High Performance User Level Protocol on Gigabit Ethernet

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Piyush Shivam, MSc.(Tech.) Information Systems

* * * * *

The Ohio State University

2002

Master's Examination Committee:

Dr. Dhabaleswar Panda, Adviser

Dr. Pete Wyckoff

Dr. P. Sadayappan

Approved by

Adviser

Department of Computer

and Information Science

© Copyright by

Piyush Shivam

2002

ABSTRACT

Modern interconnects like Myrinet and Gigabit Ethernet offer Gigabits per second (Gb/s) speeds which has put the onus of reducing the communication latency on messaging software. With the advent of programmable NICs, many aspects of protocol processing can be offloaded from kernel and user space to the NIC leaving the host processor to dedicate more cycles to the application. Many host-offload messaging systems exist for Myrinet; however, nothing similar exists for Gigabit Ethernet. In this thesis, we present a new Ethernet Message Passing (EMP) protocol, which not only implements OS-bypass but also supports zero-copy. This protocol has been implemented using the multi-CPU Alteon NICs for Gigabit Ethernet. Using the single CPU of the Alteon NIC, with the base protocol, we obtain latency of 24 μs, and a throughput of 880 Mb/s. However, the two CPUs of the Alteon NIC raise an open challenge whether performance of user-level protocols can be improved by taking advantage of a multi-CPU NIC. We highlight the intrinsic issues associated with such a challenge and explore different parallelization and pipelining schemes to enhance the performance of the basic EMP protocol. We consider four different alternatives: splitting the send path only (SO), splitting the receive path only (RO), splitting both the send and receive paths (SR), and assigning dedicated CPUs for send and receive (DSR).

The performance results indicate that parallelizing the receive path of the protocol can deliver 964 Mbps of bandwidth, close to the maximum achievable on Gigabit Ethernet. To the best of our knowledge, this is the first research in the literature to exploit the capabilities of multi-CPU NICs to improve the performance of user-level protocols. Results of this research demonstrate significant potential to design scalable and high performance clusters with Gigabit Ethernet.

High Performance User Level Protocol on Gigabit Ethernet

By

Piyush Shivam, M.S.

The Ohio State University, 2002

Dr. Dhabaleswar Panda, Adviser

Modern interconnects like Myrinet and Gigabit Ethernet offer Gigabits per second (Gb/s) speeds which has put the onus of reducing the communication latency on messaging software. With the advent of programmable NICs, many aspects of protocol processing can be offloaded from kernel and user space to the NIC leaving the host processor to dedicate more cycles to the application. Many host-offload messaging systems exist for Myrinet; however, nothing similar exists for Gigabit Ethernet. In this thesis, we present a new Ethernet Message Passing (EMP) protocol, which not only implements OS-bypass but also supports zero-copy. This protocol has been implemented using the multi-CPU Alteon NICs for Gigabit Ethernet. Using the single CPU of the Alteon NIC, with the base protocol, we obtain latency of 24 μs, and a throughput of 880 Mb/s. However, the two CPUs of the Alteon NIC raise an open challenge whether performance of user-level protocols can be improved by taking advantage of a multi-CPU NIC. We highlight the intrinsic issues associated with such a challenge and explore different parallelization and pipelining schemes to enhance

the performance of the basic EMP protocol. We consider four different alternatives: splitting the send path only (SO), splitting the receive path only (RO), splitting both the send and receive paths (SR), and assigning dedicated CPUs for send and receive (DSR).

The performance results indicate that parallelizing the receive path of the protocol can deliver 964 Mbps of bandwidth, close to the maximum achievable on Gigabit Ethernet. To the best of our knowledge, this is the first research in the literature to exploit the capabilities of multi-CPU NICs to improve the performance of user-level protocols. Results of this research demonstrate significant potential to design scalable and high performance clusters with Gigabit Ethernet.

This is dedicated to God and my parents

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for his guidance and support through the course of my graduate studies. I appreciate the time and effort he invested in steering my research work.

I am grateful to Dr. Pete Wyckoff for his continuous encouragement and involvement in this work. This research would not have been possible without his expertise.

I am thankful to Prof. P. Sadayappan for agreeing to serve on my Master's examination committee.

I would like to express a special thanks to my wife, Chris, who provided me with the much needed moral and emotional support.

I also acknowledge the student members of the Network-Based Computing Laboratory, particularly Darius Buntinas for his willingness to help with everything at all times.

This thesis was made possible by the financial support I received from various sources. I am indebted to the Department of Computer and Information Science for awarding me teaching assistantship in the first year of my M.S., and to Prof. Panda for supporting me as a research associate in the following year.

Finally, I would like to thank all my friends who made my stay at OSU enjoyable.

VITA

August 29, 1977	Born - Moradabad, India
1999	MSc.(Tech.) Information Systems, Birla Insititute of Technology and Science (BITS), Pilani, India.
Fall 2000 - Winter 2000	Graduate Teaching Associate, The Ohio State University.
Winter 2000 - Spring 2002	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Research Publications

P. Shivam, P. Wyckoff and D. Panda " EMP:Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing". *Proceedings of SC2001, Denver, Colorado, November '01.*

P. Shivam, P. Wyckoff and D. Panda " Can User Level Protocols Take Advantage of Multi-CPU NIC?". *IPDPS 2002, Fort Lauderdale, Florida, April '02.*

FIELDS OF STUDY

Major Field: Computer and Information Science

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Traditionally high performance applications have required the use of supercomputers. However, with the availability of high-speed commodity network technologies and GHz processors, one can think of connecting these high performance components over a network to run these high performance applications. Thus, one can obtain a cluster of workstations connected by System, Local or Wide Area Networks. Their cost effectiveness and incremental scalability make them a viable alternative to the more expensive supercomputers.

High-Performance computing on a cluster of workstations requires that the communication latency be as small as possible. The communication latency is primarily composed of two components: time spent in processing the message and the network latency (time on wire). Modern high speed interconnects such as Myrinet [4] and Gigabit Ethernet [24] have shifted the bottleneck in communication from the interconnect to the messaging software at the sending and receiving ends. In older systems, the processing of the messages by the kernel caused multiple copies and many context switches which increased the overall end-to-end latency.

This led to the development of user-level network protocols where the kernel was removed from the critical path of the message. This meant that the parts of the protocol or the entire protocol moved to the user space from the kernel space. Examples of some of the user-level protocols are: FM [17] for Myrinet, U-Net [26] for ATM and Fast Ethernet, GM [4] for Myrinet, etc. During the last few years, the designs and developments related to user-level protocols have been brought into an industry standard in terms of the *Virtual Interface Architecture* (VIA) [25]. Many hardware, software, and firmware implementations of VIA are currently available. Examples include M-VIA [15], GigaNet VIA [22], and FirmVIA [2]. An extension to the VIA interface is already included in the latest *InfiniBand Architecture* (IBA) [10] as the Verbs layer.

Another development which has taken place is the improvement of the Network Interface Card (NIC) technology. In the traditional architecture, the NIC would simply take the data from the host and put it on the interconnect. However, modern NICs have programmable processors and memory which makes them capable of sharing some of the message processing work with the host. Thus, the host can give more of its cycles to the application, enhancing application speedup. Under these developments, modern messaging systems are implemented outside the kernel and try to make use of available NIC processing power. In fact, the success of user-level protocols, VIA, and Verbs layer of the IBA relies heavily on the performance, programmability, and "intelligence" associated with modern network interface cards (NICs).

Moreover, as the processor technology is moving towards GHz speeds and network technology is moving towards 10–30 Gbits/sec [10] it is becoming increasingly important to exploit the capabilities of the NIC. In current generation systems, the PCI bus serves as a fundamental limitation to achieving better communication performance. This aspect is being alleviated in the IBA standard where Host Channel Adapters (HCAs) (equivalent to NICs) will be directly connected to the memory through the system bus. Thus, the design of the NICs and their interfaces are getting increased attention. As NIC processors are becoming more powerful and NICs are built with more memory, it is becoming easier to off-load significant portions of communication protocol processing to the NIC processor and thus, achieve improved communication performance.

1.1 Background and Related Work on Gigabit Ethernet

Since its inception in the 1970s, Ethernet has been an important networking protocol, with the highest installation base. Gigabit Ethernet builds on top of Ethernet but increases speed multiple times (Gb/s). Since this technology is fully compatible with Ethernet, one can use the existing Ethernet infrastructure to build gigabit per second networks. A Gigabit Ethernet infrastructure has the potential to be relatively inexpensive, assuming the industry continues Ethernet price trends and completes the transition from fiber to copper cabling.

Given so many benefits of Gigabit Ethernet it is imperative that there exist a low latency messaging system for Gigabit Ethernet just like GM [4], FM [17], AM2 [7], and others for Myrinet. However, no NIC-level, OS-bypass, Zero-copy messaging software exists on Gigabit Ethernet. All the efforts until now like the GAMMA [6],

MESH [5], MVIA [15], GigaE PM [23], and Bobnet [8] do not use the capabilities of the programmable NIC to offload protocol processing onto the NIC. Table 1.1 shows the classification of existing message passing systems.

	Programmable NIC	Non Programmable NIC
Myrinet	GM [4], FM [17], AM2 [7]	(n/a)
Gigabit Ethernet	EMP (This Work)	MVIA [15], GigaE-PM [23], Bobnet [8] MESH [5], GAMMA [6]

Table 1.1: Classification of existing message passing systems.

These Ethernet messaging systems use non-programmable NICs. Thus, no more processing can be offloaded onto the NIC than which has been already been burned into the card at design time. There was work done in the area of message fragmentation where the NIC firmware was modified to advertise a bigger MTU to IP which was then fragmented at the NIC [9]. The Arsenic project [19] extends a packet send and receive interface from the NIC to the user application. And a port of the ST protocol has been written which extends the NIC program to bypass the operating system [18]. However, these are not complete reliable NIC-based messaging systems. As indicated in [14] it is desirable to have most of the messaging overhead in the NIC, so that the processing power at the host can be used for applications.

1.2 Problem Statement and our Approach

As outlined earlier, there is a need for a high performance messaging layer for Gigabit Ethernet. To make this layer deliver low latency, high bandwidth and high host CPU utilization numbers, one needs to exploit the programmable NIC to the

maximum. One of the popular NICs for Gigabit Ethernet was developed by Alteon (now owned by Nortel Networks) which offered a unique architecture in terms of having a two-CPU core.

The aforementioned goal and the availability of such a novel NIC architecture led us to ask ourselves the following questions.

1. How best can we develop a very high performance user-level protocol on Gigabit Ethernet?

2. Can such a user-level protocol be better implemented by taking advantage of a multi-CPU NIC?

3. What are alternative strategies for parallelizing and pipelining of such user-level protocols with a two-CPU NIC, and what are the intrinsic issues?

4. How much performance benefit can be achieved with such parallelization and pipelining?

In this thesis we take on a challenge of designing, developing and implementing a zero-copy, OS-bypass, NIC-level messaging system for Gigabit Ethernet with Alteon NICs. We call this messaging layer EMP (Ethernet Message Passing). OS bypass means that the OS is removed from the critical path of the message. Data can be sent/received directly from/into the application without intermediate buffering making it a true zero-copy architecture. The protocol has support for retransmission and flow control and hence is reliable. All parts of the messaging system are implemented on the NIC, including message-based descriptor management, packetization, flow control and reliability. In fact, the host machine does not interfere with a transfer after ordering the NIC to start it. Using this messaging layer we obtain latency

of 24.31 μs, bandwidth of around 840 Mbps and excellent CPU utilization results. These results have been obtained with back to back connected hosts. We have done the experiments with the switch also and with that the latency increases by around 5-6 μs for small message sizes. The bandwidth remains unchanged though.

The basic EMP protocol was developed using only one of the two CPUs of the Alteon NIC. To answer our remaining questions we analyze, design, implement, and evaluate a parallel version of EMP with two-CPU Alteon NICs for Gigabit Ethernet.

We analyze the send and receive paths of the EMP messaging layer to determine the costs associated with the basic steps. Next, we analyze the challenges involved in parallelizing and/or pipelining user-level protocols for the two-CPU Alteon NIC. This leads to four alternative enhancements: splitting up the send path only (SO), splitting up the receive path only (RO), splitting both the send and receive paths (SR), and assigning dedicated CPUs for send and receive (DSR). We implement these strategies on our cluster testbed and evaluate their performance benefits.

The best results were obtained with the RO scheme for unidirectional traffic, giving a small message (10 bytes) latency of 22.62 μs and bandwidth of 964 Mbps. This is compared to the base case (EMP-single cpu) latency of 24.31 μs (a gain of 7.0%) and bandwidth of 840 Mbps. For large messages the latency improvement was around 8.3%. For bidirectional traffic the best results were achieved with the SO scheme where the total bandwidth peaked at 1100 Mbps as compared to 940 Mbps in the base case, a gain of 17%. These numbers were obtained with 933 MHz hosts connected back to back and we get similar percentage of improvement with slower 700 MHz hosts connected via packet engine switch.

