Exploiting NIC Architectural Support for Enhancing IP based Protocols on High Performance Networks

HYUN-WOOK JIN[†], PAVAN BALAJI[†], CHUCK YOO[‡], JIN-YOUNG CHOI[‡], AND DHABALESWAR K. PANDA[†]

[†]Computer Science and Engineering,
The Ohio State University,
Columbus, OH 43210
{jinhy, balaji, panda}@cis.ohio-state.edu

[‡]Computer Science and Engineering,
Korea University,
Seoul, 136-701, Korea
{hxy@os, choi@formal}.korea.ac.kr

Technical Report
OSU-CISRC-5/04-TR37

Exploiting NIC Architectural Support for Enhancing IP based Protocols on High Performance Networks[*]

H. -W. Jin[†] P. Balaji[†] C. Yoo[‡] J. -Y. Choi[‡] D. K. Panda[†]

[†]Computer Science and Engineering,
The Ohio State University
{jinhy, balaji, panda}@cis.ohio-state.edu

[‡]Computer Science and Engineering,
Korea University
{hxy@os, choi@formal}.korea.ac.kr

Abstract

While a number of user-level protocols have been developed to reduce the gap between the performance capabilities of the physical network and the performance actually available, their compatibility issues with the existing sockets based applications and IP based infrastructure has been an area of major concern. To address these compatibility issues while maintaining a high performance, a number of researchers have been looking at alternative approaches to optimize the existing traditional protocol stacks. Broadly, previous research has broken up the overheads in the traditional protocol stack into four related aspects, namely: (i) Compute requirements and contention, (ii) Memory contention, (iii) I/O bus contention and (iv) System resources' idle time. While previous research dealing with some of these aspects exists, to the best of our knowledge, there is no work which deals with all these issues in an integrated manner while maintaining backward compatibility with existing applications and infrastructure. In this paper, we address each of these issues, propose solutions for minimizing these overheads by exploiting the emerging architectural features provided by modern Network Interface Cards and demonstrate the capabilities of these solutions using an implementation based on UDP/IP over Myrinet. Our experimental results show that with our implementation of UDP, termed as E-UDP, can achieve up to 94% of the theoretical maximum bandwidth. We also present a mathematical performance model which allows us to study the scalability of our approach for different system architectures and network speeds.

Keywords: *Clusters, UDP/IP, Myrinet, Protocol Offload, Overhead Pipelining*

[*]This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #CCR-0204429, and #CCR-0311542 and the Post-doctoral Fellowship Program of Korea Science and Engineering Foundation (KOSEF).

1 Introduction

Commodity off-the-shelf (COTS) clusters have been accepted as a feasible and cost effective approach to mainstream supercomputing for a broad subset of applications. Most of the success of these COTS clusters is derived from the high performance-to-cost ratio achievable through them. With the advent of the several modern high speed interconnects such as Myrinet [8], InfiniBand [4, 2], Quadrics [24, 3, 27, 26, 28], 10-Gigabit Ethernet [1, 17, 14] and others, the bottleneck in the data communication path in such clusters has shifted to the messaging software at the sending and the receiving side.

Researchers have been looking at alternatives by which one could increase the communication performance delivered by clusters in the form of low latency and high bandwidth user-level protocols such as the Virtual Interface Architecture (VIA) [9], FM [23], and GM [12] for Myrinet, U-Net [35] for ATM and Ethernet, EMP [32, 33] for Gigabit Ethernet and others. While this approach is good for writing new applications which completely reside inside the cluster environment, these have several limitations with respect to compatibility with existing applications and infrastructure. In particular, we look at the following incompatibilities:

1. A number of applications have been developed in a span of several years over the traditional protocols using the sockets interface. Developing new high performance protocols might not be directly beneficial for such applications.

2. IP is the most widely accepted and used network protocol today. However, the above mentioned user-level protocols are not compatible with existing IP infrastructures, i.e., an application using GM over Myrinet or EMP over Gigabit Ethernet cannot communicate across clusters where the intermediate nodes/switches are IP based and do not understand these user-level protocols.

3. Several security mechanisms such as IPsec have been developed over IP. Using user-level protocols instead of IP based protocols might require such security mechanisms to be re-developed for these new protocols.

Researchers have looked at some of these issues in a segregated manner. For example, user-level sockets over high performance networks [6, 7, 5, 20, 31] and other substrates [30] have been developed to allow sockets based applications to take advantage of the high performance networks. This approach tries to solve the first issue (allowing existing sockets based applications to take advantage of the high performance networks), but does not address the remaining issues. Similarly, the Trapeze project [11, 10] by Chase, et. al., tries to address issues two and three (compatibility with the existing IP infrastructure), but modifies the sockets interface resulting in incompatibility with existing applications. These and other related work are discussed in Section 6 in more detail.

To address these compatibility issues while maintaining a high performance, a number of researchers have been looking at alternative approaches to optimize the existing traditional protocol stacks. Broadly, previous research has broken up the overheads in the traditional protocol stack into four related aspects, namely: (i) Compute requirements and contention, (ii) Memory contention, (iii) I/O bus contention and (iv) System resources' idle time.

In this paper, we first utilize the earlier proposed techniques, in particular those specific to Protocol Offload Engines (POEs) [37, 29, 34], to implement a partial offload of the UDP/IP protocol stack over Myrinet to address the first two issues, i.e., compute requirements and memory contention. Next, we modify the Myrinet device driver to allow a delayed posting of descriptors in order to reduce the contention at the I/O bus. Finally, we implement a fine-grained overhead pipelining technique on the firmware of the NIC to minimize the link idle time. In this paper, we refer to this implementation as E-UDP (standing for enhanced UDP). This IP based protocol implementation is not only compatible with existing sockets applications, but also with the traditional UDP/IP stack.

In this paper, we focus on the following key questions:

- *How does the performance of E-UDP compare with that of the traditional UDP stack?*
- *How does the performance of E-UDP compare with that of the existing user-level high performance protocols?*
- *Would the feasibility of fine-grained pipelining in E-UDP be specific to the application communication pattern, i.e., is it formally verifiable that a fine grained pipelining would be possible for any communication pattern?*
- *How does E-UDP perform for various other system and network configurations, e.g., for 10-Gigabit networks, faster I/O buses, etc?*

To answer the first two questions, we analyze the performance impact of the above mentioned techniques in UDP/IP over Myrinet. To answer the third question, we present a formal verification model and show the pipelining capabilities of the Network Interface Card (NIC) architecture in a generic communication pattern. Finally, to answer the fourth question, we propose an analytical model in order to study the performance of our design for various system and network configurations.

The remaining part of the paper is organized as follows: In Section 2 we present background information about the traditional UDP/IP implementation, Protocol Offload Engines and the Myrinet network. In Section 3, we discuss the architectural interaction and implications of the UDP/IP protocol implementation. In Section 4, we present several solutions for the system resource contention and other inefficiencies introduced by the UDP/IP stack. We present the experimental and analytical results in Section 5, other related work in Section 6 and some concluding remarks in Section 7.

2 Background

In this section, we present a brief background about the traditional UDP/IP implementation, Protocol Offload Engines (POE) and the functionality of the Myrinet NIC. More details about each of these can be found in [19].

2.1 Traditional UDP/IP Implementation

Like most networking protocol suites, the UDP/IP protocol suite is a combination of different protocols at various levels, with each layer responsible for a different facet of the communications.

To allow standard Unix I/O system calls such as `read()` and `write()` to operate with network connections, the file-system and networking facilities are integrated at the system call level. Network connections represented by sockets are accessed through a descriptor in the same way an open file is accessed through a descriptor. This allows the standard file-system calls such as `read()` and `write()`, as well as network-specific system calls such as `sendmsg()` and `recvmsg()`, to work with a descriptor associated with a socket. Figure 1a illustrates the interaction between the sockets layer and the actual TCP and UDP protocol stacks in the kernel.

On the transmission side, the message is copied into the socket buffer, data integrity ensured through checksum computation (to form the UDP checksum) and passed on to the underlying IP layer. The IP layer fragments the data to MTU sized chunks, extends the checksum to include the IP header and form the IP checksum and passes on the IP datagram to the device driver. After the construction of a

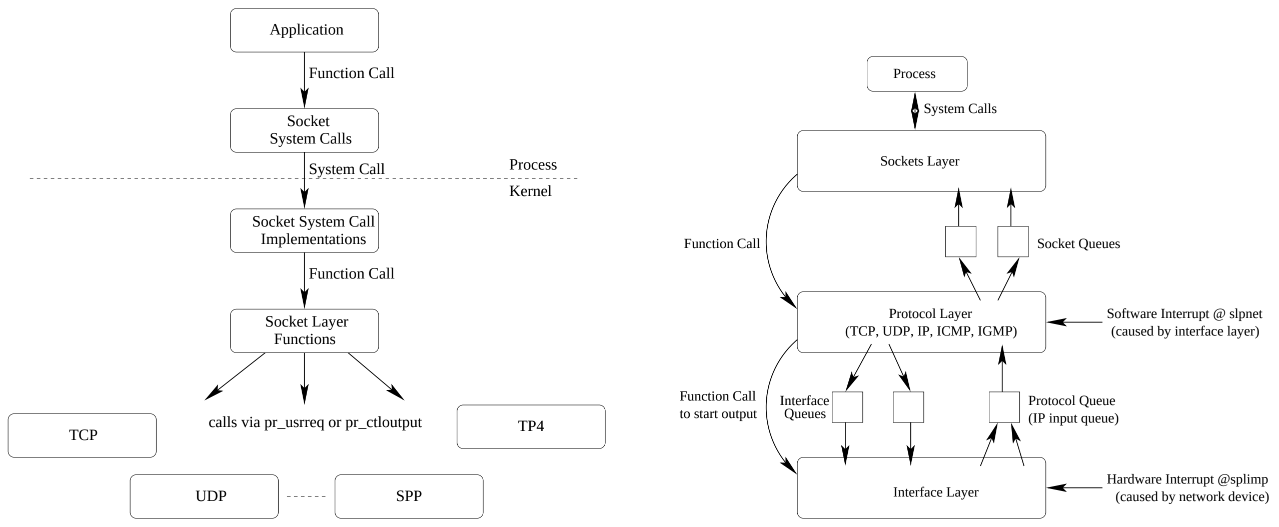

Application
Function Call
Socket System Calls
System Call
Process
Kernel
Socket System Call Implementations
Function Call
Socket Layer Functions
calls via pr_usrreq or pr_ctloutput
TCP
TP4
UDP
SPP
Process
System Calls
Sockets Layer
Function Call
Socket Queues
Protocol Layer (TCP, UDP, IP, ICMP, IGMP)
Software Interrupt @ slpnet (caused by interface layer)
Function Call to start output
Interface Queues
Protocol Queue (IP input queue)
Interface Layer
Hardware Interrupt @splimp (caused by network device)

Figure 1. (a) Interaction between the sockets layer and the TCP/UDP protocol stack; (b) Data path in the traditional protocol stack

packet header, the device driver makes a descriptor for the packet and passes the descriptor to the NIC using a PIO (Programmed I/O) operation. The NIC performs a DMA operation to move the actual data indicated by the descriptor from the socket buffer to the NIC buffer. The NIC then ships the data with the link header to the physical network and raises an interrupt to inform the device driver that it has finished transmitting the segment.

