Device Driver Issues in High-Performance Networking

John Michael Tracey Arindam Banerji
Distributed Computing Research Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556-5637
(219) 631-5273 (voice) (219) 631-9260 (FAX)
{jmt,axb}@cse.nd.edu

Abstract

High-performance networking requires attention to operating system support at the device driver level. Existing driver models, such as those of Unix, are not necessarily well-suited to supporting high-speed network interfaces. In fact, current drivers may represent a significant obstacle between applications and the high-speed network adapters they seek to exploit. Yet existing models cannot simply be discarded. Certain trends in RISC processor design can also tend to make existing device driver implementations less efficient. Specifically, many drivers make extensive use of operations which are becoming relatively more costly as RISC architectures evolve. This paper describes an effort currently underway to develop device drivers specifically designed to support high-speed network interfaces on RISC architectures. We have analyzed the operation of some existing commercial device driver implementations and modified one using several techniques. Taken together, these techniques promise to produce network device drivers which deliver the high level of performance demanded by today's high speed networks.

1. Introduction

The increasingly prevalent demands for high bandwidth, and in some cases low-latency network connectivity by a wealth of applications is well known. These growing application demands have been met by significant advancements in network hardware. Modern network architectures generally provide bandwidths at least an order of magnitude greater than traditional networks such as Token-Ring and Ethernet. Interconnections capable of providing gigabit bandwidths to the desktop over long distances have been demonstrated [Clark et al. 93]. A subset of the emerging architectures, including ATM, provide significantly improved latency characteristics as well [Cooper et al. 91].

Hardware, however, is only part of the picture. Without adequate software support, improved hardware capabilities remain underutilized or even inaccessible. Thus progress in hardware has been accompanied by advances in communication protocols and development of application programming interfaces (APIs) well suited to high bandwidth and low latency interconnections [Clark & Tennenhouse 90]. No less important than these components is the element which ties the software and hardware together, namely the device driver. But while protocols and APIs specifically designed for high-performance networks have been developed, the model after which device-drivers are patterned has remained largely static. The combination of these factors have increased the proclivity for network device drivers become the weak link in the performance chain [Banerji et al. 93].

This paper describes ongoing work to develop UNIX device drivers specifically designed

to support high-performance network interfaces. The work began with a performance analysis of the Token-Ring, Ethernet, and Serial Optical Link device drivers of AIX 3.2. Subsequently, the operation of the Ethernet device driver was studied in much greater depth. Currently we are developing techniques to improve the device driver's performance. Ultimately, the techniques being developed to improve the Ethernet driver will be applied to the other drivers as well.

The remainder of this paper proceeds as follows. The next section describes the operation of the IBM Ethernet High-Performance LAN Adapter device driver of AIX 3.2.5 [IBM 93c]. This driver is taken as representative of current network device driver implementations in commercial UNIX operating systems. It is used as the initial basis for our modifications. Section 3. presents the proposed techniques for improving communication device driver performance. The emphasis is on techniques which are not specific to any particular communication protocol or network or machine architecture, but rather apply to computer networking in general. In section 4., we present the results we have achieved from the modifications we have made to date. Finally, in Section 5., we summarize our contributions.

2. Ethernet Device Driver Operation

The IBM Ethernet High-Performance LAN Adapter serves as an interface between the Micro Channel expansion bus and an Ethernet local area network. The adapter includes 16 k bytes of static RAM and a 16-bit microprocessor running at 12.5 MHz. It features a 32-bit bus master interface to the Micro Channel and has an on-board direct memory access (DMA) controller. The adapter is capable of either memory-mapped, or DMA modes of operation. The driver which supports the adapter in AIX 3.2 uses DMA.

Granted, application of the term "high-performance" to an Ethernet device no matter how advanced may be objectionable to those accustomed to modern networks such as FDDI and ATM. The operation of the High-Performance Ethernet adapter as pertains to its interaction with its host processor, however, is similar to that of many adapters used with higher bandwidth networks. Also, the operation and structure of the Ethernet driver epitomizes the classic Unix communication device driver model we seek to improve.

The source code for the device driver is segregated into two distinct components: a device-independent common I/O (CIO) portion, and an adapter specific part. A modified version of the CIO portion is also used in the device driver for IBM's integrated Ethernet adapter. The code appears to have been written as a general-purpose top half capable of being used for LAN adapter device drivers in general. The code is not reused, however, in either the Token-Ring High-Performance LAN adapter, or Serial Optical Link device drivers.[1] The CIO portion of the driver provides an interface between the high-level abstractions of the Unix device driver model and the low-level device-specific constructs. As with most Unix device drivers, the code can also be separated into device management and an interrupt handler portions, referred to as the top and bottom halves respectively.

