protocols. We analyze the protocol software (code) in the kernel and perform critical-path profiling. Our analysis differs from the existing studies in that we consider the entire path from a sender to a receiver that include protocol processing, operating system, NICs and network transmission. We provide a panorama of the complete transmit/receive path, vividly showing the time consumed by each part with CPU clock accuracy. By doing so, we are able to

- understand the exact operations by different parts of a protocol software.
- identify the timing and bottleneck along the critical path.
- design and optimize different parts of network software to reduce latency.

For the purpose of code instrumentation, we adopt the *checkpoint-based method* in which a small piece code is injected into the points of interest. We demonstrate that M-VIA results in substantial saving in protocol processing time over UDP/IP for small messages. The difference is relatively small for large messages.

The remainder of this paper is organized as follows. Section II describes our experimental setup and profiling methodology. Section III and IV present the detailed analysis of UDP and M-VIA respectively, with a comparison and contrast of these results presented in section V. Finally section VI presents our conclusion.

II. Experimental Setup and Profiling Methodology

To evaluate the latency performance of M-VIA and TCP/UDP/IP, we setup an experimental system as shown in Figure 1. Two PCs are connected with cross-over cable, each equipped with a Pentium Celeron 400-MHz CPU, 256-MB PC100 SDRAM, and a DEC 21140 chip based Fast

Ethernet NIC. On top of the hardware, we install RedHat Linux 9.0 (with updated kernel version 2.4.20-30.9, TCP/IP is implemented in kernel) and M-VIA (implemented as a kernel module). DEC 21140 NIC is used because M-VIA provides the driver for this card. A brief description of this NIC is presented in section III.

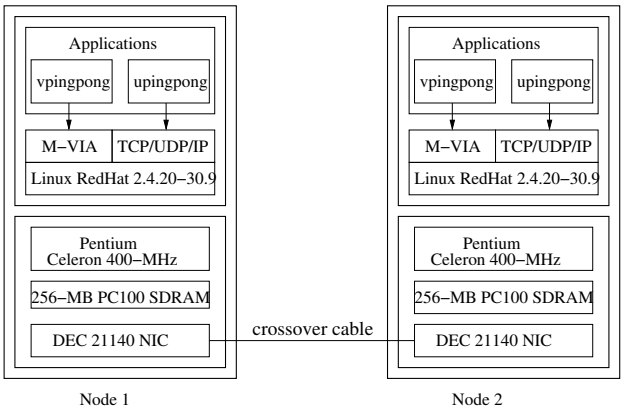


Fig. 1. Experimental Setup

In this paper, we report the end-to-end latency of UDP and M-VIA over 100Mb Fast Ethernet. Choosing Fast Ethernet and low-cost PCs clearly does not match today's technology. But in term of latency analysis, it is still valid because the latencies of transferring small packets are not much different among different networks as shown in the introduction.

To measure the end-to-end latency, we created two ping-pong programs in which one machine sends a message to another machine that echos the message back to the sender (`upingpong` for UDP/IP, and `vpingpong` for M-VIA unreliable service). The Linux kernel is also patched to include our profiling functions (see below). To minimize the amount of interference in our measurement, we eliminate all other network traffic and minimize the number of processes to the ping-pong program and a few essential Linux services. When processing the collected data, we consider 90% of the data to minimize the unavoidable interference such as OS context switch. Measurements run long enough to ensure the 95% confidence interval of the average latency with $\pm 5\%$ width.

The aim of this work is to measure and present a detailed analysis of the execution profile of M-VIA and UDP/IP. For user-level applications, `gprof` is a usual choice. For kernel codes, the Linux 2.4.x kernel provides profiling information if booted with `profile=X` option (where

X $\geq$ 2). However, these tools are not sufficient for our purpose. The measurement granularity of these tools is too coarse. They provide accumulative times for each function and are unable to distinguish multiple paths or profile specific events within a function. In addition, we want to profile user application and kernel code at the same time, preferably recording the user and kernel profiling results into one buffer. Using the above tools produces two profiling lists that ultimately have to be combined manually. Even if this process were to be automated, the profiling lists will have measurements of different precision as user-space measurements can easily be four orders of magnitude more coarse than kernel-space measurements.

While existing kernel-profiling tools such as the Linux Trace Toolkit (LTT) [30] exhaustively instrument the entire Linux kernel, such instrumentation is neither necessary nor recommended due to the Heisenberg principle, i.e., by exhaustively taking measurements from a given system, we adversely perturb the system itself, thus "invalidating" the measurements that are made.

We adopt a *checkpoint-based approach*, where a small piece of checkpoint code is manually inserted into the points of interest. This code records the current time-stamp of the measured point into a buffer. To facilitate user and kernel co-profiling, the buffer is allocated in the user application and passed to the kernel using a system call. The system call pins the related pages to the main memory and gets the corresponding kernel-space address of this buffer, so that the kernel can read/write this buffer directly. This is in fact how M-VIA sends messages without copying them from the user space to the kernel space.

The checkpoint-based approach was also used in [10]. Their checkpoint code sends a pattern and an event identifier to a logic analyzer, which then recognizes the pattern and stores the event identifier along with a time-stamp. Our approach, on the other hand, directly records the time-stamp using a "read-timestamp-counter" (`rdtsc`) instruction, thus providing a more convenient method of measurement with a time-stamp granularity on the order of only 2.5 nanoseconds (i.e., 1 / 400 MHz). The overhead of the checkpoint code is small (only 43 CPU cycles) and is subtracted from the measurement.

A tool for monitoring, debugging and analyzing system, called MAGNET [31], has been developed at Los Alamos National Laboratory. MAGNET uses the same checkpoint-based method. A `magnet_add()` call is inserted before the measured point to record the event into a circular buffer in kernel space, and a user-level program `magnet-read` is executed periodically to read the event records from the buffer and save them to the disk for post-processing. A major

difference between our method and MAGNET is that we allocate buffer in user space and pass it to kernel while MAGNET allocates buffer directly in kernel. We can instrument user application and kernel code at the same time. User application can also easily control when to start/end profiling.

III. UDP/IP Analysis

Before UDP analysis, it is necessary to briefly introduce the communication process between the host CPU and the DEC 21140 NIC used in our experiment. Fig. 2 show the diagram of the buffer management between the host and the NIC.