On the receiver side, the NIC receives the IP datagrams, DMAs them to the socket buffer and raises an interrupt informing the device driver about this. The device driver strips the packet off the link header and hands it over to the IP layer using a software interrupt mechanism. The interrupt handler for this software interrupt is typically referred to as the bottom-half handler and has a higher priority compared to the rest of the kernel. The IP layer verifies the IP checksum and if the data integrity is maintained, defragments the data segments to form the complete UDP message and hands it over to the UDP layer. The UDP layer verifies the data integrity of the message and places the data into the socket buffer. When the application calls the `read()` operation, the data is copied from the socket buffer to the application buffer. Figure 1b illustrates the sender and the receiver paths.

2.2 Protocol Offload Engines

The processing of traditional protocols such as TCP/IP and UDP/IP is accomplished by software running on the central processor, CPU or microprocessor, of the server. As network connections scale beyond Gigabit Ethernet speeds, the CPU becomes burdened with the large amount of protocol processing required. Resource-intensive memory copies, checksum computation, interrupts and reassembling of out-of-order packets puts a tremendous amount of load on the host CPU. In high-speed networks, the CPU has to dedicate more processing to handle the network traffic than to the applications it is running. Protocol Offload Engines (POE) are emerging as a solution to limit the processing required by CPUs for networking links.

The basic idea of a POE is to offload the processing of protocols from the host processor to the hardware on the adapter or in the system. A POE can be implemented with a network processor and firmware, specialized ASICs, or a combination of both. Most POE implementations available in the market concentrate on offloading the TCP and IP processing, while a few of them focus on other protocols such as UDP/IP, etc.

As a precursor to complete protocol offloading, some operating systems have started incorporating support for features to offload some compute intensive features from the host to the underlying adapters. TCP/UDP and IP checksum offload implemented in some server network adapters is an example of a simple offload. But as Ethernet speeds increased beyond 100Mbps, the need for further protocol processing offload became a clear requirement. Some Gigabit Ethernet adapters complemented this requirement by offloading TCP/IP and UDP/IP segmentation on the transmission side on to the network adapter as well.

POE can be implemented in different ways depending on the end-user preference between various factors like deployment flexibility and performance. Traditionally, firmware-based solutions provided the flexibility to implement new features, while ASIC solutions provided performance but were not flexible enough to add new features. Today, there is a new breed of performance optimized ASICs utilizing multiple processing engines to provide ASIC-like performance with more deployment flexibility.

2.3 Myrinet Network

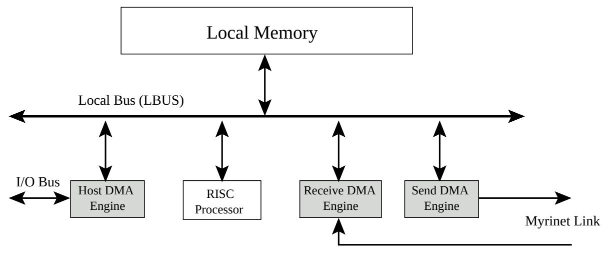


Figure 2. Myrinet NIC Architecture

Myrinet is a high performance Local Area Network (LAN) developed by Myricom Incorporation. The Myrinet NIC consists of a RISC processor named LANai, memory, and three DMA engines (host DMA engine, send DMA engine, and receive DMA engine). Figure 2 illustrates the Myrinet NIC architecture.

The LANai processor executes the Myrinet Control Program (MCP), i.e., the firmware on the NIC. The NIC memory stores the data for sending and receiving. The host DMA engine is responsible for the data movement between the host and the NIC memories through the I/O bus. On the other hand, the send DMA engine deals with moving the data from the NIC memory to the Myrinet link. Similarly, the receive DMA engine deals with moving the data from the Myrinet link to the NIC memory.

There are several emerging features provided by the Myrinet NIC. First, the programmability provided by the Myrinet NICs can be utilized to modify the implementation of existing features and/or add more features and functionality to the NIC. Such programmability can provide plentiful opportunities to enhance existing IP based protocols.

The second interesting feature is the capability of the memory on the Myrinet NIC. The memory on the NIC runs at the same clock speed as the RISC processor. Further, the LBUS (shown in Figure 2) operates at twice the chip clock speed (two LBUS memory cycles for every clock cycle). The host DMA engine, the receive DMA engine and the send DMA engine each can request a maximum of one memory access per clock cycle. The on-chip processor can request up to two memory accesses per clock cycle. Further, the memory itself can serve up to two memory accesses per clock cycle. This means that two DMA engines, e.g., the host DMA engine and the send DMA engine, can access the memory simultaneously.

Yet another interesting feature provided by Myrinet is the capability of the host DMA engine on the Myrinet NIC. The host DMA engine allows checksum computation during the DMA operation itself. In addition, to specify the end of the buffer for a send DMA operation, the Myrinet NIC provides two kinds of registers. One is the SMLT register, which specifies not only the end of the buffer in the DMA operation but also the end of the message segment. The other register, SML, only indicates the end of the buffer. Therefore, several chunks of data sent with the SML register set, are notified as parts of the the same segment on the receiver side.

3 Architectural Viewpoint of UDP/IP

While the UDP/IP data and control path is fairly straightforward with respect to the operating system functionality, it has a number of implications on the architectural requirements of the system which implements this. These architectural requirements deal with several issues in the system such as the computational requirements of the protocol stack, memory contention caused by the stack, the I/O bus contention, etc.

3.1 Interaction of UDP/IP with System Resources

UDP/IP interacts with several system resources such as CPU, host memory, I/O bus and the network link. In this section, we briefly point out the requirements and extent of these interactions using the Linux UDP/IP stack as an example implementation.

CPU Interaction: As described in Section 2.1, a number of components in the UDP/IP data path tend to be compute intensive. For example, the copy of the data from/to the application buffer occurring in the UDP/IP layer has large computation requirements. Similarly, the checksum computation occurring as a part of the bottom-half (described in Section 2.1) on the receiver side requires compute resources as well. The bottom-half typically has a higher priority compared to the rest of the kernel. This means that checksum computation for incoming packets is given a higher priority as compared to copying of the data to the application buffer. This biased prioritization of providing CPU resources for the different components has interesting implications as we will see in Section 3.2.

Host Memory Interaction: In cases where the sender buffer is touched before transmission or when the same buffer is used for transmission several times (e.g., in a micro-benchmark test), the application buffer can be expected to be in cache during the transmission operation. In such cases the data copy from the application buffer to the socket buffer is performed with cached data and does not incur any memory references. However, the case on the receiver side is quite different. On receiving data, the NIC initiates a DMA operation to move the received data into the host memory. If the host memory area for DMA is in the cache of any processor, the cache lines are invalidated and the DMA operation allowed to proceed to memory. The checksum computation, which follows the DMA operation, thus accesses data that is not cached and always requires memory accesses. Thus, the checksum computation, the DMA operation and also the data copy operation in some cases compete for memory accesses.

I/O Bus Interaction: As described in Section 2.1, once the IP layer hands over the data to the link layer, the device driver forms a descriptor corresponding to the data and posts it to the NIC using a PIO operation over the I/O bus. The NIC on seeing this posted descriptor performs a DMA operation on the actual data from the host memory to the NIC memory. Both these operations as well as other DMA operations corresponding to incoming data use the same I/O bus and essentially contend for its ownership.

NIC and Link Interaction: The host DMA engine on the Myrinet NIC performs the DMA operation to fetch the data from the host memory to the NIC memory. The send DMA engine waits for the DMA operation to complete before it can transmit the data from the NIC memory to the link. This delay in transmitting the data can lead to the link being idle for a long period of time, in essense underutilizing the available link bandwidth.

3.2 Implications of System Resources on UDP/IP Performance

The interaction of the UDP/IP stack with the various system resources has several implications on the end performance it can achieve. In this section, we study these performance implications based on some experimental results using the traditional UDP/IP stack in Linux.

Broad Perspective: Figures 3a and 3b present the time flow chart on the transmission and reception sides of the traditional UDP/IP implementation. The figures show the time chart of the host and the NIC when 10 UDP packets (of 32KB each) are transmitted in a burst. The time is set to zero when the first packet is sent by the UDP application. The *y-axis* represents the layers that each packet goes through in order to be processed. A rectangle is the time spent in the corresponding layer to process the packet. On the receiver side, the bottom-half is not dealt as a part of UDP/IP to clearly point out the overheads caused by the checksum operation (a part of the bottom-half handler) and by the data copy operation (a part of UDP/IP). The rectangles are shaded alternatively for clarity. In these figures we can see that the different rectangles in the same layer are of different sizes especially on the NIC of sender and on bottom half and UDP of receiver. This is attributed to the contention between the host and the NIC for the various system resources including the host CPU, host memory, I/O bus, etc. We will deal with each of these resources in the following few subsections.

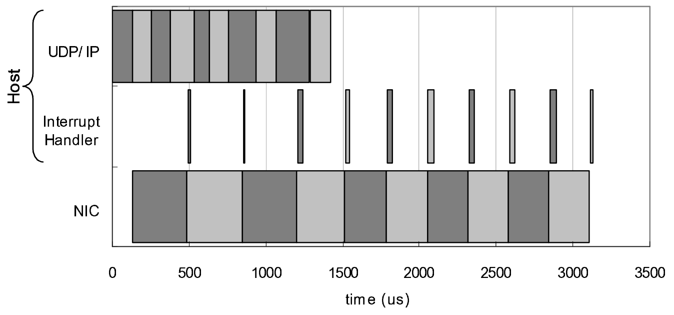
3.2.1 Compute Requirements and Contention

We first analyze the case of the host CPU contention. The host performs several compute intensive operations such as checksum computation, copying the data to the application buffer, etc. Within these, some of the operations such as the checksum computation are performed as soon as the data arrives within a higher priority context (bottom-half). During bulk data transfers where segments are received continuously, this might imply that the CPU is completely devoted to the bottom-half handler resulting in a starvation for the other operations with compute requirements such as data copy from the socket buffer to the application buffer.

We use the results depicted in Figures 3a and 3b to understand this. If we first consider the transmission side, the contention between the host and the NIC for the different system resources causes the NIC to take more time for the first four packets compared to the other packets. This delay in the NIC for the first four packets is also reflected on the receiver side (Figure 3b) where the NIC has a significant amount of idle time for the first few packets (shown as the gaps between the rectangles in the figure). These gaps in the receiver NIC's active time are in turn reflected on the bottom-half handling time on the receiver side, i.e., since the packets are not coming in at the maximum speed on the network, the host can perform the bottom-half and still would have enough time to copy the data to the application buffer before the next packet comes in. In short, in this case the sender is not sending out data at the peak rate due to resource contention; this reduced data rate gives the receiver ample time to receive data and place it in the application buffer. Thus, in this case there is no contention for the CPU on the receiver side.

