Afterburner: Architectural Support for High-Performance Protocols

Chris Dalton, Greg Watson, Dave Banks,
Costas Calamvokis, Aled Edwards, John Lumley
Networks & Communications Laboratories
HP Laboratories Bristol
HPL-93-46
July, 1993

network interfaces, TCP/IP, Gb/s networks, network protocols

Current workstations are often unable to make link-level bandwidth available to user applications. We argue that this poor performance is caused by unnecessary copying of data by the various network protocols. We describe three techniques that can reduce the number of copies performed, and we explore one - the single copy technique - in further detail.

We present a novel network-independent card, called Afterburner, that can support a single-copy stack at rates up to 1 Gbit/s. We describe the modifications that were made to the current implementations of protocols in order to achieve a single copy between application buffers and the network card. Finally, we give the measured performance obtained by applications using TCP/IP and the Afterburner card for large data transfers.

Afterburner: Architectural Support for High-performance Protocols

Many researchers have observed that while the link level rates of some networks are now in the Gbit/s range, the effective throughput between remote applications is usually an order of magnitude less. A number of components within computing systems have been postulated as the cause of this imbalance. Several years ago the transport and network protocols came under great scrutiny as they were considered to be 'heavyweight' and thus computationally expensive. This line of thought encouraged many researchers to explore ways to execute protocols in parallel, or to design new 'lightweight' protocols. Other sources of problems were thought to be poor protocol implementations, high overheads associated with operating system functions, and a generally poor interface between applications and the network services.

Clark *et al.* [2] suggested that even heavyweight protocols, such as the widely used TCP/IP protocol combination, could be extremely efficient if implemented sensibly. More recently, Jacobson has shown that most TCP/IP packets can be processed by fewer than 100 instructions [4]. It is now widely believed that while a poor implementation will impede performance, protocols such as TCP are not inherent limiting factors.

One reason many implementations fail to achieve high throughput is that they access user data several times between the instant the data are generated and the instant the data are transmitted on the network. In the rest of this paper we analyse this behaviour in a widely-used implementation of TCP, and consider three proposals for improving its performance. We describe our experimental implementation of one of these proposals, which uses novel hardware together with a revised implementation of the protocol. To conclude, we present measurements of the system's performance.

The bottleneck: copying data

We believe that the speed of protocol implementations in current workstations limited not by their calculation rate, but by how quickly they can move data. This section first reviews the design of a popular protocol implementation, then examines its behaviour with reference to workstation performance.

The conventional implementation

Our example is the HP-UX implementation of TCP/IP, which, like several others, is derived from the 4.3BSD system [7]. This overview focuses on how it treats data, and is rather brief.

Figure 1 shows the main stages through which the implementation moves data. On the left are listed the functions which move data being transmitted; on the right are those for received data. Curved arrows represent copies from one buffer to another; straight arrows show other significant reads and writes.

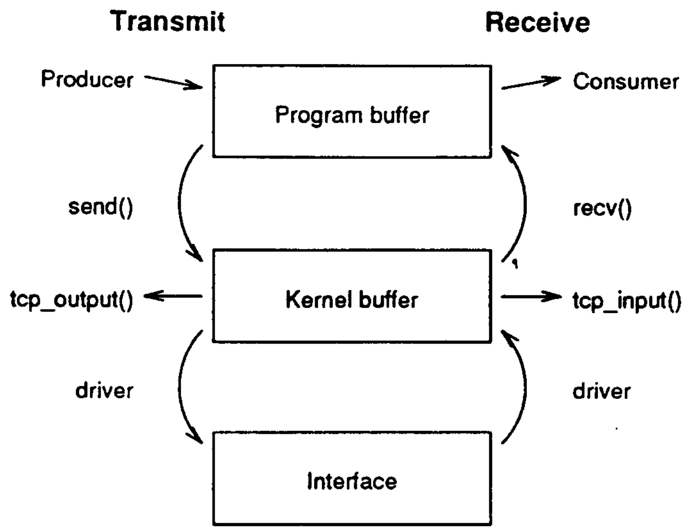

Figure 1: *Data movements in a typical TCP/IP implementation*

Transmission Producer is a program which has a connection to another machine via a stream socket. It has generated a quantity of data in a buffer, and calls the send function to transmit it.