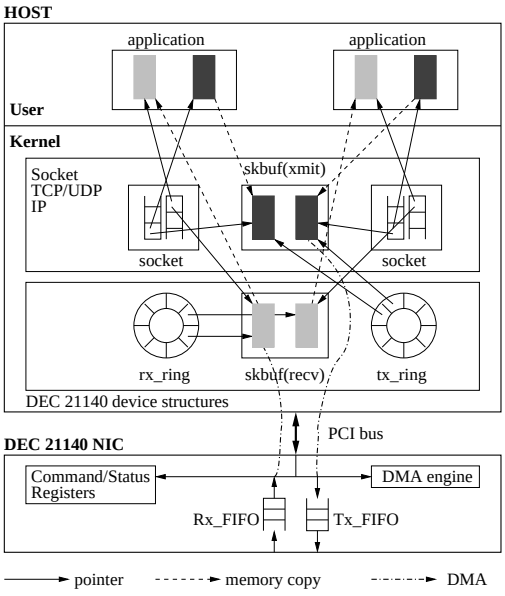


Fig. 2. The Linux implementation of TCP/UDP/IP over DEC 21140 NIC.

The DEC 21140 chip is a PCI bus-mastering Fast Ethernet controller capable of transferring frames to and from host memory via DMA. It includes a few on-chip command and status registers (CSRs), a DMA engine, a receive FIFO queue (`Rx_FIFO`) and a transmit FIFO queue (`Tx_FIFO`). It maintains a circular send ring buffer (`tx_ring`) and a receive ring buffer (`rx_ring`) containing descriptors which point to buffers for data transmission and reception. All rings and buffers are stored in the host kernel memory.

Linux TCP/IP implementation provides the standard BSD socket as the user interface. Each socket contains two queues for send and receive. Each queue entry points to an important buffer

called skbuf where all control and data information are stored. To send a message, the send process first copies the message from a user buffer to a skbuf , assemble the outgoing packet, pushes it into the tx_ring and then sends a transmit request to the NIC, which DMAs the packet from the skbuf to the Tx_FIFO and then to the network. At the receiver side, when the NIC detects an incoming packet, it first stores the packet into the Rx_FIFO and then DMAs the packet to a skbuf pointed by the tail rx_ring descriptor. Finally the OS copies the packet from the skbuf to a user-space buffer.

Now let's walk through the UDP/IP stack to see how a packet is transferred from a sender to a receiver. Fig. 3 shows a detailed time line of transferring a 1-byte message. We divide the time line into two categories: *critical time* and *non-critical time*. The former is the time that directly contributes to the one-way, end-to-end latency along a critical path while the latter is necessary for functionality but not in the critical path. We assume that the socket has been created, and the connection is also made to the remote address (Note: the UDP connection is just for the purpose of avoiding routing lookup for each packet. It is not actually connected to a remote host as in TCP). Since these are setup operations, they are not included in the critical path. To get the minimum latency, we use polling for receive. i.e., call a non-blocking receive in a loop. Also, note that the absolute times shown in Fig. 3, may be different if a more powerful PC is used because the execution time of the respective software segments will be reduced.

A. UDP/IP send processing

The sending process starts from send() , which performs a system call to sys_sendto() . The time to make a system call is not negligible, about 0.85 μs (t_1 in Fig. 3).

The sys_sendto() starts the socket layer processing. It gets the socket, builds a message header which contains various control information, and then invokes the UDP send function udp_sendmsg() . The socket layer spends about 0.77 μs (t_2 in Fig. 3). Note that in Linux, a socket actually consists of two socket structures: BSD socket and INET socket. BSD socket serves as a user interface and includes only fields that are meaningful to all network architectures. INET socket, on the other hand, contains information for every specific network architecture.

The UDP send processing is very fast, about 0.50 μs (t_3 in Fig. 3). Its main job is to verify the address, get the routing entry from the socket cache (for non-connected socket, it must goes to routing table to get the entry), build the UDP header, and invoke the IP send function

