Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
J. Parallel Distrib. Comput. 65 (2005) 1290 – 1298
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Research note

Anatomy of UDP and M-VIA for cluster communication☆

Xiao Zhang[a], Laxmi N. Bhuyan[a], Wu-Chun Feng[b,*]

[a]*Department of Computer Science and Engineering, University of California, Riverside, CA 92521, USA*
[b]*Los Alamos National Laboratory, Los Alamos, NM 87545, USA*

Received 1 April 2005; accepted 1 April 2005
Available online 20 June 2005

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. The aim of our experiments is to present the protocol overheads in details rather than to suggest new techniques to reduce overheads.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Cluster communication; TCP/UDP/IP; Virtual Interface Architecture (VIA); Measurement; Performance evaluation

1. 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 transferred among cluster nodes. In addition, to enable quick synchronization and coordination 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 rapid increase of the network bandwidth from 10 Mb to 10 Gb. 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 100 Mb Ethernet with 400-MHz Celeron CPU and 33-MHz PCI, while a latency of 19 µs was reported using 10 Gb Ethernet, dual 2.2-GHz Xeon CPUs and 133-MHz PCI-X [5]. 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 latency. 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 [1] and InfiniBand [6], more and more clusters are built using PCs and Ethernet. With the advent of 10 Gb Ethernet [5], this trend will continue.

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

* Corresponding author.
E-mail addresses: xzhang@cs.ucr.edu (X. Zhang), bhuyan@cs.ucr.edu (L.N. Bhuyan), feng@lanl.gov (W.-C. Feng).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.04.009

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 used in cluster: 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 suite is TCP/IP designed for transferring packets over Internet. For compatibility, it is also heavily used in clusters. TCP [12] is a connection-oriented protocol designed to provide reliable end-to-end transmission over an unreliable network, and has been studied extensively in the literature [3,7]. UDP [11], as a connectionless transport layer protocol in TCP/IP, provides user applications the simplest way to send a packet to another application. Although UDP does not guarantee reliable message delivery, it is widely used in the cluster environment because of the inherent reliability of the cluster hardware. Network File System (NFS) [15], Parallel Virtual Machine (PVM) [16] and some implementations of Message Passing Interface (MPI), such as LAM/MPI [8] and MPICH-SCore [10], 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/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 have been proposed. A survey of these messaging software for Myrinet can be found in [13]. All of these efforts finally led to an industry standard called Virtual Interface Architecture (VIA) [4] by Intel, Compaq and Microsoft. Therefore, we analyze VIA in addition to UDP in this paper.

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 to some extent [2].

In this paper, we perform an anatomical analysis of the UDP and 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

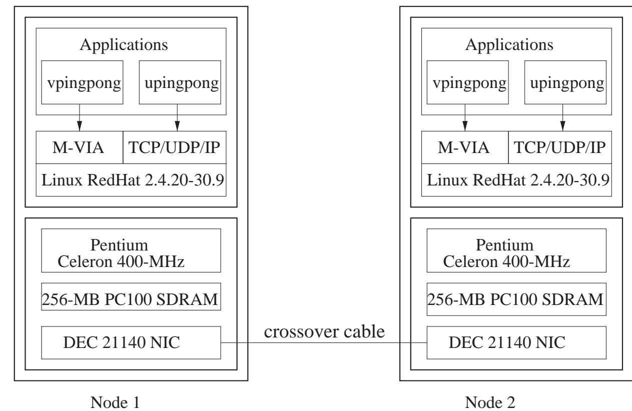

Fig. 1. Experimental setup

path from a sender to a receiver that includes 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 (1) understand the exact operations by different parts of a protocol software; (2) identify the timing and bottleneck along the critical path, and (3) design and optimize different parts of network software to reduce latency.

The remainder of this paper is organized as follows. Section 2 describes our experimental setup and profiling methodology. Sections 3 and 4 present the detailed analysis of UDP and VIA, respectively, with a comparison and contrast of these results presented in Section 5. Finally, Section 6 presents our conclusion.

