A High-Performance Network Architecture for a PA-RISC Workstation

David Banks and Michael Prudence

Abstract— **With current low-cost high-performance workstations, application-to-application throughput is limited more by host memory bandwidth than by the cost of protocol processing. Conventional network architectures are inefficient in their use of this memory bandwidth, because data is copied several times between the application and the network. As network speeds increase further, network architectures must be developed that reduce the demands on host memory bandwidth.**

In this paper, we discuss the design of a single-copy network architecture, where data is copied directly between the application buffer and the network interface. Protocol processing is performed by the host, and transport layer buffering is provided on the network interface. We describe a prototype implementation for the HP Apollo Series 700 workstation family that consists of an FDDI network interface and a modified 4.3BSD TCP/IP protocol stack, and we report some early results that demonstrate twice the throughput of a conventional network architecture and significantly lower latency.

I. Introduction

As new and faster computer networks are developed, we are starting to see inadequate performance with conventional network interface architectures and communication protocol implementations. More precisely, workstation users replacing 10 Mb/s Ethernets with 100 Mb/s FDDI (Fiber Distributed Data Interface) networks are not observing a tenfold increase in application-to-application throughput.

The interprocess communication facilities in many versions of Unix, including Hewlett-Packard's HP-UX, are based on those provided by the University of California Berkeley Software Distribution 4.3BSD [1]. Their socket abstraction allows networked applications to be developed independently of the underlying networks and protocols. Several different socket types exist and these provide services that include in-order, unduplicated, and reliable delivery of packets. For the reliable delivery of large amounts of data, applications predominantly use stream sockets. In the Internet communications domain, this reliable stream-based service is provided by the Transmission Control Protocol (TCP) [2].

One of the most demanding aspects of network interface and protocol stack design is the provision of high throughput all the way up the protocol stack to the application. With a conventional network interface, the socket layer and network protocols are implemented entirely in software on the host. Overheads are incurred whenever data is sent or received, because the host is involved in these higher-layer protocols. One way to reduce these overheads is to off-load them from the host, typically using an intelligent network adaptor with some front-end processing capabilities. Several front-end network adaptors [4], [5] have been proposed to relieve the host of the task of *protocol processing,* yet there is little evidence that they will be successful in making more network bandwidth available to the application. One of the drawbacks is that front-end processors tend to be slower than the host processor, thus increasing network latency. In addition, it is often difficult to partition a protocol in a clearcut way. So, the front-end processor ends up communicating with the host through a complex protocol anyhow.

In this paper, we argue that it is main memory bandwidth limitations, rather than protocol processing overheads, that cause the communication bottlenecks seen on current workstations. This is likely to continue for the foreseeable future because processor performance is increasing more rapidly than main memory speeds, resulting in proportionally less time being spent on protocol processing. To achieve a much higher application-to-application throughput, it will be necessary to adopt new network architectures that make more efficient use of the available memory bandwidth.

The rest of this paper is organized as follows. In Section II, we outline the overheads of the 4.3BSD protocol stack running over a conventional network interface, and we show how these costs will scale with improvements in workstation technology. We then, in Section III, present a network interface architecture that eliminates many of the data accesses yet is simple, cost effective, and not protocol specific. In Section IV, we describe the options for implementing such an interface for the HP Apollo Series 700 Workstation. In Sections V and VI, we go on to describe our prototype FDDI network interface, known as *Medusa,* based on this architecture and the implementation of the single-copy TCP/IP protocol stack needed to support it. In Section VII, we present some experimental results of application-to-application throughput and latency, and derive some simple models to explain these results. Finally, in Section VIII, we draw our conclusions.

II. The Cost of Interprocess Communication

Various studies [3] have been undertaken to discover the costs of running protocol stacks on the host, and to understand which operations are expensive. These studies indicate that, for bulk data transfers, the cost of operations on the data itself dominate header and protocol processing and the related operating system overheads.

Manuscript received February 15, 1992; revised August 31, 1992.

The authors are with Hewlett-Packard Laboratories, Bristol BS12 6QZ United Kingdom.

IEEE Log Number 9205970.

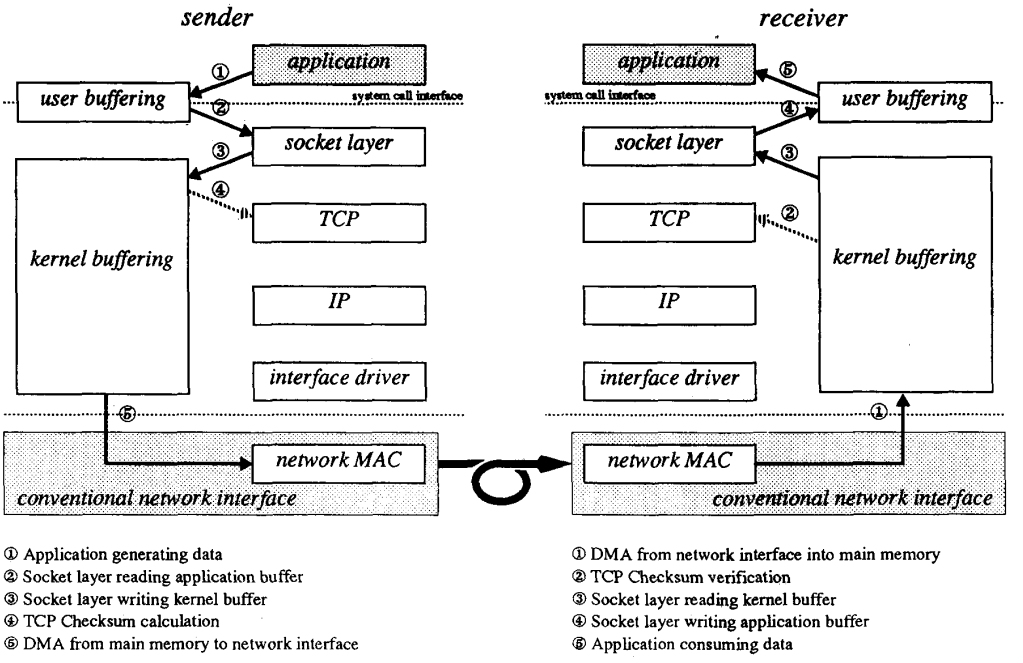


Fig. 1. Data paths in a conventional protocol stack.

The path taken by data as it passes through a conventional protocol stack is illustrated in Fig. 1. The application writes data into its buffer (called the *application buffer*) and then invokes a system call to send the data. The socket layer copies the application data into a buffer in kernel space called the *socket buffer*. Then, the transport layer reads the data in the socket buffer to compute a checksum and, finally, the data is copied out to the network interface using DMA. Thus, the memory system is accessed five times for each word of data sent. On the receive path, data is copied first from the network interface into kernel memory using DMA. The transport layer checksum is verified and, when the application is ready, the socket layer copies data from the socket buffer in kernel memory to the application buffer in user memory. Finally, the application reads the data. Thus, the memory system is also accessed five times for each word of data received.