Send begins by copying the data into a kernel buffer. The amount of data depends on the program – not on the network packet size – and it may be located anywhere in the program's data space. The copy allows Producer to reuse its buffer immediately, and gives the networking code the freedom to arrange the data into packets and manage their transmission as it sees fit.

Tcp_output gathers a quantity of data from the kernel buffer and begins to form it into a packet. Where possible, this is done using references rather than copying. However, tcp_output does have to calculate the packet's checksum and include it in a header; this entails reading the entire packet.

Eventually, the network interface's device driver receives the list of headers and data pointers. It copies the data to the interface, which transmits it to the network.

Reception The driver copies an incoming packet into a kernel buffer, then starts it moving through the protocol receive functions. Most of these only look at the headers.

Tcp_input, however, reads all the data in the packet to calculate a checksum to compare with the one in the header. It places valid data in a queue for the appropriate socket, again using pointers rather than copying.

Some time later, the program **Consumer** calls the function **recv**, which copies data from the kernel buffer into a specified area. As with **send**, **Consumer** may request any amount of data, regardless of the network packet size, and direct the data anywhere in its data space.

Where does the time go?

The standard implementation of TCP/IP copies data twice and reads it once in moving it between the program and the network. Clearly, the rate at which a connection can convey data is limited by the rates at which the system can perform these basic operations.

As an example, consider a system on which the **Producer** program is sending a continuous stream of data using TCP. Our measurements show that an HP 9000/730 workstation can copy data from a buffer in cache to one not in the cache at around 50 Mbyte/s[1], or 19 nanoseconds per byte. The rate for copying data from memory to the network interface is similar. The checksum calculation proceeds at around 127 Mbyte/s, or 7.6 ns per byte. All of these operations are limited by memory bandwidth, rather than processor speed.

Each byte of an outgoing packet, then, takes at least 45.6 ns to process: the fastest this implementation of TCP/IP can move data is about 21 Mbyte/s (176 Mbits/s). Overheads such as protocol handling and operating system functions will ensure it never realizes this rate.

Several schemes for increasing TCP throughput try to eliminate the checksum calculation. Jacobson [5] has shown that some processors, including the HP 9000/700, are able to calculate the checksum while copying the data without reducing the copy rate. Others add support for the calculation to the interface hardware. Still others propose simply dispensing with the checksum in certain circumstances.

Our figures, however, suggest that for transmission, the checksum calculation accounts for only about one-sixth of the total data manipulation time: getting rid of it increases the upper bound to around 25 Mbyte/s (211 Mbits/s). Each data copy, on the other hand, takes more than a third of the total. Eliminating one copy would increase the data handling rate to more than 36 Mbyte/s (301 Mbits/s), and removing both a copy and the checksum calculation would increase it to 50 Mbyte/s (421 Mbits/s). Clearly, there are considerable rewards for reducing the number of copies the stack performs.

For a better idea of the effect the changes would have in practice, we need to include the other overheads incurred in sending packets. In particular, we need to consider the time taken by each call to **send**, and the time needed to process each packet in addition to moving the data. On a 9000/730, these are roughly 40 μs and 110 μs respectively. These times are large, but include overheads such as context switches, interrupts, and processing TCP acknowledgements.

[1] We use the convention that Kbyte and Mbyte denote 2^{10} and 2^{20} bytes respectively, but Mbit and Gbit denote 10^6 and 10^9 bits.

Table 1 gives estimates of TCP throughput for three implementations: the conventional one, one without a separate checksum calculation ("two-copy" for short), and one using just a single copy operation. The estimates assume a stream transmission using 4 Kbyte packets, with each call to **send** also writing 4 Kbytes. Even with such small packets and large per-packet overheads, the single-copy approach is significantly faster.

Implementation	Time per packet (μs)				Throughput
	send()	packet	data	Total	(Mbyte/s)
Conventional	40	110	187	337	11.6
Two-copy	40	110	156	306	12.8
Single-copy	40	110	78	228	17.1

Table 1: Estimated TCP transmission rates for three implementations

Analysing the receiver in the same way gives similar results, as shown in table 2. The main differences from transmission are that copying data from the interface to memory is slower, at around 32 Mbyte/s, or 30 ns per byte, and that the overheads of handling an incoming packet and the **recv** system call are also smaller, approximately 90 μs and 15 μs respectively.