2. Experimental setup and profiling methodology

To evaluate the latency performance of UDP and VIA, we setup an experimental system as shown in Fig. 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). We also install M-VIA, a high-performance modular VIA for Linux over Ethernet [9]. M-VIA provides a driver for DEC 21140, allowing us to make a fair comparison between UDP and VIA using the same hardware. Of course, choosing Fast Ethernet and low-speed PCs clearly does not match today's technology. But in term of latency analysis, it may still be 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. We adopt a *time-stamp-based approach*, where a small piece of code is manually inserted into the points of interest. This code records the current time-stamp of the measured point into a buffer using the "read-time-stamp-counter" (`rdtsc`) instruction, thus providing a more convenient method of measurement with a time-stamp granularity on the order of only 2.5 ns (i.e., 1/400 MHz). The overhead of the time-stamp code is small (only 43 CPU cycles) and is subtracted from the measurement.

3. 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 shows the diagram of the buffer management between the host and the 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 also maintains in the host kernel memory 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.

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 structure `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`, assembles 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 us 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*

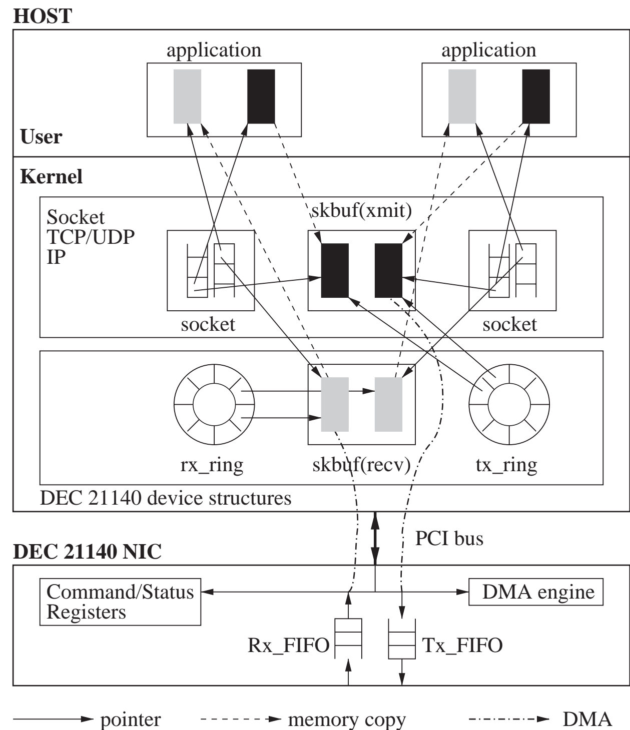

Fig. 2. Linux TCP/UDP/IP over DEC 21140 NIC.

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. Due to page limit, we only give a very brief description of the entire processing path. Interested readers can refer to the extended version of this paper [14] for more details.

3.1. UDP/IP send processing

The sending process starts from the system call `send()` (t_1) which goes through the following layers: (1) the socket layer function `sys_sendto()` (t_2) gets the socket and builds a message header which contains various control information. (2) the UDP send function `udp_sendmsg()` (t_3) verifies the address, gets the routing entry and builds the UDP header. (3) the IP send function `ip_build_xmit()` (t_4, also shown in Fig. 4) allocates a `sk_buf`, fills the IP header, and then calls `udp_getfrag()` to copy the mes-