Results published in 1989 [3] indicate that for a Sun 3/60 transmitting maximum-length Ethernet packets, about 65% of the communication time is spent on these data movements and the remaining 35% on TCP/IP protocol processing and operating system overheads. The data copy and checksum operations are memory-intensive and their cost will scale with improvements in host memory bandwidth. In contrast, the protocol processing and operating system overheads will depend more directly on the host CPU's processing power. Over the last three years, there has been an order of magnitude increase in the processing power of a typical workstation. This has not, however, been matched by as large an increase in memory bandwidth, which has only improved by a factor of three. The effect of memory bandwith lagging behind processor performance is illustrated by the following example.

We will take as a starting point a typical workstation of 1987, based on a 20MHz 68020 (essentially a 2 MIPS processor) with a memory bandwidth of 8 MByte/second. Fig. 2(a) is obtained by projecting processor performance increasing by a factor of ten every three years (115% per annum), and in Fig. 2(b) memory bandwidth is projected to increase by a factor of three every three years (44% per annum). We assume that the cost of protocol processing and related operating system overheads is fixed at 1000 instructions per packet. This is a reasonable estimate, and is consistent with [3]. We will also assume that data is accessed five times between the application and the network, as shown in Fig. 1.

We will consider the communication costs for three different networks: Ethernet, FDDI, and HIPPI with data link rates of 10, 100, and 800 Mb/s, respectively, and with maximum packet sizes of 1500, 4500, and 65,536 bytes, respectively. Fig. 2(c) shows the host processor overhead produced by sending one second's worth of data (ignoring the effect of packet headers, this is $\sim$ 1.25 MBytes for Ethernet, $\sim$ 12.5 MBytes for FDDI, and $\sim$ 100 MBytes for HIPPI). The point at which each curve intersects with the dotted line indicates when link saturation occurs. This happened in 1988 for Ethernet, in 1992 for FDDI, and can be projected to happen in 1998 for HIPPI. In Fig. 2(d), we plot the proportion of time spent accessing data. At Ethernet rates, packet header processing is still a significant part of the total (up to about 35%), but for FDDI and HIPPI with their longer packets, it is dominated by the data path costs.

As advances in processor performance continue to outpace improvements in memory bandwidth and networks support longer packets, then proportionally less and less time is going

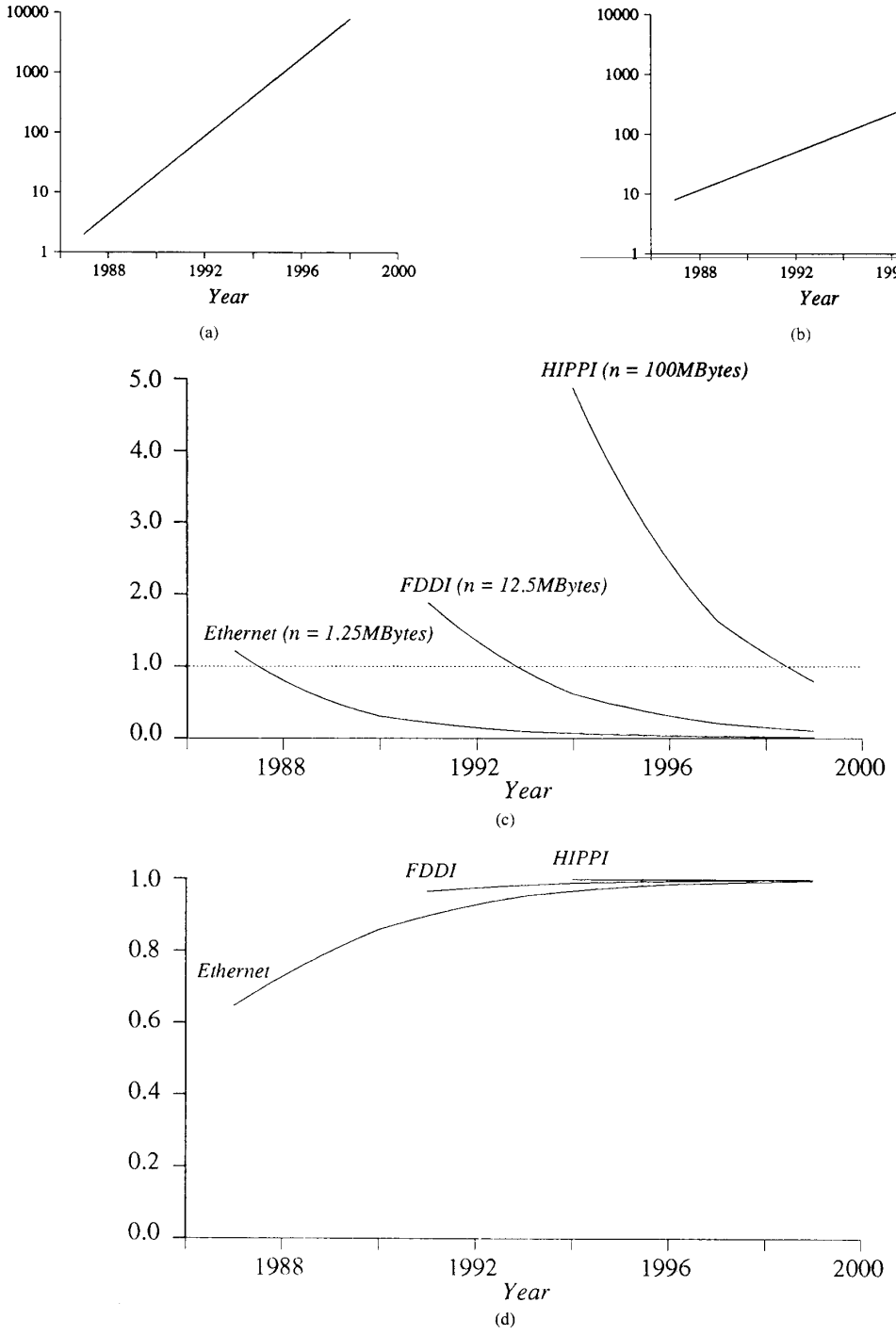


Fig. 2. Trends in workstation performance. (a) CPU performance (MIP's). (b) Main memory bandwidth (MB/s). (c) Cost to the host of sending n bytes of data (CPU seconds). (d) Proportion of time on data paths.

to be spent on protocol processing and related operating system overheads. The data path is the major component of communication costs and will remain so for the foreseeable future. The only way to substantially increase application-to-application throughput is to eliminate some of the memory accesses in the data path.