However, for the later packets (rectangles 5 to 10 for the NIC), as seen in Figure 3a, the host has completed performing its checksum and copy operations; so the NIC can proceed with its operations without any contention from the host. This reduced contention for the system resources ensures that the data can be transmitted at a faster rate by the NIC, i.e., the time spent for each packet is lesser in this case. This increased transmission rate also reflects as a lesser idle time for the NIC on the receiver side (rectangles 5-10 for the NIC in Figure 3b). Further, this reduced idle time means that the NIC continuously delivers packets to the host, thus keeping the bottom-half handler active. This results in the starvation of lower priority processes in the UDP/IP stack such as the copy of the data from the socket buffer to the application buffer. This starvation for CPU is reflected in the third rectangle in Figure 3b where the copy operation for the third packet has to wait until all the data corresponding to the other packets has been delivered to the sockets layer.

To further verify these observations, we have also used the Performance Measurement Counters (PMCs) for the Pentium processor to measure the actual time taken by the data copy operation, due to CPU starvation, cache miss effects, etc. Figure 4 shows the impact of CPU starvation. Figure 4a shows the overhead associated with the copy operation when there is a CPU starvation as compared to the case when there is no CPU starvation. The increase in the copy overhead is due to not only the wait time associated



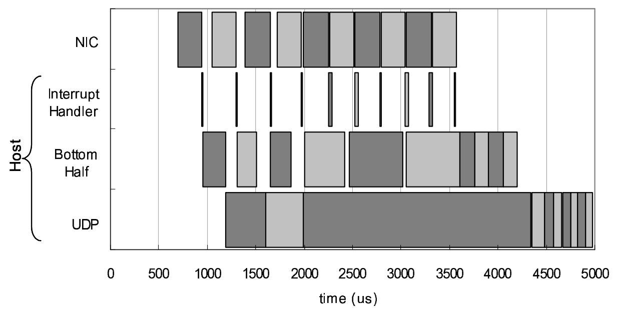


Figure 3. Time flow chart of the host and the NIC overheads: (a) Sender side; (b) Receiver side

with the process in order to get access to the host CPU but also cache misses. When UDP tries to copy the data from the kernel to the user buffer after the CPU starvation, the cached data during the checksum computation has already been removed from the cache by other checksum comutations. Figure 4b shows this wait time associated with the process before it fetches the data into the cache.

3.2.2 Memory Contention

We next look at the host memory contention. Several operations in the UDP/IP stack such as checksum computation, data copy to the application buffer, etc., as well as the DMA operations to and from the network adapter compete for memory accesses. We again refer to Figure 3 for understanding these contention issues. Observing Figure 3b, we notice that when the rate of the incoming packets increases, the time taken for the bottom-half increases (rectangles 4-6 in the bottom-half). This is because the checksum computation in the bottom-half handler competes for memory accesses with the DMA operations carried out by the NIC (rectangles 5-10 for the NIC overhead in Figure 3b). This figure shows the impact memory contention can have on the performance of the UDP/IP stack.

Again, to re-verify these results, we have used PMCs to measure the actual checksum computation overhead and the wait time for fetching data from the memory to cache. The actual impact of memory contention is depicted in Figure 5. Figure 5a shows the overhead associated with the checksum operation during memory contention periods (later segments in Figure **??**) as compared to the case when there's no memory contention. Again, there is no change in the actual checksum computation time. However, the increase in the checksum overhead is associated with the wait time in the process in order to get access to the host memory. Figure 5b shows this wait time associated with the process before it gets access to the host memory.

3.2.3 I/O Bus Contention

Posting of descriptors by the device driver as well as DMA of data by the NIC to/from the NIC memory uses the I/O bus causing contention. The transmission side in Figure 3a shows the increased time taken by the NIC for rectangles 1-4. However, this increase in the time taken by the NIC can be because of both I/O bus contention and memory contention. In order to understand the actual impact of the I/O bus contention, we modified the UDP/IP stack to offload the checksum computation to the NIC and allow a zero-copy implementation. These modifications completely get rid of the memory accesses by the host avoiding any memory contention that might be possible.

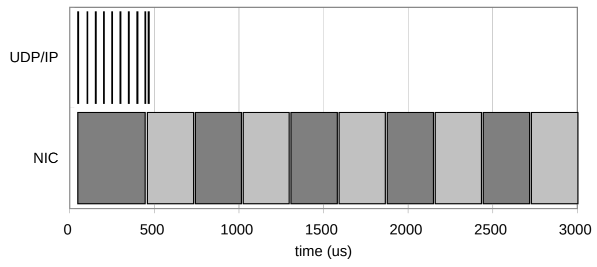


Figure 6. Time flow chart of the host and the NIC overheads on the transmission side with the modified UDP/IP stack (checksum offloaded and zero-copy implementation)

Figure 6 shows the I/O bus contention for the modified UDP/IP stack. We can see that the UDP/IP overhead is negligible in this case because of the offloading of the data touching components. On the other hand, the NIC overhead for the first rectangle is significantly larger than the rest of the rectangles due to the contention between the posting of the descriptors and the DMA of the actual data to the NIC. Since the host does not touch its memory at all, we can say that this overhead is completely due to the I/O bus contention.

3.2.4 Link Idle Time

Figure 7 shows the time flow chart for the NIC and the link overheads for the traditional UDP/IP stack. As discussed earlier, the earlier rectangles (1-4) showing the NIC overhead on the transmission side are larger than the later ones (5-10) because of the memory and I/O bus contentions. This

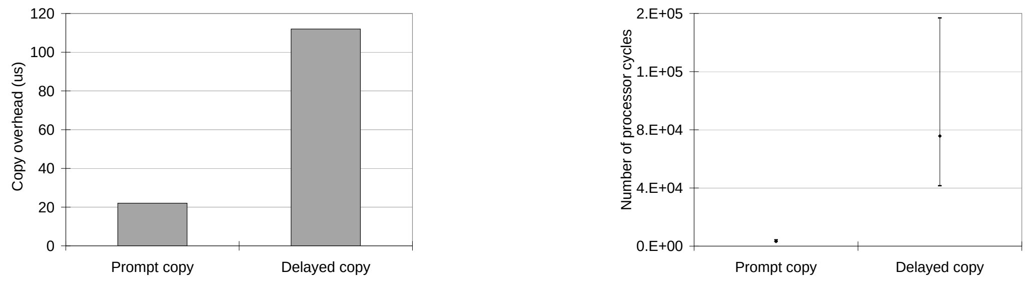


Figure 4. Impact of CPU starvation on (a) Overall copy overhead; (b) Cache line fill wait time for performing the copy operation

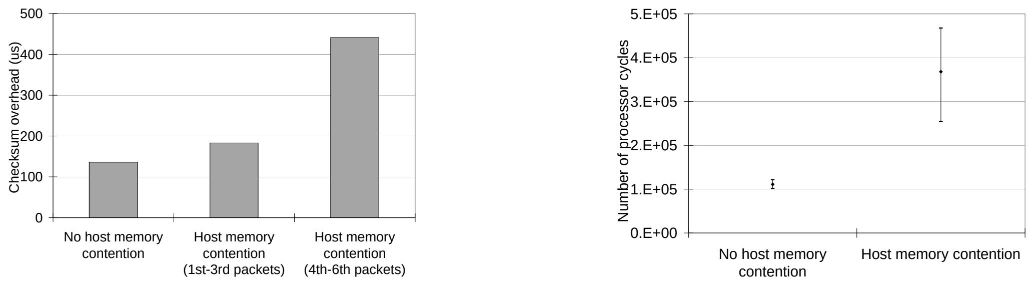


Figure 5. Impact of memory contention on: (a) Checksum overhead on the receiver side; (b) Cache line fill wait time for performing the checksum operation

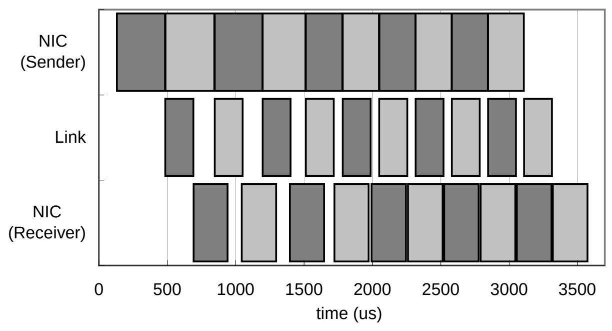


Figure 7. Time flow chart for the NIC and the link overheads for traditional UDP/IP

overhead on the sender NIC is reflected as idle time for the link and the receiver NIC (shown as gaps between rectangles in the figure). This figure shows that memory and I/O contention at the host can result in the link being idle for nearly 50% of the time, in essense dropping the effective link bandwidth to half. Further, the startup overhead for the link (shown as the gap before the first rectangle) becomes especially important for applications which do not carry out bulk data transfers, e.g., latency micro-benchmark test.

Overall, these results attempt to demonstrate the various inefficiencies present in the current UDP/IP implementation due to contention for the existing system resources. In Section 4 we present several solutions to address these issues and implement a high performance UDP/IP stack using the features provided by the Myrinet network.

4 Enhancing UDP/IP Performance: Design Overview

In the following few subsections, we discuss several design solutions for the issues pointed out in the previous section. In particular, we look at the issues of (i) Compute requirements and contention, (ii) Memory contention, (iii) I/O bus contention and (iv) System resources' idle time and suggest efficient designs in the context of the Myrinet network to address these issues.

4.1 Partial Protocol Offloading for Avoiding CPU and Memory Contention

As discussed in Section 3, the host CPU performs per-byte operations such as data copy and checksum computation which result in not only an intensive consumption of CPU resources, but also memory contention with the DMA operations that access the host memory from the NIC. Further, as described earlier, checksum offload and zero-copy transmission and reception implemented by some protocol offload engines allows a significant reduction in these resource usage requirements. We regard basic checksum offloading and zero-copy data transmission as pre-requisites to our work and implement these on the Myrinet NIC firmware.

We can consider two design alternatives to offload the checksum computation on to the NIC. The first alternative is to let the NIC firmware perform the checksum computation on the data in the NIC memory. This can be performed in parallel with the protocol processing by the host CPU. Though this approach would reduce the contention for the CPU requirement on the host, it would merely shift the memory contention issue from the host memory to the NIC memory causing the checksum computation to contend with the host and network DMA operations. In addition, since the NIC processor is significantly slower than the host CPU, it is difficult to expect this approach to reduce the overhead for checksum computation.