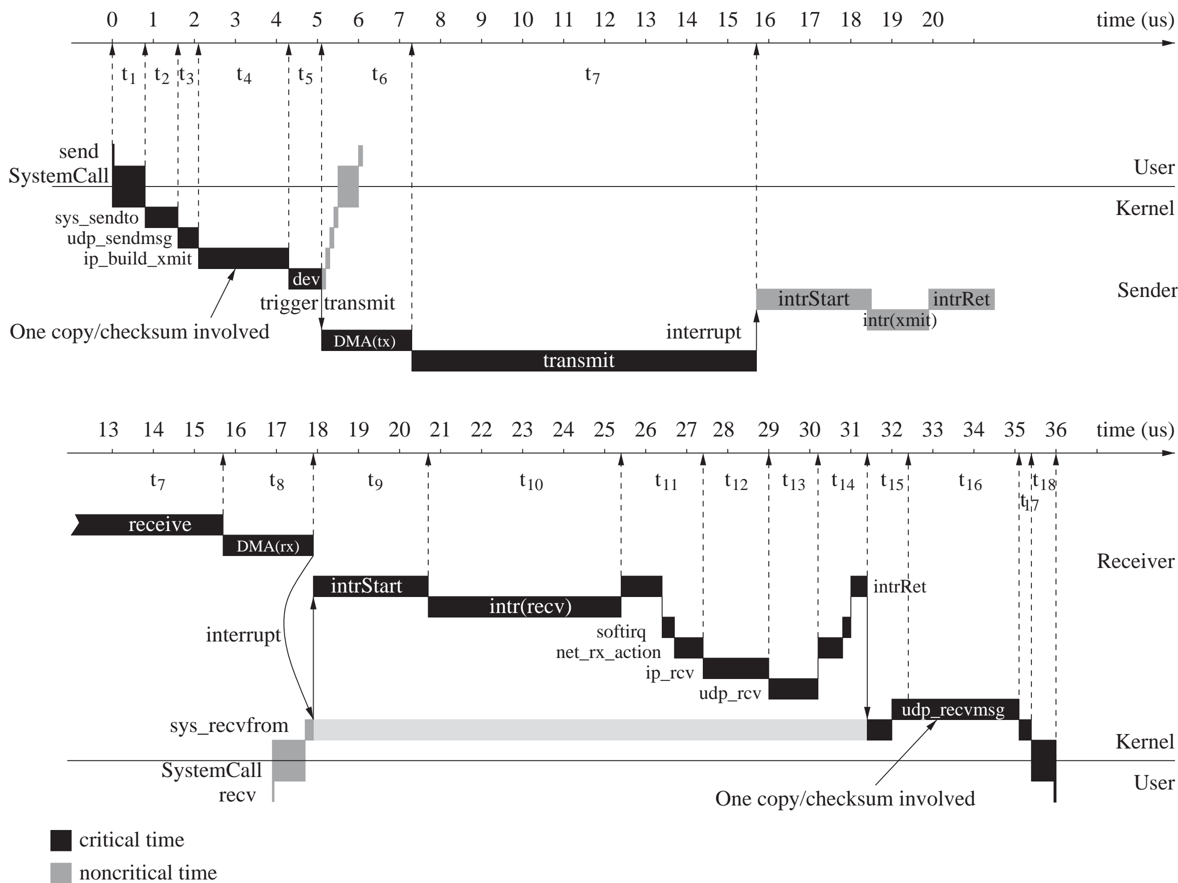

Fig. 3. Time line analysis of transferring a 1-byte message using UDP/IP (non-blocking receive). *Note*: with 14-byte Ethernet header, 20-byte IP header, 8-byte UDP header, 17-byte Ethernet padding and 4-byte Ethernet trailer, the packet received at receiver side is 64 bytes. The packet on the wire also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

sage from the user buffer to the sk_buf and compute the checksum, and finally fills the UDP header. (4) the data link layer (t_5) prepares the Ethernet header, calls the NIC send function tulip_start_xmit() to put sk_buf into the tx_ring and finally triggers the transmit by resetting the CSR1 register on the NIC.

3.2. NIC transmit and receive processing

The NIC then engages the DMA to fetch the packet from the host memory to the Tx_FIFO and then send it onto the wire. At the receive side, when the NIC detects an incoming packet, it pushes the packet into the Rx_FIFO, and then DMAs the packet to the host memory, and finally raises an interrupt to trigger the receive processing. Fig. 5 shows the DMA/transmit time as a function of message size.

