Can User Level Protocols Take Advantage of Multi-CPU NICs?

Piyush Shivam
Dept. of Comp. & Info. Sci.
The Ohio State University
Columbus, OH 43210
shivam@cis.ohio-state.edu

Pete Wyckoff
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Dhabaleswar Panda
Dept. of Comp. & Info. Sci.
The Ohio State University
Columbus, OH 43210
panda@cis.ohio-state.edu

Longer version of paper in IPDPS02. Revised 02 Feb 02

Abstract

Modern high speed interconnects such as Myrinet and Gigabit Ethernet have shifted the bottleneck in communication from the interconnect to the messaging software at the sending and receiving ends. The development of user-level protocols and their implementations on smart and programmable network interface cards (NICs) have been alleviating this communication bottleneck. Most of the user-level protocols developed so far have been based on single-CPU NICs. One of the more popular current generation Gigabit Ethernet NICs includes two CPUs, though. This raises an open challenge whether performance of user-level protocols can be improved by taking advantage of a multi-CPU NIC. In this paper, we analyze the intrinsic issues associated with such a challenge and explore different parallelization and pipelining schemes to enhance the performance of our earlier developed EMP protocol for single-CPU Alteon NICs. Four different strategies are proposed and implemented on our testbed. Performance evaluation results indicate that parallelizing the receive path of the protocol can deliver 964 Mbps of bandwidth, close to the maximum achievable on Gigabit Ethernet. This scheme also delivers up to 8% improvement in latency for a range of message sizes. Parallelizing the send path leads to 17% improvement in bidirectional bandwidth. 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.

Keywords: *Gigabit ethernet, message passing, OS bypass, user-level network protocol, parallel protocol design*

1. Introduction

High-performance computing on a cluster of workstations requires that the communication latency be as small as possible.[1] 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 [2] and Gigabit Ethernet [11] have shifted the bottleneck in communication from the interconnect to the messaging software at the sending and receiving ends. In earlier generation protocols, the processing of the messages by the kernel used to cause multiple copies and many context switches. Thus, the communication latency was quite high. Over the years, researchers and developers of communication subsystems detected this bottleneck and this led to the development of user-level network protocols. Examples of some of the user-level protocols are: FM [7] for Myrinet, U-Net [14] for ATM and Fast Ethernet, GM [2] for Myrinet, our recent work on EMP [9] for Gigabit Ethernet, 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) [13]. Many hardware, software, and firmware implementations of VIA are currently available. Examples include M-VIA [4], GigaNet VIA [10], and FirmVIA [1]. An extension to the VIA interface is already included in the latest *InfiniBand Architecture* (IBA) [3] as the Verbs layer.

It is to be noted that 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). 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.

As processor technology is moving towards gigahertz speeds and network technology is moving towards 10–30

[1] This research is supported by a grant from Sandia National Labs (contract number 12652 dated 31 Aug 2000).

Gbits/sec [3] it is becoming increasingly important to exploit the capabilities of the NIC to achieve the best possible communication performance. 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.

Most of the older and current generation NICs support only one processor. Thus, to the best of our knowledge, all user-level communication protocols including VIA implementations have been centered around single-CPU NICs. One popular current generation NIC design is the two-CPU core from Alteon [6] for Gigabit Ethernet. This leads to the following interesting challenges:

1. Can user-level protocols be better implemented by taking advantage of a multi-CPU NIC?

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

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

In this paper, we analyze, design, implement, and evaluate a parallel version of the user-level protocol layer with two-CPU Alteon NICs for Gigabit Ethernet. We enhance the *Ethernet Message Passing* (EMP) [9] protocol which we have recently developed for Gigabit Ethernet using Alteon NICs. EMP was developed by taking into account only one of the two available CPUs in the NIC. To the best of our knowledge, this paper documents the first attempt to parallelize user-level protocols on modern interconnects with multi-CPU NICs.

In this paper, first 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 with 933 MHz Intel PIII systems 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 us and bandwidth of 964 Mbps. This is compared to the base case latency of 24.31 us (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%.

