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

Abstract

User-level protocols and their implementations on programmable network interface cards (NICs) have been alleviating the communication bottleneck for high speed interconnects. 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. 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.

1. Introduction

High-performance computing on a cluster of workstations requires that the communication latency be as small as possible. This has led to the development of a range of user-level network protocols: FM [4] for Myrinet, U-Net [9] for ATM and Fast Ethernet, GM [1] for Myrinet, our recent work on EMP [7] 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) [8]. An extension to the VIA interface is already included in the latest *InfiniBand Architecture* (IBA) [2] as the Verbs layer.

As processor technology is moving towards gigahertz speeds and network technology is moving towards tens of gigabits per second 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; however, these new bus standards will alleviate that problem, bringing increased attention to high-performance NIC design.

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 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 parallelization 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 a user-level protocol layer on the two-CPU Alteon Gigabit Ethernet NIC, enhancing the recently developed *Ethernet Message Passing* (EMP) [7] protocol. EMP was initially developed to use only one of the two available CPUs in the NIC. 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 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 μs and bandwidth of 964 Mbps. This is compared to the base case latency of 24 μs (a gain

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

of 7%) and bandwidth of 840 Mbps. For large messages the latency improvement was around 8%. 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%.

2. Architectural overview

Alteon Web Systems, now part of Nortel, produced a Gigabit Ethernet network interface chipset based around a general purpose embedded microprocessor design called the Tigon2. It 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. 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 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 available at the NIC. For more details see [6].

3. Overview of the EMP protocol

In this section we provide an overview of the implementation of the EMP protocol [7]. 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. Send protocol steps

Here we outline the basic EMP protocol for a sender.

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 per 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.
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.
Transmission. The steps involved in transmission are:
DMA from HOST to the NIC: Along with the two DMA channels the NIC also contains the "DMA Assist" state machine to help off-load some of the tasks from the NIC processor. 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 descriptor to keep track of frames being sent to the serial Ethernet interface. 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. This operation happens per frame and each frame uses one MAC descriptor, hence the overhead incurred will increase with increasing message size.
Receive acknowledgment. Once the sender knows that the receiver has successfully received the frames it can release the resources related to the sent data. Receive acknowledgment introduces only minimal per-frame overhead, again, because acknowledgment is a process which applies only to groups of frames [7].

3.2. Receive protocol steps

Similarly the receiving side steps are as follows.
Receive bookkeeping. This refers to the operations which need to be performed before the frame can be sent to the host. These operations are:
Handle preposted receive descriptor: This step is initiated by the host for all the messages expected to arrive in future. Here the state information which is necessary for matching an incoming frame is stored at the NIC. If a frame arrives and does not find a matching preposted descriptor, it is dropped to avoid buffering at the NIC [7].
Classify frame: This step 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 preposted receive to which the incoming frame belongs by going through all the preposted records. In the process it also identifies if the frame has already arrived and, if so, drops it. 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 frame header information (message sequence number, frame sequence number, etc.) is stored in the receive data structures for reliability and other bookkeeping purposes. After recording this information, this step also initiates the DMA of the incoming frame data. This step is done per 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. 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 but in the reverse direction.

Send acknowledgment. This step 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.

3.3. Timing analysis

We did a complete profiling of our protocol to find out how much time is spent in each of the steps for a 64-byte ping-pong test. 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. The experimental platform is described in Section 6.

Receive bookkeeping is more time-consuming than send bookkeeping due to the need to perform extra operations including MPI tag matching, possibly re-ordering frames, and acknowledgment generation.

4. Challenges of using a multi-CPU NIC

In order to take advantage of a multi-CPU NIC, the basic protocol steps need to be distributed across the processors. However, these steps need to share some common state information at some point in the execution. Typically, NICs have 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.

Table 1. Major functional operation timings.

Operation	Time (us)
Send bookkeeping	5.25
Transmission	5.50
Receive acknowledgment	5.75
Recv bookkeeping	6.25
Receive frame	4.25
Receiving	2.75
Send acknowledgment	2.50

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

There are many scenarios where even after distributing the send and receive functions on different processors, the sending and receiving state information needs to be shared.