III. Architectural Changes

A conventional protocol stack generates five memory accesses for each word of data that passes between the ap-

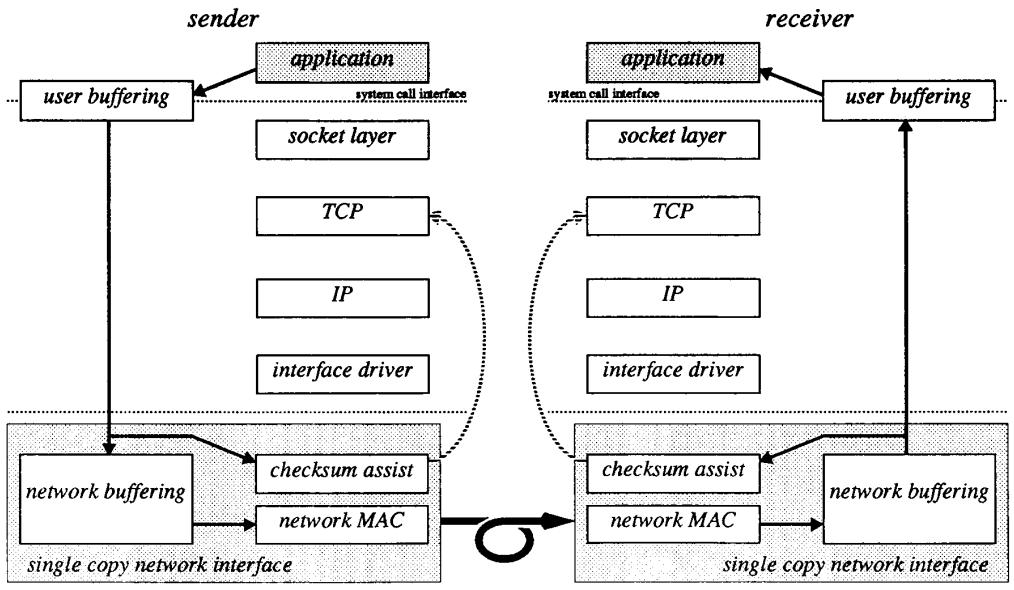


Fig. 3. A single-copy network architecture.

plication and the network. The number of memory accesses could be reduced by eliminating the socket buffers and moving data directly between the application buffer and the network interface. In this section, we discuss the purpose of the socket buffers and suggest that, if they cannot be eliminated, then they should be moved out of system memory and on to the network interface.

The function of the socket buffer is to decouple the application from the network. This is easiest to understand at the receiver. When a packet arrives from the network, there may be some delay before its data can be consumed by an application process: the application might be processing previous data, or the host may be running another process. This asynchrony between network and application is overcome by appending the new data to the appropriate socket buffer, which is then processed as is convenient.

The transmit socket buffer serves a similar purpose. By decoupling the sender from the network, delays in gaining access to the network do not necessarily slow down the application. The transmit socket buffer also has another purpose. TCP provides a reliable transfer service by the use of positive data acknowledgments. If an acknowledgment is not received within a given period, then a timer will expire and the sender must retransmit all unacknowledged data. If data were transmitted directly from the application buffer, then this buffer could not be reused until the data had been acknowledged. During this period, the application must be prevented from overwriting its buffer. This could be achieved by suspending the application or, alternately, marking the buffer as read-only so that any attempt to overwrite it would result in an access violation, at which time the trap handler could suspend the application. Both of these approaches severely limit throughput over connections with long latency, since the application will spend most of its time suspended awaiting the acknowledgment. The only alternative is for the kernel to maintain a separate copy of the data, and in the 4.3BSD implementation this is held on the transmit socket buffer in kernel memory space. Only when it has been acknowledged is the data discarded. This allows application processing and further generation of data to continue while previous data and acknowledgments are still in transit.

Since the socket buffer cannot be eliminated, one solution is to move these buffers onto the network interface, as shown in Fig. 3. The network interface contains a block of memory large enough to hold many packets. We refer to this memory as the *network buffer memory*. Data is copied directly from the application buffer to the network buffer memory, where it is held until an acknowledgment is received. When a packet is received, it is held in network buffer memory until it can be copied to the application buffer. The function of this buffer is logically identical to the socket buffer in kernel memory, but because it is physically located on the network interface we can eliminate the socket layer copy and thus reduce the number of system memory accesses. Data is copied only once, from the application buffer to network buffer memory, giving rise to the term *single-copy protocol stack*.

The other essential data operation is the computation of the transport-layer checksum. This could be combined with the data copy between the application buffer and network buffer memory. By doing so, no additional memory activity is generated, and the extra computation may not actually increase the cost of the copy if it can be done in parallel with the memory access latency. This is possible to achieve with many RISC processors but does require that the copy be performed by the processor, which may not be the most efficient means if the host provides a block move capability or the interface supports DMA. In fact, it turns out to be cheap to provide hardware support for the Internet checksum function on the

interface. The checksum can be calculated on the fly as data is moved to or from the interface, and the result can be cached for inclusion in the TCP header.

IV. Implementing a Single-Copy Network Interface

In this section, we discuss the different approaches that could be taken to implement a network interface with support for a single-copy protocol stack. The host system we will be considering is the HP Apollo Series 720/730/750 workstation family, as described in [7]–[11]. This system implements the PA-RISC 1.1 architecture, as described in [12].

There are two locations in a Series 700 workstation where a block of network buffer memory can be usefully placed: within the *main memory system* or on the *system I/O connection* [8].

A. A Single-Copy Network Interface in Memory Space

If the network buffer memory was placed within the main memory system, then data could be moved between network buffer memory and main memory by the processor using load and store instructions. This approach has the advantage that accesses to network buffer memory would be cached, thus providing very efficient access to the memory. However, because the cache is a write-back cache, data must be explicitly flushed back to network buffer memory prior to transmitting a packet. Similarly, when a packet is received, the cache must be purged so that new data is read from memory. The overhead of this software cache management will increase the cost of the data path between the network interface and the application.

There are some more practical difficulties with a memory-based interface. The memory cards in Series 700 workstations are very closely coupled to the VLSI memory controller, allowing timing margins to be much tighter than if a memory bus existed. This results in a higher performance memory system, but complicates the design of a network interface in memory space. In addition, the memory system uses an error correction scheme. Therefore, network interface would have to generate correct check bits for incoming data, which would further complicate the design.

B. A Single-Copy Network Interface in I/O Space

The I/O system of the Series 700 workstation family has been optimized to improve graphics performance. For example, stores to I/O space were designed to incur only a single-cycle pipeline penalty, resulting in a very high I/O store bandwidth. In addition, the VLSI memory controller supports a block move function, allowing data to be moved between memory and I/O space without the overhead of having to pass through the processor. Although these features were designed with graphics in mind, they are of benefit to any device in I/O space. If the network buffer memory was placed in I/O space, then three mechanisms exist for accessing data: programmed I/O, direct memory access (DMA), and the memory controller block move instructions.