The other approach is to utilize the checksum computation capability of the host-DMA engine. The checksum computation by the DMA engine does not generate any additional computation overhead or memory accesses since the data checksum is performed together with the DMA operation and before being placed into the NIC memory. This allows us to achieve both offloading as well as high performance. At the same time, the checksum offloading does not introduce any additional memory contention on the NIC memory.

Together with checksum computation, the data copy present in the traditional UDP implementation is another source of CPU and host memory contention. One way to remove the copy operation is to utilize the memory mapping mechanism between the application and the kernel memory spaces and use the mapped kernel memory space for all data communication. Another alternative is to directly move the data in the application buffer through DMA with the address and the length of the application buffer. Without the copy operation, the protocol layer can simply take the virtual address of the application buffer and translate it into the corresponding physical address by traversing the page directory and table. This physical address is used by the NIC to perform the DMA operations. Here, since we allow applications to use arbitrary application buffers, we have to consider the linearity of the buffer. In cases where the buffer is not physically linear, we construct gather or scatter lists so that the NIC can perform a DMA operation for each linear chunk in the application buffer separately. Both approaches have their own advantages and disadvantages, but for efficiency and ease of implementation, we chose the second approach.

4.2 Scheduling I/O Bus Transactions

As mention in Section 2.1, once the IP layer presents the information about the outgoing data to the link layer, the device driver forms a descriptor for the data message and pushes this descriptor to the NIC through a PIO operation. While this approach is quite efficient when there is little contention on the I/O bus, this might lead to a significant degradation of the performance when the I/O bus gets more heavily utilized. For example, bursty communication requests from applications can generate a high rate of descriptor posting to the network adapter with each descriptor posting requiring a PIO transaction across the I/O bus.

In order to resolve this problem, we propose a two-level queue structure to allow an efficient scheduling of the rate of descriptor posting without sacrificing the performance. Figure 8 shows the basic structure of two-level queue, where the level-1 (L1) queue is large and located in the host memory, while the level-2 (L2) queue is small and located in the NIC memory.

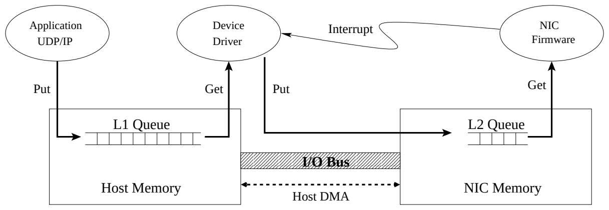


Figure 8. Two level queues to control I/O bus contention

When an application issues a send request, if the L2 send queue is not full, the device driver puts the descriptor corresponding to the request in the L2 send queue through a PIO operation. On the other hand, if the L2 send queue is full, the descriptor is just queued in the L1 queue. When the NIC firmware completes a send, an interrupt is triggered, which means that a vacancy is created in the L2 send queue. Accordingly, the interrupt handler gets a descriptor from the L1 send queue and puts it into the L2 send queue through a PIO operation. This approach ensures that the L1 send queue behaves in a self-clocking manner, i.e., it uses the interrupt for send completion as a clock to put a new descriptor into the L2 send queue. The description of the two-level queue on the receiver side is similar.

Consequently, the two-level queue structure can mitigate the rate of PIO to less than:

$$Rate_{max} = \frac{n + sizeof(L2)}{t}$$

where n is the maximum number of requests that the NIC can process in time t. And `sizeof(L2)` is the number of requests that L2 queue can hold.

The kernel and the NIC do not need to know any details about the queue structure because the device driver allocates and manages the L1 queue and can decide the size of the L2 queue. Accordingly, it is easy to apply to various operating systems and NICs. While we understand the importance of the performance impacts the size of the L2 queue might have, in this paper, we do not deal with its variation and fix it to a predefined value for all experiments.

4.3 Fine-Grained Pipelining to Minimize Link Idle Time

As mentioned earlier, the Myrinet NIC memory allows two memory accesses per clock cycle while each DMA engine can only request one memory access per clock cycle. This means that the transmission of a data segment by the send DMA engine can be pipelined with the DMA of the next segment by the host DMA engine (since these engines can read and write to the memory simultaneously[1]). The current GM architecture follows this approach for data transmission. We term this form of pipelining as *coarse-grained pipelining*. Earlier generation protocols over Myrinet such as Berkeley VIA achieve a much lower performance as they do not utilize this information about the memory clock speed at all and carry out the host DMA and the network DMA in a serialized manner. However, as discussed in Section 3, even a coarse-grained pipelining approach does not fully avoid the idle time in the link since the transmission is not carried out till the DMA of the entire segment is completed.

In order to address the issue of the link idle time, we propose a fine-grained pipelining approach between the network adapter and the Myrinet link. In the fine-grained pipelining approach, the NIC initiates its DMA operation as soon as a sufficient number of bytes have arrived from the host memory (through a host DMA) or the network (through a receive DMA). For instance, on the sender side the send DMA engine can send a packet out to the network while the host DMA engine is still moving a later part of the same packet from the host to the NIC memory. This approach allows us to fully pipeline the overheads of the send DMA and the host DMA for the same packet. It is to be noted that this approach only aims at reducing the per-byte overhead of the packet and does not impact the per-packet overhead associated with UDP/IP.

In order to achieve a fine-grained pipelining between the NIC and the link overheads, the following conditions should be satisfied. First, the NIC architecture has to allow multiple accesses to the NIC memory at a time by different DMA engines. The Myrinet NIC allows two DMA engines to access the NIC memory simultaneously. Second, the NIC has to provide an approach to send several fragments of a packet separately and realize that these fragments belong to the same segment on the receiver side. As described in Section 2, the Myrinet NIC provides two kinds of registers, namely SMLT and SML. The SMLT and SML registers can be used to indicate the end of the segment and the fragment, respectively. Third, the NIC firmware should force the host DMA engine to perform the DMA operation in parallel with the network DMA engines within the same packet.

While it is easily verifiable that the NIC satisfies the first two conditions, it is very difficult to verify that the NIC firmware always guarantees the third property irrespective of the communication pattern followed by the application. To address this, we propose a formal verification for the fine-grained pipelining based firmware model on the Myrinet NIC firmware, MCP.

The MCP performs coarse-grained overhead pipelining for IP based protocols, where DMA overheads across different packets are overlapped. As mentioned earlier, this technique only aims at reducing the per-byte overhead of the message. Therefore, it is more effective for large messages where the per-byte overhead is the dominant part as compared to small messages where the per-packet overhead is the dominant part.

The MCP consists of multiple threads: SDMA, RDMA, SEND, and RECV. The SDMA and SEND threads are responsible for sending. The SDMA thread moves data from the host memory to the NIC memory, and the SEND thread sends data in the NIC memory to the physical network. The receiving of data is performed by the RDMA and RECV threads. The RECV thread receives data from the physical network to the NIC memory. The RDMA thread moves the received data to the host memory.

Based on this design we suggest an extended firmware design for fine-grained pipelining as shown in Figure 9. In the figure, the states in the rectangle (with the dotted line) are the newly defined states for fine-grained overhead pipelining. The shaded states in the figure are dispatching entries of each thread. Each thread starts from the initial state and when it reaches a dispatching state, yields the processor to another thread that is ready to run without waiting for the next event to translate the state of the running thread. The yielding thread starts at a later point from the state in which the thread was suspended right before.

For example, let us consider the states for fine-grained pipelining on the RDMA thread. The initial state of the RDMA thread is the Idle state. If the amount of data arrived is more than the threshold for fine-grained pipelining, the state is moved to Fine_Grained_Rdma, where the RDMA thread initiates the host DMA operation to move the data in the NIC memory to the host memory. After finishing this DMA operation, in the Fine_Grained_Rdma_Done state, if there is still more data than the threshold, the RDMA thread performs the host DMA operation again moving to the Fine_Grained_Rdma state. Otherwise, if the receive DMA has completed, the state of the RDMA thread is changed to the Rdma_Last_Fragment state. In this state, the RDMA thread does the DMA operation for the rest of the packet regardless of its size.

[1]In theory, unlike dual ported memory interfaces, the Myrinet NIC memory does not provide truly concurrent access. However, the number of requests the memory can handle is twice the number of requests each DMA engine can generate. So, the accesses can be assumed to be concurrent.

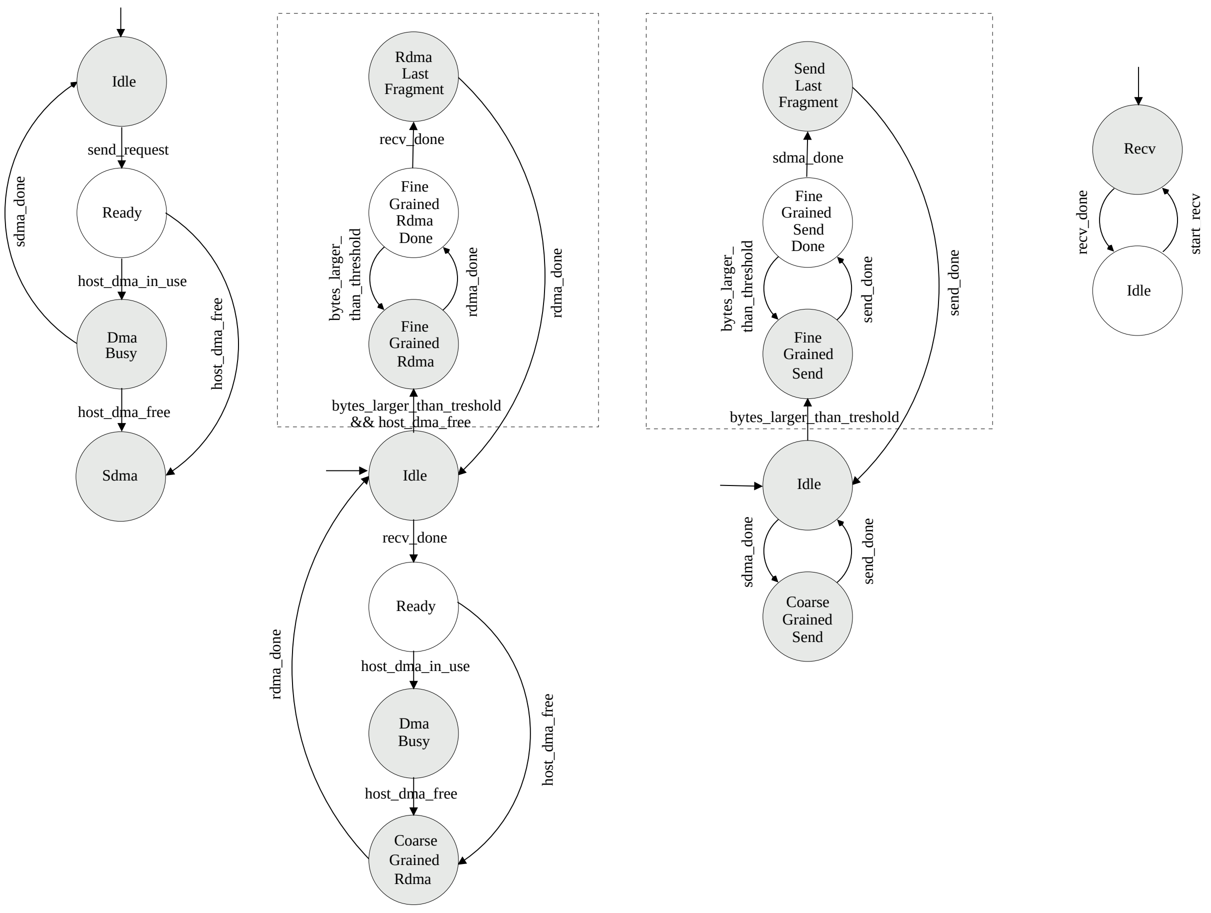


Figure 9. State Transition Model for: (a) SDMA, (b) RDMA, (c) SEND, and (d) RECV threads

In order to verify the correctness of the about state transition diagram, we used the Spin verification tool. Spin [16] is a tool for formal verification of distributed software systems, such as operating systems and data communication protocols. It uses a high level language to specify a system, called PROMELA (PROcess MEta LAnguage) [15]. Given a model specified in PROMELA, Spin verifies that the model satisfies properties written in linear temporal logic (LTL) [22]. LTL was designed for expressing temporal ordering of events and variables; it characterizes the behavior of systems by specifying some behavior that all traces of a system must obey.