The paper is organized as follows. Section 2 provides an overview of the multiprocessor support provided by the Alteon NIC. Section 3 provides the overview of the EMP protocol. Section 4 describes the new design challenges encountered for a NIC-based implementation of a parallelized messaging layer. In Section 5, we enumerate the possible alternatives for parallelizing the NIC-driven protocol. In Section 6 we examine how best we can exploit the NIC hardware capability for achieving parallelization and pipelining. In Section 7, we evaluate the alternative strategies with respect to their effectiveness while providing the results of our experiments. Related work and conclusions are presented in Sections 8 and 9, respectively.

2. Architectural overview of the Multi-CPU 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 paper, 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 Tigon chip 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.

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.

3. Overview of the EMP Protocol

In this Section we provide an overview of the implementation of the EMP protocol [9]. 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 will help us understand the challenges involved in parallelizing the EMP protocol.

3.1. 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.

3.1.1. 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.

3.1.2. 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 was added by the hardware designers 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 will queue a request for data from the HOST. When the transfer has completed, it will automatically tell the MAC to send the frame. The transfer is made in the send buffer which is updated after each transfer. This set of operations takes place for every frame and hence will take more 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 will increase with increasing message size.

3.1.3. 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.

3.1.4. 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 [9]. This step happens for every message.

- Classify frame: This step does multiple things. It looks at the header of each incoming frame and identifies if 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.

3.1.5. 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 in Section 3.1.2. 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.

3.1.6. 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 [9].

3.2. 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 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.

Table 1. Timing analysis for the major functional operations.

Operation	Time (us)

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

4. 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 two processors. However these steps need to 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.

4.1. NIC constraints

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.

4.2. 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. This 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.

4.3. 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 paper, we explore both pipelining and parallelization to enhance the performance of user-level protocols with multi-CPU NICs.

5. 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.

Table 2. Function distribution (unidirectional).

Send	**cpu A**	**(us)**	**cpu B**	**(us)**
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**	**(us)**	**cpu B**	**(us)**
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
	send ack	2.50	receiving	2.75

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.

5.1. SO

The split up of the send path in SO happens as shown in Table 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 us) and a part of receiving(2.75 us). 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.

5.2. RO

The split up of functions in RO happens as shown in Table 2. Here, we are able to achieve true parallelism. The send ack (2.50 us) happens in parallel with the receive dma (2.75 us) and a part of receive bookkeeping (4.25 us). 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.

5.3. 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 2.

5.4. 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 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.

6. 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 4.

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 attempt to this alleviate 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.

7. Performance Evaluation

7.1. Experimental setup

For the Gigabit Ethernet tests, we used 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 2.4.2. Our NICs are Netgear 620, which have 512 kB of memory. The machines were connected back-to-back with a strand of fiber.

7.2. 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.

7.2.1. 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 "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 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.

SO The unidirectional bandwidth (Figure 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.

By looking at Figure 1 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 us, which was marginally higher than the base case latency of 24.31 us, a degradation of less than one percent.

RO The unidirectional bandwidth (Figure 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 Figure 1 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 us which is an improvement over the base case latency of 24.31 us, 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.

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 (Figure 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