The adapter utilizes two circularly-linked lists of buffer descriptors in its static RAM, one each for its transmit and receive lists. One such list is illustrated below in the upper portion of Figure 1. Each on-card buffer descriptor consists of four fields. The first is a pointer to the next descriptor in the circularly linked list. Next is a control/status field which allows the device driver and adapter to communicate relevant information concerning the described buffer's state. The third field contains a count of the number of bytes contained in the buffer. Finally, the buffer descriptor quite logically contains a pointer to the buffer itself. This pointer resides in the Micro Channel memory address space and gets translated to an address in the system memory address space during DMA operations. During its initialization, the device driver programs the system's

1. Unfortunately, we do not currently have the source code for either the FDDI or ATM drivers.

DMA controller to set up a static mapping between the buffer addresses in the Micro Channel memory address space used by the card and the addresses of the corresponding buffers in the system memory address space used by the device driver.

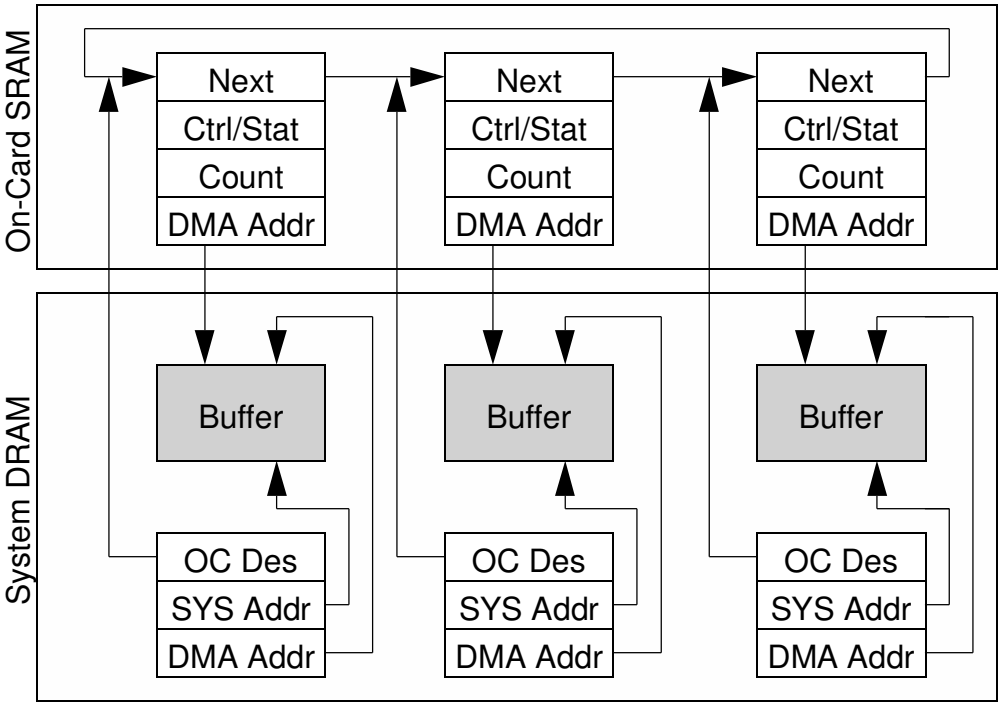

Figure 1. - On-Card and System Buffer Descriptors

Similarly, the device driver maintains two arrays of buffer descriptors in system memory, one each for transmission and reception. Each array of system buffer descriptors in system memory corresponds to a list of on-card buffer descriptors in the adapter's static RAM. The structure of these system buffer descriptors is illustrated in the lower portion of Figure 1. Each system buffer descriptor consists of three fields. The first is the offset from the beginning of the adapter card's static RAM of the corresponding on-card descriptor. A second field contains the address in the Micro Channel memory address space of the described buffer. This field contains the same value as the identical field in the corresponding descriptor on the card. The third and final field contains the address in the system memory address space of the described buffer.

To perform a transmit operation, the device driver copies the data to be transmitted into the next available transmit buffer in system memory and writes the appropriate value to the control/status field in the corresponding on-card descriptor. The card "sees" a new buffer to be sent and initiates a DMA operation to copy the data from the buffer in system memory to the adapter static RAM. Once transmission is complete or fails, the adapter informs the device driver via an interrupt. Upon reception of an incoming packet, the adapter copies the received data to the next available receive buffer in system memory via DMA, updates the control/status and count fields, and informs the driver via an interrupt.

Transmitted data is not copied directly to the dedicated low-level transmit buffers just described on its way from application code. Instead, the driver maintains a single high-level transmit queue. When a user-level client performs a write to the Ethernet device, the data to be transmitted is first copied from its initial location in the user address space to an mbuf [Leffler et al. 89] within the kernel address space. This mbuf is then appended to the single high-level transmit queue before being copied to a low-level transmit buffer. From there, the data is copied to the

adapter static RAM via a DMA operation initiated by the adapter as described above. For a client writing to the Ethernet device from within the kernel, the original copy from user to kernel space is avoided as it is assumed the data already resides in an mbuf within the kernel address space. As with user-level clients, however, data written by clients within the kernel is appended to the high-level transmit queue before being copied to low-level transmit buffers.