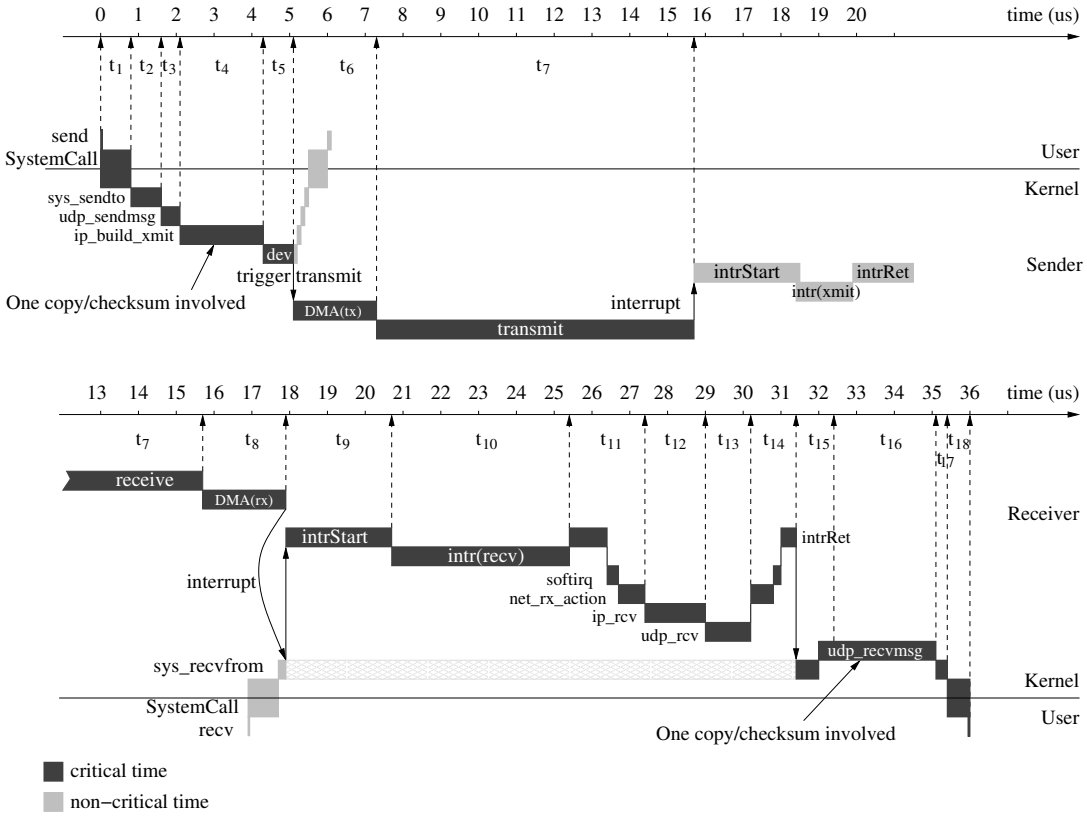


Fig. 3. Time line analysis of transferring a 1-byte message using UDP/IP (non-blocking receive). Note: with 14-byte Ethernet header, 20-byte IP header, 8-byte UDP header, 17-byte Ethernet padding and 4-byte Ethernet trailer, the packet received at receiver side is 64 bytes. The packet on the wire also includes 8 byte preamble.

`ip_build_xmit()`.

The IP send processing is quite tricky. A `skbuf` is first allocated based on the message size. The IP header is then filled into the `skbuf`. After that `udp_getfrag()` is called to copy the message from the user buffer to the `skbuf` and compute the checksum. Finally the UDP header is filled into the `skbuf`. This reversed processing order is due to the UDP specification which indicates that the UDP checksum must include the payload, the UDP header, and a pseudo-header which includes the source and destination IP addresses. So the IP layer actually performs some functions belonging to UDP. Clearly, the IP send processing time (t_4 in Fig. 3) depends on the message size, as shown in Fig. 4.

After the IP layer, the data link layer prepares the Ethernet header, enqueues the `skbuf` to a `Qdisc` queue, and invokes `qdisc_run()`, which dequeues the `skbuf` and calls the NIC send

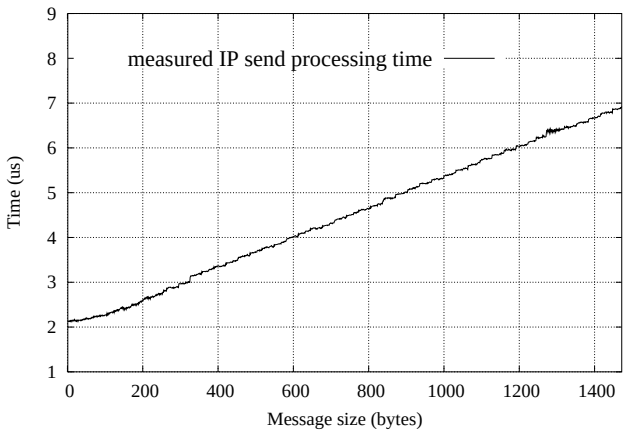


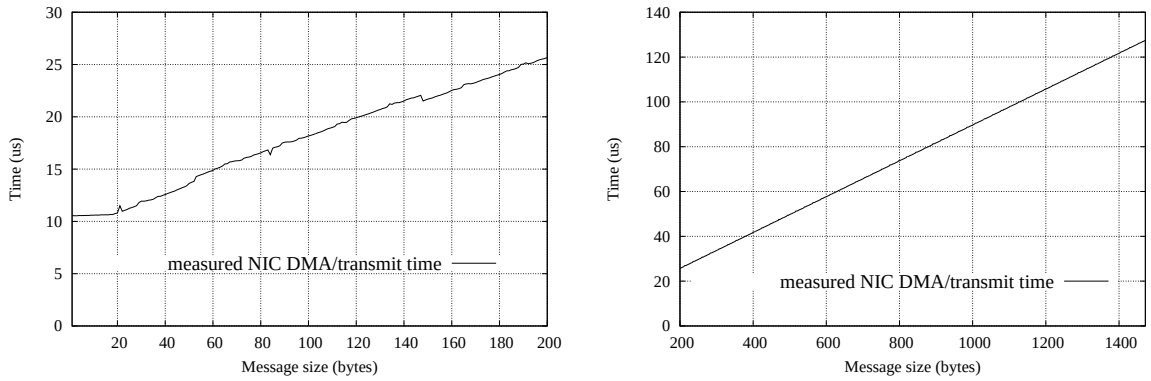
Fig. 4. IP layer processing time (t_4 in Fig. 3)

function `tulip_start_xmit()`. This function puts the `skbuf` into the `tx_ring` and then triggers the transmit by resetting the CSR1 register on the NIC. These device related processing takes about 0.73 μs (t_5 in Fig. 3). At this point the sender's work is finally done. After a number of function and system call returns, the sender program continues.

B. NIC transmit and receive process

After receiving the transmit command, the NIC engages the DMA to fetch the head `tx_ring` descriptor, and then the packet from the `skbuf` pointed by the `tx_ring` descriptor to the `Tx_FIFO` on the NIC. DMA and transmit are pipelined, i.e., when a complete packet or N bytes are received ($N = 128$ in current implementation), the NIC starts sending the packet out to the wire. After sending a packet, the NIC raises an interrupt.

Since DMA and transmit are done by the NIC, we measure them together from triggering an immediate transmit command to the starting point of the kernel interrupt handler `IRQ`*n*`_interrupt` ($t_6 + t_7$ in Fig. 3). Fig. 5 shows the DMA/transmit time as a function of message size. Note that packet size on the wire also includes 14-byte Ethernet header, 28-byte UDP/IP header and 4-byte Ethernet trailer (for CRC check). Due to the 64-byte minimum Ethernet size, for message size $\leq$ 18 bytes, Ethernet padding bytes are also included. This explains why the curve is flat for message size $\leq$ 18 bytes. The slope (0.08 μs/byte) exactly reflects the 100Mb line speed. This also implies that DMA and transmit are pipelined.



(a) 1≤ size ≤ 200

(b) 200 ≤ size ≤ 1472

Fig. 5. DEC 21140 NIC DMA/transmit time ($t_6 + t_7$ in Fig. 3).