Implementation	Time per packet (μs)				Throughput
	recv()	packet	data	Total	(Mbyte/s)
Conventional	15	90	256	361	10.8
Two-copy	15	90	193	298	13.1
Single-copy	15	90	124	229	17.1

Table 2: Estimated TCP reception rates for three implementations

Before we consider the single-copy approach in more detail, we examine the trends in two relevant technologies: memory bandwidth and CPU performance. Memory bandwidth affects the transmission of every byte and, for large packets, is arguably the limiting factor. CPU performance determines the time to execute the protocols for each packet, but this effort is independent of the length of the packet. (A more detailed look at the effect of memory systems is given by Druschel *et al.* [3] in this issue.)

Over the past few years main memory (Dynamic RAM) has been getting faster at the rate of about 7% per annum whereas CPU ratings in terms of instructions per second have increased by about 50% per annum. We believe that reducing the number of data copies in protocol implementations will yield significant benefits as long as this trend continues.

Minimizing data movement

Several suggestions have been made to reduce the number of times that application data must be accessed [3]. This section describes three of them: copy-on-write, page remapping and single-copy.

Copy-on-write When a program sends data, the system makes the memory pages that contain the data *read-only*. The data go to the network interface directly from the program's buffer. The pages are made *read-write* again once the peer process has acknowledged the data.

The program is able to continue, but if it tries to write to the same buffer before the data are sent, the Memory Manager blocks the program and the networking code copies the data into a system buffer. (In a variation called **sleep-on-write**, the Memory Manager forces the process to wait until the transmission is complete.)

Copy-on-write needs changes to the system's Memory Manager as well as the networking code. For best performance, programs should be coded not to write to buffers that contain data in transit.

Page remapping The system maintains a set of buffers for incoming data. The network interface splits incoming packets, placing the headers in one buffer and the data in another starting at a memory page boundary. When a program receives the data, if the buffer it supplies also starts on a page boundary, the Memory Manager exchanges it for the buffer containing the data by remapping the corresponding pages, i.e., by editing the system's virtual memory tables.

Page remapping needs changes to both the memory management and networking code. In addition, the network interface hardware has to be able to interpret incoming packets well enough to find the headers and data. Application programs must be written to use suitably aligned buffers.

Single-copy This technique works when both sending and receiving data. It needs a dedicated area of memory which the processor and network interface share without affecting each other's performance.

When a program sends data, the networking code copies the data immediately into a buffer in the dedicated area. The various protocol handling routines prefix their headers to the data in the buffer, then the network interface transmits the whole packet in one operation.

The interface places incoming packets in buffers in this area before informing the network code of their arrival. The data remain in the dedicated buffer until a program asks to receive them, when they are copied into the program's buffer.

The single-copy technique only affects the system's networking code. Significantly, user programs get the full benefit without being altered in any way.

All three techniques can reduce the number of copy operations needed by a protocol implementation. All need hardware support of some kind, and their relative effectiveness depends on the characteristics of that support and of the data traffic being handled. In our view, single-copy has three distinct advantages over the others, for general TCP traffic. First, it affects only the networking code in the system. Second, it speeds up both sending and receiving data. Third, most existing programs benefit without being recoded or recompiled.

The Afterburner Card

Van Jacobson has proposed WITLESS [2] [4], a network interface designed to support single-copy implementations of protocols such as TCP. We previously built an FDDI interface, Medusa [1], as a test of the WITLESS architecture. The results were excellent, and we decided to adapt the design to support link rates up to 1 Gbit/s.

Afterburner is designed for the HP 700 series of workstations. It occupies a slot in the workstation's fast graphics bus, and is mapped into the processor's memory. Figure 2 shows Afterburner's architecture.

The focus of the card is a buffer built from three-port Video RAMs (VRAMs). One port provides random access to the buffer for the workstation's CPU; the other two are high-speed serial ports connected to fast I/O pipes. To the CPU, the VRAM is one large buffer, but Afterburner itself treats the VRAM as a set of distinct, equal-sized blocks. The block size is set by software when the card is initialised, and ranges from 2 Kbytes to 64 Kbytes.

Sending and receiving data The CPU and Afterburner communicate with each other mainly through four FIFOs: two for transmission (Tx) and two for reception (Rx). An entry in one of these FIFOs is a descriptor which specifies a block of VRAM and tells how many words of information it contains. Descriptors in the Tx_Free and Rx_Free FIFOs identify blocks of VRAM available for use; those in the Tx_Ready and Rx_Ready indicate data waiting to be processed.