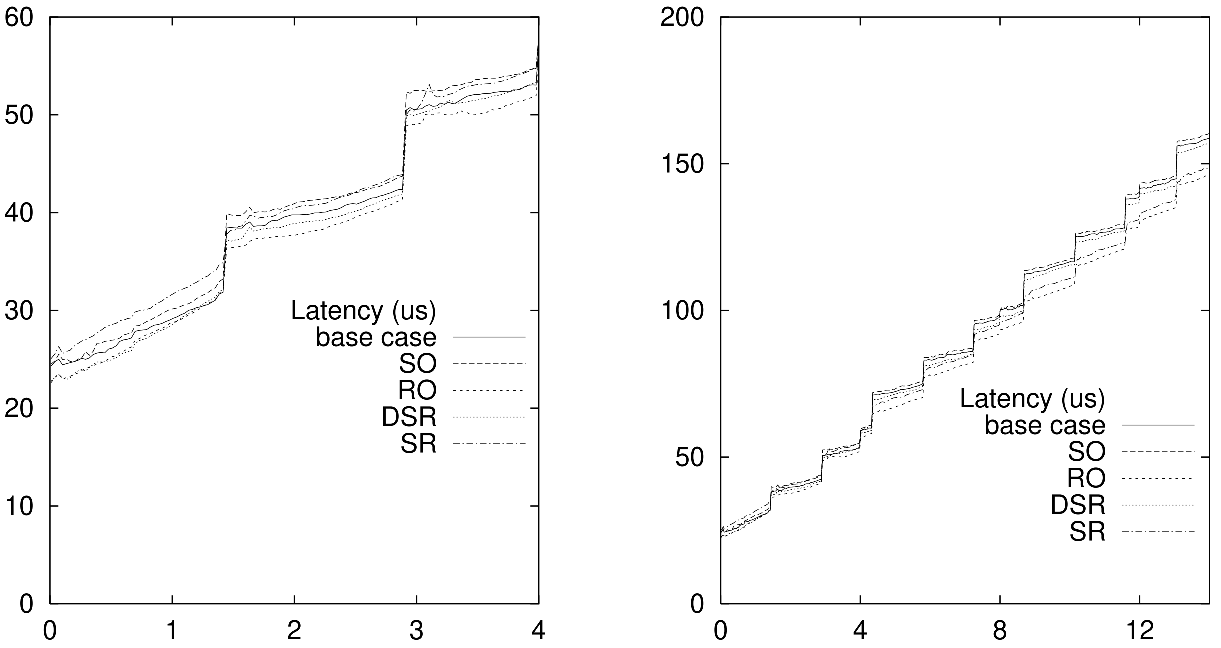

Figure 1. Latency comparisons for small and large message sizes with unidirectional traffic. The x-axis is indicates message size in kilobytes. The y-axis shows latency in microseconds.

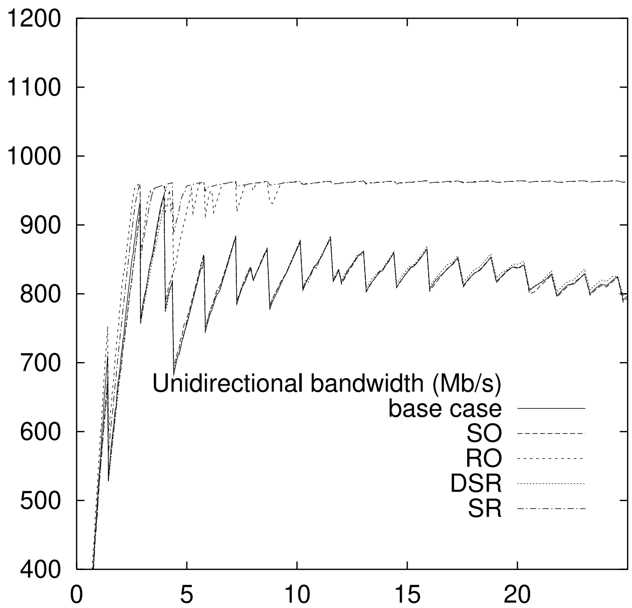

Figure 2. Bandwidth comparisons for unidirectional traffic. The x-axis is indicates message size in kilobytes. The y-axis shows bandwidth in Mb/sec.

Table 3. Function distribution (bidirectional).

Send	cpu A	(us)	cpu B	(us)
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	(us)	cpu B	(us)
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		

send acknowledgment is only a very small portion of the entire receive side processing.

By looking at Figure 1 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 us which is a 6.4% improvement over the base case latency.

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 (Figure 2). If we look at Figure 1 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.

7.2.2. 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 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.

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.

SO For bidirectional traffic, the distribution of steps is as shown in Table 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 (Figure 3) shows considerable improvement over the base case. This is happening because both the CPUs have more functions to perform in parallel (Table 3). Thus, the gain obtained more than offsets the inter-CPU communication overhead.