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. However, this cannot be guaranteed for every case.

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, which would be very expensive as the data structures are accessed frequently.

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 might be a better option because in this case the overhead is generated only when there is a need for sharing data between the CPUs.

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. 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 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 techniques 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. Both pipelining and parallelism were considered in designing this distribution. We tried to prefer parallelism 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 across the NIC CPUs.
RO: The receive path only is split across the NIC CPUs.
DSR: Dedicated processors for send and receive paths.
SR: Both send and receive path are split.

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 steps happen at the same processor.

SO. The split-up of the send path in SO happens as shown in Table 2. Here, we aim to achieve pipelining by running the bookkeeping phase of a later message with the transmission phase of an earlier message for a unidirectional flow. There is some parallelism also happening at the receiver between 'send ack' (2.50 us) and receiving (1.25 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. The send acknowledgment on the receiver is a part of the send path since the 'send ack' involves steps used in sending and not receiving.

RO. The split-up of functions in RO happens as shown in Table 2. The 'send ack' (2.50 us) happens in parallel with receiving (2.75 us) and a part of 'recv bookkeep' (4.25 us). 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.

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 way we achieve an almost complete split of the send and receive paths. The 'recv ack' 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.

Table 2. Function distribution (EMP).

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

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. We hope to gain from the benefits of pipelining on the send side and parallelization on the receive side.

6. Implementation and performance

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 GA620, which have 512 kB of memory. The machines were connected back-to-back with a strand of fiber.

6.1. 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, and we developed a pair of locking primitives 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.

Running two processors simultaneously puts more load on the memory system in the NIC. We attempt 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.

6.2. Unidirectional results

In this section we analyze the results derived from the alternatives discussed so far. Better performance than the base case (single CPU per NIC) is achieved for both latency and bandwidth measures by using at least one alternative in each of the cases.

The latency is determined by halving the time to complete a single ping-pong test. The "ping" side posts a descriptor for one receive and for one transmit operation, 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 iterated 10,000 times to produce an average round-trip time, half of which is reported as a one-way latency estimate.

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, waiting for each in turn to complete and posting more descriptors. 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 reliability mechanism causes the receiving NIC to generate 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 1) is the same as in the base case. On the sending side the benefit is obtained due to pipelining (between 'send bookkeep' and transmission) and parallelism ('recv ack' and transmission).

However the gains are offset by two factors. First, overhead is induced by requiring inter-CPU communication. Next, comparing the receive and the send path operations, we see that the receive path has more overhead. Also, acknowledgment clocking limits the potential speedup by coupling send performance to receive performance. And since the receive processing here is identical to the base case, SO does not demonstrate much benefit.

RO. The unidirectional bandwidth (Figure 1) 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 hardware and protocol headers). This happens because we can overlap sending an acknowledgment with receive DMA processing. In addition, 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 path has been parallelized effectively we are able to offset the inter-CPU communication overhead and achieve almost the maximum possible bandwidth. This implies that the receive side is the bottleneck, a fact confirmed by the SO case where we left the receive side almost unaltered and did not achieve any benefits even though we had pipelined and parallelized the send path.

By looking at Figure 1 we observe that even the latency improves for RO parallelism. For 10-byte messages we obtained a 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.

DSR. The unidirectional bandwidth (Figure 1) 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 ack' to happen on a different CPU, we are able to see the marginal improvement in bandwidth numbers. The improvement is marginal because 'send ack' is a very small portion of the receive side processing.

By looking at Figure 1 we can conclude that latency also benefits with this approach. For a 10-byte message, we get a latency improvement of 6%.

SR. This alternative gives the best unidirectional bandwidth, allowing almost complete utilization of what is physically available. 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 1).