When an incoming packet is received, the adapter immediately copies it from the static RAM on the card to the next available low-level receive buffer in system memory via DMA as described earlier. It then issues an interrupt to inform the driver a packet has arrived. Upon receiving the interrupt, the driver allocates an mbuf and copies the received data into it. If the packet is destined for a user-level client, the driver appends the mbuf to that client's high-level receive queue and wakes the client process if it is sleeping on a read. If the packet is for a client within the kernel, the driver calls that client's receive routine directly with a pointer to the mbuf as an argument.

3. Proposed Techniques for Improving Device Driver Performance

In this section, we propose several techniques intended to improve network device driver performance. The first concerns the implementation and use of watchdog timers which are manipulated along the critical paths of some key device driver entry points. The second technique concerns the algorithms used by the driver to manage its table of *network identifiers* (netids). [IBM 93c] The concept of a netid is introduced and explained in this section. Third we address the need for mutual exclusion within device driver critical sections and describe the technique of *optimistic interrupt protection* introduced by researchers at Carnegie Mellon University. [Stodolsky et al. 93] The fourth technique is that of providing optimized fast paths designed to handle given specific instances more efficiently than code which supports the general case. Saving the most important for last, we finally discuss the use of shared memory to eliminate the cost of copying data between the kernel and user address spaces.

3.1. Watchdog Timers

In order to detect adapter failures, the driver sets a watchdog timer every time it initiates an operation which the adapter must respond to. Such operations include packet transmission. Every time the driver passes a packet to the adapter to be transmitted, it starts a watchdog timer with a duration of ten seconds. Under normal operation, the driver stops the timer when it receives an interrupt indicating the transmission is complete. If the interrupt fails to arrive within ten seconds, the timer expires generating its own interrupt and alerting the driver of the error. Such use of watchdog timers is common in nearly all device drivers. Researchers have recognized the importance of reducing the overhead associated with timer management and developed efficient implementations. [Varghese & Lauck 87]

During periods of high transmit activity, the driver may set the watchdog timer to expire in ten seconds, thousands of times a second. This would seem unnecessary. It is, of course, not important that the timer expire in exactly ten seconds. Rather, this number has been chosen somewhat arbitrarily. It would be sufficient for the driver to be assured a timer would expire within some relatively broad range, say five to twenty seconds. It is also not essential that the driver stop the timer immediately when an anticipated interrupt arrives especially in cases where most of the timer's initial duration remains. (This case can be identified using a small number of instructions.) Instead it can simply set a flag indicating the timer is inactive and ignore its expiration in the unlikely event it expires before being stopped during the processing of a subsequent anticipated interrupt.

We thus propose the following technique for reducing the number of instances in which

the driver starts and stops the watchdog timer. Instead of restarting the timer on every transmission, it checks to see if the timer is already set to expire within the acceptable range. It does so by reading the current time, subtracting the time at which the timer was started, and comparing the difference to some constant value. Similarly instead of stopping the timer each time an anticipated interrupt is received, the driver simply sets a flag indicating the timer is inactive, and checks to see if the timer has some minimum threshold duration remaining. If so, the driver refrains from stopping the timer. The cost of handling watchdog timer expirations which are ignored is expected to be offset by the reduced number of instances in which the timer is started and stopped.

3.2. Network ID Management

Before receiving frames from the Ethernet device, a client must perform a `start ioctl`. This call allows the client to specify a network identifier or netid to be associated with an opening of the device. The netid is a field in the Ethernet header, also known as the type field, which essentially identifies a network protocol. The term *service access point* or SAP is used for a field with the same purpose in other link level protocols. To give an example, IP packets are encapsulated in Ethernet frames with a netid field of 0x0800. Netware packets travelling over Ethernet have a netid value of 0x8137. When a frame arrives, the driver examines its netid field to determine which client, if any, should receive it.

Analysis of the unmodified Ethernet driver exposed one so-called optimization which may, in fact, decrease performance in some cases instead of enhancing it. When determining which client is to receive an inbound packet, the driver first checks to see if the packet is associated with the same netid as the last packet received. If so, the appropriate client is found immediately, eliminating the need to perform a linear search through a table of netids. This optimization is based on the idea that data for a given high-level protocol is often fragmented into multiple frames which results in a stream of frames for the same netid. Some netids, however, are used by protocols which generally, or always package messages within a single network packet. The netids associated with the ARP and RARP protocols on Ethernet are two examples. When the Ethernet driver receives an ARP packet, it assumes, the next frame received will contain an ARP request also which, in the common case is exactly the wrong assumption to make.

This optimization directly concerns the separation of protocol-specific knowledge from communication device drivers which is viewed by many [Peterson et al. 90] (including the authors) as a desirable property. Assuming the next packet will be destined for the same netid requires no protocol-specific knowledge. Assuming the packet immediately following an ARP request will be for a different netid rather than for the same, requires the device driver be contaminated with protocol-specific code. We are investigating the use of statistics such as the averaged number of packets received in a row for each particular netid as an improvement over the current scheme. This information along with more efficient search methods, such as a has function, may yield improvement.