Generally, the network may introduce delay. In the case of the connection by a cross-over cable, the network delay is negligible and can be regarded as 0,

At the receive side, when the NIC detects an incoming packet, it pushes the packet into the `Rx_FIFO` . When 64 bytes are received, the NIC DMAs the packet to a `skbuf` pointed by the tail `rx_ring` descriptor while receiving the rest of the packet. When a complete packet is received, the NIC raises an interrupt to trigger the receive processing. The DMA receive time (t_8 in Fig. 3) cannot be measured directly. Instead, we measure the round-trip time and all other time intervals, and t_8 is the difference between half of the round-trip time and all other intervals.

C. UDP/IP receive processing

The receiving process is much more complex and time-consuming than the sending process. As shown in Fig. 3, for small messages, half of the one-way latency is spent on receive. It consists of two threads: one is from the interrupt (we call it bottom thread) and the other is from the `recv()` system call in the user application (we call it upper thread).

The bottom thread starts from the interrupt handler `IQR`n`_interrupt` , where n is the IRQ number. It saves all the CPU registers on the stack and invokes the `do_IRQ()` , which in turn calls the NIC interrupt service routine `tulip_interrupt()` to handle the incoming packet. The processing time before `tulip_interrupt()` is very large, about 2.82 μs (t_9 in Fig. 3).

The basic job of `tulip_interrupt()` is to get the `skbuf` pointed by the head `rx_ring`

descriptor, determine the packet protocol id, push the skbuf into a per-cpu queue, and raise a NET_RX_SOFTIRQ soft interrupt. There is a special treatment for small packets (size $\leq$ 57 bytes). In this case, a new skbuf is allocated and the incoming packet will be copied with checksum into this buffer, so that the old buffer will be used for receiving the next packet. This operation is clearly shown in Fig. 6. After the copy break point, the processing time is constant, about 4.4 μs (t_{10} in Fig. 3).

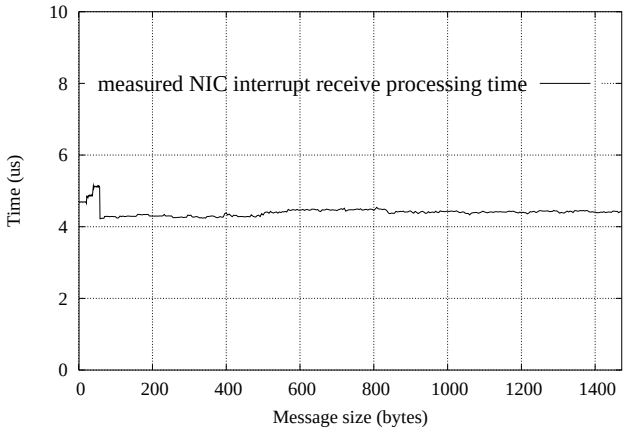


Fig. 6. DEC 21140 NIC interrupt receive processing time (t_{10} in Fig. 3).

The protocol processing is done in the NET_RX_SOFTIRQ soft interrupt which can be deferred according to the system status. In our test, it is executed at the end of do_IRQ() . The NET_RX_SOFTIRQ soft interrupt is implemented by the net_rx_action() function which dequeues the first packet from the queue, determines the network layer protocol number and invokes a suitable function of the network layer protocol, in our case, ip_rcv() . The overhead of do_IRQ() , the soft interrupt and net_rx_action() is about 2.03 μs (t_{11} in Fig. 3). Note that net_rx_action() is necessary because it needs to handle network layer protocols other than TCP/IP.

The ip_rcv() function first checks the length and the checksum of the packet and discards it if it is corrupted or truncated, then invokes ip_route_input() to determine whether the packet must be forwarded to another host or passed to the transport layer function (in our case, udp_rcv()). It also handles IP options and reassemble IP fragments. This function takes about 1.81 μs (t_{12} in Fig. 3) without IP options and fragments.

The udp_rcv() function checks the UDP header, finds the INET socket that matches the address/port, pushes the packet into the socket queue, and wakes up a sleeping process waiting for packets. Our ping-pong program uses non-blocking receive, so there is no wake-up operation. This function takes about 1.17 μs (t_{13} in Fig. 3).

At this point the bottom thread is done. After a number of function and interrupt returns, the interrupted program continues. This OS overhead is about 1.25 μs (t_{14} in Fig. 3).

The upper thread starts from recv() in the user application. It makes a socket system call to sys_recvfrom(), which performs the socket layer processing, very similar to sys_send(). The sys_recvfrom() eventually invokes the UDP receive function udp_recvmsg(), which checks the socket queue. If the queue is empty and the receive is blocking, it sleeps; if the receive is non-blocking, it returns immediately. If the queue is not empty, it dequeues the first skbuf from the socket queue and copies the message from the skbuf to the specified user buffer. Checksum is also performed during copy.

Our ping-pong program uses non-blocking receive, so sys_recvfrom() just returns if the queue is empty. To get a packet, recv() is called in a while loop. When a packet arrives, the ping-pong program is interrupted by the OS to execute the bottom thread as we discussed before. In Fig. 3 we show that the interrupt happens during sys_recvfrom(), but in fact it can happen at any time during a recv() call. If the interrupt occurs right before udp_recvmsg() checks the socket queue. we get the packet immediately. If the interrupt occurs after that point, we will spend another recv() system call to get the packet. Therefore the time from the end of the interrupt handler until udp_recvmsg() getting a packet can vary from almost 0 to 2.2 μs (t_{15} in Fig. 3).

Fig. 7 records the variation of the processing time (t_{16} in Fig. 3) after udp_recvmsg() gets a packet as a function of message size. The saw-shaped curve shows the the impact of alignment to checksum computation. There is a jump at message size of 58 bytes due to the fact that the checksum of small packets has already been computed in tulip_interrupt() (see Fig. 6).

After udp_recvmsg() returns, sys_recvfrom() performs some socket layer post processing, such as copy the sender address to the user program. It takes about 0.26 μs (t_{17} in Fig. 3). Finally the system call returns with overhead of 0.6 μs (t_{18} in Fig. 3), and the whole receiving process is done! Table I summarizes the cpu cycles and time spent in each layer.