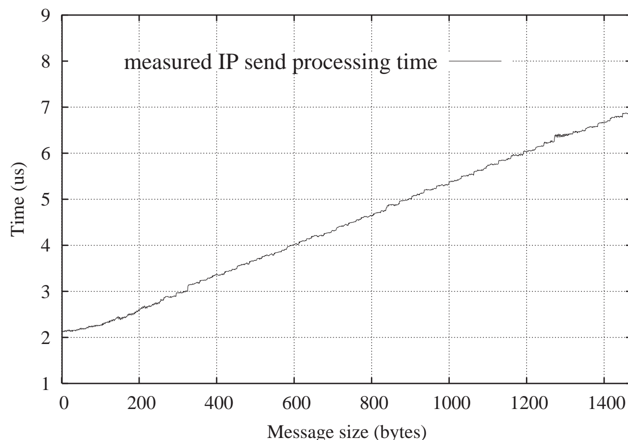

Fig. 4. IP layer processing time (t_4 in Fig. 3).

3.3. 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

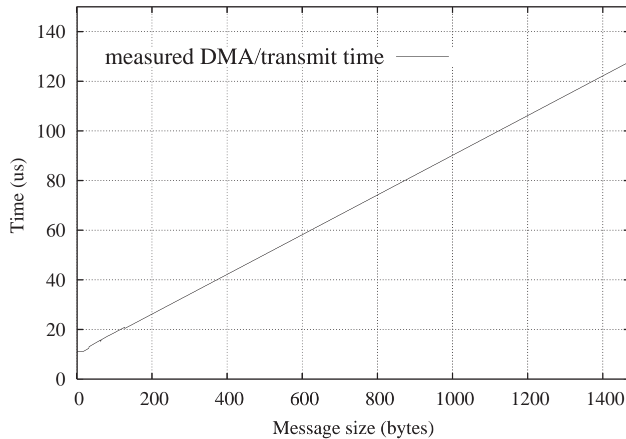

Fig. 5. DEC 21140 NIC DMA/transmit time ($t_6 + t_7$ in Fig. 3).

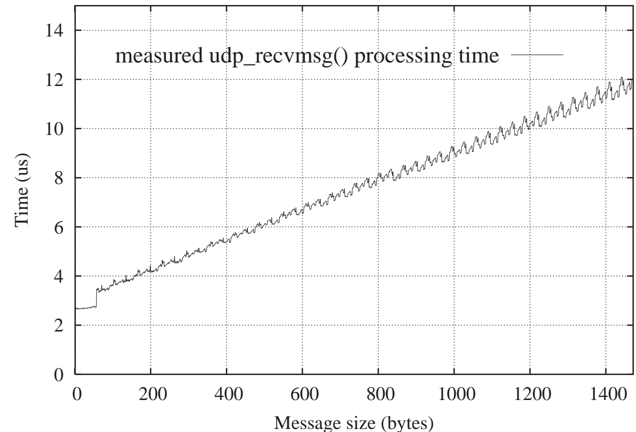

Fig. 6. `udp_recvmsg()` processing time after getting a packet from the socket queue (t_{16} in Fig. 3).

the CPU registers on the stack and invokes the `do_IRQ()`, which in turn calls the NIC interrupt service routine `tulip_interrupt()` to get the `skbuf` pointed by the head `rx_ring` descriptor, determine the packet protocol id and raise the `NET_RX_SOFTIRQ` soft interrupt. The `NET_RX_SOFTIRQ` soft interrupt is implemented by the `net_rx_action()` function which invokes a suitable network layer function, in our case, `ip_rcv()` which in turn calls the transport layer function (in our case, `udp_rcv()`). The `udp_rcv()` checks the UDP header, finds the socket that matches the address and port, pushes the packet into the socket queue, and wakes up a sleeping process waiting for packets if necessary.

The upper thread starts from the system call `recv()`. It eventually invokes `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. Fig. 6 records the variation of the processing time (t_{16} in Fig. 3) after `udp_recvmsg()` gets a packet as a function of message size.