For the formal verification, we first translate the state transition diagrams of Figure 9 into specifications written in PROMELA. We then define propositional symbols and derive verification formulas written in LTL as follows:

Symbol Definitions:

```
#define sdma
        (SDMA_state == Sdma)
#define coarse_rdma
        (RDMA_state == Coarse_Grained_Rdma)
#define fine_sdma
        (RDMA_state == Fine_Grained_Rdma)
#define coarse_send
        (SEND_state == Coarse_Grained_Send)
#define fine_send
        (SEND_state == Fine_Grained_Send)
#define recv
        (RECV_state == Recv)
```

Formula 1: <> (sdma && (coarse_send || fine_send))
Can the SEND thread initiate a send DMA while the SDMA thread performs a host DMA, and vice versa?

Formula 2: <> ((coarse_rdma || fine_rdma) && recv)
Can the RECV thread initiate a receive DMA while the RDMA thread performs a host DMA, and vice versa?

Formula 3: <> ((coarse_send || fine_send) && recv)
Can the SEND thread initiate a send DMA while the RECV thread performs a receive DMA, and vice versa?

Formula 4: [ ] (sdma && (coarse_rdma || fine_rdma))
The SDMA thread cannot use the host DMA engine while RDMA thread utilizes it, and vice versa.

Formulas 1 and 2 represent the properties that the suggested model performs the fine-grained pipelining. Formula 3 represents that the model utilizes the Myrinet link in full-duplex mode. Formula 4 ensures correctness, i.e., only one thread between SDMA and RDMA should occupy the host DMA engine at a time. Using these formulas with Spin, we formally verified that the above presented model performs fine-grained pipelining with any generic communication pattern.

4.4 Performance Modeling

To analyze the performance of our design described in Sections 4.1, 4.2, and 4.3 on various systems, we propose a mathematical performance model. First, we derive the performance model for coarse-grained pipelining as a base model to compare against. Next, we describe the performance model for fine-grained pipelining. Both models implement the partial protocol offloading as well as the two-level queuing and differ only in the pipelining mechanism.

4.4.1 Coarse-Grained Pipelining

In the coarse-grained pipelining model, pipelining between the overheads for the $(p+1)^{th}$ packet at the i^{th} layer and the p^{th} packet at the $(i+1)^{th}$ layer occurs as shown in Figure 10a, where the smaller numbered layer is the upper layer. In the figure, g_i and G_i denote the per-packet and the per-byte overheads at the i^{th} layer, respectively. B is the byte size of the packet.

In this case, the one-way latency and the bandwidth are given by Equations 1 and 2, respectively, where n is the number of layers that perform the overhead pipelining. The subscript b represents the bottleneck layer, and m is the number of packets. We analyze the bandwidth for a large number of packets (i.e., $m \to \infty$) since we assume that the test would use a massive data transmission.

4.4.2 Fine-Grained Pipelining

Based on the implementation of partial protocol offloading and two-level queuing, we can achieve fine-grained pipelining with the firmware model described in Section 4.3. In the fine-grained pipelining approach, a layer initiates its per-byte processing as soon as a sufficient number of bytes have arrived from the upper layer. Therefore, the overhead of a packet at the i^{th} layer and the same packet at the $(i+1)^{th}$ layer are fully pipelined except the per-packet overhead as shown in Figure 10b.

In the figure, S_i is the start-up overhead that is required to start a per-byte operation such as a DMA operation, g'_i is the per-packet overhead excepting the start-up time S_i (i.e., $g'_i = g_i - S_i$) and $x_{f,p,i}$ is the size of f^{th} fragment of the p^{th} packet in the i^{th} layer.

For the first step, in order to model the one-way latency for fine-grained pipelining, let us define the parameters for the first packet (i.e., $p = 1$). A characteristic of fine-grained pipelining is that the per-byte overhead of a layer is affected by the per-byte overhead of the upper layer. Accordingly, we define a new per-byte overhead, $\Gamma_{p,i}$ as follows:

$$\Gamma_{1,i} = \begin{cases} \frac{B \cdot G_1 + S_1}{B} & \text{if } (i = 1) \\ \frac{B \cdot G_i + k_{1,i} \cdot S_i + (k-1) \cdot y_{1,i}}{B} & \text{if } (i > 1) \end{cases} \quad (3)$$

where the number of fragments, k, is:

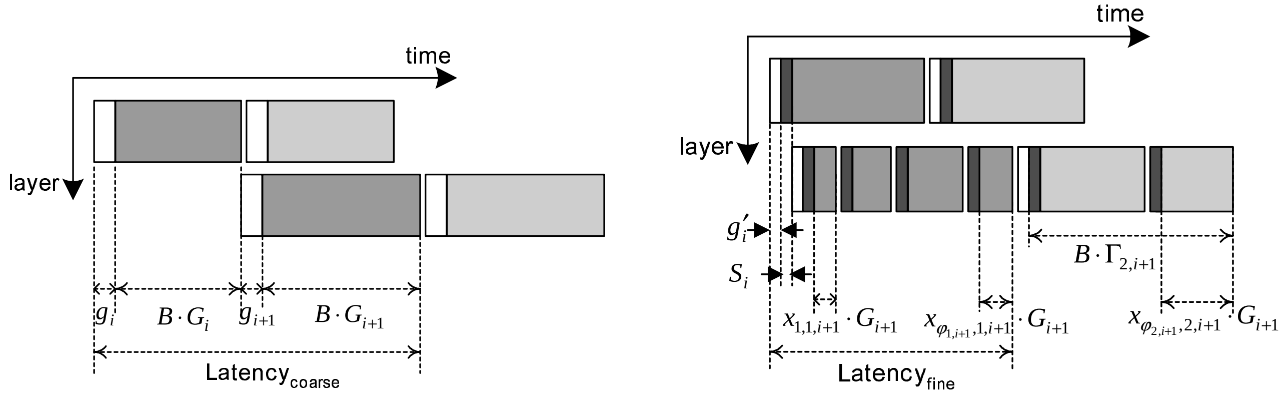


Figure 10. Overhead Pipelining: (a) Coarse Grained; (b) Fine Grained

$$Latency_{coarse} = \sum_{i=1}^{n}(g_i + B \cdot G_i), \tag{1}$$

$$\begin{aligned} Bandwidth_{coarse} &= \lim_{m \to \infty} \frac{B \cdot m}{(m-1)\cdot(g_b + B \cdot G_b) + \sum_{i=1}^{n}(g_i + B \cdot G_i)} \\ &= \frac{B}{g_b + B \cdot G_b} \end{aligned} \tag{2}$$

$$k_{1,i} = \begin{cases} 1 & \text{if } (i = 1, \Gamma_{1,i-1} = 0, or G_i = 0) \\ \frac{B}{t} & \text{if } \left(\frac{t \cdot G_i + S_i}{\Gamma_{1,i-1}} < t\right) \\ \alpha_i & \text{if } \left(\frac{t \cdot G_i + S_i}{\Gamma_{1,i-1}} \geq t\right) \end{cases} \tag{4}$$

t is the threshold, i.e., the minimum number of bytes that make a layer start its per-byte processing; α_i is the number of fragments when the data processing of the upper layer is faster than the current layer, so it satisfies Equation 5.

The $y_{1,i}$ parameter in the Equation (3) denotes the dormant time between fragments. In other words, $y_{1,i}$ is the waiting time for t bytes to arrive from the upper layer. Therefore, $y_{1,i}$ is defined as shown in Equation (8).

As a result, the one-way latency of the fine-grained pipelining is as follows:

$$Latency_{fine} = \sum_{i=1, i \neq e}^{n} (g_i + t \cdot G_i) + g'_e + B \cdot \Gamma_{1,e} \tag{9}$$

where e is the last layer such that $\Gamma_{1,e} \neq 0$.

In order to model the bandwidth, we define a parameter, u, which is the number of packets after which the fine-grained pipelining is changed into the coarse-grained pipelining. This occurs because when a layer passes the data to the next layer which happens to be slower, data accumulates in the buffer of the next layer triggering progressively larger transfers in a ripple effect. u is the minimum integer satisfying the following equation:

Therefore, for all packets from the first to the u^{th}, the per-byte overheads (i.e., $\Gamma_{p,i}$) are the same regardless of p. On the other hand, after the u^{th} packet, the per-byte overheads are the same as that of coarse-grained pipelining (i.e., G_i). Thus, the bandwidth of fine-grained pipelining is given by Equation 15.

An interesting result is that Equation (15) is the same with Equation (2). This is because fine-grained pipelining switches to coarse-grained pipelining after the u^{th} packet when a large number of packets are transmitted in burst. As a result, fine-grained pipelining can achieve a low latency without sacrificing the bandwidth.

5 Experimental Results

In this section, we first present the capability of our E-UDP implementation to avoid the various inefficiencies pointed out in Section 3. Next, we present the performance achieved by E-UDP compared to other protocol implementations. Finally, we present the results of our analytical model showing the performance of E-UDP for various system and network configurations.