With programmed I/O, the processor may read or write from I/O space using single-word load and store instructions, just as it would to memory space. The only difference is that accesses to I/O space are uncached, so there will be a significant latency involved with loads. With direct memory access (DMA), an I/O device can read or write main memory directly without involving the processor. Cache coherence during DMA is maintained by the operating system, with the aid of purge cache and flush cache instructions. Finally, an I/O device could use memory controller block moves to move data between memory and I/O space. This block move operates on 32-byte (one cache line) blocks, and movement in either direction is supported.

C. Analysis and Conclusions

By considering the data path between the application process and the network, the authors have been able to quantitatively compare the different options outlined previously. For the sender, this data path starts with writes to the application buffer in main memory. Data is then copied from the application buffer to the network buffer memory. At the receiver, the reverse occurs, finishing with reads from the application buffer. The analysis consisted of tabulating the operations needed to send and receive 32 bytes (one cache line) of data, and counting the instruction cycles consumed.

The results of this analysis have shown that the most efficient scheme for outbound transfers is an I/O-based interface, with data being moved to the interface using programmed I/O. This is because all of the other schemes involve flushing either the application buffer or the network buffer from cache, which will cause a cache miss when the buffer is reused. With programmed I/O, there is a good chance that the application buffer will remain cache resident and, as a result, data never has to be written back to memory.

For inbound transfers, the analysis has shown that the I/O-based interface performs poorly if data is read out one word at a time using programmed I/O. However, if data is moved from the interface in larger blocks by using the memory controller's block move hardware, then a much higher throughput can be expected. In fact, this mechanism is so effective that it is difficult to justify the interface supporting DMA.

On the basis of this analysis, we decided that the best all-around solution was a simple I/O-based interface where the network buffer memory is accessed using either programmed I/O or memory controller block moves.

V. The Medusa FDDI Interface

A. Introduction

The Medusa FDDI interface is a research prototype that was designed for the HP Apollo 9000 Series 700 workstations at Hewlett-Packard Labs in Bristol. It is loosely based on the WITLESS (Workstation Interface That's Low-cost, Efficient, Scalable and Stupid) architecture proposed in [6]. It contains the network buffer memory that is required to support a single-copy protocol stack, and appears to the host as a block of memory in I/O space. All network, transport, and socket layer processing is performed by the host and, thus, the architecture can support many different protocols.

The main reason for developing the Medusa FDDI interface was to demonstrate that application-to-application communi-

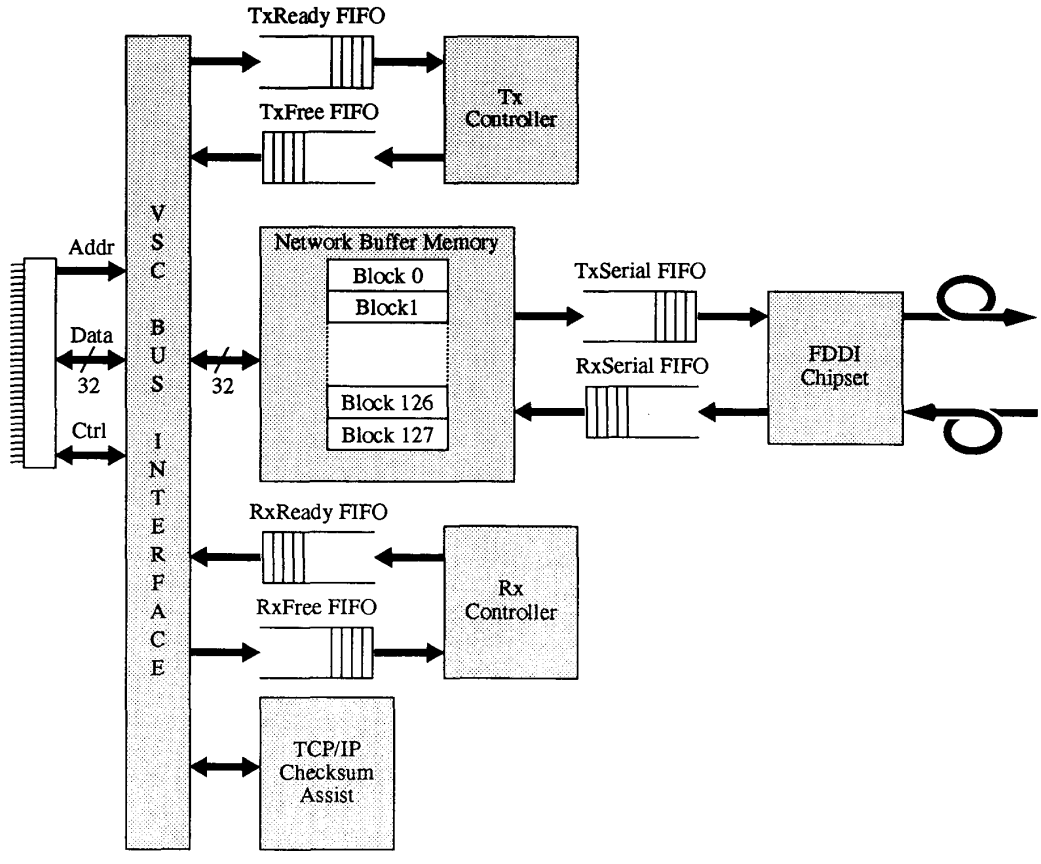


Fig. 4. The Medusa FDDI interface.

cation can be achieved at FDDI rates with a low-cost interface using standard protocols such as TCP.

B. Design Overview

The Medusa FDDI interface (see Fig. 4) attaches to the *I/O System Connect* of a Series 700 workstation and is based on a large block of multiported memory with a single wait-state access. This is implemented efficiently and cheaply using triple-ported video RAM's. The video RAM's contain a parallel port as well as two serial ports. The parallel port is interfaced to the host, allowing random access to the network buffer memory using fast page mode read and write cycles, whereas the two serial ports provide independent transmit and receive paths to the FDDI chipset. The memory system is organized as 256K 32-bit words and is constructed from eight 256K×4 devices.

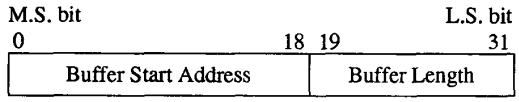
A video RAM is capable of sustaining high data rates from its serial port with very little effect on the parallel port. In our memory system, a single transfer cycle can move 2 KBytes of data from the VRAM memory into the VRAM serial register, or vice versa. There is an additional overhead for generating refresh cycles, but this is small. Hence, even when transmitting data at FDDI link rates, most of the parallel port bandwidth is available to the host.