RO The bidirectional bandwidth (Figure 3) 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), 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:

- 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 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

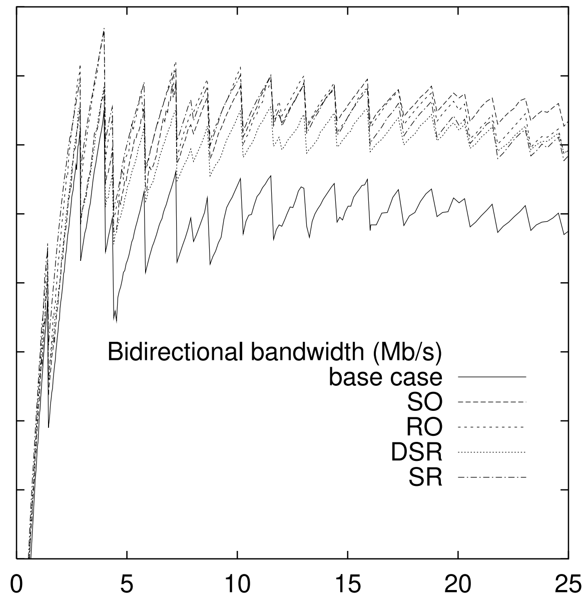

Figure 3. Bandwidth comparisons for bidirectional traffic. The x-axis indicates message size in KBytes. The y-axis shows bandwidth in Mb/sec.

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 (Figure 3) 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 (Figure 3 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.

8. Related Work

All the previous efforts to parallelize the network protocols have been focussed on WAN protocols such as TCP/IP and Symmetric Multi Processor (SMP) systems. The research directions have identified various mechanisms of achieving the parallelization [12]. We will outline some of these approaches in this Section. One of the approaches has analyzed in detail packet level parallelism where the packets are distributed across the processors. This gives the capability to the protocol to process multiple packets in parallel [5]. This approach has been shown to achieve speedup both with multiple connections and with a single connection. For layered protocols following the TCP/IP and the OSI model, attempts have been made to analyze the parallelism between the execution of different layers. This is also know as vertical decomposition of the network protocol [15]. The approach in [15] divides the send path from the receive path and tries to do them in parallel. This has come to be known as horizontal subdivision of a layer in a multi-layer protocol. Another approach has been to split the components of the protocol into different functions and distributing these functions across multiple processors [8]. This turns out to be a good approach for feature rich protocols.

It is to be noted that all these research directions focus on host-based layered protocols and try to exploit SMP systems. The focus of our work is how to take advantage of multi-CPU NIC to enhance user-level protocols. It must be remembered that the goal of fast communication is not for its own end, but rather to enable cooperating host machines to exchange data quickly. If the host CPUs are involved in moving the data across the network, they have fewer cycles available to devote to the parallel application itself.

9. Conclusions and Future Work

In this paper, we have presented how to take advantage of a multi-CPU NIC, as available in Alteon NIC core implementations, to improve point-to-point communication performance on Gigabit Ethernet. We have considered our earlier developed EMP protocol (valid for single-CPU NIC) and analyzed different alternatives to parallelize and pipeline different steps of the communication operation. The 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. Similarly, dedicated assignment of send and receive functionalities to different CPUs delivers very good bidirectional bandwidth.

As a result of our investigations into work distribution strategies on the multi-CPU Alteon NIC, we have determined multiple promising paths for future study. However, the distribution of steps in each path happens at compile time. operation. 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.

We are also exploring the benefits of multi-CPU NICs to support collective communication operations efficiently. We plan to perform application-level performance evaluation for our scheme. Another interesting direction is to explore the required architectural support at the NIC to parallelize both point-to-point and collective communication operations.

10. Acknowledgments

We would like to thank Sandia National Laboratories for sponsoring this project. We would also like to thank the graduate students and the faculty of the NOW (Network of Workstations) lab at the Ohio State University for giving us their invaluable advice on numerous issues related to the project.

References

[1] 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.

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

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

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

[5] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First USENIX Symposium on OSDI*, pages 125–137, November 1994.

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

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

[8] T. Porta and M. Schwartz. A high-speed protocol parallel implementation: design and analysis. In *Fourth IFIP TC6.1/WG6.4 International Conference on High Peformance Networking*, pages 135–150, December 1992.

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

[10] 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.

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

[12] J. Touch. Protocol parallelization. In *Protocols for High Speed Networks IV*, pages 349–360, 1995.

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

[14] 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.

[15] M. Zitterbart. High-speed protocol implementations based on a multiprocessor architecture. In *Protocols for High Speed Networks*, pages 151–163, 1999.