Though this parallelization is done specifically for EMP, the results and design strategies we suggest are applicable to any other user-level protocol. To the best of our knowledge, this thesis documents the first attempt to not only develop such a messaging layer for Gigabit Ethernet but also to parallelize such a protocol with multi-CPU NICs.

1.3 Thesis Organization

The rest of the thesis is organized in the following manner. Chapter 2 describes the basic EMP and its development, along with the performance analysis of this messaging layer on the single CPU of the Alteon NIC. Chapter 3 outlines the parallelization of the basic EMP on the two CPUs of the Alteon NIC along with performance analysis. Finally, conclusions and future work are presented in Chapter 4.

CHAPTER 2

EMP (ETHERNET MESSAGE PASSING)

In this chapter we describe the design, implementation and the performance evaluation of our zero-copy, OS-bypass and NIC-driven messaging system, EMP. First we look at the Alteon NIC architecture, on which we develop our high performance messaging layer. This is followed by the detailed description of EMP.

2.1 Architectural overview of the Multi-CPU Alteon NIC

Alteon Web Systems, now owned by Nortel Networks, produced a Gigabit Ethernet network interface chipset based around a general purpose embedded microprocessor design which they called the Tigon2. It is novel because most Ethernet chipsets are fixed designs, using a standard descriptor-based host communication protocol. A fully programmable microprocessor design allows for much flexibility in the design of a communication system. This chipset was sold on boards by Alteon, and also was used in board designs by other companies, including Netgear and 3Com. When we speak of the "Alteon NIC" later in the thesis, it is understood that all of the implementations which use this chip are equivalent. Broadcom has a chip (5700) which implements the follow-on technology, Tigon3, which should be similar enough to allow use of our messaging environment on future gigabit Ethernet hardware.

The core of the Alteon (Figure 2.1)is the Tigon chip which is a 388-pin ASIC consisting of two MIPS-like microprocessors running at 88 MHz, an internal memory bus with interface to external SRAM, a 64-bit, 66 MHz PCI interface, and an interface to an external MAC. The chip also includes an instruction and data cache, and a small amount of fast per-CPU "scratchpad" memory. The instruction set used by the Tigon processors is essentially MIPS level 2 (as in the R4000), without some instructions which would go unused in a NIC application.

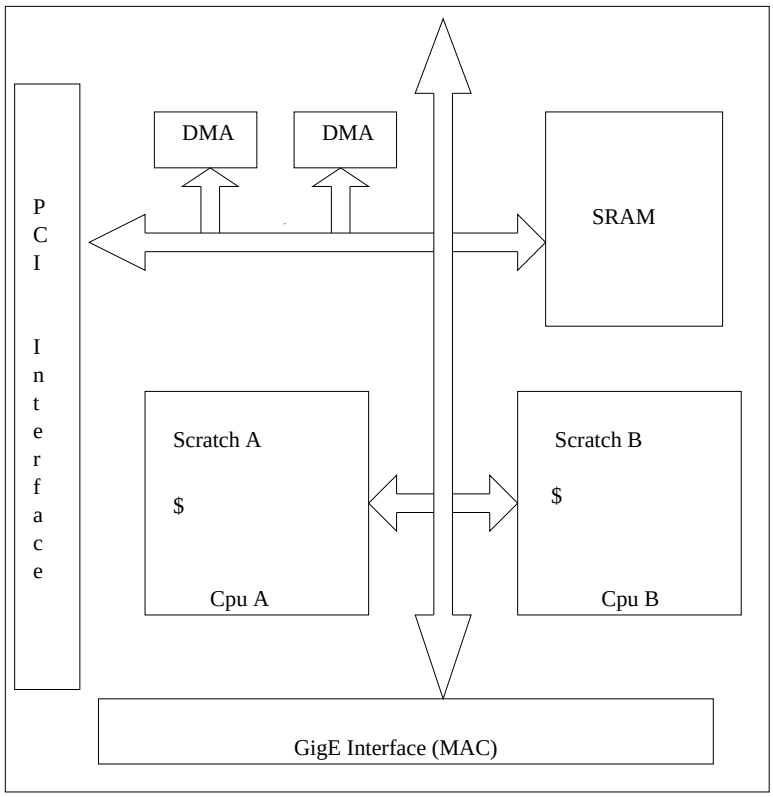

Figure 2.1: Alteon Architecture.

Hardware registers can be used by the processor cores to control the operation of other systems on the Tigon, including the PCI interface, a timer, two host DMA

engines, transmit and receive MAC FIFOs, and a DMA assist engine. Our particular cards have 512 kB of external SRAM, although implementations with more memory are available. The NIC exports a 4 kB PCI address space, revealing to the host: 1 kB of Tigon hardware registers, a fixed mapping of the lowest 1 kB of SRAM (including event-generating mailboxes), and a 2 kB memory "window" which can be positioned by the host to map into any section of the full 512 kB external SRAM.

The hardware provides a single semaphore which can be used to synchronize the two CPUs. Each CPU has its own register which it writes with any value to request ownership of the semaphore, then must loop until a read from the semaphore register is non-zero, indicating successful ownership. This is the only general locking mechanism; in particular, the memory system on the NIC does not support locked bus cycles.

The Tigon2 has many features useful for implementing event-driven execution (as opposed to interrupt-driven or threaded, for example). Two new instructions were added to facilitate fast event dispatch: `pri` selects the highest bit in a word subject to a mask, and `joff` jumps through a function table using that high bit. Each processor has a register which includes bits for each event that a processor might want to handle. These bits report hardware readiness, such as an arriving frame from the network, or a buffer low watermark condition. The event register also has bits which software can use to define its own events for handling from within the main dispatch loop.

2.2 Design Challenges

Our goal was to develop a design for a new firmware for the Tigon, and associated code running in the host, to facilitate OS-bypass high-throughput and low-latency message passing communications, with no buffer copies and no intervention by the

operating system. We reuse none of the original firmware written by Alteon, and know of no other original firmware designs, although quite a number of projects have made small modifications to the original Alteon firmware.

The following sections highlight some of the major challenges we faced in designing both the network protocol. Figure 2.2 shows the core components addressed in the following. The existing solution in the figure refers to the current messaging layer implementations on the Gigabit Ethernet.

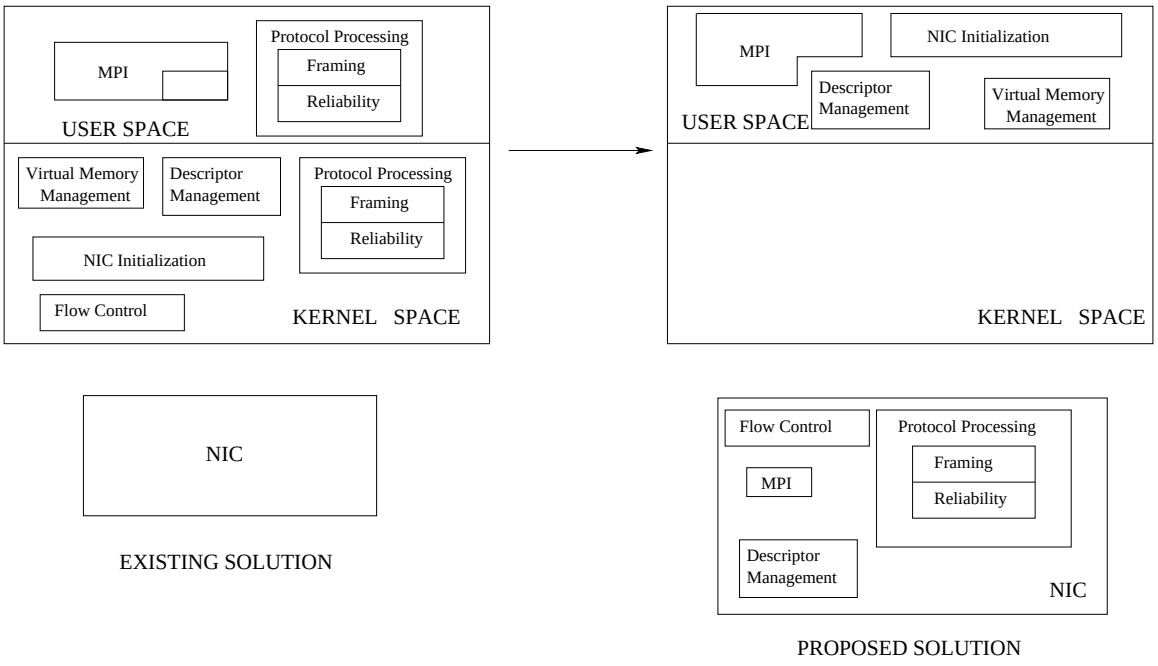

Figure 2.2: Our approach in the redistribution of the various components of the messaging system.

2.2.1 Protocol processing

One of the most important design challenges was to decide how to distribute the various functions related to protocol processing such as reliability, fragmentation, *etc.*: on the host, NIC, or shared in both? In EMP we have offloaded all the protocol processing to the NIC because of its unique processing capabilities (two 88 MHz MIPS processors). That is the reason for calling EMP NIC-driven.

Framing

Once the NIC has been informed that there is data to send, it retrieves the data from the host in preparation for moving the data to the network. Here, we had to decide whether we should buffer the data at the NIC while sending and receiving the message. We decided against it because buffering would add greatly to the system overhead. Instead, whenever the NIC has data to send, it pulls the data from the host one frame at a time and puts it on the wire. Similarly on the receiving side, if there is no pre-posted receive for the incoming frame, the frame is simply dropped to avoid buffering. MPI programs frequently arrange to pre-post all receives to avoid buffering by the messaging system, because users know that it hurts performance and that the buffering limit varies among implementations, thus our design caters to this type of usage. VIA [25] also imposes this condition in its design.

Reliability

EMP presents a reliable transport interface to the host. While designing the reliability of EMP, one of the design issues [3] we faced was the unit of acknowledgment (the number of frames to acknowledge at one time). We decided to acknowledge a collection of frames instead of a single frame (each message is made up of multiple

frames). There were two main reasons for that. The Ethernet frame size is small so acknowledging every frame would introduce a lot of overhead. Moreover, since we were developing our messaging layer for LAN clusters where the rate of frame loss is not as excessive as WAN, we could decide not to acknowledge every frame. EMP has the flexibility of configuring the unit of acknowledgment on the fly depending upon application and network configuration. This feature also allows us to research reliability issues in the future for different kinds of applications.

Flow control

In this Section we outline the flow control design for EMP. First we look at the drawbacks of the existing schemes. Then we look at some of the issues related with the flow control at NIC given our goals of low latency, high bandwidth and high host CPU utilization. We look at our solution and its advantages in the Section 2.3.

Limitations of the current schemes: The most common schemes in the literature [11] are the Credit based schemes and the Rate control based schemes [12]. In Credit based scheme an explicit VC (virtual circuit) is set up for each flow. Before forwarding any data cell over the link, the sender needs to receive credits for the VC from the receiver. Periodically, the receiver sends credits to the sender indicating the availability of the buffer space. The sender then is allowed to forward some data corresponding to the number of credits it has. Each send results in the current credit reducing by one. Each VC is associated with a fixed amount of buffer. This scheme is very difficult to implement with good performance given the limited resources of the NIC. When we allocate buffer for the VC it is difficult to come out with a near optimal allocation. There will be too much overhead in dealing with the buffer management

for each VC. Moreover, the limited memory of the NIC restricts the number of flows which can be achieved. The static allocation of buffers is also ruled out because of this limitation. Doing things dynamically further adds a lot of overhead at the NIC.

In Rate based scheme the exact rate of traffic for each VC is determined. Thus, it assumes that the rates for the VC sharing a link can be made to converge on sensible values. However, on very fast networks, the network load may change faster than the control system can react. Moreover, its very complex to determine all the parameters for rate control. Again the kind of processing available at the NIC precludes the implementation of this scheme.

Issues related with NIC-based flow control

1. Minimum overhead

   The main motivation for using high speed interconnects, programmable NICs and user level protocols is to achieve low latency, high bandwidth and high host CPU utilization. Incorporating the flow control in the basic state machine of the protocol causes an increase in overhead. This is due to the following factors.

   - The EMP protocol is implemented at the NIC and the NIC has limited resources (NIC CPU speed - 88MHz, limited buffers).
   - The extra work introduced due to flow control disturbs the critical path of send and receive actions.

   Any NIC-driven flow control scheme should cause minimum burden on the limited NIC resources and try to disturb the critical path of the protocol as little as possible. Once again, we stress that the reason for implementing the flow

control at the NIC is that we want to devote the host cycles for real application work as much as possible.

2. Effective criteria for flow control

   There are many factors on which we can base our flow control scheme. Before going in detail on this we briefly describe some of the basic steps involving these resources.

   - Host posts the send descriptor, a software resource limited by the amount of memory at the NIC. There are currently 200 send descriptors and 440 receive descriptors.
   - The NIC reads the posted descriptor and based on the size of the message it schedules DMA for the message using the DMA descriptors (a hardware resource). There are a total of 64 DMA descriptors.
   - The message gets DMAed in the NIC memory which is a total of 512kB. This includes all the data structures, descriptors and a variety of other things needed for implementing EMP at the NIC.
   - When the DMA gets completed the MAC engine of the network interface schedules MAC descriptors for the final transmission of the frames. There are a total of 256 MAC descriptors.