The network buffer memory is organized as a number of fixed-size (8 KByte) blocks, each capable of holding a maximum-length packet. By using fixed-size blocks, we eliminate the problem of external memory fragmentation. It also means that the starting address and length of a block can be encoded into a single word, allowing transactions on blocks to be atomic. Control of the interface is achieved with four small (256 word) FIFO's: TxReady, TxFree, RxReady, and RxFree FIFO's. The format of the data in these control FIFO's is shown in Fig. 5.

The only protocol-specific part of the design is hardware support for the transport-layer checksum. It would be possible to support several checksum functions; however, the current Medusa FDDI interface supports only the Internet checksum function, as used by TCP [2]. A checksum function not supported by the hardware can always be performed in software by the host, but with a reduction in throughput. The TCP and XTP [13] checksum functions are straightforward to implement in hardware but the ISO TP4 [14] checksum function is not, because it is byte-oriented and involves modulo 255 arithmetic.

C. Transmit Operation

A packet is constructed in a VRAM block by the host, and



To send a packet:

```
Medusa->TxReady = buffer_start | buffer_length
```

To receive a packet:

```
buffer = Medusa->RxReady

buffer_start = buffer & 0xFFFFE000

buffer_length = buffer & 0x00001FFF
```

N.B. PA-RISC 1.1 bit numbering is big endian

Fig. 5. Format of the control FIFO's.

queued for transmission by writing the block's start address together with its length into the TxReady FIFO. This is done with a single atomic write, so there is no need for a semaphore to control access to these FIFO's. Outstanding transmission requests are handled in the order in which they were issued. Each transmission request is processed by the transmit control logic, which streams the packet from the VRAM's transmit port into the FDDI MAC for transmission on the network. When the packet has been transferred to the MAC, the transmit control logic writes the block address into the TxFree FIFO, to be subsequently read by the host.

D. Receive Operation

At the receiver, the host will have written the addresses of some free blocks into the RxFree FIFO. When a packet arrives, the receive control logic will read the first of these addresses, and stream the data from the FDDI MAC into the VRAM's receive port. When the end of the packet is reached, the start address of the block together with the packet length will be written into the RxReady FIFO by the receive control logic. Again, because of the coding of the address and length into a single word, this transaction is atomic. An interrupt will be generated only when the RxReady FIFO goes from empty to nonempty. This reduces the overhead when a packet arrives and the host is still processing the previous one.

VI. Implementation of a Single-Copy Protocol Stack

To fully realize the potential of the Medusa FDDI interface, it is necessary to modify the TCP/IP protocol stack to reduce the number of data copies to one: from the application buffer to network buffer memory. This gives us the expected increase in performance, and also highlights a number of issues for implementing high-speed network protocols in a conventional UNIX environment.

Management of the buffers in network buffer memory is a key area in the single-copy stack implementation. The traditional mechanism of *mbufs* [1] can be used, simply by pointing the offset field of an mbuf cluster to network buffer memory. However, it is advantageous to treat a packet as a set of two different types of mbuf: those that hold data in network buffer memory, called *Medusa mbufs,* and normal mbufs that hold protocol headers in system memory. The latter are accessed a number of times by the network protocols, and will reside in cache for the duration of protocol processing. Cache accesses are far quicker than I/O accesses, so having the header information cache resident speeds up the protocol processing.

The device driver supports packets containing data in normal mbufs, as well as packets containing data that are already on the interface and pointed to by a Medusa mbuf. This allows protocols other than TCP/IP to be run over the Medusa network interface, without having to rewrite those protocols for the single-copy architecture.

A. Data Transmission

A user process presents data to the socket layer by means of a *send()* system call. This causes the socket layer procedure *sosend()* to be called. In a conventional stack, *sosend()* would check the amount of space available in the send socket buffer, and copy as much data as possible from the application buffer into normal mbufs. These are appended to the socket buffer, and the protocol output routine is called.

In a single-copy stack, the socket layer looks at the routing information maintained by IP to find the network interface that the data is destined for. If this interface has network buffer memory for supporting a single-copy protocol stack, the socket layer copies data directly to this memory. Space is left at the front of the network buffer for the protocol headers, and data is copied in blocks not larger than the data-link layer maximum transmission unit. A Medusa mbuf pointing to this data is appended to the send socket buffer, and the protocol output routine is called as usual.

By allowing the socket layer to perform the copy operation, we are effectively packetizing the data before passing it to TCP. This is a change from the conventional stack, where TCP takes data from the socket buffer byte stream and packetizes it.

During the data copy operation, a checksum is computed on the fly. This can either be done using a software copy-and-checksum routine, or using the hardware checksum unit on the Medusa interface itself. This checksum is stored with the data in network buffer memory for later use. TCP and IP work as usual, with TCP using the checksum calculated by the socket layer instead of calling *in_cksum()* directly. The data, with protocol headers prepended in normal mbufs, is passed to the device driver.

The device driver is very simple. If the packet data is in network buffer memory (pointed to by a Medusa mbuf), then the device driver copies the headers into the space reserved at the front of the buffer and causes the packet to be transmitted.

If the packet presented is not in network buffer memory, a buffer is allocated by the driver and data is copied from kernel memory to network buffer memory. The packet is then transmitted as normal.

B. Data Reception

The arrival of a packet in network buffer memory causes the device driver interrupt routine to be invoked. A block of data from the front of the packet is copied into a normal mbuf, on the assumption that it contains a valid TCP/IP header. The driver then performs a partial parse of the packet header, looking for a valid TCP/IP packet with data in it. If this is successful, the data is left in the network buffer memory with a Medusa mbuf pointing to it. Packets received that are not TCP/IP, or that do not contain any data such as TCP acknowledgments, are copied to normal mbufs and passed up the protocol stack to the IP layer.

IP passes the packet to TCP as usual. Normally, TCP checksums the data to make sure it is correct and then appends this data to the socket buffer for the connection. This cannot be done in the single-copy architecture because the checksum is produced as a result of the data copy from network buffer memory to the user buffer. Until the socket layer is invoked, with the address of a user buffer to place the data in, we cannot do the data copy and, hence, cannot obtain the checksum.

Consequently, acknowledgment of the data to the peer TCP must be delayed until the socket layer has copied the data and verified the checksum. This means that the socket layer becomes responsible for acknowledging data for TCP, rather than TCP itself. We need to extend the notion of a TCP window to cover data that has been received but not yet acknowledged. This covers data that is *in transit* in the system, allowing the TCP layer to properly reject any duplicate data received.

VII. Experimental Results

In this section, we present the results of some performance measurements of the single-copy protocol stack (as described in Section VI) running over the Medusa FDDI interface. For comparison purposes, we have made similar measurements of a conventional protocol stack running over both Medusa and the built-in Ethernet interface.