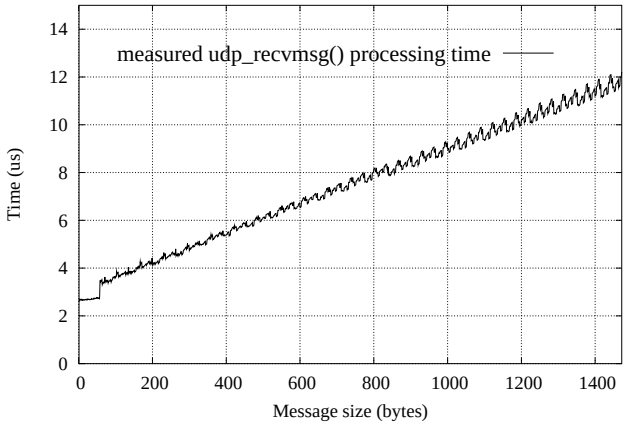


Fig. 7. `udp_recvmsg()` processing time after getting a packet from the socket queue (t_{16} in Fig. 3).

TABLE I

UDP/IP CRITICAL TIME BREAKDOWN USING PENTIUM CELERON 400MHZ CPU

Interval	**Description**	**Time**	
		cycles	μs
t_1	system call	342	0.85
t_2	socket send processing	307	0.77
t_3	UDP send processing	200	0.50
t_4	IP send processing	see Fig. 4	
t_5	device send processing	291	0.73
$t_6 + t_7$	NIC DMA/transmit	see Fig. 5	
t_8	NIC DMA receive	896	2.24
t_9	interrupt start time	1131	2.82
t_{10}	NIC interrupt service routine	see Fig. 6	
t_{11}	do_IRQ/softirq/net_rx_action processing	379+148+283	0.95+0.37+0.71
t_{12}	IP receive processing	726	1.81
t_{13}	UDP receive processing (in bottom thread)	470	1.17
t_{14}	net_rx_action/softirq/interrupt return	238+100+150	0.60+0.25+0.38
t_{15}	polling time before getting a message (average)	432	1.10
t_{16}	UDP receive processing (in upper thread)	see Fig. 7	
t_{17}	socket receive post processing	106	0.26
t_{18}	system call return	239	0.60

IV. M-VIA Analysis

M-VIA implements the standard user interface specified in the VIA specification [25]. End-to-end communication is through a Virtual Interface (VI). Similar to socket, a VI consists of a send queue and a receive queue. To start communication, a user program first creates a VI connection. Unlike a socket program which uses `send()` and `recv()`, a program using VI posts requests, in the form of descriptors, to the queues to send or receive messages. A descriptor contains all information, such as pointers to data buffers, for processing the request. To be specific, the sender first calls `VipPostSend()` to post a descriptor to its send queue and then calls `VipSendwait()` to wait for the send completion. To receive a packet, the receiver first calls `VipPostRecv()` to post a descriptor to its receive queue and then calls `VipRecvwait()` waiting for an incoming packet.

Fig. 8 shows the diagram of the buffer management between the host and the NIC when M-VIA is used. A major difference compared to UDP/IP is that there is no additional memory copy during the send process for M-VIA. The data buffer allocated by the user application will be directly accessed by the NIC. This is done by the `VipRegisterMem()` function, which registers the user buffer into the kernel. What M-VIA does in `VipMemRegister()` is to pin the user buffer into the memory to avoid swap so that the physical address of the user buffer can be obtained by the NIC for DMA operation. This memory registration is normally done during the setup. Fig. 9 shows the detailed time line of transferring 1 byte message in M-VIA. Compared to Fig. 3, the send/receive process in M-VIA is much simpler than that in UDP. Note that `VipPostRecv()` must be used before `VipRecvwait()` to receive a packet, but it doesn't have to be in the critical path because we can post a receive descriptor at any time before a packet arrives as shown in Fig. 9.

A. M-VIA send processing

The sender starts with `VipPostSend()` which does two things: 1) push a send descriptor to the end of the send queue. It is a typical linked-list operation, which takes very little time, about 0.04 μs (t_1 in Fig. 9); 2) inform the NIC to send the message. For NICs that support VIA in hardware, a doorbell mechanism is provided to trigger the NIC. For traditional NICs, such as those used in M-VIA, additional device related processing is necessary. This is done by fast

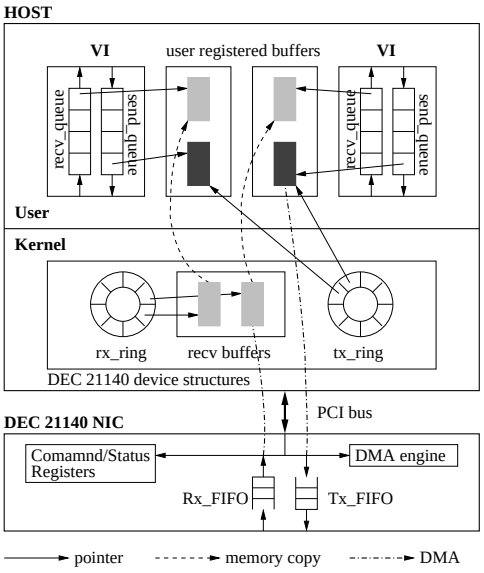


Fig. 8. M-VIA over DEC 21140 NIC.