After `udp_recvmsg()` returns, `sys_recvfrom()` performs some socket layer post processing, such as copy the sender address to the user space. Finally the system call returns, and the whole receiving process is done. Table 1 summarizes the CPU cycles and time spent in each segment.

4. M-VIA Analysis

M-VIA implements the standard user interface specified in the VIA specification [4]. 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. 7 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. 8 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 does not 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. 8.

4.1. 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 (t_1 in Fig. 8); (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

Table 1
UDP/IP critical time breakdown using Pentium Celeron 400 MHz 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	1780	4.44
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. 6	
t_{17}	Socket receive post processing	106	0.26
t_{18}	System call return	239	0.60

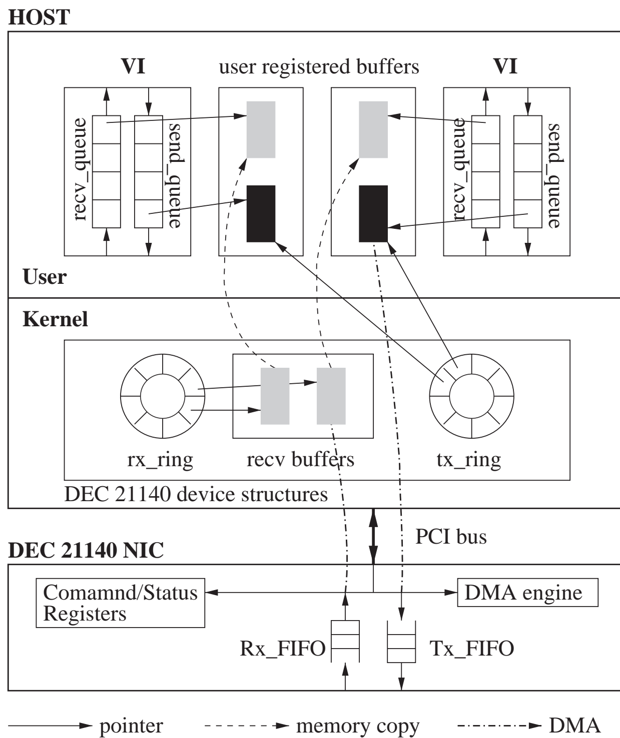

Fig. 7. M-VIA over DEC 21 140 NIC.

those used in M-VIA, additional device-related processing is necessary. This is done by fast trapping to a kernel function `VipkERingtulipPostSend()` (t_2 in Fig. 8).

`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 (t_3 in Fig. 8). Comparing Fig. 8 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.

4.2. NIC transmit and receive processing

Fig. 9 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.

4.3. 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 the NIC DMAs a packet to a buffer in the host memory. The NIC interrupt service routine is also `tulip_interrupt()` which is modified to add VIA functionalities. Unlike UDP which raises a soft interrupt and may postpone the receive processing, this function does all the receiving 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;