To transmit a message, the CPU writes it into the VRAM starting at a block boundary, then puts the appropriate descriptor into the Tx_Ready FIFO. Afterburner takes the entry from the FIFO and streams the message from the VRAM to the Tx_Data FIFO. When finished, Afterburner places the block address into the Tx_Free FIFO.

Similarly, when a message arrives on the Rx_Data FIFO, Afterburner takes the first entry

[2] Workstation Interface That's Low-cost, Efficient, Scalable, and Stupid

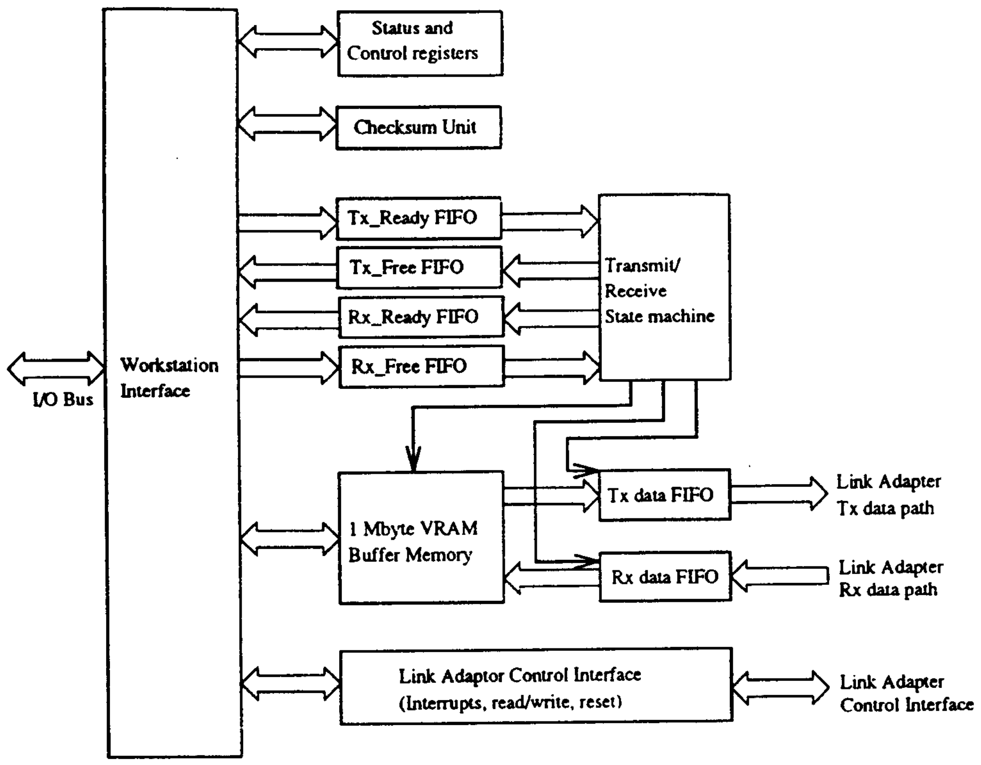
Status and Control registers
Checksum Unit
Tx_Ready FIFO
Tx_Free FIFO
Rx_Ready FIFO
Rx_Free FIFO
Transmit/ Receive State machine
Workstation Interface
I/O Bus
1 Mbyte VRAM Buffer Memory
Tx data FIFO
Rx data FIFO
Link Adapter Tx data path
Link Adapter Rx data path
Link Adaptor Control Interface (Interrupts, read/write, reset)
Link Adapter Control Interface

Figure 2: *Afterburner Block Diagram*

from the Rx_Free FIFO, and streams the data into the corresponding block of VRAM. At the end of the message, Afterburner fills in the descriptor with the message's length, then puts it in the Rx_Ready FIFO. When the CPU is ready, it takes the descriptor from the Rx_Ready FIFO, processes the data in the block, then returns the descriptor to the Rx_Free FIFO. The CPU has to prime the Rx_Free FIFO with some descriptors before Afterburner is able to receive data.

Large messages Although Afterburner allows VRAM blocks up to 64 Kbytes in size, it provides a mechanism for handling large packets that is better suited to the wide range of message sizes in typical IP traffic. Packets can be built from an arbitrary number of VRAM blocks.