   Thus the following hardware resources can be overrun and hence result in the activation of flow control at any node.

   - The NIC memory space
   - The DMA descriptors

- The MAC descriptors

Coming out with the correct parameters is not trivial since the network behavior varies depending on the number of nodes, the burstiness of the traffic and the size of the messages in the network.

3. Current event mechanism of the Alteon NIC

   The way the current Alteon NIC is designed, it always does a send before a receive. This is something fixed in the hardware and cannot be changed as such. There are two kinds of events among many other events. One is the MAILBOX event which is set when a send is scheduled by the host. The second is the MAC_RX_COMP event which gets set when a frame arrives at the node. In the hardware, the MAILBOX event always gets processed earlier to the MAC_RX_COMP event. Thus, any control information to the sender does not get noticed till the sends are complete. This way many sends might proceed even when there is no buffer space at the receiver NIC.

   The Alteon NIC provides a provision for defining software events, which can be put anywhere in the event priority table. Thus one way of making sure that receives get noticed before all the sends are complete is to define the MAILBOX event as a software event and give it a lower priority than the MAC_RX_COMP event. This way all the receives will get processed before any sends. While this does not affect bandwidth measurements, this has the potential of increasing the latency numbers.

4. Flow control vs. retransmissions

The idea behind flow control is to avoid the receiver from getting flooded by the sender. This means that sender should not send the data if the receiver is full and wait for the receiver to free up some space.

This implies that on the sender side a regular check is required for all kinds of messages to make sure that sender does not send the data when the receiver does not have space. This will hinder the latency and bandwidth for cases when there is no need for flow control especially for small to medium sized messages. This fact motivated the design of EMP whereby sender is allowed to send to the receiver even when the receiver might not have enough buffer size to keep the incoming data. This poses the problem of receiver dropping data. However since EMP allows for retransmission of packets, we are able to solve this problem.

There is another problem however. Consider the case where the sender does a lot of retransmits before completely sending the data (that is if the retransmits also get dropped and again have to be retransmitted). In that case it might be helpful to moderate the rate of these retransmissions. This requires that there be additional checks for receiver and sender buffer sizes at regular intervals which will cause an overhead. Also, there will be the extra overhead of the delay introduced to slow down the rate of retransmissions. This total overhead is to be weighed against the overhead of possible multiple retransmissions for the same packet. We investigate this tradeoff in Section 2.4.2.

2.2.2 Virtual memory management

There are two mechanisms available by which data can be moved to the NIC to allow it to be sent to the network, or the reverse, as defined by the PCI specification.

The first is programmed input/output (PIO), where the host CPU explicitly issues a write instruction for each word in the message to be transferred. The second is direct memory access (DMA), where the NIC can move the data directly from the host memory without requiring the involvement of the host CPU. We quickly decided that PIO would require extremely high host overhead, and thus use DMA for the movement of all message data.

However, this decision led us to solve another problem. All widely used operating systems employ virtual memory as a mechanism of allowing applications to exceed physical memory constraints, and to implement shared libraries, and to provide a wide array of other handy features. Although an application on the host machine does not need to know about the actual physical layout of its data, the NIC is required to perform all its DMA transfers to and from the host using physical addresses. Some entity must translate from user virtual to PCI physical addresses for every page in the transfer.

One option would be to have the NIC maintain a copy of the page tables which the kernel and processor use to establish the mapping. If we devoted all of the memory on the NIC to page tables, we could map no more than 256 MB of process address space on the x86 architecture. This may be sufficient for some applications, but was discarded as being too constraining. We considered caching just the "active" parts of the page tables, meaning only those pages which are being used in send/receive messages are available on the NIC, again with some sort of kernel interaction. For instance, if the kernel decides that it must swap out an application page due to memory pressure, it must inform the NIC that its cached copy is invalid, and conversely if an incoming frame is destined for a user page which is not in core, the NIC must request

it from the kernel. Kernel virtual memory operation overhead, complexity, and again limited NIC memory all suggest that even a caching version would not perform well.

This led us to mandate that the user application specifically informs the NIC of the physical address of each page involved in the transfer. However, user space does not have access to this information which resides in the kernel. In the implementation section below we point out that it does not cost too much to use a system call to ask the kernel for the translations.

Applications which use the NIC with our firmware must ensure that the physical pages comprising a message, either for transmit or receive, stay resident at all times during their potential use by the NIC. Unix provides a system call, `mlock`, which selectively guarantees this for individual pages, or the variant, `mlockall`, which can be used to inform the operating system that all parts of the address space should remain in physical memory. We choose the latter method which requires only one system call during initialization and is appropriate for most memory-resident codes, but that technique precludes the use of swap space. The alternative of individual page mapping is potentially more expensive depending on the exact distribution of memory used for communication with the network, although the per-page mapping cost is less than a microsecond on our hardware.

2.2.3 Descriptor management

We use the term "descriptor" to refer to the pieces of information that describe message transfers, and which must pass between the host system and the NIC to initiate, check, and finalize sends and receives. Descriptors are usually small in comparison to the message payloads they represent.

To initiate a send, or to post a receive, the host must give to the NIC the following information: location and length of the message in the host address space, destination or source node, and the MPI-specified message tag.

This information is passed to the NIC by "posting" a descriptor. The NIC needs access to descriptor fields, which are generated in the host. It can get this information either by individually DMAing fields on demand from the host or it can keep a local copy of the descriptor in NIC memory. The tradeoff is between expensive DMA operations (1–3 μs per transfer) and the number of descriptors which can reside in the limited NIC memory. Too many accesses to descriptor fields via DMA will degrade performance immensely.

We found that the NIC indeed needs to access the fields of a descriptor at many stages during the lifetime of a send or receive message, thus it requires a copy of this descriptor locally and can not use DMA on demand to retrieve it with any reasonable performance. The host, on the other hand, fills descriptors and does not need to be concerned with them until message completion time, *i.e.,* at the conclusion of a possibly asynchronous operation.

Since we require a local copy on the NIC, and since the host will not frequently use the descriptor, we chose to allocate permanent storage for an array of descriptors in the NIC memory. The host accesses these using PIO, which is slow in bulk compared to DMA, but provides lower latency for small transfers. Details of our extended descriptor formats for transfers greater than a page are provided in the implementation sections below.

The one piece of information that the host needs from the NIC after an asynchronous send or receive has been initiated is, "Is it done yet?" Traditional Ethernet

devices communicate this by an interrupt to the operating system kernel, which is clearly inappropriate given our low latency goals. Other systems (*e*.g., VIA, Infiniband [10]) use a completion queue, which is a list describing the transfers that have completed. Our project goals state that no buffering will ever be performed, which implies that no unexpected messages will ever be received at any node, which reduces the concept of a completion queue to simply reporting statuses of transfers that the host explicitly initiated. Since the host has this knowledge, it can instead remember which descriptors it has posted, and explicitly poll them when it is interested in determining if the transfer has completed.

2.3 Implementation

Our firmware consists of 12,000 lines of C code, and another 100 lines of MIPS assembly. Roughly half of the code deals with the basic hardware functionality of the various Tigon subsystems, including DMA engines, MAC interface, link negotiation and control. The core of the code is an event loop written in three assembly instructions which tests for the next pending event and dispatches control to the appropriate handling routine. Details of our implementation of fundamental modules are provided in the following sections.

2.3.1 Protocol processing

A message is composed of multiple frames. A message is a user defined entity whereas the frame is defined by the hardware. The length of each frame is 1500 bytes, but could be larger if the particular Ethernet switch infrastructure provides support for 9000 byte "jumbo" frames. The first frame of the message contains a message header which embeds frame header. All frames contain a frame header, and the

first frame of a message also contains extra per-message information as shown in Figure 2.3.

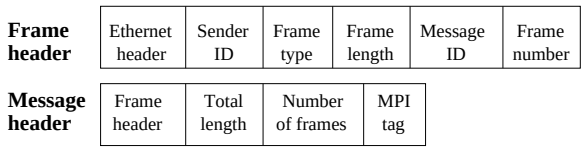

Figure 2.3: Two main headers used in EMP.

For each host we keep a small amount of information including expected next message ID, but we have no concept of a connection, and thus expect not to be limited in scalability by that. Other protocols generally arbitrate a unique connection between each pair of communicating hosts which preserves the relatively large state for that pairwise connection. The per-host state we keep is on each NIC in the cluster is: remote MAC address, send and receive sequence numbers (message ID).

For each message we keep a data structure describing the state of the transfer for just that message, which includes a pointer to the descriptor filled by the host. The limits to scalability of the system thus mirror the scalability limits inherent in each application, as represented by the number of outstanding messages it demands. Each of these message states is much larger than the per-host state mentioned above.

Once the NIC is informed that a message is there to send, it DMAs the data from the application space and prepends the message/frame header before sending it on the wire. As mentioned before we have something called a collection of frames. In the current implementation this number is 3. Thus, once 3 consecutive frames have been sent the timer for that collection starts. Any time the receiver receives

all the frames in that collection, the acknowledgment for that collection is sent. The acknowledgment consists of the frame header of one of the frames for that collection. Essentially it is the header of the last received frame of that collection. If the sender receives an acknowledgment it verifies whether all the frame collections have been acknowledged. If yes, then the host is informed that the message has been sent and it is removed from consideration by the NIC. If any collection times out then all the frames of that collection are resent. We have provided support for out of order frames too in EMP.

The sequence of operations involved in sending/receiving are shown in Figure 2.4. The USER (application space) posts a transmit or a receive. The data structures corresponding to that operation are updated (BOOKKEEPING). In case of transmit a DMA is done and the message is sent to the wire (PHY) as explained above. A timer corresponding to that transmit starts and a RETRANSMIT is performed if that timeout value is reached before the remote ACK arrives. Similarly, on the receive side when the frame arrives, a check is made if it is the last frame (BOOKKEEPING), in which case the receive is marked DONE and an acknowledgment is sent (SEND ACK).

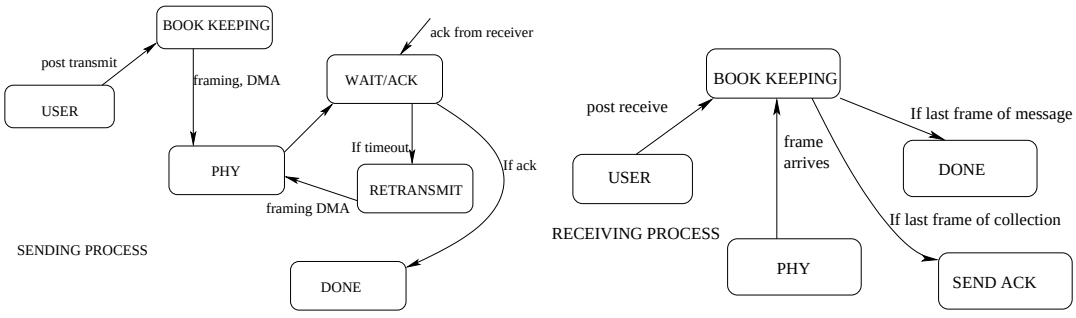

Figure 2.4: Processing flow for sending and receiving.

Flow control scheme for EMP: To determine the effective criteria for flow control we gathered the statistics of the three main parameters; the remaining buffer size, DMA descriptors and MAC descriptors for varied message sizes. We gathered these statistics after sending and receiving every packet on the sending and receiving side respectively. One limitation though was that, we did this with just two nodes which were connected back to back. We aim to improve on this in future. While gathering the statistics we realized that it is always the NIC buffer on the receiver side which gets overrun while sending huge messages at a very fast pace. The other parameters were always available in sufficient numbers. This led us to base our flow control scheme on the receiver buffer size.

The basic flow control scheme consists of the receiver sending the state of the receive buffers to the sender in a piggyback fashion along with the acknowledgment. This information is sent with every acknowledgment. The sender rate is determined based on this information received along with the acknowledgment sent by the receiver. This information is maintained for every receiver. If at any instant a particular receiver gets congested, only the rate of transmission for that receiver will be varied while allowing the other receiver flows to continue at the same rate. We also define two levels of control in this scheme. The acknowledgment might come with a feedback information which will tell the sender by how much to slow down. Thus, we can define the marks based on which the receiver might want to slow down the sender accordingly.

The advantages of this scheme are enumerated as follows.

- We are able to reuse the existing reliability infrastructure for sending flow control information.

- We are able to avoid the overhead associated with forming a new control frame for sending the receiver state to the sender.

- The acknowledgments are sent every so often and hence we are able to supply the latest state of the receiver to the sender at constant intervals without much overhead.

- We don't have to worry about the acknowledgment frame getting lost and hence the state information getting lost as the acknowledgment frames are sent every so often and hence the sender will always come to know the latest receiver buffer state.

When we implemented the above scheme we noticed that we were getting too much. This led us to look more carefully at our scheme and we discovered that some calculations (e.g. remaining buffer size) were taking too much time. One required the use of a division instruction for doing these calculation. However, the Alteon NIC does not provide such an instruction so we had simulated one in the software, using repeated subtraction. However, this was adding too much overhead. We avoided this by calculating the remaining buffer size using the bitwise operations.