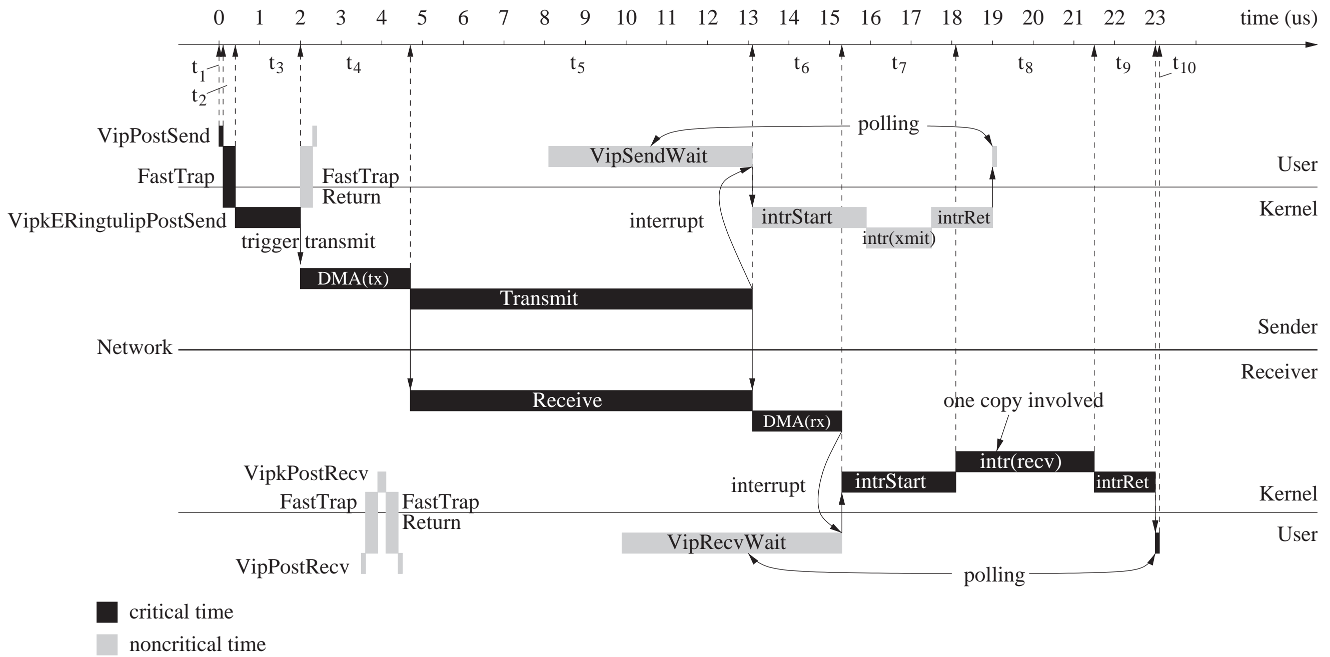

Fig. 8. Time line analysis of transferring a 1-byte message using M-VIA (unreliable service, non-blocking receive). *Note*: with 14-byte Ethernet header, 28-byte M-VIA header, 17-byte Ethernet padding and 4-byte Ethernet trailer, the packet received at the receiver side is 64 bytes. The packet on the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

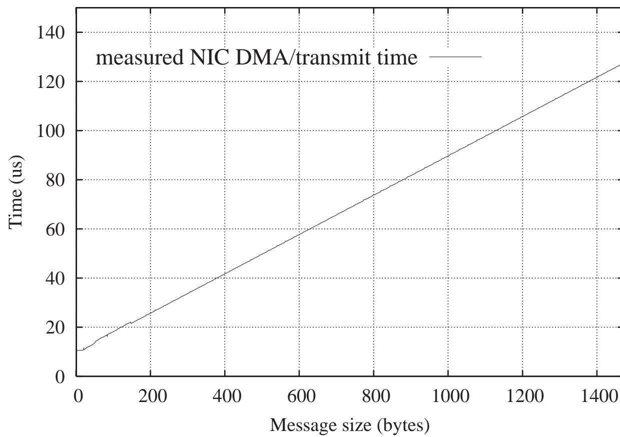

Fig. 9. M-VIA DEC 21140 NIC DMA/transmit time ($t_4 + t_5$ in Fig. 8).

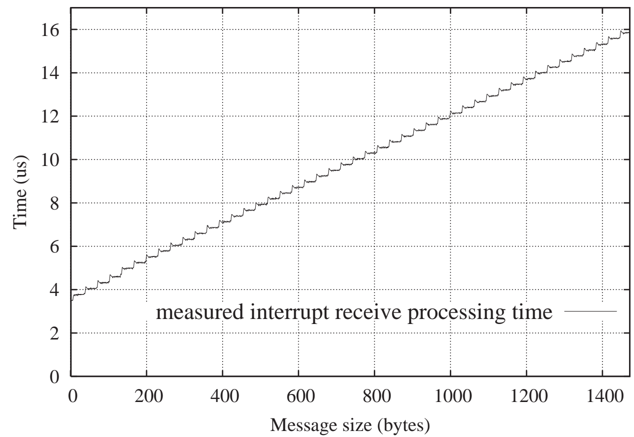

Fig. 10. M-VIA NIC interrupt receive processing time (t_8 in Fig. 8)

(6) and, if last segment, mark the VI descriptor completion bit, increment the receive count, and wake up a block process if necessary.