In addition to specifying a block and the size of its payload, the descriptor in a FIFO contains a flag to indicate whether the next block in the queue belongs to the same message. To send a message larger than a VRAM block, the CPU writes the data in several free blocks – they need not be contiguous – then puts the descriptors in the right order into the Tx_Ready FIFO, setting the "continued" flag in all but the last. Afterburner transmits the contents of the blocks in order as a single message. Long incoming messages are handled in a similar way.

Interrupts When Afterburner has received data, it has at some point to interrupt the host. Because interrupts can be expensive – several hundred instructions – it is important for the card to signal only when there is useful work to do. Afterburner provides several options. The simplest is to interrupt when Rx_Ready becomes non-empty. When long packets are common, the most useful is to interrupt when Rx_Ready contains a complete message, i.e., Rx_Ready contains a block with the "continued" bit not set.

The card is also able to interrupt the host when it has transmitted a block (the Tx_Free FIFO becomes non-empty) or when it has transmitted all it had to do (the Tx_Ready FIFO becomes empty). Normally, the card would be configured to interrupt only for incoming data.

Link Adapters So far, we have not mentioned the connection to the physical network. When we began to design Afterburner there was no obvious choice for a network operating at up to 1 Gbit/s. Rather, there were several possibilities. So, Afterburner is not designed for a particular LAN: it has no MAC or Physical layer devices. Instead, it provides a simple plug-in interface to a number of "Link Adapters", each designed to connect to a particular network.

The interface consists of three connectors. One provides a simple address and data bus for the host CPU and Link Adapter to communicate directly. The other two connect the Adapter to Afterburner's input and output streams. When Afterburner and an Adapter are mated, the combined unit fits into the workstation as a single card.

To date, three link adapters have been designed: one for HIPPI [8], one for ATM, and one for Jetstream, an experimental Gbit/s LAN developed at HP Labs in Bristol.

One substantial benefit of the separation between Afterburner and the link adapter is that network interfaces do not need to be redesigned for each new generation of workstation. Only the Afterburner card needs to be redesigned and replaced, and typically, the redesign affects only the workstation interface.

A Single-copy Implementation of TCP-IP

This section describes an implementation of TCP/IP that uses the features provided by Afterburner to reduce the movement of data to a single copy. The changes we describe were made to the networking code in the 8.07 release of HP-UX, itself derived from that in the 4.3BSD system.

Ours is not a complete reimplementation of TCP, but simply adds a single-copy path to the existing TCP code. We did this for practical reasons, but a side-effect is that protocol processing hasn't changed: any changes in performance are mainly from changes in data handling.

The principles of the single-copy implementation are simple: put the data on the card as early as possible, leave it there as long as possible, and don't touch it in between. Figure 3 gives a very simple view of the single-copy route. Compared with the conventional implementation (figure 1), the socket functions **send** and **receive** do most of the work, including the one data movement. The other protocol functions handle only a small amount of control information, represented by the dotted lines.

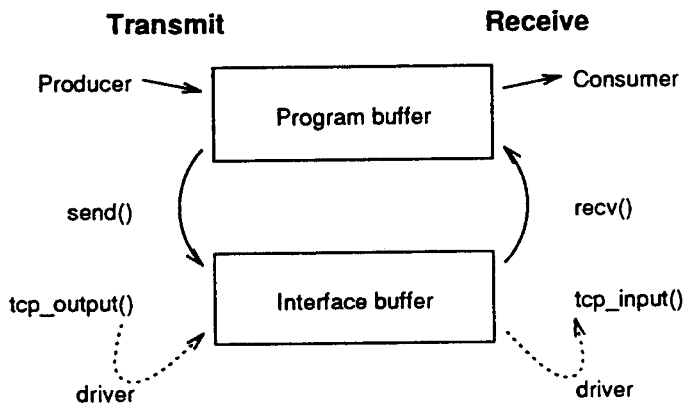

Figure 3: *The single-copy approach*

In the remainder of this section, we give an overview of the main features of our single-copy stack compared with the standard one. We also discuss a number of issues which have emerged during the course of the work.

Data structures – mbufs and clusters

The networking code keeps data in objects called **mbufs**. An mbuf can hold about 100 bytes of data, but in another form, called a **cluster**, it can hold several Kilobytes. Most networking data structures, including packets under construction and the kernel buffer in figure 1, consist of linked lists, or chains, of mbufs and clusters. The system provides a set of functions for handling mbufs, e.g, for making a copy of a chain, or for trimming the data in a chain to a particular size.