2.3.2 Virtual memory and descriptors

As discussed in the design section, we translate addresses from virtual to physical on the host. This is done by the kernel and invoked from the user application (library) as an `ioctl` system call. This overhead is included in the performance results below.

The descriptors which are used to pass the information between the host and the NIC each require 20 bytes of memory, and we currently allocate static storage on the NIC for 640 of these to serve both message receives and sends. The physical address of

every page in a transfer must be passed to the NIC, which leads us to define multiple descriptor formats to include these addresses.

The first simply includes a single 8-byte address in the 20-byte descriptor and can describe a message up to a page in size. The second is built by chaining up to four descriptors which among them describe a message up to seven pages long (28 kB on x86). The descriptor format used for messages larger than this is similar to the short format in that only one address is given to the NIC, but this address is a pointer to a page full of addresses which the NIC uses indirectly to get the address of any single page in the transfer. In all these formats we always provide a 16-bit length field along with each page address which allows us to do scatter/gather transfers, so that if an application specifies a message which is not even contiguous in user memory (such as with an arbitrary MPI Datatype), the NIC can gather the individual chunks when preparing a message for the network, and the reverse when receiving a network frame.

A diagram of the descriptor layout is shown in Figure 2.5, where all the descriptors physically reside in NIC memory, but the host will take ownership of some when it needs to post a new send or receive. After writing the relevant information for the transfer, ownership is passed to the NIC (by modifying the "stat" field), which will then carry out the transfer and return ownership to the host.

2.3.3 Host library

A small library provides a simple interface to the functionality provided by the firmware. It includes calls to post transmits and receives, test and wait for messages to complete, and manage descriptors. Other functionality includes initialization of

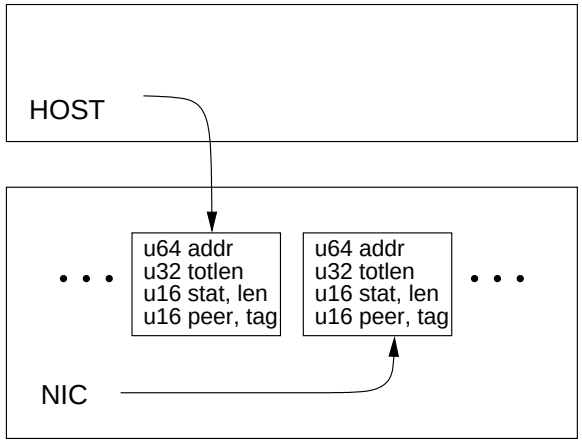

Figure 2.5: Some short-format descriptors in the NIC memory, showing ownership by either the HOST or the NIC.

the NIC, and loading a "route" table which gives an ordered list of host names and Ethernet hardware addresses which are used when generating Ethernet frames.

The steps involved in a short message transmission by the host are:

- Convert the user message address to a physical address.
- Get a new descriptor from the host-managed free pool.
- Fill the fields with address, length, destination, tag.
- Write the 20 bytes to NIC memory through the window in a PIO burst.
- Poke the NIC address of the descriptor in the transmit mailbox.
- Wait until the NIC acknowledges receipt of the descriptor.

Later, when the application must be sure that the message has been sent, it can poll the status field in the descriptor, then release the descriptor back to the free pool. (Each poll of a descriptor status takes 380 ns on our hardware, which is described

later.) On the receive side, the application must first pre-post an area into which a message will be received, following the same steps as for a transmission. When a message arrives which matches the designated source and tag (or wildcards), the NIC will place it directly in the pre-arranged message area and mark the status field of the descriptor. Frequently, applications will send to or receive from the same location multiple times (persistent transfers in MPI parlance). In this case, subsequent transfers are simplified and require only poking the NIC mailbox and waiting for acknowledgment.

2.3.4 Initialization

We facilitate the interaction of an arbitrary user application and the network interface card by a piece of code resident in the kernel. Our module is inserted into a running linux, or compiled statically, and provides services to applications through a character device interface. No operations beyond this module insertion require root privileges. The kernel module probes for a known PCI card with the Alteon chipset and initializes the basic PCI interface settings. Then, as each user application wants to use the NIC, it will open a character device `/dev/emp-alteon`. The kernel makes sure that only one user at a time can access the hardware during this open. Next the user application calls `mmap` to get a mapping to the PCI address space of the NIC. Reads and writes into this mapped page go directly to the NIC without further kernel intervention. The final piece of functionality provided by the kernel is a single ioctl which is used to convert a user address into a physical address.

2.4 Performance Evaluation

We compare basic networking performance with two other systems which are frequently used for MPI message passing. The base case for using Gigabit Ethernet is to use standard TCP/IP, which uses exactly the same Alteon NIC, but running the default firmware coupled with a TCP stack in the kernel. The second system is GM over Myrinet, which uses different hardware, but shares with EMP the NIC programmability features and the focus on message passing.

2.4.1 Experimental setup

We had two experimental setups. The first one had two dual 933 MHz Intel PIII systems built around the ServerWorks LE chipset which has a 64-bit 66 MHz PCI bus, and unmodified Linux version 2.4.2. The hosts in this setup were connected back to back (no switch) for all the tests. For tests with Myrinet we use the same setup with LANai 7.2 cards.

The second setup had two quad 700 MHz Intel PIII systems built around the ServerWorks LE chipset which has a 64-bit 66 MHz PCI bus, and unmodified Linux version 2.4.18. The hosts in this setup were connected via a packet engine switch for Gigabit Ethernet and a Myricom switch for Myrinet. For tests with Myrinet we use the same setup with LANai 9.2 cards.

For both the setups our Gigabit Ethernet NICs were Netgear 620 [16], which have 512 kB of memory.

All tests using Ethernet were performed with a maximum transfer unit (MTU) of 1500 bytes. The TCP tests used 64 kB socket buffers on both sides; more buffering did not increase performance.

2.4.2 Results and discussion

In this Section first we discuss the role of flow control in EMP and then we discuss the bandwidth and latency results followed by the host CPU utilization results.

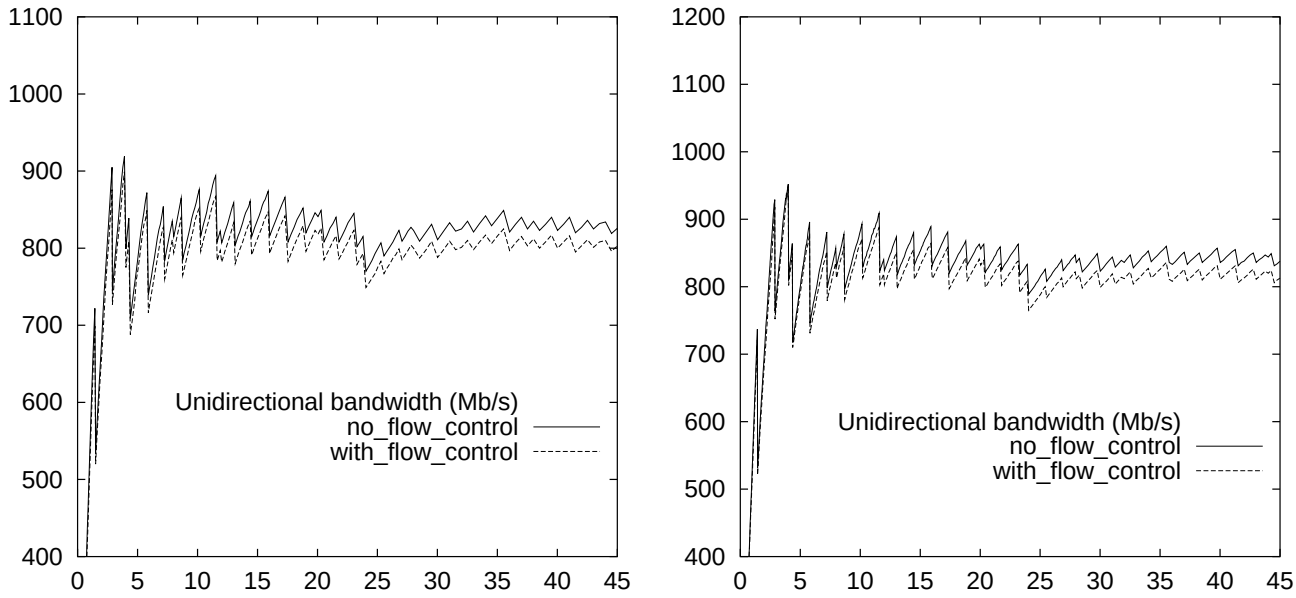

Figure 2.6: Comparison of bandwidth with flow control vs. no flow control. The x-axis is message size in Kb and y-axis is bw in Mb/s. These results are on 933 MHz hosts with no switch in between (left) and 700 MHz hosts with the switch (right).

Impact of Flow Control

Figure 2.6 shows the bandwidth measurements with the flow control and without the flow control for both the setups. In Section 2.2.1 we indicated that there is a tradeoff involved between flow control and multiple retransmissions. By looking at the figure we can conclude that EMP without the flow control but with multiple

retransmissions does better than EMP with the flow control and minimum retransmissions. Thus, for the remaining part of the evaluation, we consider cases without the flow control.

Latency and Bandwidth Measurements

Figure 2.7 shows plots for the latency and bandwidth for each of the three systems as a function of the message size. The latency is determined by halving the time to complete a ping-pong test, and the bandwidth is calculated from one-way sends with a trailing return acknowledgment. Each test is repeated 10 000 times to average the results.

The latency for TCP over Gigabit Ethernet is much higher than the other two systems due to the overhead of invoking the operating system for every message. GM and EMP both implement reliability, framing, and other aspects of messaging in the NIC. In the very small message range under 256 bytes, GM imposes a latency which is up to 10 μs lower than that of EMP, but the per-byte scaling of both EMP and GM is similar and message latencies are comparable.

Figure 2.8 shows the same plots with the switch in between and on the 700 MHz hosts. One can see that the switch and the slower host do not have effect on the peak bandwidth attained but it does affect the latency numbers for EMP by 5-6 μs for small message sizes. However, the latency in the presence of the switch does increase significantly for larger message sizes.

The theoretical wire speed of our test version of Myrinet is 1200 Mb/s (LANai 7.2) and 2000 Mb/s (LANai 9.2), and GM delivers very near to that limit (Figures 2.7 & 2.8). Gigabit Ethernet is limited at 1000 Mb/s, but EMP only delivers 80% of that limit. The packet size used in GM is 4 kB, and it exhibits the same periodic

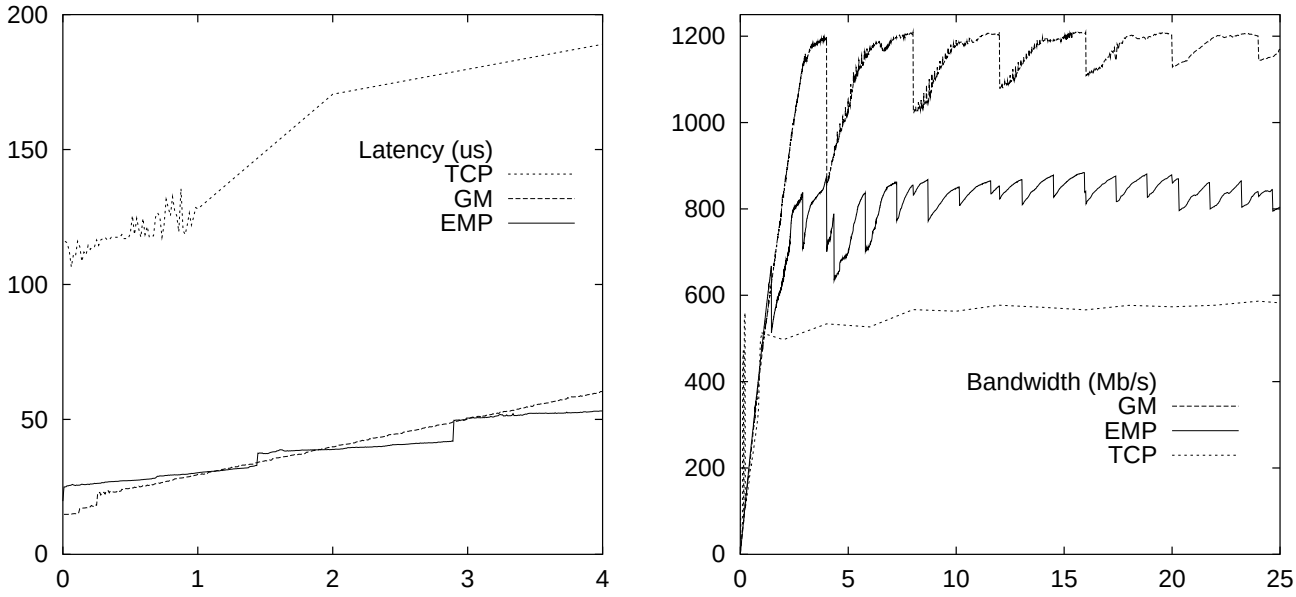

Figure 2.7: Latency and bandwidth comparisons. The quantity measured on the abscissa is message size in kilobytes, for all three plots. These results are on 933 MHz hosts with no switch in between.