Fig. 10 shows this processing time (t_8 in Fig. 8) as a function of message size. The step shaped curves clearly shows the impact of cache line (32 bytes) to the memory copy. After the NIC interrupt service routine, the OS returns to execute the interrupted program.

The upper thread starts from `VipRecvWait()` which just checks whether or not the descriptor at the head of the receive queue is completed. It has two stages. First it polls the completion bit for a number of times (50 000 times in the current implementation). Then, if polling timeouts, `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. 8). This conclude the whole receiving process. Table 2 summarizes the CPU cycles and time spent in each part.

5. Comparison between UDP and M-VIA

Based on the previous data and analysis, we now conduct a detailed comparison between UDP and M-VIA. Fig. 11

Table 2
M-VIA critical time breakdown using Pentium Celeron 400 MHz 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. 9	
t_6	DMA receive time	896	2.24
t_7	Interrupt starting time	1119	2.80
t_8	Interrupt receive processing	see Fig. 10	
t_9	Interrupt return time	600	1.50
t_{10}	VipRecvWait() processing (polling)	16	0.04

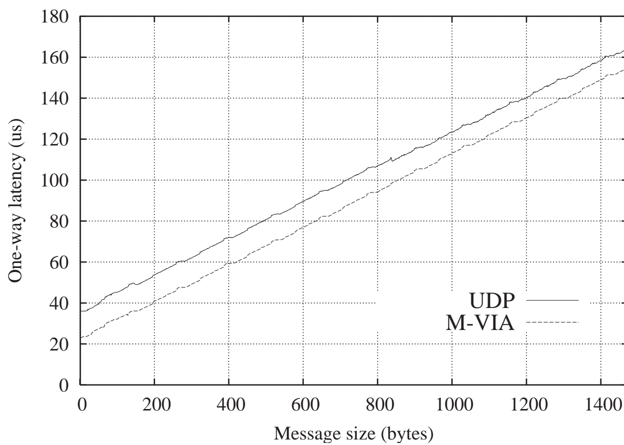

Fig. 11. UDP and M-VIA one-way latency.

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 3, we put each time interval into one of the six 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 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.

- First, M-VIA has less OS overhead than UDP. In send, M-VIA uses fast trap (0.33 μs) instead of normal system call (0.85 μs). In receive, M-VIA avoids soft interrupt by handling all receive functionalities in the interrupt handler. However, since M-VIA interrupt processing time (see Fig. 10) is variable, this may cause slow response time to other processes when receiving large packets. Note that the technique that M-VIA uses can also be applied to UDP. In addition, the main OS overhead is the interrupt overhead ($t_7 + t_9$ in Fig. 8). This overhead can only be eliminated by using programmable NICs which handle all receive functionalities locally. That is what a VIA capable NIC needs to do.
- Second, M-VIA spends 5.17 μs on protocol processing while UDP spends 15.76 μs. This big saving is mainly due to the reduction of the protocol layers. UDP, though simple, still needs to go through a number of layers, while M-VIA directly handles packets in the device layer. In addition, M-VIA eliminates the data copy in send (t_3 in Fig. 8 vs. $t_2 - t_5$ in Fig. 3) and uses more efficient polling in receive (t_{10} in Fig. 8 vs. t_{15} in Fig. 3). However, VIA achieves this at the expense of lacking 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 implies that M-VIA is the protocol specifically optimized for LANs.

6. Conclusion

In this paper, we performed an anatomical analysis of UDP and M-VIA. Our analysis clearly presented different parts of a send/receive communication used in parallel computing. 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 layer. 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 to reduce the latency. 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. Although this conclusion is not new, our paper described

Table 3
Critical time comparison between UDP and M-VIA for transferring a 1-byte message over Fast Ethernet using Pentium Celeron 400-MHz CPU

Categories	UDP/IP				M-VIA			
	Intervals in Fig. 3	Cycles	μs	%	Intervals in Fig. 8	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	35.53	$t_4 + t_5 + t_6$	5320	13.30	57.58
Total		14408	36.02			9240	23.10	

what exactly UDP and M-VIA differ and why M-VIA saves time over UDP.