We do not expect this modification to yield any significant performance improvement when subjected to practical use at least not in most environments where the vast majority of frames do, in fact, have the same netid as the frame which immediately proceeded them. In environments which do use multiple netids, however, the current scheme may tend to limit performance unnecessarily.

3.3. Optimistic Interrupt Protection

Researchers at Carnegie-Mellon University [Stodolsky et al. 93] have proposed an optimization which reduces overhead associated with assuring mutually exclusive access to critical sections of code by changing the processor interrupt level. Such interrupt level manipulation is a

common technique for guaranteeing mutual exclusion. The problem with the technique, as pointed out by the researchers, is that the process of changing the interrupt level on a RISC processor is getting relatively more expensive as RISC implementations evolve. Increasing the desire to avoid this cost is the realization that it yields value only in the relatively uncommon, but admittedly important, case where a mutual exclusion would otherwise be violated due to an interrupt A method of ensuring mutual exclusion in the more common case where critical sections are not interrupted is desirable.

The idea proposed by the researchers is to set a software flag instead of changing the interrupt level and code interrupt service routines to check the software flag. If an interrupt service routine discovers it has interrupted an operation which should not be interrupted, it sets another flag which identifies the particular interrupt, suspends its own execution and resumes at the routine which was running when the interrupt occurred now with the interrupt priority raised to prevent nested interrupts. When the so-called uninterruptible sequence completes, it checks to see if it must continue a suspended interrupt. This optimization eliminates the need to pay the cost associated with changing the interrupt level for every sequence, when they are in fact rarely interrupted.

3.4. Device Driver Fastpaths

This technique is more of an overall approach, than a specific optimization. The approach is to provide *fast paths,* device driver entry points designed to efficiently handle specific situations. Fast paths allow a device driver client to use knowledge of its behavior to allow operations to be performed more efficiently than is possible in the general case. One example of a fast path in the existing Ethernet driver [IBM 93a] is the fast write routine. The driver supports an IOCTL operation which returns a pointer to a transmit routine. This IOCTL is only supported for clients within the kernel. The transmit routine, also known as the fast write routine, is optimized for kernel clients. It contains none of the code needed only to support user-level clients such as calls which copy data from user to kernel-level space. The existing fast write fast path supports the “special case”[1] of the device driver client is within the kernel.

Further performance improvements can be gained by supporting additional special cases with their own fast paths. One such fast path we are adding is intended to support transmission of very small messages with minimal latency. The need for this path was identified while developing a distributed shared memory facility for AIX. The performance of the facility relies on the ability to send small messages containing page requests quickly. The special purpose nature of the fast path allows a number of important optimizations to be made. Due to the priority of the message to be sent, and its small size, the routine bypasses the high-level transmit queue and copies the data to be sent directly to a dedicated low-level transmit buffer from which the adapter transfers data via DMA. In effect, the ability to bypass the high-level transmit queue implements a simplistic two-level priority scheme at the frame level. The routine either waits for a transmit buffer to become available, or depending on how the driver has been configured, is assured at least one transmit buffer is always available exclusively for its use. One final optimization is that the fast path routine assumes the packet to be sent is of a specific pre-determined length which eliminates the need for the length to be calculated within the transmit routine.

Implementation of fast path routines on the receive side are complicated by the inability to know what process or netid an incoming packet is destined for before it arrives. This inability does not, however, preclude the possibility of fast path, or at least faster path, receive routines. One consideration is that although it is impossible to tell *with certainty* what device driver client the next inbound packet will be destined for, it is often possible to use the expectation that a given client will be receiving a packet soon as when awaiting a response to a request. The benefits of assuming

1. In fact, the case of the device driver client being within the kernel is, by far, the most common.

the next packet received will be for a particular client must be weighed against the cost if the assumption is incorrect. We have not investigated the potential of this optimization.

It is a common case that the vast majority of inbound packets will be destined for a single device driver client, for example the network interface layer of IP on most Unix workstations. Even in this scenario it is possible to improve the performance of receive operations by eliminating code to support the general case when it is known to be unneeded. As an example, the receive routine of the Ethernet driver in AIX includes code to support *promiscuous mode* clients which receive a copy of every packet on the network. Despite the fact this mode is rarely used, the receive routine contains a check to see if there are any promiscuous mode clients for which a copy of the received packet must be made. Similar checks are made for packets sent to multicast addresses which represent another rarely-used feature. The simple strategy is to preclude the presence of such infrequently used code on the critical path when it is known to be used. The code to handle promiscuous clients, for example, can be removed from a commonly used receive routine and relocated to a specialized receive routine which is used only when the adapter has been put into promiscuous mode.