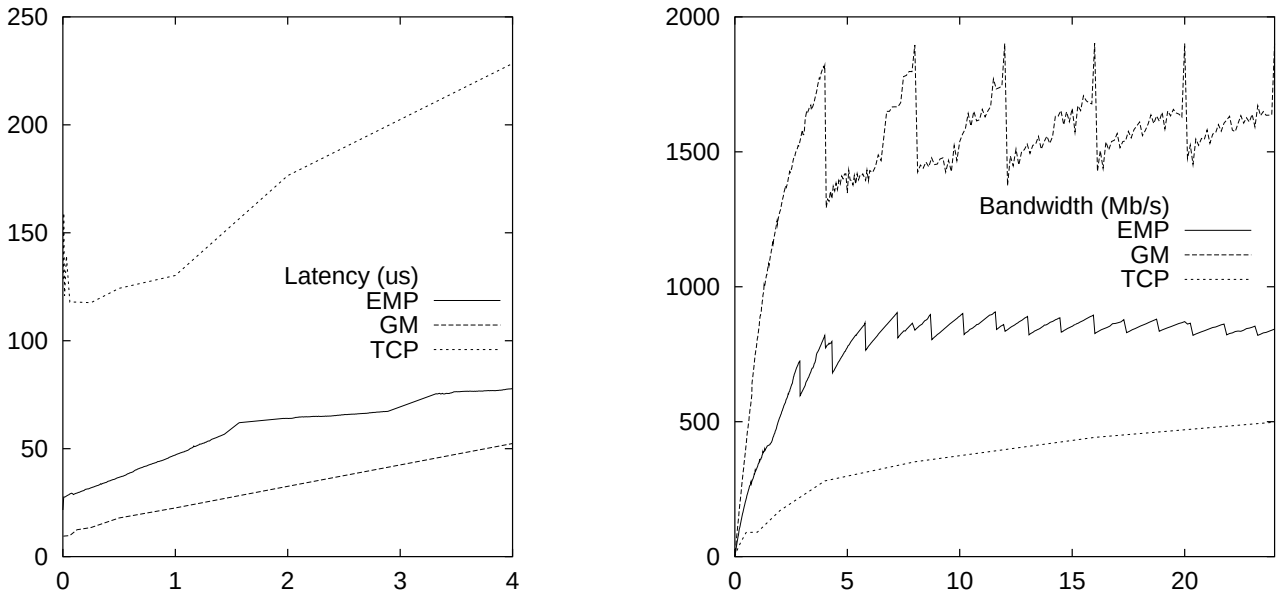

Figure 2.8: Latency and bandwidth comparisons. The quantity measured on the abscissa is message size in kilobytes, for all three plots. These results are on 700 MHz hosts with the switch in between.

performance drop as does EMP, since that coincides with the host page boundary. For EMP, the jumps at the frame size of 1500 bytes where the NIC must process an extra frame are clearly visible, and jumps at the page size of 4096 bytes where the NIC must start an extra DMA are also visible. The bandwidth of TCP never exceeds about 600 Mb/s.

CHAPTER 3

PARALLELIZATION OF EMP

In this chapter we outline the parallelization of the EMP protocol which we described in the previous chapter. We first explain in detail the breakup of the basic EMP protocol in different steps. Then we look at the new design challenges encountered for a NIC-based implementation of a parallelized messaging layer. This is followed by the possible alternatives for parallelizing the NIC-driven protocol. Next, we look at how best to exploit the NIC hardware capability for achieving parallelization and pipelining. Finally, we evaluate the different strategies with respect to their effectiveness while providing the results of our experiments.

3.1 EMP Protocol

In this section we provide the detailed steps of the basic EMP protocol [20] [21]. We first provide an overview of the basic steps. Next, we discuss these steps in detail. Finally, we present a timing analysis of these steps on a single-CPU NIC. The description of the steps and the timing analysis help us to understand the challenges involved in parallelizing the EMP protocol.

3.1.1 Basic steps at the sending and receiving side

Here we outline the basic steps happening at the sending side and the receiving side of the EMP protocol. The sending side performs the following steps.

1. Send bookkeeping: The process of preparing a frame before transmitting. Here we keep a record of all the information which is necessary for reliability purposes.

2. Transmission: This step involves the actual sending of data to the wire once all the bookkeeping is over.

3. Receive acknowledgment: This step happens when the receiving side acknowledges that it has received a certain number of frames.

Similarly, the receiving side performs the following steps.

1. Receive bookkeeping: The process of keeping track of incoming frames for reliability purposes and allowing for the acceptance of out of order frames.

2. Receive: Here the data is communicated to the host via DMA after the bookkeeping phase is over.

3. Send acknowledgment: Once the receiver has processed a known number of frames it sends an acknowledgment to the sender. This step is required for reliability.

These steps are described in detail below.

Send bookkeeping

Send bookkeeping refers to the operations which take place for preparing the frame for being sent. The bookkeeping operations can be outlined as:

- Handle posted transmit descriptor: This step is initiated by the host which operates asynchronously with the NIC. The introduction of each new transmit request leads to the rest of the operations. This operation takes place for every message.

- Message fragmentation: The host desires to send a message, which is a user-space entity corresponding to some size of the application's data structures. The NIC must fragment this into frames, which is a quantity defined by the underlying Ethernet hardware as the largest quantum of data which can be supported in the network, 1500 bytes in our system. Thus, the NIC determines how many frames will be necessary to send this message. The overhead incurred for large messages is more since they contain a larger number of frames. This implies that the bookkeeping effort will increase with increasing message size.

- Initialize transmission record: Each message which enters the transmit queue on the NIC is given a record in a NIC-resident table which keeps track of the state of that message including how many frames, a pointer to the host data, which frames have been sent, which have been acknowledged, the message recipient, and so on. The NIC prepares this structure for each message, then updates it as the message is processed through the various stages of transmission. This record is maintained for each message which is being transmitted.

Transmission

The steps involved in the transmission of the frames to the wire can be outlined as:

- DMA from HOST to the NIC: The NIC contains two DMA channels to transfer data between its local memory and the host memory. Managing these channels in order to keep them active can require significant resources of the internal processor. To help off-load some of these tasks from the internal processor, the "DMA Assist" state machine is used to perform the most time critical tasks. DMA descriptors are used by firmware to pass the relevant information about a DMA to the assist logic. These DMA descriptors reside in a small portion of the local memory and are organized into a ring structure. Once the bookkeeping steps for the frame are over, the DMA assist engine queues a request for data from the HOST. When the transfer completes, it automatically informs the MAC to send the frame. The transfer is made in the send buffer which is updated after each transfer. These set of operations take place for every frame and hence will take time for large message sizes.

- MAC to wire: The NIC uses MAC transmit descriptors to keep track of frames being sent to the serial Ethernet interface. The format of these descriptors is fixed to allow the hardware to directly reference the fields within the descriptors. The MAC is responsible for sending frames to the external network interface by reading the associated MAC transmit descriptor and the frame from the local memory buffer. Frames are sent only when a valid descriptor is ready and the send buffer indicates that the data is available. The send buffer is updated after the DMA is completed by the DMA assist engine which informs the MAC to start sending the data. There are 256 MAC transmit descriptors which can be used for transmitting data. This operation happens for every single frame

and each frame uses one MAC descriptor, hence the overhead incurred increases with increasing message size.

Send acknowledgment

This step, though it happens on the receiver, involves a combination of bookkeeping and transmission. The acknowledgment is sent as a single frame with some control information but no data. Hence the overhead involved in this step is not as large as that for any data frame. Moreover, this does not involve per-frame overhead because an acknowledgment is sent only for complete groups of frames.

Receive bookkeeping

The receive bookkeeping refers to the operations which need to be performed before the frame can be sent to the host. These operations are:

- Handle pre-posted receive descriptor: This step is initiated by the host for messages it expects to receive in future. Here the state information which is necessary for matching an incoming frame is stored at the NIC. In the current setup if a frame arrives and it does not find a matching pre-posted descriptor, it is simply dropped. This is done to avoid buffering at the NIC [20]. This step happens for every message.

- Classify frame: This step does multiple things. It looks at the header of each incoming frame and identifies whether it is a header frame, data frame, acknowledgment frame or negative acknowledgment. It also identifies the pre-posted receive to which the incoming frame belongs by going through all the pre-posted records. In the process it also identifies if the frame has already arrived and,

if so, drops it. In case a data frame arrives arrives before the corresponding header frame, it is dropped as well. Classify frame is performed for every frame and hence the overhead per message increases with increasing message size.

- Receive frame: Once the frame has been correctly identified in the previous step, the information in the frame header is stored in the receive data structures for reliability and other bookkeeping purposes. Receive frame also initiates the DMA of the incoming frame data after filling in the receive data structures with fields including message sequence number, frame sequence number, etc. After this step the frame is ready to be DMAed to the host. Receive frame is also done for every frame and the overhead increases as the message size increases.

Receiving

The step comprising the actual receiving process involves the following operations:

- Wire to MAC: Similar to transmission, the NIC uses MAC receive descriptors to keep track of frames being received from the serial Ethernet interface. Again, the format of these descriptors is fixed like the transmit descriptors to allow the hardware to directly reference the fields within the descriptors. Error conditions are monitored during frame reception and reported to the firmware through the status word located in the descriptors. Before the data is given to the NIC the 32 bit CRC is verified and noted in the status word.

- NIC to HOST: Here the DMA Assist engine comes into play exactly like in the transmit case. The only difference is that the DMA assist engine operates in the reverse direction, moving the data to the host instead of to the NIC.

Receive acknowledgment

Once the sender knows that the receiver has successfully received the frames it can release the resources related to the sent data. In this step there is no data to be DMAed to the host and hence the overhead is lower than that of receiving any data frame. Receive acknowledgment introduces only minimal per-frame overhead, again, because acknowledgment is a process which applies only to groups of frames [20].

3.1.2 Timing analysis of the messaging layer components

We did a complete time profiling of our protocol to find out how much time is spent in each of the steps. As we discussed, each of the steps consists of one or more operations. But for the sake of clarity we are showing only the timings for the major steps. Table 3.1 shows the analysis. These numbers correspond to two dual 933 MHz Intel PIII systems, built around the ServerWorks LE chipset which has a 64-bit 66 MHz PCI bus, and using unmodified Linux 2.4.2.

Receive bookkeeping is more expensive than send bookkeeping because while sending, the frames are sent in order but they can arrive out of order on the receive side (due to switch dropping and reordering of frames). So extra effort is needed per frame to accept these out of order frames and put them in the correct order. Moreover, since the frames can be out of order, for each frame one has to go through all the pre-posted records to see if it belongs to any of them which also contributes to a large overhead.

3.2 Challenges in Taking Advantage of a Multi-CPU NIC

In order to take advantage of a multi-CPU NIC, the basic steps in sending and receiving need to be distributed across the processors. However these steps need to

Operation	Time (μs)
Send bookkeeping Handle posted transmit descriptor Message fragmentation Initialize transmission record	5.25
Transmission DMA from host to NIC Queue frame to MAC	5.50
Receive acknowledgment	5.75
Recv bookkeeping Handle posted receive descriptor Classify frame Receive frame	10.50
Receiving Receive frame from MAC DMA from NIC to host	2.75
Send acknowledgment	2.50

Table 3.1: Timing analysis for the major functional operations.

share some common state information at some point in the execution. Typically, NICs have very limited hardware resources to assist in this operation without introducing additional overhead. Here, we take a critical look at the limitations of the Alteon NIC and the potential alternatives for achieving our objective.

3.2.1 NIC constraints

As indicated in Section 2.1, the Alteon NIC does not provide hardware support for concurrency. There is only one lock, hence fine-grained parallelism is expensive. Coarse-grained parallelism is inappropriate for the kind of operations performed at the NIC, due to its limited resources. Shared resources (MAC, DMA) do not have hardware support for concurrency, and use the only available lock, thus overloading that single semaphore.

3.2.2 Achieving concurrency

Consider a simple unidirectional flow scenario for reliable communication. While the send is happening on the sender side, a receive is also actually taking place (e.g. receive acknowledgments) on the same side. Thus the process of sending data (or acknowledgments) can be overlapped with the process of receiving data (or acknowledgments) on different processors on the sending side and/or the receiving side. During this overlap there are scenarios where the state information needs to be shared between the conceptual sending steps and the receiving steps.

To minimize sharing of such state one may keep separate data structures for send bookkeeping and receive bookkeeping so that both the operations can happen in parallel without needing to access the other's data structure for state information. However, this cannot be guaranteed for every case. Thus, even while the data structures might be different for send and receive processing, there will still be a need for some form of mechanism for sharing information.

One way to solve this problem would be to share the data structures across the CPUs. However this would mean that each access to the data structure requires synchronization. This would be very expensive since the data structures are accessed frequently, and each access would lead to synchronization overhead.