References

[1] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.-K. Su, Myrinet: a gigabit-per-second local area network, IEEE Micro 15 (1) (January/February 1995) 29–36.

[2] P. Buonadomna, A. Geweke, D. Culler, An implementation and analysis of the virtual interface architecture, in: Proceedings of SC ’98, Orlando, FL, November 7–13 1998.

[3] D. Clark, V. Jacobson, J. Romkey, H. Salwen, An analysis of TCP processing overhead, IEEE Comm. Mag. 27 (6) (June 1989) 23–29.

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

[5] J. Hurwitz, W. Feng, End-to-end performance of 10-gigabit Ethernet on commodity systems, IEEE Micro 24 (1) (January/February 2004) 10–22.

[6] InfiniBand Trade Association, Infiniband architecture specification, release 1.0 [Online]. Available: http://www.infiniband.org.

[7] J. Kay, J. Pasquale, Profiling and reducing processing overheads in TCP/IP, IEEE/ACM Trans. Networking 4 (6) (December 1996) 817–828.

[8] LAM/MPI parallel computing [Online]. Available: http://www.lam-mpi.org/.

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

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

[11] J. Postel, User datagram protocol, RFC 768, August 1980.

[12] J. Postel, Transmission control protocol, RFC 793, September 1981.

[13] R. dos Santos, R. Bianchini, C.L. Amorim, A survey of messaging software issues and systems for Myrinet-based clusters, Parallel Distributed Comput. Practices, Special issue High-Performance Comput. Clusters 2 (2) (June 1999).

[14] X. Zhang, L. Bhuyan, W. Feng, Anatomy of UDP and M-VIA for cluster communication, UCR, Technical Report, May 2004 [Online]. Available: http://www.cs.ucr.edu/~xzhang/publications/tr-via-udp.pdf.

[15] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, {NFS} Version 4 Protocol, RFC, 3010, Dec, 2000.

[16] PVM, Parallel Virtual Machine, url:http://www.csm.ornl.gov/pvm/pvm_home.html.

Xiao Zhang received the B.E degree in Computer Science from the Shanghai Jiao Tong University, Shanghai, P.R. China, in 1991, and the M.S. degree in Computer Science from the University of California, Riverside, in 2001. He is currently working toward the Ph.D. degree at the University of California, Riverside.
His research interests include high-performance network, switch scheduling, high availability cluster, and distributed system.

Laxmi N. Bhuyan received the Ph.D. degree in Computer Engineering from the Wayne State University, Detroit, MI, in 1982.
He has been Professor of Computer Science and Engineering at the University of California, Riverside, since January 2001. Prior to that, he was a Professor of Computer Science at Texas A&M University, College Station (1989–2000) and Program Director of the Computer System Architecture Program at the National Science Foundation (1998–2000). He has also worked as a Consultant to Intel and HP labs. His research addresses multiprocessor architecture, network processors, Internet routers, web servers, parallel and distributed computing, and performance evaluation.
Dr. Bhuyan is a Fellow of the IEEE, the ACM and the AAAS. He has also been named as an ISI Highly Cited Researcher in Computer Science.

Dr. Wu-chun (Wu) Feng is a Technical Staff Member and Team Leader of Research & Development in Advanced Network Technology (RADIANT) in the Computer & Computational Sciences Division at Los Alamos National Laboratory and a fellow of the Los Alamos Computer Science Institute. His research interests span the areas of high-performance networking and computing.
He received a B.S. in Electrical & Computer Engineering and Music (Honors) and an M.S. in Computer Engineering from the Pennsylvania State University in 1988 and 1990, respectively. He earned a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1996. He is a Senior Member of the IEEE and a Member of the ACM.