In Section VII-A, we discuss what sort of performance measurements are representative of real applications, and explain why maximum socket-to-socket throughput tests can result in pessimistic estimates of communication cost. In Section VII-B, we describe a simple profiling technique used to determine the amount of time spent in particular sections of code. Using this, we have measured the amount of time spent in the driver, IP, TCP, and socket layers. In Section VII-C, we examine TCP stream performance. We present a simple model of socket-to-socket throughput based on the results of the profiling, and verify this with experimental results. We show that FDDI link saturation is achievable. Finally, in Section VII-D, we examine TCP request/response transaction rates, again using a simple model of socket-to-socket latency.

A. Network Performance Measurements

Usually, network performance measurements are taken to determine the maximum socket-to-socket throughput. With low-cost workstations and very high-speed networks such as FDDI, this can only be achieved if the application performs minimal processing on the data. This is not representative of real applications, which take time to generate and consume data, and will also lead to higher protocol overheads. If the receiver is doing nothing other than executing *recv()* calls, then there will be little chance of packets building up in the receive socket buffer. Consequently, most *recv()* calls will return only a single packet of data rather than several. This is inefficient, and will result in acknowledgments being sent more frequently (probably one every other packet rather than one per socket buffer). If the throughput bottleneck is the network, then this may not affect maximum throughput figures but will lead to a pessimistic view of communication overhead, in terms of processor time per kilobyte of data transferred. We have tried to make our performance measurements more realistic by including a delay between *recv()* calls to simulate the application taking time to consume data.

B. Detailed Profiling of a Network Protocol Stack

To be able to understand and explain socket-to-socket performance, it is useful to track the progress of a packet through the protocol stack, noting the time at which particular sections of code are executed. This *profiling* can be done with the aid of a powerful logic analyzer, set up to record particular instructions being fetched. Alternately, there are software-based schemes where additional codes must be added to the protocol stack. The hardware scheme can only be used if instruction fetches are visible, which is unlikely to be the case if the processor has an on-chip instruction cache. The obvious advantage, though, is that no modifications are required to the protocol stack. Software-based schemes usually involve calling *gettimeofday()*, which returns a time in microseconds. This will impact the performance, and is not really accurate enough for our needs.

The scheme we have used for profiling is software-based, yet is extremely accurate and involves very little overhead. PA-RISC processors have a hardware interval timer register. On the model 720, this register is incremented every processor clock cycle (20 ns) and wraps around every 85.9 seconds. Both the application and kernel can read this register because it is a nonpriviledged operation. Our profiling scheme involves a single function, *med_trace(n)*, that simply reads the interval timer register and writes its value, together with n, into a global data array. Trace points are recorded by calling *med_trace()* at various points in the protocol stack, using a different trace point number, n, each time. The overhead of *med_trace()* is very low, taking only 15 instructions including the function call and return. Since each call to *med_trace()* writes two words of data into the data array and a cache line contains eight words, then a cache miss is likely to occur every fourth call. Taking this into account, the average time spent in a call to *med_trace()* will be 0.41 μs, and overall we estimate

	per packet (us)	*per sys. call (us)*	*total (us)*	*percentage*
scale with processor speed:				
Socket	5.01	40.42	80.50	4.1
TCP	24.05	89.36	281.76	14.4
IP	7.23	16.25	74.09	3.8
Driver	66.33	76.03	606.67	30.9
scale with memory speed:				
Data copy	114.67		917.36	46.8
totals:	217.29	222.06	1960.38	100.0

Fig. 6. TCP stream test—send path summary (assuming 32 KByte socket buffer).

	per packet (us)	*per sys. call (us)*	*total (us)*	*percentage*
scale with processor speed:				
Socket	9.80	40.97	119.37	5.4
TCP	13.90	23.76	134.96	6.1
IP	13.34	7.62	114.34	5.2
Driver	51.06	63.82	472.30	21.5
scale with memory speed:				
Data copy	169.93		1359.44	61.8
totals:	258.03	136.17	2200.41	100.0

Fig. 7. TCP stream test—receive path summary (assuming 32 KByte socket buffer).

that tracing will result in a 2.2% increase in communication overhead.

To make the measurements, a simple application was written to transfer data over a stream socket connection. The send and receive socket buffers were both set to 32 KBytes, and data was transferred in one direction using *send()* and *recv()* system calls. The application buffers and send size were also set to 32 KBytes, and each packet transmitted contained 4 KBytes of data. The amount of time spent processing data in between *recv()* calls was sufficient to ensure that the receive socket buffer had filled up completely before the next *recv()* call was made.

Because the first *send()* call will fill up the send socket buffer, the sender will be put to sleep at the start of the next *send()* call. When an acknowledgment is received, the sender will be awakened and more data may be sent. We have not included the sleep time in our results.

In Fig. 6, we summarize our measurements of the send path, resulting from four successive calls to *send()*. We have grouped the overheads according to whether they are incurred just once per *send()* system call or for every packet sent. Similar measurements of the receive path are given in Fig. 7.

In both the send and receive cases, the driver contributes a significant proportion to the total cost. There are known inefficiencies with our driver implementation, and with some optimizations we should be able to greatly reduce this cost. However, the driver is only perceived as expensive when compared to the IP, TCP, and socket layers. What this really illustrates is that the protocol processing itself is not expensive. The cost of even a single data copy dominates everything else. As we noted earlier, processor speeds are increasing faster than memory speeds so we should expect protocol processing costs to continue to drop.

In the following two sections, we present empirical models for socket-to-socket throughput and socket-to-socket latency and verify the models with experimental measurements. Our measurement system consisted of a pair of HP9000 Series 720 workstations, running a version of the HP-UX 8.05 operating system that was modified to include the Medusa device driver and the single-copy TCP/IP protocol stack. To ensure our measurements were repeatable, we conducted all tests on a private network, with only the two workstations involved in the tests attached. For Ethernet, this consisted of a short length of coaxial cable interconnecting the two nodes. For FDDI, two single attach nodes were connected directly together, without the use of a concentrator, to form a two-node ring.

C. TCP Stream Performance

Where the socket-to-socket throughput is limited by the rate at which the receiving application can consume data, then

$$\text{throughput} = \frac{p}{\frac{a}{n} + b + x}$$

p = packet size (KBytes)

n = number of packets received per system call

a = per system call overhead (seconds)

b = per packet overhead (seconds)

x = application per packet processing time (seconds).

Taking the values of $a = 136.17\,\mu$s and $b = 258.03\mu$s from Fig. 7, $p = 4$ KBytes, and $n = 8$ packets per socket buffer, we have varied x from 1 μs to 10 ms, giving the graph shown in Fig. 8. The other curves shown on this graph are our experimental measurements of socket-to-socket throughput. Of primary importance is the curve for the single-copy stack running over Medusa. For comparison purposes, we have also shown the performance of a conventional protocol stack running over Medusa and of the built-in Ethernet interface.