For the evaluation, we used a pair of machines equipped with an Intel Pentium III 1GHz processor on an ASUS

$$\sum_{j=1}^{\alpha_i} x_{j,1,i} = t + \sum_{j=2}^{\alpha_i} \left(\frac{t \cdot G_i^{j-1}}{\Gamma_{1,i-1}^{j-1}} + \sum_{n=2}^{j} \left(\frac{S_i \cdot G_i^{n-2}}{\Gamma_{1,i-1}^{j-1}} \right) \right) = B \tag{5}$$

Proof of Equation (5).

$$\begin{aligned}
x_{1,1,i} &= t \\
x_{2,1,i} &= \frac{x_{1,1,i} \cdot G_i + S_i}{\Gamma_{1,i-1}} = \frac{t \cdot G_i}{\Gamma_{1,i-1}} + \frac{S_i}{\Gamma_{1,i-1}} \\
x_{3,1,i} &= \frac{x_{2,1,i} \cdot G_i + S_i}{\Gamma_{1,i-1}} = \frac{t \cdot G_i^2}{\Gamma_{1,i-1}^2} + \frac{S_i \cdot G_i}{\Gamma_{1,i-1}^2} + \frac{S_i}{\Gamma_{1,i-1}} \\
x_{4,1,i} &= \frac{x_{3,1,i} \cdot G_i + S_i}{\Gamma_{1,i-1}} = \frac{t \cdot G_i^3}{\Gamma_{1,i-1}^3} + \frac{S_i \cdot G_i^2}{\Gamma_{1,i-1}^3} + \frac{S_i \cdot G_i}{\Gamma_{1,i-1}^2} + \frac{S_i}{\Gamma_{1,i-1}} \\
&\vdots \\
x_{\alpha,1,i} &= \frac{x_{\alpha_i-1,1,i} \cdot G_i + S_i}{\Gamma_{1,i-1}} = \frac{t \cdot G_i^{\alpha_i-1}}{\Gamma_{1,i-1}^{\alpha_i-1}} + \frac{S_i \cdot G_i^{\alpha_i-2}}{\Gamma_{1,i-1}^{\alpha_i-1}} + \frac{S_i \cdot G_i^{\alpha_i-3}}{\Gamma_{1,i-1}^{\alpha_i-2}} + \cdots + \frac{S}{\Gamma_{1,i-1}}
\end{aligned} \tag{6}$$

$$\therefore \sum_{j=1}^{\alpha_i} x_{j,1,i} = t + \sum_{j=2}^{\alpha_i} \left(\frac{t \cdot G_i^{j-1}}{\Gamma_{1,i-1}^{j-1}} + \sum_{n=2}^{j} \left(\frac{S_i \cdot G_i^{n-2}}{\Gamma_{1,i-1}^{n-1}} \right) \right) \tag{7}$$

□

$$y_{1,i} = \begin{cases} (k_i - 1) \cdot \left(t - \frac{t \cdot G_i + S_i}{\Gamma_{1,i-1}} \right) \cdot \Gamma_{1,i-1} & \text{if} \left(\frac{t \cdot G_i + S_i}{\Gamma 1,i-1} < t \right) \\ 0 & \text{if} \left(\frac{t \cdot G_i + S_i}{\Gamma 1,i-1} \geq t \right) \end{cases} \tag{8}$$

$$\frac{g_b + B \cdot \Gamma_b}{(g_b + B \cdot \Gamma_b) - (g_{b-1} + B \cdot \Gamma_{b-1})} \leq u < \frac{2 \cdot (g_b + B \cdot \Gamma_b) - (g_{b-1} + B \cdot \Gamma_{b-1})}{(g_b + B \cdot \Gamma_b) - (g_{b-1} + B \cdot \Gamma_{b-1})} \tag{10}$$

Proof of the lower bound of Equation (10).

$$\begin{aligned}
u \cdot (g_{b-1} + B \cdot \Gamma_{b-1}) &\leq (u-1) \cdot (g_b + B \cdot \Gamma_b) \\
g_b + B \cdot \Gamma_b &\leq u \cdot (g_b + B \cdot \Gamma_b - g_{b-1} - B \cdot \Gamma_{b-1})
\end{aligned} \tag{11}$$

$$\therefore \frac{g_b + B \cdot \Gamma_b}{(g_b + B \cdot \Gamma_b) - (g_{b-1} + B \cdot \Gamma_{b-1})} \leq u \tag{12}$$

□

Proof of the upper bound of Equation (10).

$$\begin{aligned}
(u-1) \cdot (g_{b-1} + B \cdot \Gamma_{b-1}) &> (u-2) \cdot (g_b + B \cdot \Gamma_b) \\
u \cdot (g_{b-1} + B \cdot \Gamma_{b-1} - g_b - B \cdot \Gamma_{b-1}) &> (g_{b-1} + B \cdot \Gamma_{b-1}) - 2 \cdot (g_b + B \cdot \Gamma_b)
\end{aligned} \tag{13}$$

$$\therefore u < \frac{2 \cdot (g_b + B \cdot \Gamma_b) - (g_{b-1} + B \cdot \Gamma_{b-1})}{(g_b + B \cdot \Gamma_b) - (g_{b-1} + B \cdot \Gamma_{b-1})} \tag{14}$$

□

$$\begin{aligned}
Bandwidth_{fine} &= \lim_{m\to\infty} \frac{B \cdot m}{\sum_{i=1}^{n-1} g'_i + \sum_{p=1}^{m} (g'_n + B \cdot \Gamma_{p,n})} \\
&\approx \lim_{m\to\infty} \frac{B \cdot m}{\sum_{i=1}^{n-1} g'_i + \sum_{p=1}^{m} (g'_b + B \cdot G_b + k_{p,b} \cdot S_b)} \\
&= \lim_{m\to\infty} \frac{B \cdot m}{\sum_{i=1}^{n-1} g'_i + m \cdot (g'_b + B \cdot G_b) + \sum_{p=1}^{u} (k_{p,b} \cdot S_b) + \sum_{p=u}^{m} S_b} \\
&= \frac{B}{g_b + B \cdot G_b}
\end{aligned} \tag{15}$$

motherboard (Intel 815EP chipset). Each machine has a Myrinet NIC (LANai 9.0) set to a 32bit 33MHz PCI slot, and the NICs are directly connected to each other through the Myrinet-1280 link. The Linux kernel version used is 2.2, and we adopt GM (version 1.4) for the device driver and the firmware on the Myrinet NIC. The MTU size is set to 32KB.

5.1 System Resource Consumption in E-UDP

To analyze the effect of partial protocol offloading on the CPU overhead, we measured the overhead on both the host and the NIC CPUs. Figures 11a and 11b compare the CPU overheads of the original UDP and E-UDP for small (1B) and large (32KB) message sizes respectively. For small messages, though the copy operation and the checksum computation overheads are small, we can see a slight reduction in the CPU overhead, especially on the receiver side. On the other hand, the NIC overheads of E-UDP are increased due to the offload of the copy operation and the checksum computation. Further, some other functionalities such as a part of the UDP/IP header manipulation also have been moved to NIC. Overall, the accumulated overhead on both the host and the NIC are nearly equal (44.7us on E-UDP Vs. 42.3us on original UDP).

For large messages, however, we can see a large benefit through partial protocol offloading. By offloading the per-byte operations from the host, we achieve a very small host overhead regardless of the message size. At the same time, there is no significant increase in the overhead on the NIC.

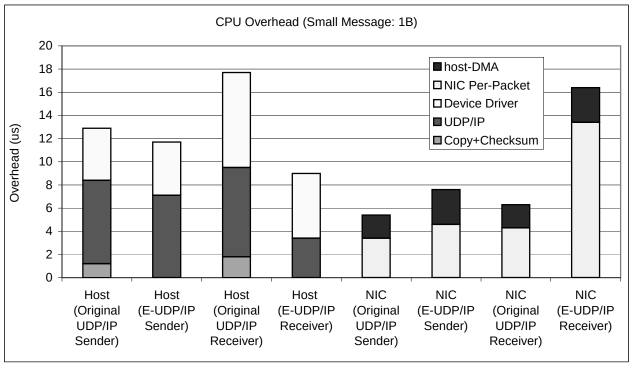
To observe whether E-UDP resolves the resource contention and the idle resource problems, we study the time flow chart of E-UDP for 10 packets each of 32KB size. The time flow chart of the original UDP has already been shown in Section 3. Figure 12 shows the time chart of E-UDP. We can see that the overhead of each layer can be fully pipelined with the others from the sender side to the receiver side. This is due to the fact that E-UDP eliminates the CPU, the memory, and the I/O bus contentions. In addition, E-UDP performs a fine-grained overhead pipelining to overlap the NIC and the link overheads. Consequently, the largest overhead (i.e., the NIC overhead) hides the smaller overheads and allows us to achieve a performance close to the theoretical maximum.

5.2 Latency and Bandwidth

In this section, we compare the performance of E-UDP with that of the original UDP, GM and Berkeley-VIA [9], a well-known implementation of VIA. Since there is no implementation of Berkeley-VIA on LANai9, we measured its performance with a LANai4 based Myrinet NIC on the same platform.

Figure 13 compares the latency of E-UDP, GM, Berkeley-VIA, and original UDP with the theoretical minimum latency of the experimental system for large and small messages respectively. Since the bottleneck of the system is the PCI bus (1007Mbps of PCI Vs. 1310Mbps of Myrinet-1280), the theoretical maximum performance in this subsection is derived from that achievable by the PCI bus used. We can observe that the latency of E-UDP is smaller than the others and almost even with the theoretical minimum latency. An interesting result is that the latency of E-UDP is even smaller than that of the user-level protocols, such as GM and Berkeley-VIA for large message sizes. This is because E-UDP performs fine-grained overhead pipelining on the NIC. On the other hand, in the case of small data sizes (Figure 13b), GM shows the smallest one-way latency. This is because for small messages, the per-packet overhead becomes the dominant factor. User-level protocols have a lower per-packet overhead compared to UDP/IP following which they are able to achieve a lower latency.