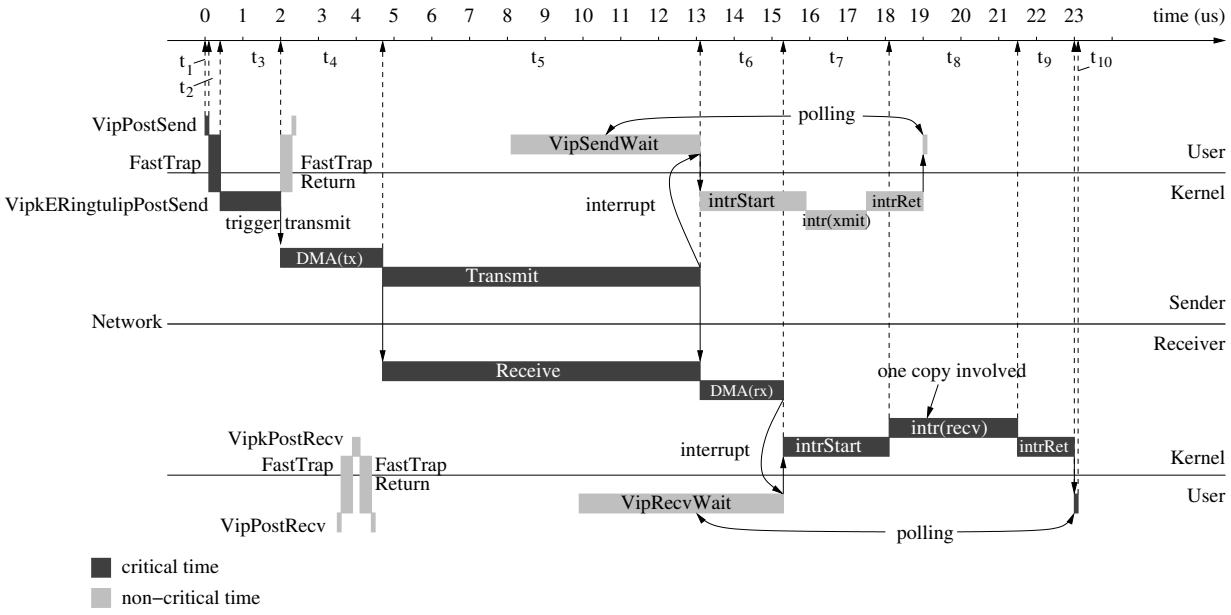


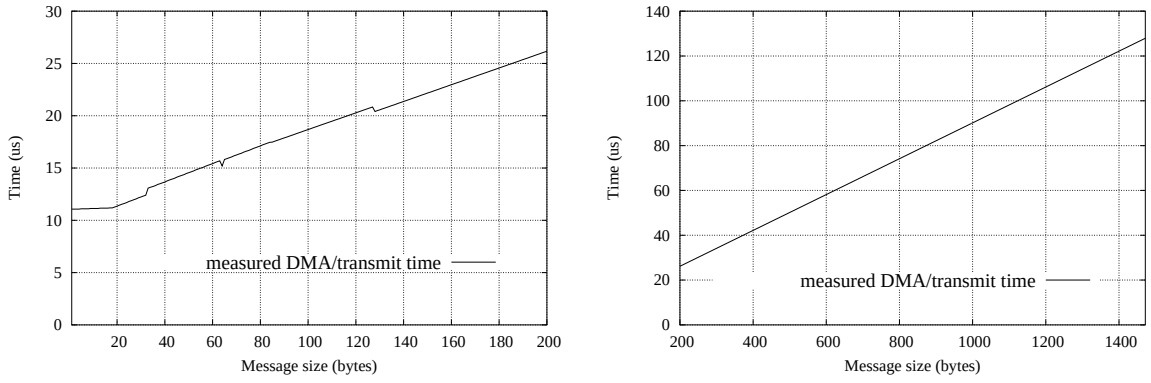
Fig. 9. Time line analysis of transferring a 1-byte message using M-VIA (unreliable service, non-blocking receive). Note: with 14-byte Ethernet header, 28-byte M-VIA header, 17-byte Ethernet padding and 4-byte Ethernet trailer, the packet received at the receiver side is 64 bytes. The packet on the wire also includes 8 byte preamble.

trapping to a kernel function `VipkERingtulipPostSend()`. The fast trap takes about 0.33 μs (t_2 in Fig. 9).

`VipkERingtulipPostSend()` first prepares the send header, waits for device stop, gets the next send descriptor entry, and fills in the header. Then it segments large packets if necessary. Segmentation is based on maximum transmit unit (MTU) and page boundary. A packet larger than MTU or across a page boundary will be segmented into chunks. For each chunk, the function first puts the chunk address into an available `tx_ring` entry, and then triggers an immediate transmit demand by resetting CSR1 on the NIC. Because of pipelining, the time which `VipkERingtulipPostSend()` contributes to the critical path is constant, about 1.56 μs (t_3 in Fig. 9). Comparing Fig. 9 and Fig. 3, it may be noticed that M-VIA reduces the send time from 5 μs to 2 μs before DMA, and that the send time in M-VIA is constant because of the copy elimination.

B. NIC transmit and receive

Fig. 10 shows that DMA/transmit time. The graph is almost the same as that in the case of UDP. Close examination of the measured data shows that there is a 0.5 μs increase in the latency in the case of M-VIA. This is due to the fact that M-VIA puts header and data into two buffers, so that the NIC needs to DMA twice to get all of them.



(a) $1 \leq$ size ≤ 200 (b) $200 \leq$ size ≤ 1472

Fig. 10. M-VIA DEC 21140 NIC DMA/transmit time ($t_4 + t_5$ in Fig. 9).

C. M-VIA receive processing

Like UDP, the M-VIA receiving process also consists of two threads. The bottom thread is triggered by the NIC interrupt after it DMAs a packet to a buffer in the host memory. The interrupt starting time is about 2.80 μs (t_7 in Fig. 9).

The NIC interrupt service routine is also `tulip_interrupt()` which is modified to add VIA functionalities. Contrary to UDP, this function does all the following receive work as follows:

1) get the VI handle from the received packet;
2) get the corresponding VI structure;
3) check the sequence number of the received packet;
4) get the VI descriptor from the VI receive queue;
5) copy the message from the buffer pointed by the head `rx_ring` descriptor to the buffer specified by the VI descriptor;
6) and, if last segment, mark the VI descriptor complete bit, increment the receive count, wake up a block process if necessary.

Fig. 11 shows this processing time (t_8 in Fig. 9) as a function of message size. The step shaped curves clearly shows the impact of cache line (32 bytes) to the memory copy.

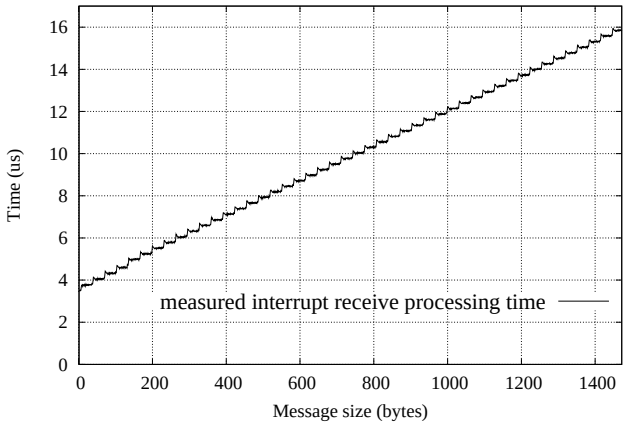


Fig. 11. M-VIA NIC interrupt receive processing time (t_8 in Fig. 9)

After the NIC interrupt service routine, the OS returns to execute the interrupted program. This interrupt return overhead is about 1.5 μs (t_9 in Fig. 9).