3.5. Shared Memory

The traditional device driver model requires user-level clients to incur the undesirable penalty associated with copying data between the user and kernel address spaces. Ideally, user-level clients should be afforded the same luxury in terms of avoiding that copy as are clients within the kernel. Two pertinent considerations highlight the need to eliminate this copy. The first is the emergence of multimedia applications happy to consume network bandwidth which is a significant fraction of the bandwidth available on the memory bus. This leads directly to the somewhat alarming conclusion that user-level clients should be able to create data directly in the kennel address space. The second is the widespread adoption of alternate operating system structures such as microkernels [Acetta et al. 86]. Such structuring often entails the prohibitively expensive practice of copying communication data across multiple (that is, more than two) protection domains. Given these considerations, we have adopted a shared I/O buffer management scheme[1] that extends from user-level applications to device-drivers.

The *shared heap buffer management* utility allows execution threads in different protection domains, uniform access to shared communication data. It is realized as a kernel-level pageable heap that gets mapped at the same addresses in all application-domain processes. This mapping is chosen to be the same as that used by kernel-domain entities. Thus, a uniform mapping is maintained across both protection domains. There are two major issues in the implementation of the heap - security and buffer allocation.

Simplistic security schemes, such as private areas within the shared heap, are not effective against malicious entities [Druschel & Peterson 93]. A page re-mapping scheme that mimics a MOVE operation is required. This entails dynamically changing the protection bits on the pages involved in a cross-domain transfer. Such an operation guarantees that malicious entities cannot corrupt shared communication data after it has been passed into the kernel. The only fool-proof solution is a page remapping mechanism based on per-process page tables. Unfortunately, most such implementations require some changes to the virtual memory management component of existing operating systems. However, in order to efficiently handle application such as multimedia, similar changes will become necessary in other operating system components as well [Rangan & Vin 91].

As is obvious, such page-remapping schemes introduce some overhead. Given that most messages on the internet are smaller than 200 bytes [Kay & Pasquale 93a] (memory-memory

1. This scheme is closely based upon the fbuf facility [Druschel & Peterson 93]

copying time for a 200 byte packet is approximately 1 µsec on a 25 Mhz RS/6000), it is almost certain that a page-remap scheme takes longer than a memory-memory copy for very small messages. However, this disadvantage can be offset by the allocation mechanisms provided by the shared heap utility.

Cross-domain allocation on the shared heap is separate but co-ordinated. The separation is required to ensure that allocation itself does not require a protection domain switch, such as a system call. The co-ordination guarantees that multiple allocators can work concurrently. This is implemented by two mechanisms - per-domain allocators that each manage a part of the heap and locking mechanisms based upon fast compare and swap[1] provided by AIX [IBM 93b]. Such mechanisms speed up allocation; however, allocation costs are fixed and not dependent upon the amount of data allocated. Hence, sophisticated allocation mechanisms that can reduce some of the data-touching [Kay & Pasquale 93b] overhead of transport protocols are necessary.

Specialized shared heap allocators are being constructed to utilize application and device specific information, in order to optimize memory management. One such allocator based upon the x-kernel [Peterson et al. 90] message objects, internally maintains non-contiguous chunks of data while allowing clients to treat the data as contiguous. Such non-contiguous chunks can then be easily mapped on to specific MTU sizes of a device, thus avoiding fragmentation.

4. Experience With Techniques for Improving Device Driver Performance

In this section, we report the results from the modifications whose performance we have analyzed to date. Several of the modifications remain to be fully tested. We start by describing the testing procedure. We then report our experience with watchdog timers, optimistic interrupt protection, address space reservation, and shared memory.

4.1. Testing Procedure

We used a single metric to evaluate device driver performance: round-trip latency between user-level clients. In order to measure this value, we ran a pair of user-level processes, an *initiator* and an *echo* process, on two machines attached to the same Ethernet segment. Each process opened the Ethernet device ("`/dev/ent0`") using the UNIX `open` system call and set the network identifier it would receive packets for by issuing an `ioctl` call. The initiator process then recorded the current time and sent a packet of sixty bytes in length, including the Ethernet header, to the echo process on the remote machine. Sixty bytes was chosen because it is the minimum size allowed for an Ethernet packet. Upon receiving the packet, the echo process would respond by sending a packet of the same length back to the initiator which would record the current time immediately after receiving the reply.

Before starting the test, both processes wrote the source Ethernet addresses and destination network identifier fields of the Ethernet header at the beginning of the message buffer. The initiator process filled in the destination Ethernet address of the first packet is sent to the echo process. Subsequently, each process copied the source Ethernet address from the last packet it had received into the destination Ethernet address of the next packet to be sent. Data transmission was initiated using the `write` system call.