To reduce the synchronization overhead, the bookkeeping data structures can be fine-grained so that locking one data structure does not lead to halting of other operations which can proceed using other unrelated data structures.

One may also accomplish synchronization by allocating a special region in the NIC SRAM where one CPU would write the data needed by the other CPU, which would then read the common data from there. This would help in communicating the

common data across the CPUs without causing the overhead related to the sharing of data structures. There is an overhead involved in this operation, of course, but this overhead is only explicitly generated when there is a need for data sharing between the CPUs. This might be a better option because if we allow the send and receive data structures to be shared there will be overhead for each access to them even when there is no need simply because it happens to be shared data. One problem with this solution, though, could be the contention for the common area.

3.2.3 Exploiting pipelining and parallelization

Amdahl's law states that the speed-up achievable on a parallel computer can be significantly limited by the existence of a small fraction of inherently sequential code which cannot be parallelized. In any reliable network protocol there will be a lot of steps which have to be executed sequentially. In fact, serially constrained operations become the norm. As an example, in transmission, before the frame can be sent, one has to attach the frame header and perform other bookkeeping operations for reliability purposes. This puts a limit on the amount of work which can be scheduled in parallel. This limitation forces us to think about the underlying implementation and make appropriate changes so that we can perform the maximum number of operations in parallel. In addition to parallelization, pipelining can also be exploited, where the operations happen one after another but not in parallel. In the previous example, if the bookkeeping steps for a frame happen on one processor and the actual transmission on another it will be an example of pipelining, because for the same frame both these steps cannot happen at the same time. Bookkeeping and transmission can happen in parallel but for different frames, hence must be categorized as pipelining as

opposed to parallelism. In this thesis, we explore both pipelining and parallelization to enhance the performance of user-level protocols with multi-CPU NICs.

3.3 Schemes for Parallelization and Pipelining

In this section, we propose and analyze alternative schemes to enhance the performance of the EMP protocol with the support of a two-CPU NIC. The basic approach was to distribute the major steps of send and receive paths to achieve a balance of work on the two processors. This break-up was done with the goal of achieving pipelining or parallelism—whichever would be possible depending on the implementation. We tried to achieve the latter as much as possible but were limited by the inherent sequentiality of the protocol in many cases.

We analyzed the send path and the receive path for parallelization based on our timing analysis and recognized the following four alternatives.

- SO: The send path only is split up across the NIC CPUs.
- RO: The receive path only is split up across the NIC CPUs.
- DSR: The send path and receive path have dedicated processors to themselves.
- SR: Both send and receive path are split up.

For each of these alternatives, we illustrate how different components (steps) are distributed over two processors at both the sending and receiving sides. We compare our schemes with the base case scheme where all the sending-side components happen on the same CPU, as do the receive-side operations. CPU B is not used.

3.3.1 SO

The split up of the send path in SO happens as shown in Table 3.2. Here, we are aiming to achieve pipelining by running the bookkeeping phase of a later message with the transmission phase of an earlier message for a unidirectional flow. The idea is to have another message ready for transmission by processor A while the previous message is actually being transmitted by B. There is some parallelism also happening at the receiver between 'send ack' (2.50 μs) and a part of receiving(2.75 μs). The receive path for SO remains the same as in the base case. One needs to distinguish the difference between the receive path and receive side. The receive path is made up of receive bookkeeping and actual receiving (DMA). The assignment of send acknowledgment on the receiver is a part of the SO scheme since send ack involves steps which are used in sending and not receiving.

3.3.2 RO

The split up of functions in RO happens as shown in Table 3.2. Here, we are able to achieve true parallelism. The send ack (2.50 μs) happens in parallel with the receive dma (2.75 μs) and a part of receive bookkeeping (4.25 μs). The split-up of receive bookkeeping helps in achieving pipelining also. We are able to achieve a very good balance of functions on the receiving side. The send path remains the same as in the base case. Again, similar to the receive path scenario one needs to distinguish between the send path and sending side. The sending side has a receive step happening which is a part of the receive path and hence happening as in the base case as well.

Send	**cpu A**	(μs)	**cpu B**	(μs)
SO	send bookkeep	5.25	transmission	5.50
	recv ack	5.75		
RO	send bookkeep	5.25		
	transmission	5.50		
	recv ack	5.75		
DSR	send bookkeep	5.25	recv ack	3.25
	transmission	5.50		
	recv ack	2.50		
SR	send bookkeep	5.25	transmission	5.50
	recv ack	5.75		
Recv	**cpu A**	(μs)	**cpu B**	(μs)
SO	recv bookkeep	6.25	send ack	2.50
	recv frame	4.25		
	receiving	2.75		
RO	recv bookkeep	6.25	recv frame	4.25
	send ack	2.50	receiving	2.75
DSR	send ack	2.50	recv bookkeep	6.25
			recv frame	4.25
			receiving	2.75
SR	recv bookkeep	6.25	recv frame	4.25
			receiving	2.75
			send ack	2.50

Table 3.2: Function distribution (unidirectional).

3.3.3 DSR

In this case, we are dedicating one CPU each for the send path and the receive path on the sending as well as receiving side. This helps us to achieve an almost complete split of the send and receive paths. The receive acknowledgment step is split on the sending side because a part of it needs to update the send data structures and hence it is scheduled at the send processor. The functions are distributed as shown in Table 3.2.

3.3.4 SR

Here we combine the optimized send path and receive path together to see if we can benefit from the overall optimization of the protocol. It is a combination of SO and RO as depicted in Table 3.2. This is an attempt to extract the maximum benefit by putting together the individually optimized send and receive paths. We hope to gain from the benefits of pipelining on the send side and parallelization on the receive side.

3.4 Exploiting the NIC Hardware Capability

To solve the problem of synchronization we allocated a special common area in the NIC SRAM through which the CPUs can communicate common data. The benefits of such an approach were discussed in Section 3.2.

We developed a pair of calls, `spin_lock` and `spin_unlock`, which are used to gain exclusive access for protected code regions. We would have preferred to have multiple points of synchronization to implement object-specific locking, but the hardware provides exactly one point for inter-CPU synchronization through a semaphore.

Thus accesses to protected regions become potentially very expensive due to high contention for this single lock.

The other communication mechanism we used was to set bits in the event register of each processor. These calls use spin locks to guarantee exclusive access to the event register, whereby one CPU sets a bit in the event register of the other. The second CPU will notice this event in its main priority-based dispatch loop, clear the bit, and process the event. We used an "edge-triggered" model, and guarantee that events do not get lost by using the lock and by having the setting processor check to make sure that the bit is clear first.

Running two processors simultaneously puts more load on the memory system in the NIC. We attempted to alleviate this pressure somewhat by moving frequently used variables to the processor-private "scratchpad" memory area in each CPU. This small region (16 kB on cpu A, 8 kB on cpu B) also has faster access times, so we put frequently-called functions there too. Important functions that are used by both processors are replicated into both scratchpads. Source code annotations and a special linker script are used to position the functions in the various memory areas.

3.5 Performance Evaluation

3.5.1 Experimental setup

We had two experimental setups. The first one had two dual 933 MHz Intel PIII systems built around the ServerWorks LE chipset which has a 64-bit 66 MHz PCI bus, and unmodified Linux version 2.4.2. The hosts in this setup were connected back to back (no switch) for all the tests.

The second setup had two quad 700 MHz Intel PIII systems built around the ServerWorks LE chipset which has a 64-bit 66 MHz PCI bus, and unmodified Linux version 2.4.18. The hosts in this setup were connected via a packet engine switch.

For both the setups our Gigabit Ethernet NICs were Netgear 620 [16], which have 512 kB of memory.

All tests using Ethernet were performed with a maximum transfer unit (MTU) of 1500 bytes.

3.5.2 Results and discussion

In this Section we analyze the results derived from the alternatives discussed so far. We tested each of our alternatives for unidirectional as well as bidirectional flows. For unidirectional flow, we evaluated latency as well as bandwidth. For bidirectional flow, we evaluated bandwidth. We get better performance than the base case (single CPU per NIC) by using at least one alternative in each of the cases.

In all the alternatives the gain achieved due to pipelining/parallelism is offset to some extent by the overhead involved in switching control between CPUs. This happens because during the execution of the protocol one CPU might come across a task which is to be scheduled on the other processor. Hence, there is an extra overhead involved in this communication. This overhead is different for the various alternatives depending on how the components have been distributed across the CPUs.

Unidirectional Traffic

The latency is determined by halving the time to complete a single ping-pong test. The "ping" side posts two descriptors: one for receive, then one for transmit, then a busy-wait loop is entered until both actions are finished by the NIC. Meanwhile the

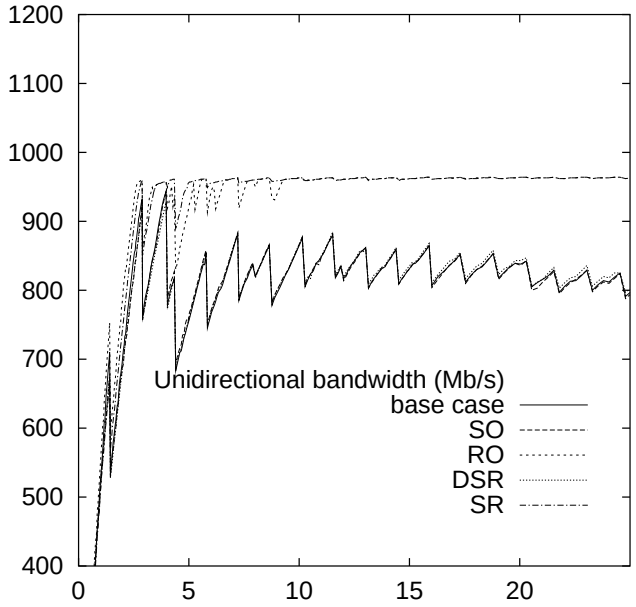

Figure 3.1: Bandwidth comparisons for unidirectional traffic. The x-axis is indicates message size in kilobytes. The y-axis shows bandwidth in Mb/sec. These results are on 933 MHz hosts with no switch in between.

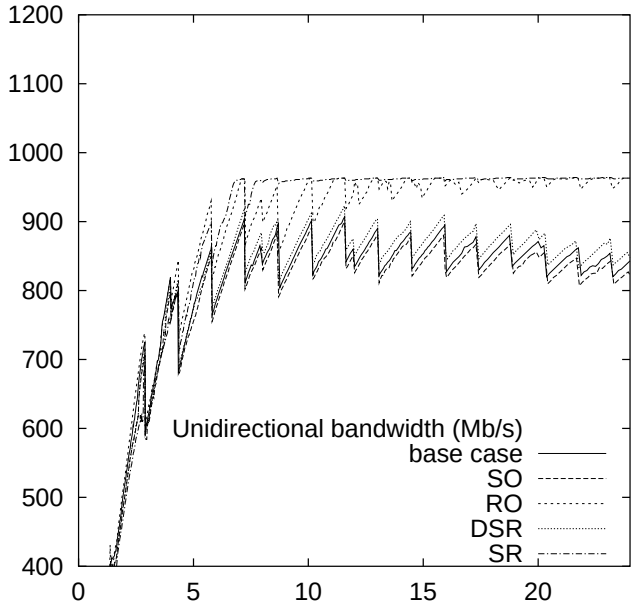

Figure 3.2: Bandwidth comparisons for unidirectional traffic with the switch. The x-axis is indicates message size in kilobytes. The y-axis shows bandwidth in Mb/sec. These results are on 700 MHz hosts with the switch in between.

"pong" side posts a receive descriptor, waits for the message to arrive, then posts and waits for transmission of the return message. This entire process is run in a loop of 10 000 iterations from which an average round-trip time is produced, then divided by two to estimate one-way latency.

The trend observed for latency numbers with the switch and slower hosts is the same as with no switch and faster hosts. The only difference is that, because of the switch delay the absolute latency readings are higher in the measurements done with the switch.

The unidirectional throughput is calculated from one-way sends with a trailing return acknowledgment. The user-level receive code posts as many receive descriptors as possible (about 400), and continually waits for messages to come in and posts new receives as slots become available. The transmit side posts two transmit descriptors so that the NIC will always have something ready to send, and loops waiting for one of the sends to complete then immediately posts another to take its place. Each transmit is known to have completed because the receiving NIC generates an acknowledgment message which signals the sending NIC to inform the host that the message has arrived. This is iterated 10 000 times to generate a good average.

Like the latency, the unidirectional throughput results have been presented with and without the switch in between. The switch or the host speed has little or no effect on the bandwidth measurements.

SO: The unidirectional bandwidth (Figures 3.1 & 3.2) is the same as in the base case. To analyze this case we need to consider the factors which are speeding up execution and the factors which are impeding it. On the sending side the benefit is

obtained due to pipelining (when send bookkeeping and transmit work one after the other) and parallelism (ack receive happens at the same time as transmit).