6.3. Bidirectional results

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

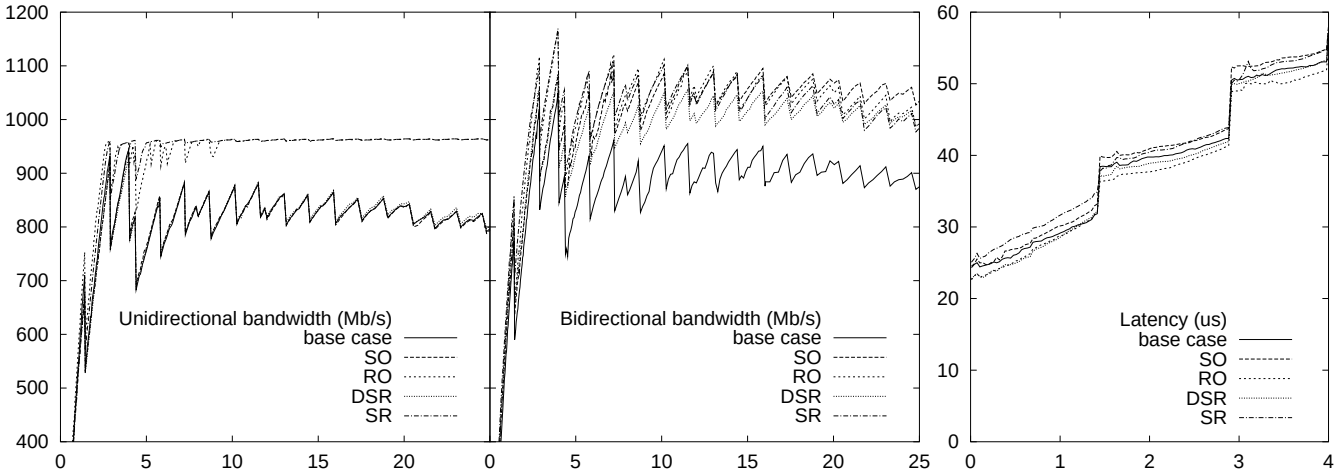

Figure 1. Bandwidth (Mbps) and latency (us) comparisons as a function of message size (kB).

each host ensures that we do not suffer from operating system scheduler decisions.

Bidirectional traffic is more complex than unidirectional traffic. A lot more steps are happening at the sender and the receiver as compared to the unidirectional case. These steps are send bookkeeping, transmission, acknowledgment receive, sending acknowledgment, receive bookkeeping and dmaing of the receive as well as send data.

Figure 1 shows considerable improvement for all the alternatives. This is so because of more number of steps and hence more opportunities to parallelize/pipeline in a bidirectional traffic as compared to the unidirectional traffic. In the bidirectional traffic we encounter the same interplay of factors which was present in the unidirectional traffic. However, for large message sizes the inter-CPU communication overtakes the gain. This does not happen in the unidirectional case because the amount of inter-CPU communication is less as compared to the bidirectional case.

7. Related work

All previous efforts to parallelize network protocols have focused on WAN protocols such as TCP/IP and on large shared memory systems. These approaches involve processing multiple packets in parallel [3], executing different layers in parallel for deeply layered protocols [10] and breaking the protocol functions across multiple processors [5]. These cited research activities concern themselves with multi-CPU *hosts*, while our work takes advantage of multi-CPU *NICs* to enhance user-level protocols. The key idea is to free all host CPUs as much as possible so they can devote their cycles to the parallel application itself.

8. 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 benefits for unidirectional as well as bidirectional traffic. In fact, this scheme allows us to reach the theoretical throughput of the medium.

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. Currently the split-up of tasks in done 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. We are also exploring the benefits of multi-CPU NICs to support and to parallelize collective communication operations.

References

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

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

[3] E. Nahum, D. Yates, and J. Kurose. Performance issues in parallelized network protocols. In *USENIX OSDI*, 1994.

[4] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Fast Messages, 1995.

[5] T. Porta and M. Schwartz. A high-speed protocol parallel implementation: design and analysis. In *ICHPN*, 1992.

[6] P. Shivam, P. Wyckoff, and D. Panda. Can user level protocols take advantage of multi-CPU NICs? Technical Report OSU-CISRC-08/01-TR09, The Ohio State University, 2001.

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

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

[9] T. von Eicken, A. Basu, and V. Buch. U-Net: A user-level network interface for parallel computing. In *ASPLOS*, 1995.

[10] M. Zitterbart. High-speed protocol implementations on a multiproc. arch. In *Protocols for Highspeed Networks*, 1999.