The upper thread starts from `VipRecvWait()` which just checks whether or not the descrip-

tor at the head of the receive queue is completed. It has two stages. First it polls the completion bit for a number of times (50000 times in the current implementation). Then, if polling fails, `VipRecvWait()` will trap to `VipkRecvWait()` and sleep there, and finally be waked up by the bottom thread. In our test, `VipRecvWait()` always returns in the polling phase. One polling iteration takes very little time, about 0.04 μs (t_{10} in Fig. 9). This conclude the whole receiving process. Table II summarizes the cpu cycles and time spent in each layer.

TABLE II

M-VIA CRITICAL TIME BREAKDOWN USING PENTIUM CELERON 400MHZ CPU

Interval	**Description**	**Time**	
		cycles	μs
t_1	VipPostSend() processing	24	0.06
t_2	fast trap	131	0.33
t_3	device send processing	627	1.56
$t_4 + t_5$	DMA/transmit	see Fig. 10	
t_6	DMA receive time	896	2.24
t_7	interrupt starting time	1119	2.80
t_8	interrupt recv processing	see Fig. 11	
t_9	interrupt return time	600	1.50
t_{10}	VipRecvWait() processing (polling)	16	0.04

V. COMPARISON BETWEEN UDP AND M-VIA

Based on the previous data and analysis, we now conduct a detailed comparison between UDP and M-VIA, and propose two optimizations to reduce the UDP latency. Fig. 12 shows the one-way end-to-end latency of UDP and M-VIA. Note that the one-way latency of M-VIA for short messages is much less than UDP. Since most of the interprocess communication in parallel computing or grid environments consists of small messages, M-VIA has enormous advantage over UDP. When we move to a server workload transferring big messages, this advantage tends to reduce.

In Table III, we put each time interval into one of the five categories to show the difference between UDP and M-VIA in each category. As we can see, M-VIA has some processing in user space before trapping to kernel. It also spends a little more time in DMA/transmit/receive because

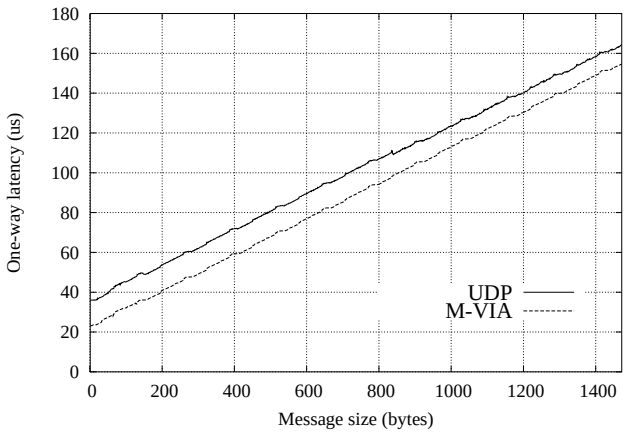


Fig. 12. UDP and M-VIA one-way latency

M-VIA stores header and data in two buffers. However, these time increases are negligible. In other categories, the saving of M-VIA over UDP is significant as described below.

TABLE III

CRITICAL TIME COMPARISON BETWEEN UDP AND M-VIA FOR TRANSFERRING A 1-BYTE MESSAGE OVER FAST EHTERNET USING PENTIUM CELERON 400-MHZ CPU.

Categories	**UDP/IP**				**M-VIA**			
	Intervals in Fig. 3	cycles	μs	%	Intervals in Fig. 9	cycles	μs	%
User application					$t_1 + t_{10}$	40	0.10	0.43
Protocol processing (send)	$t_2 + t_3 + t_4 + t_5$	1688	4.22	11.72	t_3	627	1.56	6.75
Protocol processing (recv)	$t_{10} + t_{12} + t_{13} + t_{15} + t_{16} + t_{17}$	4616	11.54	32.04	t_8	1404	3.51	15.20
OS overhead (send)	t_1	342	0.85	2.36	t_2	131	0.33	1.43
OS overhead (recv)	$t_9 + t_{11} + t_{14} + t_{18}$	2644	6.61	18.35	$t_7 + t_9$	1719	4.30	18.61
DMA/transmit/receive	$t_6 + t_7 + t_8$	5120	12.80	32.76	$t_4 + t_5 + t_6$	5320	13.30	57.58
Total		14408	36.02			9240	23.10	

- First, M-VIA has less OS overhead than UDP because M-VIA uses fast trap (0.33 μs) instead of normal system call (0.85 μs); and M-VIA also avoids soft interrupt by handling all receive functionalities in the interrupt handler. However, since M-VIA interrupt processing time (see Fig. 11) is variable, this may cause slow response time to other processes when receiving large packets.
- Second, M-VIA spends 5.17 μs on protocol processing while UDP spends 15.76 μs. It is

because UDP, though simple, still needs to go through a number of layers, while M-VIA directly handles packets in the device layer. However, M-VIA lacks IP routing. It depends on the MAC address to route packet, which means that it can only be used in LAN, where nodes are connected by switches. UDP, on the other hand, can be used anywhere. This shows that M-VIA is the protocol specifically optimized for LANs.

Based on the above comparison, we can propose the following simple optimizations for UDP to reduce its latency by 10%.

- *Use fast trap and avoid soft interrupt*: For heavily used protocols like UDP, we can provide a fast path by writing a fast trap for `send()` and `recv()` and processing protocol stack directly in the interrupt handler instead of soft interrupt. As shown in Table I, the `ip_rcv()` and `udp_rcv()` processing times are small, so putting them into the interrupt handler is not a problem. The OS overhead thus can be reduced by 3 μs.
- *Reduce receive polling cycle*: In our ping-pong test, both UDP and M-VIA use non-blocking receive. However, M-VIA does this more efficiently. It just checks the complete bit of the head descriptor in each polling. The UDP polling cycle, on the other hand, can be as large as system call cycle with average of 1.1 μs (t_{15} in Fig. 3). If we let the `udp_recvmsg()` check the queue status in a loop like that in M-VIA, we can remove t_{15} and save 1 μs.