The theoretical and experimental curves are coincident to the point where the application processing time drops below about 320 μs/4 KByte packet. This is the point at which the receive socket buffer has just been refilled when the next *recv()* call is made. If the processing time is reduced further, then the amount of data returned by each *recv()* decreases. Our model has assumed n is fixed, hence the difference in results. The other effect illustrated by the graph is that of network saturation at 11 700 KBytes/s. The transmission of protocol headers and acknowledgment packets account for this being less than the FDDI link rate of 100 Mb/s.

A different way of presenting this data is to plot CPU utilization along the horizontal axis, where

$$\text{utilization} = 1 - \text{throughput} \times \left(\frac{x}{p}\right).$$

This has been done in Fig. 9, which shows a direct correlation between data rate and processor utilization, until the point of network saturation. Although the curves extend beyond this point, this does not mean that the protocol overheads continue to increase. Most of the time the receiver will be sleeping, waiting for the next packet to arrive, and other processes could be run.

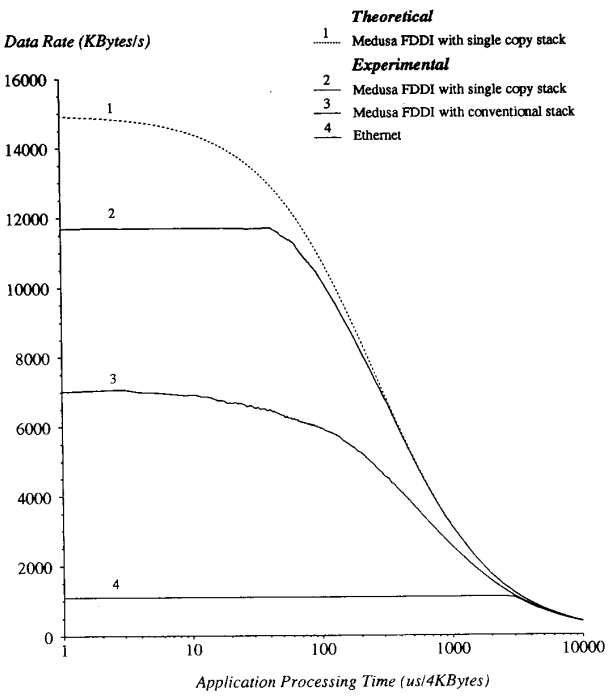


Fig. 8. Graph of data rate versus application processing time.

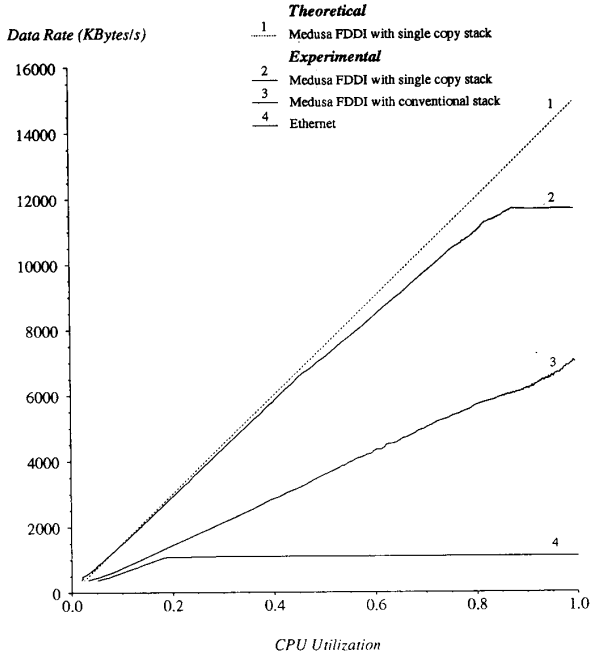


Fig. 9. Graph of data rate versus CPU utilization.

D. TCP Request/Response Performance

Often, it is not socket-to-socket bandwidth that is important to an application, but socket-to-socket latency. This is the case with many distributed applications, where most of the communication takes the form of short request/response messages. Since the amount of data being transferred is small, data path performance is not critical, and we would not expect to see a great deal of difference between a single-copy architecture and a conventional architecture. However, the extremely simple design of the Medusa interface will introduce far less latency than an intelligent interface adaptor, which may have several processors operating on a pipelined data path.

	scale with memory speed (us)	*scale with processor speed (us)*	*scale with network speed (us)*	*percentage*
fixed overheads:				
Send socket layer		28.29		3.1
TCP output		28.44		3.1
IP output		7.72		0.8
Driver output		42.15		4.6
Network Latency			15.60	1.7
Driver Input		48.93		5.4
IP input		15.05		1.7
TCP input		52.00		5.7
Receive socket layer		56.91		6.3
scale with message size:				
Send data copy	115.08			12.6
Transmission of data			327.68	36.0
Receive data copy	172.12			19.0
totals:	287.20	279.49	343.28	100.0

Fig. 10. TCP request/response critical path (4 KBytes messages).

Using the same profiling techniques as before, we have identified all of the components of this latency. The applications in this case simply exchange messages, using *send()* and *recv()* calls. Data packets flow in both directions and so no explicit acknowledgments need to be sent. The components of the send–network–receive critical path are identified in Fig. 10, where they have been categorized according to whether they scale with memory, network, or processor speed.

With the 4 KByte messages used, the data copy and network transmission times are significant. However, for small messages these will be dominated by the path through the protocol stack. The socket-to-socket latency can be modeled as

$$\text{latency} = \left(\frac{d}{m_1}\right) + s + n_1 + \left(\frac{d+h}{n_2}\right) + r + \left(\frac{d}{m_2}\right)$$

d = data size
h = header size
s = send path through protocol stack
r = receive path through protocol stack
m_1 = main memory to network interface bandwidth
m_2 = network interface to main memory bandwidth
n_1 = network latency
n_2 = network bandwidth.