Normally, clusters are fixed-size blocks in an area of memory reserved by the operating system. We enhanced the mbuf-handling code to treat blocks of Afterburner's VRAM buffer as clusters. Code is able to tell normal clusters from single-copy ones.

Single-copy clusters carry additional information, for example, the checksum of the data in the cluster. However, the main difference in handling them is that, in general, code should not try to change their size nor move their contents. Either of these operations would imply having to make an extra pass over the data, either to copy it, or to recalculate a checksum.

This characteristic also has an effect on the behaviour of the protocol which we discuss further below.

We have had to alter several of the mbuf functions in the kernel to take these differences into account. To support the use of the on-card memory as clusters, we have written a small number of functions. The most important is a special copy routine, functionally equivalent to the BSD function bcopy. It is optimised for moving data over the I/O bus, and also optionally uses the card's built-in unit to calculate the IP checksum of the data it moves. Another function converts a single-copy cluster into a chain of normal clusters and mbufs; it also calculates the checksum.

Sending data

As before, Producer has already established a socket connection with a program on another machine, and calls the socket send function to transmit it.

The socket layer – send Send decides, from information kept about the connection, to follow the single-copy course. It therefore obtains a single-copy cluster and copies the data from Producer's buffer into it, leaving just enough room at the beginning of the cluster for headers from the protocol functions. The amount of space needed is a property of the connection and is fixed when the connection is established. It depends on the TCP and IP options in use[3]. The copy also calculates the checksum of the data, and send caches this in the cluster along with the length of the data and its position in the stream being sent.

The data in the single-copy cluster are now physically on the interface card. Logically, however, the cluster is still in the the send socket buffer – the queue of data waiting to be transmitted on this connection. Often, the last mbuf in the queue is a single-copy cluster with some room in it, so, when possible, send tries to fill the cluster at the end of the socket buffer before obtaining a new one.

TCP_output In general, the send socket buffer is a mixture of normal mbufs and both normal and single-copy clusters. To build a packet, tcp_output assembles a new chain of mbufs that are either copies of mbufs in the socket buffer or references to clusters there. It also ensures that the packet's data is either in normal mbufs and clusters or in a single-copy cluster – never both.

Tcp_output sets the size of normal packets based on its information about the connection, and collects only as much as it needs from the socket buffer. Conversely, it treats single-copy clusters as indivisible units, and sets the size of the packet to be that of the cluster[4].

[3] In future, using Afterburner's ability to form packets from groups of VRAM blocks will remove the need to leave this space

[4] This can have undesirable effects, which are discussed under "Issues".

As in the normal stack, **tcp_output** builds the header in a separate (normal) mbuf, and prefixes it to the single-copy cluster. An important part of the header is the packet checksum, which covers both data and header. With normal packets, **tcp_output** reads all the data in a normal packet to calculate the checksum. A single-copy cluster already contains the data's checksum, so the calculation involves only the header and some simple arithmetic.

When the TCP header is complete, **tcp_output** passes the packet to the **ip_output** and link-layer functions. These are the same for both normal and single-copy packets, so we shall not discuss them here.

The device driver Here, the packet is sent to the network. With single-copy chains, all the driver has to do to complete the packet is to copy the various protocol headers at the beginning of the chain into the space the **send** function left at the beginning of the single-copy cluster. With chains of normal mbufs, the driver copies the contents of the mbuf chain onto the card, using a VRAM block from a small pool reserved for the driver. When ready, the driver constructs the descriptor for the packet and writes it to the Tx_Ready FIFO.

Receiving data

Receiving packets is more complicated than sending them. The sender knows everything about an outgoing packet except whether it will arrive safely, and it can reasonably expect its information to be accurate. The receiver, on the other hand, has to work everything out from the contents of the packet, and - until it knows better - it has to assume that information may be wrong or incomplete.