However the gains are offset by two factors. First, the overhead in inter-CPU communication. Next, while comparing the receive path and the send path, we can see that the receive path has more overhead than the send side. The sending side cannot send at any rate since it will swamp the receiver. It waits for the acknowledgment from the receiving side for a certain number of messages (two in our case) before it sends out more messages. Thus whatever speedup which can be gained due to pipelining or parallelism is limited by the reception of acknowledgments from the receiver which is again dependent on receive processing. Since the receive processing is happening in the same way as the base case except for a very small amount of parallelism (which is offset by the inter-CPU communication overhead), the pipelining/parallelization does not demonstrate much benefit for the SO case.

The pipelining benefits will not be seen for a latency test because we are timing only one message. However, because of inter-CPU communication there will be some degradation of latency. By looking at Figure 3.3 we can conclude that latency does not degrade much even for large message sizes. For a 10-byte message we see a latency of 24.52 μs, which was marginally higher than the base case latency of 24.31 μs, a degradation of less than one percent. These results are with 933 MHz hosts and no switch. Even with the switch in between, the latency shows a degradation of less than one percent though the absolute numbers are higher due to the switch delay (Figure 3.4).

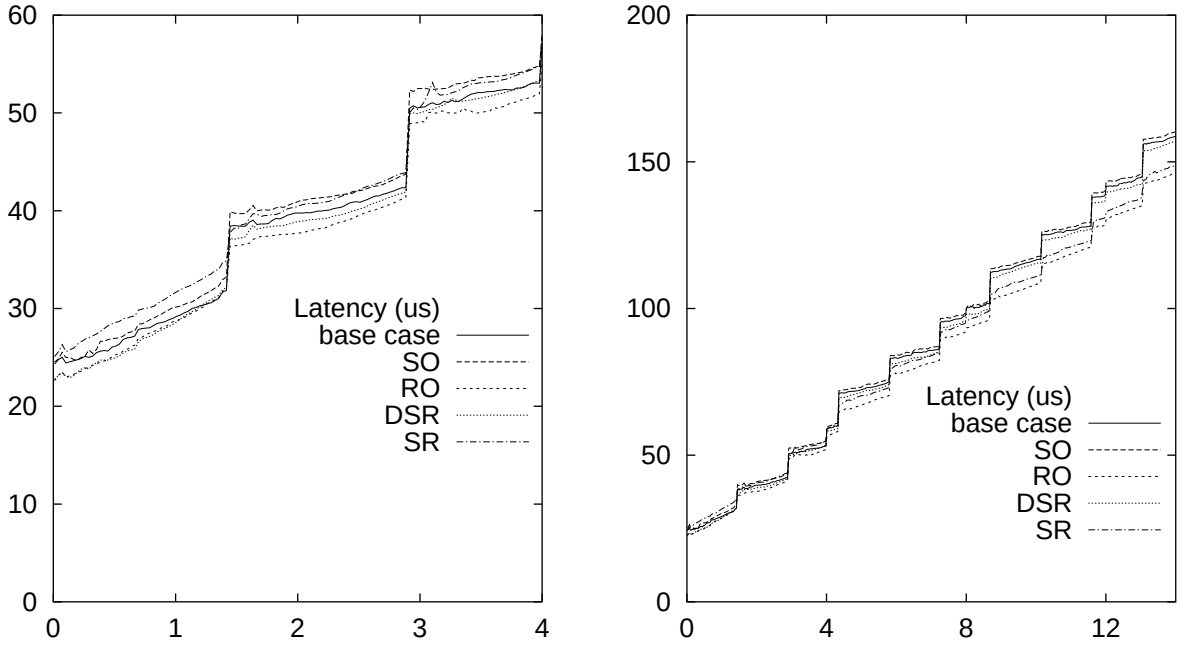

Figure 3.3: Latency comparisons for small and large message sizes with unidirectional traffic. The x-axis is indicates message size in kilobytes. The y-axis shows latency in microseconds. These results are on 933 MHz hosts with no switch in between.

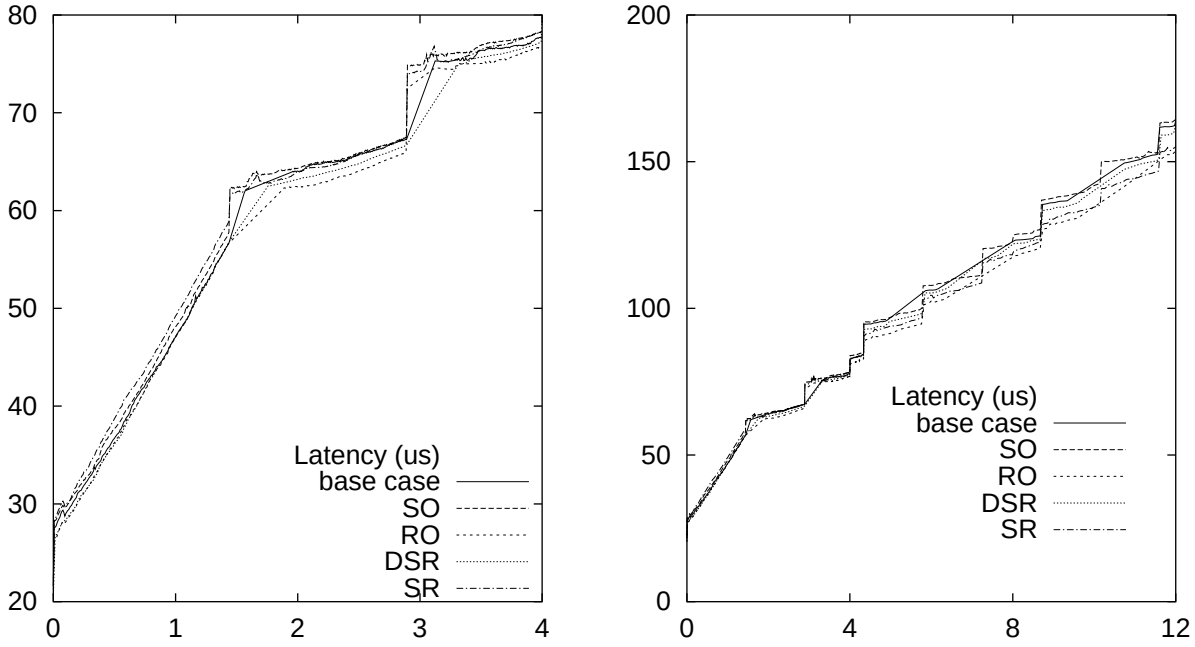

Figure 3.4: Latency comparisons for small and large message sizes with unidirectional traffic with the switch. The x-axis is indicates message size in kilobytes. The y-axis shows latency in microseconds. These results are on 700 MHz hosts with the switch in between.

RO: The unidirectional bandwidth (Figures 3.1 & 3.2) is much better than the base case. In fact it reaches up to 99.78% of the theoretical throughput limit on Gigabit Ethernet (taking into account the required preamble and inter-frame gap and our protocol headers). Factors which speed up the execution are:

- On the receiving side, the benefit is obtained due to parallelization of send acknowledgment on CPU A and receive bookkeeping and receive DMA on CPU B.

- The distribution of jobs on the receiving side is well balanced, resulting in both the CPUs being occupied most of the time.

Since the receive side, which is more demanding of processor cycles than is the send side, has been parallelized effectively we can achieve almost the maximum possible bandwidth. This implies that the receive side is the bottleneck which is also confirmed by our results from the SO case where we left the receive side unaltered and did not achieve any benefits even though we had pipelining and parallelism on the send side.

By looking at Figures 3.3 & 3.4 we can observe that even the latency improves for RO parallelism, both at small and large message sizes. For a 10-byte messages we obtained a latency of 22.62 μs which is an improvement over the base case latency of 24.31 μs, a gain of about 7%. For a message size of 14 kB we were able to achieve a latency improvement of about 8.3%, indicating that the rate of latency improvement increases with increasing message size. With the switch in between and a much slower host, the gains are around 6% and 7% respectively.

By this we can conclude that whatever overhead is involved in inter-CPU communication is more than offset by the parallelism between receive bookkeeping, receive DMA, and send acknowledgment.

DSR: The unidirectional bandwidth (Figures 3.1 & 3.2) is marginally better than the base case. Here the scenario is very similar to the SO case with the receive side being the bottleneck. However, since on the receive side we do schedule send acknowledgment to happen on a different CPU, we are able to see the marginal improvement in bandwidth numbers. The improvement is marginal because send acknowledgment is only a very small portion of the entire receive side processing.

By looking at Figures 3.3 & 3.4 we can conclude that latency also benefits with this approach though the benefit is marginal. For a 10-byte message, the latency is 22.76 μs which is a 6.4% improvement over the base case latency. With the slower hosts and the switch, the improvement is around 4%.

SR: This alternative gives the best unidirectional bandwidth. One is able to achieve almost complete utilization of Gigabit Ethernet's bandwidth. The results are very similar to the RO case but one can see the benefits of pipelining/parallelizing the send path also in the SR case (Figures 3.1 & 3.2). If we look at Figures 3.3 & 3.4 we can see that with increasing message sizes the latency also goes on reducing like in the RO case though the improvement is not as much as in the RO case.

Bidirectional Traffic

Bidirectional throughput is calculated in a manner similar to the unidirectional throughput, except both sides are busy sending to each other. After the startup

Send	**cpu A**	**(μs)**	**cpu B**	**(μs)**
SO	send bookkeep	5.25	transmission	5.50
	recv ack	5.75	send ack	2.50
	recv bookkeep	6.25		
	recv frame	4.25		
	receiving	2.75		
RO	send bookkeep	5.25	recv frame	4.25
	transmission	5.50	receiving	2.75
	recv ack	5.75		
	recv bookkeep	6.25		
	send ack	2.50		
DSR	send bookkeep	5.25	recv bookkeep	6.25
	transmission	5.50	recv frame	4.25
	recv ack	2.50	receiving	2.75
	send ack	2.50	recv ack	3.25
SR	send bookkeep	5.25	transmission	5.50
	recv bookkeep	6.25	recv frame	4.25
	recv ack	5.75	receiving	2.75
			send ack	2.50

Recv	**cpu A**	**(μs)**	**cpu B**	**(μs)**
SO	send bookkeep	5.25	transmission	5.50
	recv ack	5.75	send ack	2.50
	recv bookkeep	6.25		
	recv frame	4.25		
	receiving	2.75		
RO	send bookkeep	5.25	recv frame	4.25
	transmission	5.50	receiving	2.75
	recv ack	5.75		
	recv bookkeep	6.25		
	send ack	2.50		
DSR	send bookkeep	5.25	recv bookkeep	6.25
	transmission	5.50	recv frame	4.25
	recv ack	2.50	receiving	2.75
	send ack	2.50	recv ack	3.25
SR	send bookkeep	5.25	transmission	5.50
	recv bookkeep	6.25	recv frame	4.25
	recv ack	5.75	receiving	2.75
			send ack	2.50

Table 3.3: Function distribution (bidirectional).

pre-posting of many receive descriptors, the timer is started on one side. Then two messages are initiated at each side, and a main loop is iterated 10 000 times which consists of four operations: wait for the oldest transmit to complete, wait for the oldest receive to complete, post another transmit, post another receive. Using one application rather than two on each host ensures that we do not suffer from operating system scheduler decisions. Testing in this alternative manner gives the same results, although longer averages are necessary due to burstiness induced by context switches.

As we will see in the following discussion we obtain very good benefits for all the schemes compared to the base case for the bidirectional traffic. However, the absolute numbers with the second platform (700 MHz with the switch) are lower because of the packet engine switches, which shows that the bandwidth can vary depending on the switch in between. The slower host has no effect on the bandwidth because the entire protocol is NIC-driven and it can be seen that even with a slower host we get the same readings as with the 933 MHz hosts when we connect them back to back (Figure 3.7).

Bidirectional traffic is more complex than unidirectional traffic. In order to understand the benefits of our schemes for bidirectional traffic, let us analyze the distribution of the basic steps for these cases. These distributions are shown in Table 3.3.

SO: For bidirectional traffic, the distribution of steps is as shown in Table 3.3. This is not a different implementation but just the adjustment of steps from the unidirectional case when there is traffic in both the directions. The same is true for all other parallelization alternatives.

The bidirectional bandwidth (Figures 3.5 & 3.6) shows considerable improvement over the base case. This is happening because both the CPUs have more functions to perform in parallel (Table 3.3). Thus, the gain obtained more than offsets the inter-CPU communication overhead.

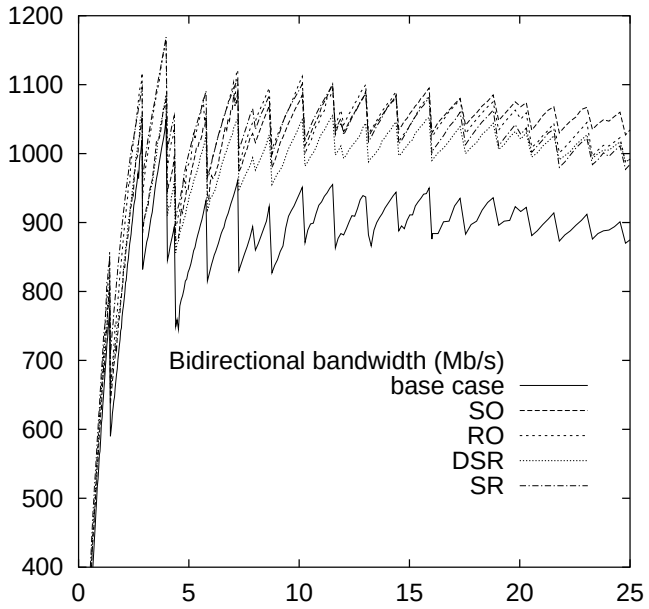

Figure 3.5: Bandwidth comparisons for bidirectional traffic. The x-axis indicates message size in KBytes. The y-axis shows bandwidth in Mb/sec. These results are on 933 MHz hosts with no switch in between.