Using values for FDDI and from Fig. 10,

$h = 70$ Bytes
$s = 106.6\,\mu$s
$r = 172.89\,\mu$s
$m_1 = 4096$ Bytes per $115.08\,\mu$s

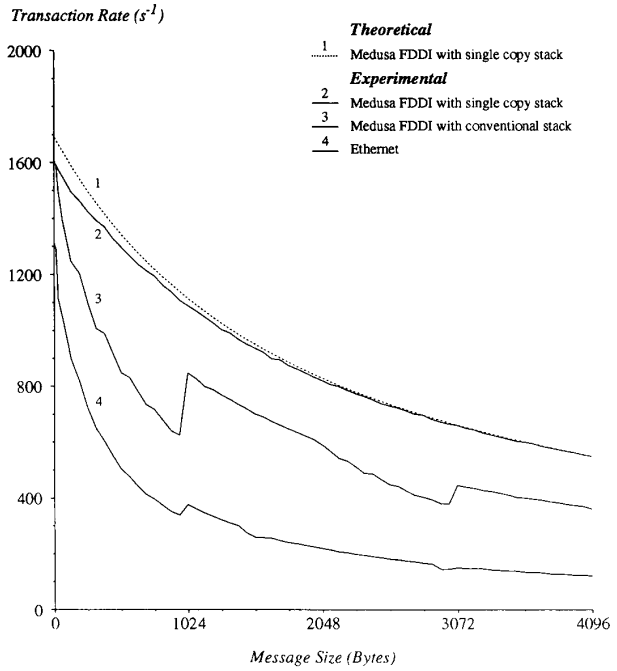


Fig. 11. Graph of TCP request/response transaction rate versus message size.

$$m_2 = 4096 \text{ Bytes per } 172.12\,\mu\text{s}$$
$$n_1 = 10\,\mu\text{s (estimated)}$$
$$n_2 = 12.500 \times 10^6 \text{ Bytes/s}.$$

This expression reduces to

$$\text{latency} = 295.09 + 0.1508d \quad (\mu\text{s}).$$

Our benchmark specifies a single transaction as a request and response; hence, the transaction rate involves twice the socket-to-socket latency. In Fig. 11, we have plotted the theoretical transaction rate verses message size, together with some experimental results. With 4 KByte messages, the model is very accurate, but with 4 byte messages some difference is observed, which corresponds to an extra 15 μs socket-to-socket latency. We suggest two reasons for this discrepancy. First, our estimate of FDDI network latency, which involves waiting to capture the token, may be slightly low. Second, we have assumed that the cost of the *uiomove()* call, which moves data between the application buffer and network buffer memory, scales proportionally with data length. This will not be the case when copying small amounts of data, since the fixed costs of checking for boundary conditions and loop initialization will dominate the cost of the data copy.

The other curves in Fig. 11 confirm that for small message sizes, there is little benefit from a single-copy architecture but, as the message size increases, then so does the performance differential. With the conventional stack, an increase in transaction rate as the message size passes 1024 bytes is caused by the socket layer changing its buffer management policy. Small messages are held on the socket buffer as a chain of normal mbufs (each one holding 96 bytes of data). When the message size is greater than 1024 bytes, then a single cluster mbuf is used. Operations on cluster mbufs are more efficient since data is contiguous, and copies of the mbuf may be taken by reference rather than by duplicating the data. This anomaly is not seen with a single-copy stack, because there is only one buffer management policy.

VIII. Conclusions

The design, implementation, and performance analysis of the Medusa network interface and the associated single-copy protocol stack has demonstrated that it is possible to provide very high network throughput between application processes running on low-cost workstations. In particular, it is the combination of an interface with network buffer memory and a single-copy protocol stack that provides twice the throughput of a conventional architecture. This type of network interface can be engineered to low cost, with the expense of providing memory on the interface being offset by the lack of complex processors and DMA engines.

We have devised a simple and effective scheme for profiling the protocol stack that makes use of the PA-RISC interval timer register. With this scheme, detailed measurements of the time taken to perform various protocol and operating system tasks were taken. We have used these measurements to derive empirical models of socket-to-socket throughput and socket-to-socket latency for our single-copy stack running over the Medusa FDDI interface. These models can be used to determine the impact of faster processors, memory systems, and physical networks on network performance to the application. The conclusion we must draw here is that the cost of the IP, TCP, and socket layers is now small when compared to the cost of even a single data copy and will continue to drop, because workstation performance is increasing at a ferocious pace. Off-loading these layers from the host, by making the network interface smart enough to perform protocol processing functions, cannot really be justified on performance grounds and would probably increase the network latency as seen by an application.

References

[1] S. Leffler, M. McKusick, M. Karels, and J. Quaterman, *The Design and Implementation of the 4.3BSD UNIX Operating System.* Reading, MA: Addison-Wesley, 1989.
[2] J. Postel, "Transmission control protocol," RFC 793, SRI Netw. Inform. Cent., Menlo Park, CA, Sept. 1981.
[3] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Commun. Mag.*, June 1989.
[4] Excelan Inc., *TCP/IP Protocol Package for VAX/VMS Systems Reference Manual*, pub. 4 200 012–00, Rev. A, July 31, 1985.
[5] A. G. Fraser, "Toward a universal data transport system," *IEEE J. Select. Areas Commun.*, vol. SAC-1, no. 5, Nov. 1983.
[6] V. Jacobson, "Efficient protocol implementation," in *Proc. ACM SIGCOMM '90*, Sept. 24, 1990.
[7] M. Forsyth, S. Mangelsdorf, E. DeLano, C. Gleason, and J. Yetter, "CMOS PA-RISC processor for a new family of workstations," in *Proc. IEEE COMPCON*, spring 1991.
[8] R. Horning, L. Johnson, L. Thayer, D. Li, V. Meier, C. Dowell, and D. Roberts, "System design for a low cost PA-RISC desktop workstation," in *Proc. IEEE COMPCON*, spring 1991.
[9] D. Odnert, R. Hanson, M. Dadoo, and M. Laventhal, "Architecture and compiler enhancements for PA-RISC workstations," in *Proc. IEEE COMPCON*, spring 1991.

[10] A. J. DeBaets and K. M. Wheeler, "Midrange PA-RISC workstations with price/performance leadership," *Hewlett-Packard J.*, Aug. 1992.
[11] C. A. Gleason, L. Johnson, S. T. Mangelsdorf, T. O. Meyer, and M. A. Forsyth, "VLSI circuits for low-end and mid-range PA-RISC computers," *Hewlett-Packard J.*, Aug. 1992.
[12] Hewlett-Packard Co., *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Man. Part 09740-90039.
[13] Protocol Engines, Inc., *XTP Protocol Definition Revision 3.4*, 1989.
[14] "Transport protocol specification for open systems interconnection for CCITT applications," CCITT Recom. X.224, Sect. 6.17 and Appendix I.

David Banks was born in Leigh, England, on Jan. 21, 1967. He received the B.Sc. (Hons) degree in computer engineering from the University of Manchester, Manchester, England, in 1988.

Since Sept. 1988, he has been a Member of Technical Staff in the Network Technology Department of Hewlett-Packard Laboratories in Bristol, England. He has been involved in research into several areas of high-speed communication and electronics, including gigabit networking and high-speed network interfaces. He is currently continuing this work, in the context of the architecture of next-generation computer systems.

Michael Prudence received the B.Sc. (Hons) degree in computer engineering from the University of Manchester, Manchester, England, in 1988.

Since 1988, he has been a Member of Technical Staff in the Network Technology Department of Hewlett-Packard Laboratories in Bristol, England. He has worked in a number of different areas, including fast packet switching, network protocol implementations in UNIX, and high-speed network interfaces. Currently, he is working in the field of asynchronous transfer mode technology for the local area.