VI. Conclusion

In this paper, we performed an anatomical analysis of UDP and M-VIA. Our experiment shows that to transfer a 1-byte message, the NIC spends about 13 μs, which is the hardware limit of Fast Ethernet. On top of that, UDP spends 23 μs in processing while M-VIA spends only 10 μs. This significant difference is mainly due to the protocol processing time reduction in M-VIA. UDP has to go through a number of layers while M-VIA directly handles packets at the device level. However, M-VIA lacks IP routine functionality, making it only feasible in specific environments, such as cluster, where nodes are connected by switches. M-VIA also incurs less OS overhead than UDP. However, the techniques used by M-VIA can also apply to UDP. Based on our analysis, we proposed two simple optimizations to reduce the UDP latency by 10%. It was also shown that the latency of both UDP and M-VIA increases as the message size increases, and that the difference in latency between the two becomes small when transferring

large messages. This means that M-VIA is much more applicable for parallel computing than server applications.

REFERENCES

[1] J. Hurwitz and W. Feng, "End-to-end performance of 10-gigabit Ethernet on commodity systems," *IEEE Micro*, vol. 24, no. 1, pp. 10–22, Jan./Feb. 2004.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Jan./Feb. 1995.

[3] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics network: high-performance clustering technology," *IEEE Micro*, vol. 22, no. 1, pp. 46–57, Jan./Feb. 2002.

[4] InfiniBand Trade Association, "Infiniband architecture specification, release 1.0." [Online]. Available: http://www.infinband.org

[5] B. Chandrasekaran, P. Wyckoff, and D. K. Panda, "MIBA: A micro-benchmark suite for evaluating infiniband architecture implementations," in *Performance TOOLS 2003*, September 2-5 2003.

[6] W. Feng, G. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low, "Optimizing 10-gigabit Ethernet for networks of workstations, clusters and grids: A case study," in *High-Performance Networking and Computing Conference (SC2003)*, Phoenix AZ, Nov. 2003.

[7] J. Postel, "Transimission control protocol," RFC 793, Sept. 1981.

[8] ——, "User datagram protocol," RFC 768, Aug. 1980.

[9] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Commun. Mag.*, vol. 27, no. 6, pp. 23–29, June 1989.

[10] J. Kay and J. Pasquale, "Profiling and reducing processing overheads in TCP/IP," *IEEE/ACM Trans. Networking*, vol. 4, no. 6, pp. 817–828, December 1996.

[11] P. Barford and M. Crovella, "Critical path analysis of TCP transactions," *IEEE/ACM Trans. Networking*, vol. 9, no. 3, pp. 238–248, June 2001.

[12] H. Xie, L. Zhao, and L. N. Bhuyan, "Architectural analysis and instruction-set optimization for design of network protocol processors," in *CODES+ISSS'03*, Newport Beach, CA, Dec. 2003, pp. 225–230.

[13] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "NFS version 4 protocol," RFC 3010, Dec. 2000.

[14] "PVM - parallel virtual machine." [Online]. Available: //www.csm.ornl.gov/pvm/pvm_home.html

[15] "LAM/MPI Parallel Computing." [Online]. Available: http://www.lam-mpi.org/

[16] PC Cluster Consortium, "SCore Cluster System Software." [Online]. Available: http://www.pccluster.org/

[17] Myricom, Inc., "GM: low-level message-passing system for Myrinet networks, http://www.myrinet.com."

[18] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *19th International Symposium on Computer Architecture*, vol. 20, Gold Coast, Australia, 1992, pp. 256–266.

[19] L. Prylli and B. Tourancheau, "BIP: A new protocol designed for high performance," in *PC-NOW Workshop, held in parallel with IPPS/SPDP98*, Orlando, USA, March 30 – April 3 1998.

[20] C. Dubnicki, A. Bilas, K. Li, and J. Philbin, "Design and implementation of virtual memory-mapped communication on Myrinet," in *Proc. of the International Parallel Processing Symposium*, April 1997.

[21] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proc. of the 15th ACM Symposium on Operation Systems Principles*, Copper Mountain, Colorado, December 1995.

[22] R. dos Santos, R. Bianchini, and C. L. Amorim, "A survey of messaging software issues and systems for Myrinet-based clusters," *Parallel and Distributed Computing Practices, special issue on High-Performance Computing on Clusters*, vol. 2, no. 2, June 1999.

[23] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd, "The virtual interface architecture," *IEEE Micro*, pp. 66–76, Mar./Apr. 1998.

[24] T. von Eicken and W. Vogels, "Evolution of the virtual interface architecture," *IEEE Computer*, vol. 31, pp. 61–68, November 1998.

[25] Compaq, Intel, and Microsoft, "Virtual interface architecture specification, draft revision 1.0," December 4 1997. [Online]. Available: http://www.viarch.org

[26] P. Buonadomna, A. Geweke, and D. Culler, "An implementation and analysis of the virtual interface architecture," in *Proc. of SC '98*, Orlando, FL, Nov. 7-13 1998.

[27] M. Banikazemi, B. Abali, and D. K. Panda, "Comparison and evaluation of design choices for implementing the virtual interface architecture (VIA)," in *Fourth Int'l Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '00)*, Jan. 2000.

[28] M. Banikazemi, J. Liu, S. Kutlug, P. S. A. Ramakrishna, H. Sah, and D. K. Panda, "VIBe: A micro-benchmark suite for evaluating virtual interface architecture (VIA) implementations," in *Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2001.

[29] National Energy Research Scientific Computing Center, "M-VIA: A high performance modular VIA for Linux." [Online]. Available: http://www.nersc.org/research/FTG/via

[30] K. Yaghmour, M. R. Dagenais, and E. P. de Montral, "Measuring and characterizing system behavior using kernel-level event logging," in *Proc. of the 2000 USENIX Annual Technical Conference*, June 2000.

[31] M. K. Gardner, W. Feng, M. Broxton, A. Engelhart, and G. Hurwitz, "MAGNET: A tool for debugging, analyzing and adapting computing systems," in *Proc. of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003, pp. 310–317.