In order to reduce the effects of caches, virtual memory translation, and interference from other device driver clients, the round trip latency test was performed twice in immediate succession. Upon receiving the first packet back from the echo process, the initiator would immediately send a second which would also get echoed back. The round-trip latency values were computed and recorded for both round trips. The value from the second round-trip was used in our perfor-

1. Such mechanisms are available on many systems.

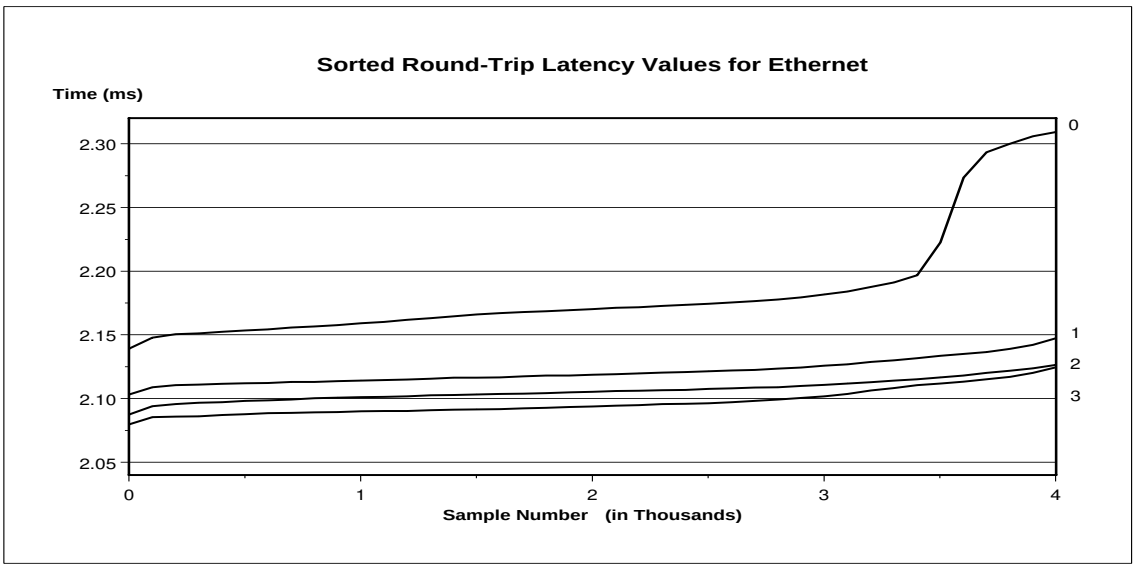

Figure 2. - Effect of Various Driver Modifications on Round-Trip Latency Values

mance analysis. We intended the first packet to "warm up the path" to be traveled by the second. Apparently it did since we found the round-trip latency value for the second packet to be approximately five to seven percent less than the one sent immediately before it on average. This difference persisted and remained relatively constant as we applied various modifications to the driver. The five to seven percent difference is particularly significant when compared to the impact of the individual modifications we were testing some of which were in the range of only one percent.

For each version of the driver which we tested, we repeated the double round-trip trial 5 k ($5*2^{10}$) times with a delay of 0.5 seconds between trials. Each test was therefore performed over a period of approximately 43 minutes. We observed that although most of the 5 k values fell within a narrow range, some small subset of them were an order of magnitude larger. We believe these drastically larger values to be due to such extraneous factors such as collisions with other network traffic and arrivals of unrelated device interrupts at inopportune times. In order to focus on the contribution to the round-trip latency made by the driver, we sorted all 5 k values and took the arithmetic mean of the lowest 4000 to produce our representative round-trip latency figure.

The tests were performed on a pair of IBM RS/6000 Model 530s (running at 25 MHz) equipped with IBM Ethernet High-Performance LAN adapters. Both machines were running AIX 3.2.5 in multi-user mode with a somewhat reduced complement of background processes running including AFS, NFS, and NIS. The initiator and echo processes were the only active user-level processes on the machines. They were run by root using the nice command (`nice -20`) to give them the highest possible user-level priority. The vital product data for the Ethernet adapters on the initiator and echo machines respectively was: part numbers 071F1182 and 00063369, engineering change numbers C73857 and C73859, read only storage levels 0028 and 0015, device driver levels 00 and 01, field replaceable unit numbers 081F7913 and 00063368, serial numbers 90002403 and 00116753, manufacturers 204491 and 204491.

Figure 2. shows a graph of the lowest 4000 sorted round-trip latency values for various versions of the driver. The top line, labeled 0, represents the original unmodified driver from AIX 3.2.5. The lowest latency value obtained with the original driver was 2.14 ms. The average of the lowest 4000 values was 2.18 ms. As can be seen on the graph, the vast majority of the latency values fall between 2.15 ms to 2.20 ms. The highest 500 or so values, however, display a sharp increase with some values in excess of 2.30 ms. The remaining lines on the graph are explained below.