RO: The bidirectional bandwidth (Figures 3.5 & 3.6) shows considerable improvement over the base case. This is happening because both the CPUs again are kept busier, and hence operate more in parallel (Table 3.3), once again offsetting the communication overhead. As we approach large message sizes the bandwidth drops below SO. This may be attributed to the following factors:

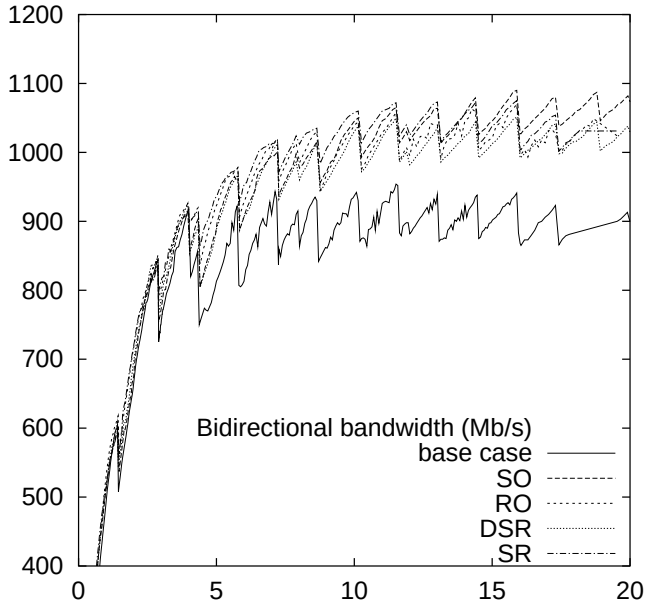

Figure 3.6: Bandwidth comparisons for bidirectional traffic with switch. The x-axis indicates message size in KBytes. The y-axis shows bandwidth in Mb/sec. These results are on 700 MHz hosts with the switch in between.

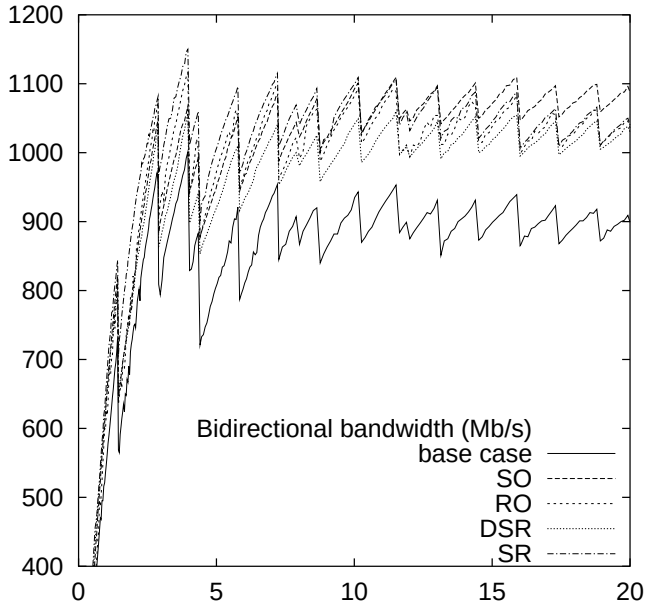

Figure 3.7: Bandwidth comparisons for bidirectional traffic with switch. The x-axis indicates message size in KBytes. The y-axis shows bandwidth in Mb/sec. These results are on 700 MHz hosts with no switch in between.

- In the RO case every frame causes communication overhead as compared with the SO case where only every third frame (acknowledgment group size) causes communication overhead.

- The amount of work which can happen in parallel on CPU B is less than that available in the SO case.

However, in Figure 3.5 the RO bandwidth numbers for medium-sized messages is larger than the SO case. This may be because for small messages there will be fewer frames and hence less communication overhead while receiving as compared to large message sizes. Also, the ratio of work distribution on A and B is better for RO as compared to SO. These factors combine to give RO better bandwidth numbers for medium-sized messages.

DSR: The bidirectional bandwidth (Figures 3.5 & 3.6) shows considerable improvement over the base case as well, for the same reasons as in the previous two strategies. However the gain is not as much as the RO scheme for medium-sized messages because the receive bottleneck offsets the gain obtained by the parallelism whereas in the RO case the gains due to distribution of receive path offsets the the inter-CPU communication overhead. However, for large messages (more frames) this communication overhead starts to affect the RO case and it starts dropping. The SO scheme outperforms all other alternatives, showing that for large messages the send path pipelining begins to have a positive impact on the bandwidth and overcomes the effect of the receive bottleneck. This is corroborated by the reduction in bandwidth in the RO case at larger message sizes indicating that splitting the receive path introduces a lot more overhead in the bidirectional case for large messages.

SR: The bidirectional bandwidth (Figures 3.5 & 3.6) shows maximum improvement over all the cases up till medium sized messages. However, it begins to drop after a certain message size. This happens because initially the gain obtained by parallelism offsets the inter-CPU communication.

Since we have combined the optimized paths of SO and RO case we are able to schedule maximum number of steps in parallel. Hence we derive the best performance initially. However, since there are more number of steps which are happening in parallel, the inter-CPU communication is also the maximum among all the alternatives. As the message size starts increasing this negates the gain obtained due to parallelism. For this reason the performance drops below all the other options for very large message sizes.

CHAPTER 4

CONCLUSIONS AND FUTURE WORK

In this chapter we present the summary of the entire thesis in the following manner. First we bring out the main conclusions and then we look at the current work being done in this area. This is finally followed by future works.

4.1 Conclusions

The objective of this thesis was to produce a reliable, low latency and high throughput, NIC-driven messaging system for Gigabit Ethernet. We came across numerous design challenges for accomplishing this which were overcome by redistributing the various functions of the messaging system in a novel way. We moved all the protocol related functions to the NIC and utilized the processing capabilities of the Alteon NIC, removing the kernel altogether from the critical path. All the work of moving the multiple frames of the message, ensuring reliability, demultiplexing incoming frames, and addressing are performed by the NIC. This resulted in our base protocol, EMP.

Given the fact that the Alteon NIC had two CPUs we showed how to take advantage of the multi-CPU NIC. Using EMP protocol (valid for single-CPU NIC) we

analyzed different alternatives to parallelize and pipeline different steps of the communication operation. We had four alternative enhancements: splitting up the send path only (SO), splitting up the receive path only (RO), splitting both the send and receive paths (SR), and assigning dedicated CPUs for send and receive (DSR).

Our study shows that parallelizing the receive path can deliver maximum benefits for unidirectional latency and bandwidth. In fact, this scheme allows us to reach the theoretical throughput of the medium, something which has not been accomplished before. Similarly, dedicated assignment of send and receive functionalities to different CPUs delivers very good bidirectional bandwidth. Though the strategies and results have been suggested for EMP, they are applicable to any user-level protocol.

The NIC handles all queuing, framing, and reliability details asynchronously, freeing the host to perform useful work. We obtained a latency of 24 μs for 4 byte message and a throughput of 880 Mb/s utilizing the single CPU of the multi-CPU Alteon NIC. With the parallelized receive path we obtained a latency of around 22 μs and bandwidth of 964 Mbps. These results were obtained with the 933 MHz hosts connected back to back. With the slower hosts (700 MHz) and a packet engine switch we obtained a latency of 28 μs for 4 byte message and bandwidth of 880 Mbps for the basic EMP protocol. The parallelized version (RO) gave a latency of around 26 μs and bandwidth of 964 Mbps.

Our results suggest that EMP, in particular EMP parallelized on the Alteon NIC holds tremendous potential for high performance applications on Gigabit Ethernet.

4.2 Current work

EMP is a lower layer which has given us excellent latency and bandwidth results. Since the parallel applications are written using higher level programming models (Sockets, MPI, DSM) we are in the process of providing higher layer implementations on top of EMP.

We have already provided an MPI implementation on top of EMP for writing MPI programs. Our initial results have shown excellent benefits, which are even comparable to GM results. We plan to bring out these results in a publication soon.

We have also implemented sockets over EMP which gives us excellent benefits compared to the native sockets API on TCP. This allows the existing TCP applications to run using the high performance EMP protocol without incurring the overhead of TCP. For more details one may refer [1].

4.3 Future work

As a result of our investigations in developing a high performance user-level protocol for Gigabit Ethernet, we have determined multiple promising paths for future study. Currently the distribution of steps in the parallelized implementation of EMP happens at compile time. We would like to produce a truly dynamic event scheduling system, where the next available event is handled by either processor when it becomes free.

The study so far has been done for point to point communication. We plan to expand this for collective communication operations also. We would like to predict if doing the collective communication operations at the NIC, given its multiple CPUs

and other resources, we can transport good benefits to the application. It would be interesting to see if the role of flow control becomes significant in this scenario.

We have shown in our analysis that the flow control for EMP is not useful for two node case. We intend to investigate the role of our flow control scheme further with bigger systems.

Our success with the multi-CPU implementation of EMP has encouraged us to predict the NICs of the future. We are working on developing a protocol simulator for Gigabit Ethernet where we can experiment with different parameters like number of CPUs at the NIC, the speed of each CPU, etc. to come out with the optimal NIC design.

BIBLIOGRAPHY

[1] P. Balaji, P. Shivam, P. Wyckoff, and D. Panda. High performance user level sockets over gigabit ethernet. In *Proceedings of Cluster2002*, June 2002.

[2] M. Banikazemi, V. Moorthy, L. Herger, D. Panda, and B. Abali. Efficient Virtual Interface Architecture support for the IBM SP switch-connected NT clusters. In *IPDPS*, May 2000.

[3] R. Bhoedjang, K. Verstoep, T. Ruhl, H. Bal, and R. Hofman. Evaluating design alternatives for reliable communication on high-speed networks. In *Proceedings of ASPLOS-9*, November 2000.

[4] N. Boden, D. Cohen, and R. Felderman. Myrinet: a gigabit per second local-area network. *IEEE Micro*, 15(1):29, February 1995.

[5] M. Boosten, R. W. Dobinson, and P. D. V van der Stok. MESH: Messaging and scheduling for fine-grain parallel processing on commodity platforms. In *Proceedings of PDPTA*, June 1999.

[6] G. Chiola and G. Ciaccio. GAMMA, http://www.disi.unige.it/project/gamma.

[7] B. Chun, A. Mainwaring, and D. Culler. Virtual network transport protocols for Myrinet. Technical Report CSD-98-988, UC Berkeley, 29 1998.

[8] C. Csanady and P. Wyckoff. Bobnet: High-performance message passing for commodity networking components. In *Proceedings of PDCN*, December 1998.

[9] P. Gilfeather and T. Underwood. Fragmentation and high performance IP. In *CAC Workshop*, October 2000.

[10] Infiniband. http://www.infinibandta.org.

[11] Srinivasan Keshav. Flow control in high-speed networks with long delays. In *Proceedings of INET*, 1992.

[12] H. T. Kung and R. Morris. Credit-based flow control for ATM networks. In *IEEE Network Magazine*, March 1995.

[13] William F. Lawry, Christopher R. Wilson, and Arthur B. Maccabe. OS bypass implementation benchmark. http://www.cs.unm.edu/~maccabe/SSL, 2001.

[14] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of ISCA*, June 1997.

[15] MVIA. http://www.nersc.gov/research/FTG/via, 1998.

[16] Netgear. http://www.netgear.com/adapters_main.asp.

[17] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet, 1995.

[18] Pekka Pietikainen. Hardware acceleration of Scheduled Transfer Protocol. http://oss.sgi.com/projects/stp.

[19] Ian Pratt and Keir Fraser. Arsenic: a user-accessible gigabit ethernet interface. In *Proceedings of Infocom*, April 2001.

[20] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proceedings of SC01*, November 2001.

[21] P. Shivam, P. Wyckoff, and D. Panda. Can user level protocols take advantage of multi-CPU NICs? In *Proceedings of IPDPS*, April 2002.

[22] E. Speight, H. Abdel-Shafi, and J. Bennett. Realizing the performance potential of a virtual interface architecture. In *Proceedings of the International Conference on Supercomputing*, June 1999.

[23] S. Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi, and Y. Ishikawa. High performance communication using a gigabit ethernet. Technical Report TR-98003, Real World Computing Partnership, 1998.

[24] TechFest. http://www.techfest.com/networking/lan/ethernet2.htm.

[25] VI. http://www.viarch.org, 1998.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.