Figure 14 shows the bandwidth achieved by E-UDP compared to the other protocols. A notable result is that E-



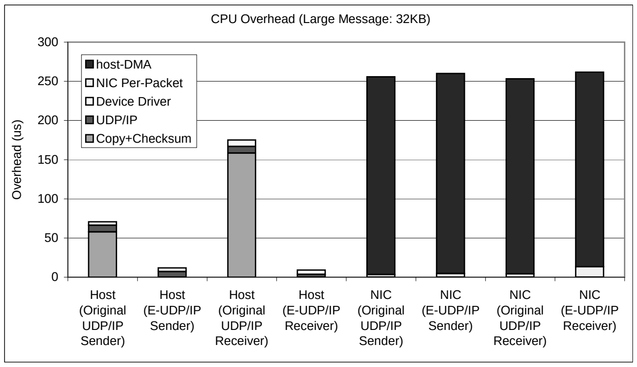


Figure 11. CPU overhead on the host and the NIC: (a) Small messages (1 byte); (b) Large messages (32 Kbytes)

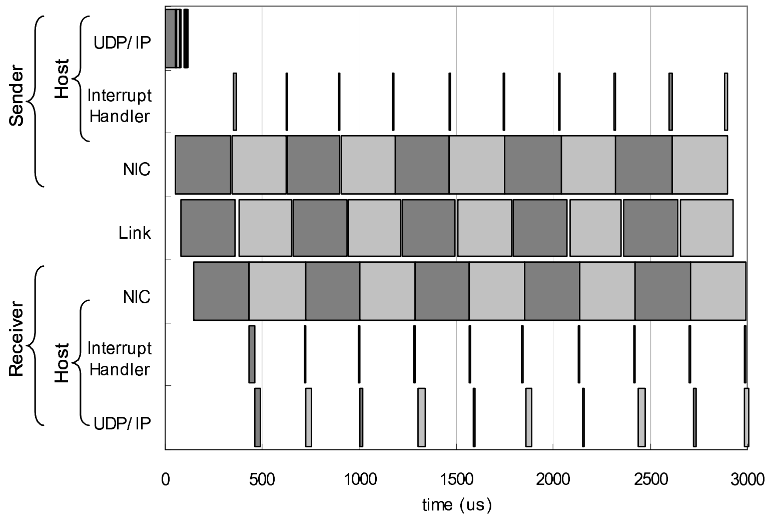
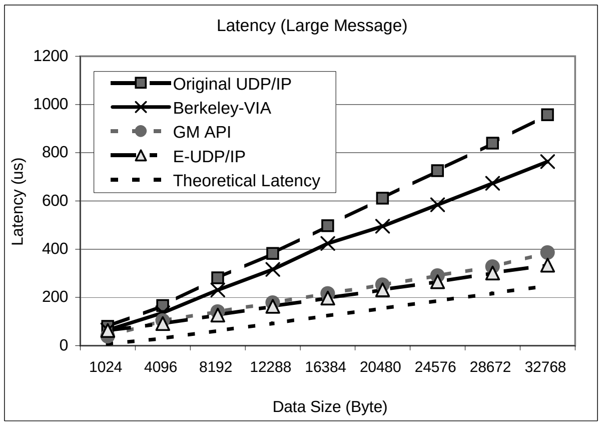


Figure 12. Time Flow Chart for E-UDP



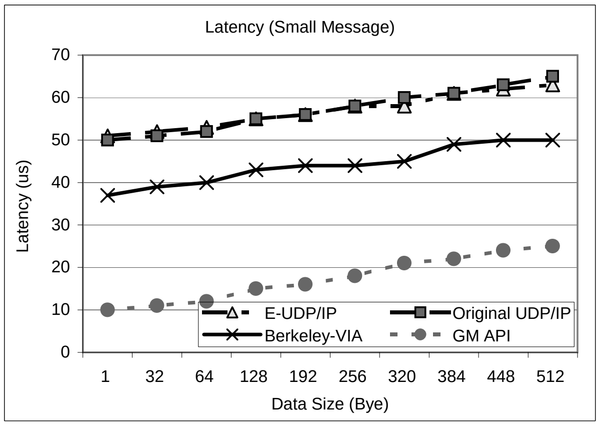


Figure 13. Latency Comparison for (a) Large messages; (b) Small messages

UDP achieves a peak bandwidth of 951Mbps which is about 94% of the theoretical maximum bandwidth of the experimental system. The bandwidth of GM is slightly lower than that of E-UDP. This is because GM employs middle-grained pipelining that splits a large packet into several segments with a fixed size of 4KB as shown in Figure 15. This segmentation increases the packet processing overhead in proportion to the number of segments. Consequently, the throughput suffers.

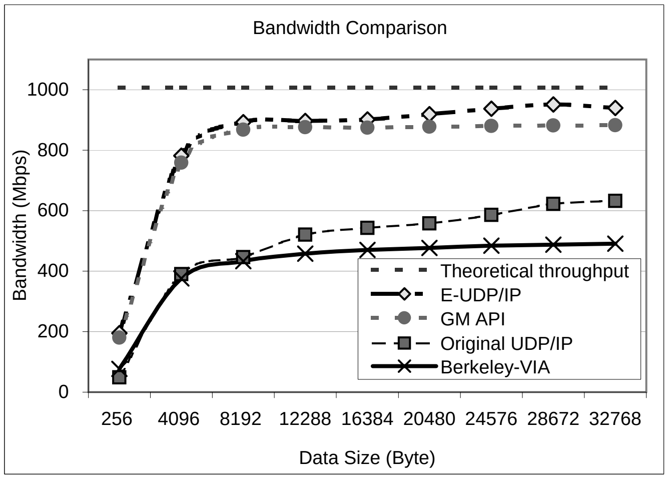


Figure 14. Bandwidth Comparison

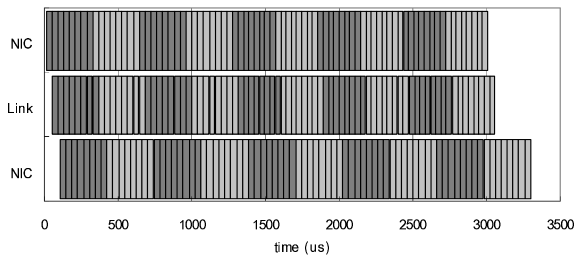


Figure 15. Middle-Grained Overhead Pipelining on GM

An unexpected result in Figure 14 is that for large data sizes (larger than 1KB) Berkeley-VIA shows a much lower throughput than the others. This can be partly because we measured its performance with a LANai4 based NIC. However, since the rest of the experimental system was the same, the data-touching portion is unchanged between LANai4 and LANai9. Based on this, the drop in bandwidth is attributed to the lack of pipelining between the NIC and the link overheads, i.e., the NIC firmware of Berkeley-VIA serializes DMA operations even for those performed by different DMA engines without taking advantage of the capabilities of the Myrinet NIC memory to allow simultaneous access to both the DMA engines. For example, Figure 16 shows that the NIC and link overheads are never overlapped on the sender running Berkeley-VIA.

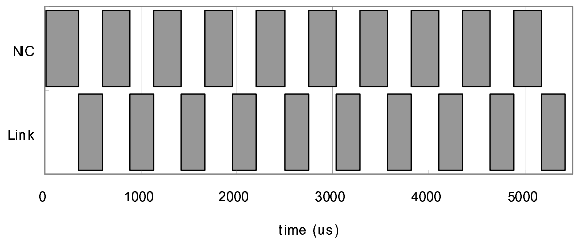


Figure 16. Serialization of NIC and Link Overheads on Berkeley-VIA

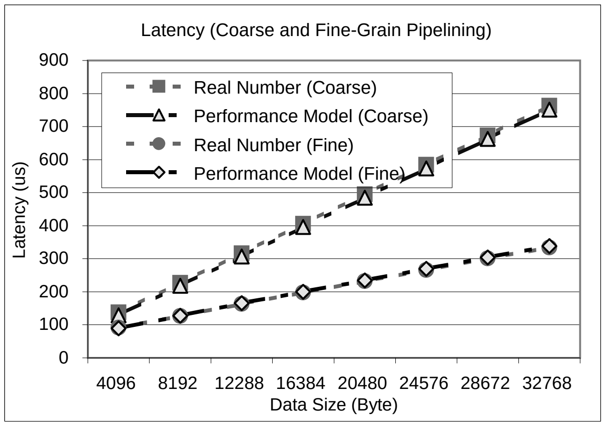
5.3 Performance Modeling Results

To verify that the mathematical performance model suggested in Section 4.4 is accurate, we compare the performance evaluated from the model with the real performance numbers on our test-bed. Figures 17a and 17b show the latency and bandwidth comparisons, respectively. Since, from the performance model equations in Section 4.4, the bandwidth of the coarse and the fine-grained pipelining mechanisms are the same, Figure 17b does not deal with coarse and fine-grain pipelining separately. Real data for coarse-grain pipelining refers to a version of UDP with partial protocol offloading and two level queuing for the I/O requests, i.e., it differs from E-UDP in only the pipelining mechanism. As we can see in the figures, the performance model matches the actual data very closely with an error of less than 5%.

Based on the performance model, we tried to analyze the latency and bandwidth of E-UDP on faster networks and I/O buses than our evaluation system. We considered 2Gbps and 10Gbps networks to reflect the characteristics of the emerging networks such as Myrinet-2000, 10-Gigabit Ethernet, and InfiniBand. In addition, we took account of 64bit/66MHz and 64bit/133MHz PCI systems. We used the per-packet overhead values measured on E-UDP and decided the per-byte overhead values according to the target network and the PCI bus speed.

Figures 18a and 18b show the latency and bandwidth on a 2Gbps network with a 64bit/66MHz PCI bus. We can see that in this case the fine-grained pipelining can achieve a very low latency for large message sizes compared to coarse-grained pipelining. In addition, the rate of increase of the overhead is equal to that of the network link latency; this shows that fine-grained pipelining is able to hide the DMA overhead on the NIC behind the link overhead successfully. Moreover, Figure 18b shows that both overhead pipelining mechanisms can achieve a near physical bandwidth.

The results on a 10Gbps network and a 64bit/133MHz



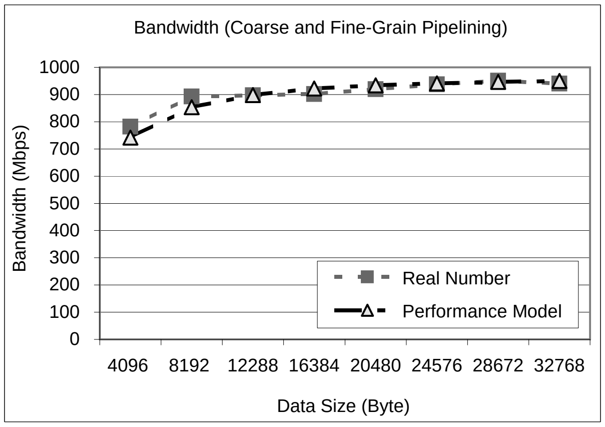
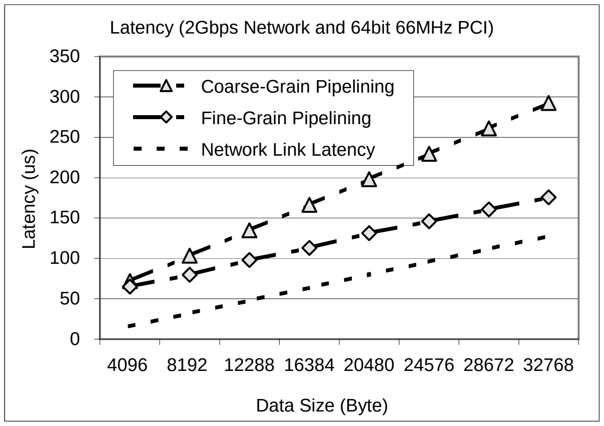


Figure 17. Performance Modeling Results Verification: (a) Latency; (b) Bandwidth



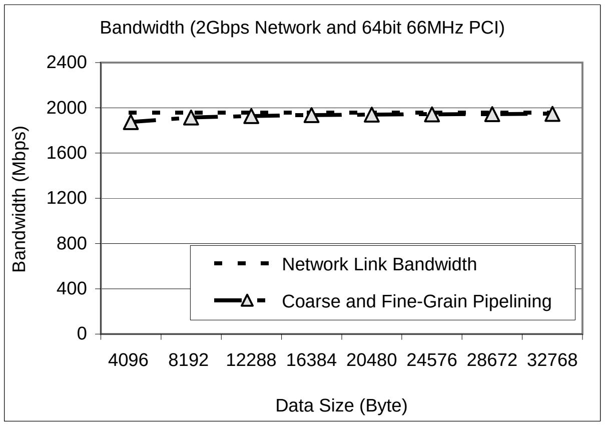


Figure 18. Latency and Bandwidth measurements on a 2Gbps network with a 64bit/66MHz I/O bus

PCI bus are shown in Figures 19a and 19b. We can see that even fine-grained pipelining cannot fully utilize the network link. The reason for this is that the PCI bus is the bottleneck and not the network. Therefore, the performance is limited by the PCI bus. Another interesting observation is that the per-packet overhead becomes more important as the speed of the network and the I/O bus become faster. As shown in Figure 19a, although fine-grained pipelining can reduce the latency, it is still high because of the large per-packet overhead compared to the per-byte overheads such as the DMA and the link overheads.