4.2. Watchdog Timers

In trying to improve the performance of the driver by reducing overhead associated with timer manipulation, we assumed the overhead was sufficient to warrant reduction. That is, we proceeded on the assumption the small cost of determining whether to start or stop the timer would be less than simply doing so. That assumption proved false. When we modified the driver to determine whether or not it should manipulate the timer, its performance got slightly worse. Specifically, the average round-trip latency increased 22.3 µsec and 18.9 µsec for the average and shortest cases respectively. This represents an increase in latency of approximately one percent. The modification was subsequently dropped. The results of performance tests made with the modification are not shown in Figure 2.

4.3. Optimistic Interrupt Protection

The device driver must ensure mutually exclusive access to data structures shared between its *top* and *bottom halves*. The top half of the driver consists of entry points called by processes such as the read and write routines. The bottom half consists of service routines which are run in response to device interrupts. These include the routines which run when a packet is received or a transmission completes. The original driver ensures mutual exclusion by using the AIX `i_disable` and `i_enable` system calls to temporarily raise the processor interrupt level. These calls provide the same functionality as the `spl` call found in many other UNIX implementations. For example, when a user-level process performs a read, the driver raises the interrupt level before accessing that process' receive queue in order to prevent an incoming frame from being concurrently appended to it by a routine handling a receive interrupt.

We sought to improve driver performance by eliminating use of the `i_disable` and `i_enable` calls from the read and write paths. In order to assure mutual exclusion, we use a routine hand coded in assembly language which performs an atomic test and set operation.[1] We call this routine to set a flag which protects the set of shared data structures being manipulated. In the simple and frequent case where the flag can be set the code simply proceeds as though it had changed the interrupt level. The case where a flag can not be obtained, indicating another thread is already active within the critical section requires more consideration.

The case of the write path was relatively simple to handle. We added a mutual exclusion flag, `xmit_mutex`, which is used to relegate access to transmit data structures such as the high-level transmit queue and low-level transmit buffer descriptors. There are three routines which need to set the flag. One is the user-level write routine. If this routine is unable to obtain the flag, it simply spins trying to do so until it succeeds. This method would fail miserably in kernel implementations where system calls are not interruptible. In such cases the routine could be coded to sleep on an event instead. The second routine is the fast write routine exported only to kernel-level clients. If this routine fails to obtain the flag on its first try, it immediately returns with an error code indicating the operation should be retried. This is exactly what the routine does if it encounters other temporary error conditions such as the transmit queue being full. The third routine handles interrupts generated when a transmission completes. This routine checks to see if there are elements on the high-level transmit queue which were waiting for transmit-buffers which are available now that a transmission is complete. If this routine is unable to claim the flag it simply does not process the elements waiting on the transmit queue. This is permissible since whatever process has already claimed the flag either has or will.

The case of the receive path was somewhat more involved. We associated a mutual exclusion flag with each user-level client's receive queue. There are only two routines which to set this

1. The POWER architecture of the RS/6000 does not include any instructions which perform an atomic test and set operation. Interestingly, the POWERPC architecture does.

flag, the user-level receive routine, and the receive interrupt handler. As with the user-level write routine, if the user-level receive routine fails to obtain the flag on its first try, it simply spins attempting to do so until it eventually succeeds. If the receive interrupt handler fails to obtain the flag, however, it cannot simply spin as it would do so forever. Instead, the routine sets a flag indicating it was unable to claim the flag, saves a pointer to the received data, calls the AIX `i_mask` kernel service to mask off the interrupt associated with reception, and returns. This allows whatever user-level process possesses the flag to continue.Before that process releases the flag, it checks to see if the interrupt handler was denied access to the receive queue. If so, that process resets the flag indicating the interrupt handler was denied, processes the received data, can reenables receive interrupts by calling the AIX `i_unmask` kernel service.

Elimination of the `i_disable` and `i_enable` calls yielded encouraging results. The second line from the top, labeled 1, in Figure 2. shows the test results from a version of the driver modified to eliminate the interrupt priority manipulation calls from the write path. The lowest and average latency values obtained with this driver were 2.10 ms and 2.12 ms respectively. These represent an improvement of 36 µsec and 62 µsec or 1.7% and 2.9% in the lowest and average cases. The third line from the top, labeled 2, in Figure 2. represents the test results from a driver which was also modified to eliminate the interrupt priority manipulation calls from the read path. This driver displayed a further improvement of 16 µsec and 14 µsec in the shortest and average cases respectively bringing the total improvement to 2.4% and 3.5%.

We believe the greater improvement in eliminating the interrupt priority manipulation calls from the write path is due to those calls being on the critical path for a write, but occurring before data had arrived in the case of a read. Besides producing lower latency values, the elimination of the calls appears to have eliminated the spike in latency values encountered with the original driver. We suspect this may be due to a reduction in the length of the overall code path, but have not confirmed this.

4.4. Address Space Reservation

Before sending or receiving a frame, the driver must first obtain access to the adapter's memory and I/O ports. It does so by invoking the AIX `vm_att` kernel service which attaches the bus memory and I/O address regions to the process' or interrupt handler's address space. In order to evaluate the overhead associated with this operation, we modified the AIX kernel to reserve a single segment register which would always provide access to the necessary regions. The fourth line from the top in Figure 2., labeled 3, represents the test results of a driver modified to eliminate the calls to `vm_att` from the send and receive paths. This modification yielded a very slight improvement of 8 µsec and 9 µsec in the shortest and average cases. These values represent less than 0.5% of the round-trip times for the original driver.