6 Related Work

Several researchers have worked on implementing high performance user-level sockets implementations over high performance networks. Balaji, et. al have worked on such pseudo sockets layers over Gigabit Ethernet [6], GigaNet cLAN [7] and InfiniBand [5]. However, these sockets layers only deal with compatibility issues with existing applications and do not focus on compatibility with the existing IP infrastructure.

Trapeze [11] has implemented zero copy, checksum offloading, and a form of overhead pipelining based on TCP/IP on Myrinet. This research is notable in the sense that this was the one of the first to show that an IP based protocol can achieve a significantly high performance. However, Trapeze provides a different API from the sockets interface and is not compatible with the traditional TCP/IP implementation. In our paper, we try to achieve a near theoretical performance while keeping the socket interface and compatibility existing the UDP/IP implementations.

For overhead pipelining, several studies have been done to achieve a middle-grained pipelining [25, 36, 18], which splits a large packet into smaller sized segments so that the overheads for a packet at a layer and the same packet at the next layer are partly pipelined. The middle-grained pipelining can achieve a lower latency than coarse-grained pipelining but is not as efficient as fine-grained pipelining. Further, since middle-grained pipelining splits a packet into separate segments, each with a separate segment header, it sacrifices some of the bandwidth.

A notable research on formal verification of the NIC firmware is performed by Kumar et al. [21], which employs a model checking approach to implement the NIC firmware. They use Event-driven State-machine Programming (ESP); a language for writing firmware for programmable devices to verify the retransmission protocol, memory safety, and the deadlock free property of the VMMC [13] firmware. We expect that their approach can effectively help implement our suggested model.

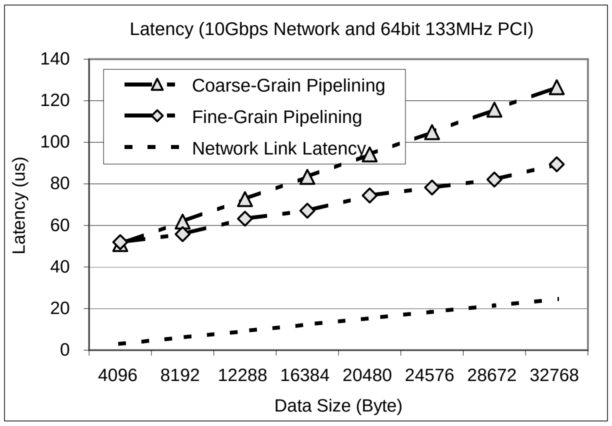
7 Concluding Remarks and Future Work

While a number of user-level protocols have been developed to reduce the gap between the performance capabilities of the physical network and the performance actually available, their compatibility issues with the existing sockets based applications and IP based infrastructure has been an area of major concern. To address these compatibility issues while maintaining a high performance, a number of researchers have been looking at alternative approaches to optimize the existing traditional protocol stacks. Broadly, previous research has broken up the overheads in the traditional protocol stack into four related aspects, namely: (i) Compute requirements and contention, (ii) Memory contention, (iii) I/O bus contention and (iv) System resources' idle time.

There has been some previous research which deals with some of these aspects. For example, Protocol Offload Engines (POEs) have been recently proposed as an industry standard for offloading the compute intensive components in protocol processing to specialized hardware. However, these approaches require network adapters supported with specialized ASIC based chips which implement the protocol processing and are not generic enough to be implemented on most network adapters. Further, these deal with only the compute requirement and memory contention issues and do not address the remaining issues. In short, to the best of our knowledge, there is no work which deals with all these issues in an integrated manner while maintaining backward compatibility with existing applications and infrastructure.

In this paper, we address each of these issues and propose solutions for minimizing these overheads. We also modify the existing UDP/IP implementation over Myrinet to demonstrate the capabilities of these solutions. We first utilize the earlier proposed techniques to implement a partial offload of the UDP/IP protocol stack to address the first two issues, i.e., compute requirements and memory contention. Next, we modify the device driver to allow a delayed posting of descriptors in order to reduce the contention at the I/O bus between descriptor posting and the DMA operations of the actual outgoing or incoming data. Finally, we implement a fine-grained pipelining technique on the firmware of the network adapter to minimize the link idle time in order to achieve a high performance. Further, all these enhancements to the UDP stack are completely compatible not only with existing applications and infrastructure, but also with the existing UDP implementations. Our experimental results show that with our implementation of UDP, termed as E-UDP, can achieve up to 94% of the theoretical maximum bandwidth. We also present a mathematical performance model which allows us the study the performance of our design for various system architectures and network speeds.

We have done some preliminary work earlier to show that several features provided by Myrinet network including link level flow control can be used to provide reliability in data transmission. This relieves us of the requirement of a heavy



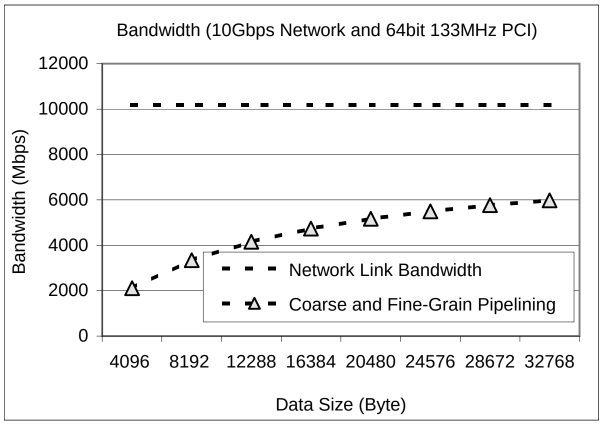


Figure 19. Latency and Bandwidth measurements on a 10Gbps network with a 64bit/133MHz I/O bus

protocol such as TCP/IP and allows us to achieve high performance. As a part of our future work, we plan to integrate this kind of reliability into E-UDP. Also, we are currently looking into offloading the per-packet overheads in the current UDP/IP stack on to the network adapter. This would not only allow us to achieve a better performance for small messages (where the per-packet overhead is dominant), but also a better scalability for faster networks.

References

[1] 10 Gigabit Ethernet Alliance. http://www.10gea.org/.

[2] InfiniBand Trade Association Specifications. http://www.infinibandta.org/estore.html.

[3] Quadrics Supercomputers World Ltd. http://www.quadrics.com/.

[4] InfiniBand Trade Association. http://www.infinibandta.org.

[5] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *the Proceedings of ISPASS 2004*, Austin, Texas, March 10-12 2004.

[6] P. Balaji, P. Shivam, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *the Proceedings of Cluster'02*, pages 179–186, Chicago, Illinois, September 23-26 2002.

[7] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *the Proceedings of HPDC-12*, pages 24–33, Seattle, Washington, June 22-24 2003.

[8] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[9] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *Proceedings of Supercomputing*, 1998.

[10] J. Chase, A. Gallatin, A. Lebek, and Y. G. Yocum. Trapeze API. Technical report, Duke University, 1997.

[11] J. Chase, A. Gallatin, and K. Yocum. End-System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 2000.

[12] Myricom Corporations. The GM Message Passing System.

[13] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection Oriented Communication. In *Proceedings of Hot Interconnects*, 1997.

[14] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *Proceedings of ICS '03*, Phoenix, Arizona, November 2003.

[15] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[16] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, May 1997.

[17] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro*, January 2004.

[18] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. E. II, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reducing Network Latency Using Subpages in a Global Memory Environment. In *Proceedings of ASPLOS-VII*, October 1996.

[19] H. W. Jin, P. Balaji, C. Yoo, J. Y. Choi, and D. K. Panda. Exploiting NIC Architectural Support for Enhancing IP based Protocols on High Performance Networks. Technical report, Ohio State University, Columbus, Ohio, May 2004.

[20] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *the Proceedings of Cluster '01*.

[21] S. Kumar and K. Li. Using Model Checking to Debug Device Firmware. In *Proceedings of OSDI*, 2002.

[22] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[23] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of Supercomputing*, 1995.

[24] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *the Proceedings of Hot Interconnects 9*, August 2001.

[25] L. Prylli, R. Westerlin, and B. Tourancheau. Modeling of a High Speed Network to Maximize Throughput Performance: the Experience of BIP over Myrinet. In *Proceedings of PDPTA '98*, 1998.

[26] Quadrics Supercomputers World Ltd. Elan Programming Manual. 1999.

[27] Quadrics Supercomputers World Ltd. Elan Reference Manual. 1999.

[28] Quadrics Supercomputers World Ltd. Elite Reference Manual. 1999.

[29] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. *IEEE Micro*, pages 24–31, 2004.

[30] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, R. S. Madukkarumukumana, and G. J. Regnier. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *the Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages pages 61–72, San Francisco, CA, March 2001.

[31] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *the Proceedings of the CANPC workshop (held in conjunction with HPCA)*, pages 91–107, 1999.

[32] P. Shivam, P. Wyckoff, and D. K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *the Proceedings of ICS*, pages 57–64, Denver, Colorado, November 10-16 2001.

[33] P. Shivam, P. Wyckoff, and D. K. Panda. Can User-Level Protocols Take Advantage of Multi-CPU NICs? In *the Proceedings of IPDPS*, Fort Lauderdale, Florida, April 15-19 2002.

[34] Y. Turner, T. Brecht, G. Regnier, V. Saletore, G. Janakiraman, and B. Lynn. Scalable Networking for Next-Generation Computing Platforms. In *Proceedings of SAN*, 2004.

[35] 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 SOSP*, December 1995.

[36] R. Y. Wang, A Krishnamurthy, R. P. Martin, T. E. Anderson, and D. E. Culler. Modeling and Optimizing Computation Pipelines. In *Proceedings of SIGMETRICS*, June 1998.

[37] E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth. Introduction to TCP/IP Offload Engine (TOE). http://www.10gea.org, May 2002.