4.5. Shared Memory

Comprehensive test results for the shared memory modifications were not availible at the time of publication. See section 6. for the address from which the latest version of the paper can be obtained.

5. Conclusion

Although still at a preliminary stage, this work amply points out two significant facts. Although, considerable effort has been directed at optimizing protocol realizations, device drivers need to be implemented to adequately deal with high-speed network interfaces. Furthermore, one single optimization cannot ameliorate existing device driver woes, but a synthesis of distinct implementation techniques are necessary.

We are continuing to implement the proposed optimizations, most notably, the addition of support for shared-memory. We intend to apply the techniques described here to other network device drivers and wish to repeat the tests reported here on faster machines.

6. Availability

The latest version of this paper is available in Postscript form via anonymous ftp at invaders.dcrl.nd.edu under the directory /pub/TechReports.

7. References

[Acetta et al. 86]Acetta, M. et. al. "MACH: A New kernel Foundation for Kernel Development." In *Proceedings of the Summer 1986 USENIX Conference* (Atlanta, GA. July). The USENIX Association, Berkeley, CA, 1986, pp. 93-112.

[Banerji et al. 93]Banerji A. et al, "High-Performance Distributed Shared Memory Substrate for Workstation Clusters" In *Proceedings of the Second International Symposium on High Performance Distributed Computing* (Spokane, WA. July 20-23). IEEE Computer Society Press, Los Alamitos, CA, 1993. also Technical Report 93-1, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, January 1993.

[Clark & Tennenhouse 90] Clark, D. D. and Tennenhouse, D. L. "Architectural Considerations for a New Generation of Protocols." In *Proceedings of ACM SIGCOMM*, (Philadelphia, PN. Sept.). 1990.

[Clark et al. 93] Clark, D. D. et al. "The AURORA Gigabit Testbed." *Computer Networks and ISDN Systems*, Vol. 25, No. 6, January 1993, pp. 599-621.

[Cooper et al. 91]Cooper, E. et al. "Host Interface Design for ATM LANs." In *Proceedings of the 16th Conference on Local Computer Networks* (Minneapolis, MN. Oct. 14-17). 1991, pp. 247-258.

[Druschel & Peterson 93]Druschel, P. and Peterson, L. L. "Fbufs - A High Performance Cross-Domain Data Transfer Facility." In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles* (Asheville, NC. Dec. 5-8). ACM Press, New York, NT, 1993, pp. 189-202.

[IBM 93a] *AIX Version 3 for RISC System/6000 - Kernel Extensions and Device Support Programming Concepts* (Seventh Edition), IBM Corporation, October 1993. (Part no. SC23-2207-00)

[IBM 93b] *AIX Version 3 for RISC System/6000 -Commands Reference Volume 1* (Seventh Edition), IBM Corporation, October 1993. (Part no. SC23-2233)

[IBM 93c] AIX Version 3.2 for RISC System/6000 - Technical Reference: Kernel and Subsystems Volume 5 (Seventh Edition), IBM Corporation, October 1993. (Part no. SC23-2386)

[Kay & Pasquale 93a] Kay, J. and Pasquale, J. "Measurement, Analysis and Improvement of UDP/IP Throughput for the DECstation 5000." In Proceedings of the Winter 1993 USENIX Conference, pp 249-258, January 1993.

[Kay & Pasquale 93b] Kay, J. and Pasquale, J. "The Importance of Non-Data Touching Processing Overheads in TCP/IP." ACM SIGCOMM, September 1993.

[Leffler et al. 89]Leffler, J. et al. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. New York: Addison-Wesley, 1989.

[Peterson et al. 90] Peterson, L. et. al., "The x-kernel: A Platform for Accessing Internet Resources." *IEEE Computer*, 23 (5), May 1990, pp. 23-33.

[Rangan & Vin 91] Rangan, P. V. and Vin, H. M. “Designing File Systems for Digital Video and Audio” In Proceedings of the Thirteenth ACM Symposium on Operating System Principles (Pacific Grove, CA. Oct. 13-16). ACM Press, New York, NY, 1991, pp. 81-94.

[Stodolsky et al. 93] Stodolsky et. al. “Fast Interrupt Priority Management in Operating System Kernels.” In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures* (San Diego, CA. Sept. 20, 21). The USENIX Association, Berkeley, CA, 1993, pp. 105-110.

[Varghese & Lauck 87] Vargheese. G. and Lauck, T. “Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of Timer Facility.” In *Proceedings of the Eleventh ACM Symposium on Operating System Principles* (Austin, TX. Nov. 8-11). ACM Press, New York, NY, 1987, pp. 25-38.