The device driver To decide whether the packet should take the single-copy or normal route, the driver examines the incoming packet to discover its protocol type, the length of the packet, and the length of its headers. There are four cases:

> **Non-IP packets.** The driver copies the entire packet into a chain of normal mbufs.
>
> **Small IP packets (less than 100 bytes).** The driver creates a chain of two normal mbufs: the first contains the link header, the second contains the whole IP packet.
>
> **Large TCP/IP packets.** The driver creates a chain of three mbufs. The first two are normal, and contain the headers. The third is a single-copy cluster - the VRAM block containing the packet.
>
> **All other IP packets** The driver creates a chain of normal mbufs. The first contains the link header, the second, the IP and other headers. The remainder contain data.

Small packets are treated specially for several reasons. Many such packets have only one or two bytes of payload, e.g, single characters being typed or echoed during a remote login. It's quicker to process these packets as one mbuf in the conventional stack than it is to process a single-copy chain. Also, copying in the data immediately frees the VRAM block for re-use. Because buffers on the card are a relatively scarce resource, this is important when the receiving application is very slow or when the transmitter is sending a rapid stream of short messages.

Tcp_input The first thing **tcp_input** normally does with an incoming packet is to calculate its checksum and compare it with the one in the TCP header. This checks the integrity of both header and data. It is possible, however, to defer the checksum calculation until later. The important thing is to ensure that an erroneous packet doesn't cause **tcp_input** to change the state of some connection.

When it receives a single-copy packet, **tcp_input** checks the header for three things: that the packet is for an established connection; that the packet simply contains data, not control information that would change the state of the connection; and that the data in the packet are the next in sequence on the connection. **Tcp_input** converts any packet that fails one of these tests into normal mbufs, calculating the checksum in the process, then processes it as usual.

A single-copy packet that passed the tests is easy for **tcp_input** to handle. It calculates the checksum of the packet's TCP header, and stores it and a small amount of information from the header in the cluster, then appends the cluster to the appropriate receive socket buffer. Even if one is eventually found to be in error, it won't have changed the connection state.

These tests are slight extensions of ones already in the conventional stack. **Tcp_input** implements a feature called "header prediction" that tests most fields in the TCP header against a set of expected values. Packets which match are able to be processed quickly; all others, including those that alter the state of the connection or that require special processing, take a slower route. In typical stream connections, the only packets needing special treatment are those which establish or close the connection.

The socket layer – recv This is the most intricate area of the single-copy code. As well as copying data from the socket buffer into a buffer in the program, **recv** has to verify the data are correct, acknowledge data, manage data the program has not yet asked for, and keep the information about the state of the connection up to date. To complicate matters, the receive socket buffer is a mixture of normal and single-copy mbufs.

In the simplest case, the socket buffer contains one single-copy cluster, and the program asks **recv** for as much data as it can provide. **Recv** copies all the data from the cluster to the program's buffer, calculating the checksum as it does so. It then compares the result with the checksum **tcp_input** placed in the cluster header. If the two match, it removes the cluster from the socket buffer and updates the socket and TCP control information. This causes

TCP to acknowledge the new data in due course. Should the checksum test fail, **recv** restores the buffer as far as possible to its original condition, although the original contents are lost. Recv then acts as if the packet had not arrived, returning EAGAIN or EWOULDBLOCK as appropriate.

When the program requests an amount smaller than that contained in a single-copy cluster, recv must honour the request while still calculating the checksum of the entire packet. The simplest way to handle this is to convert the cluster to ordinary mbufs, verify the checksum, then copy the required data into the program's buffer. This does, however, mean copying the data twice.

The same situation arises in a more general form whenever the program asks **recv** for an amount that isn't contained in an integral number of clusters. Recv's semantics, however, allow it to return less than the requested amount. A reasonable solution in this case is to return the largest amount which doesn't require a single-copy cluster to be split.

Issues

We now consider some of the more substantial issues that have arisen from our single-copy implementation of TCP.

Building packets in the socket layer

Once **send** has built a single-copy cluster, it is difficult to change what it has done. The problem is that **send** does not have all the information it needs to know how much data to place in each cluster: it has to guess. Upper bounds exist: the capacity of the cluster, the maximum segment size negotiated by TCP when the connection was set up, and the largest window the receiver has advertised. These can be too large. Consider a connection which starts with a segment size of 8 Kbytes and an initial receiver window of 32 Kbytes which later shrinks to 4 Kbytes. Following the naïve policy, **send** continues to generate 8 Kbyte packets, and TCP must send them. The receiver trims incoming packets to fit its window, so when it receives an 8 Kbyte packet, it deletes the last half. Then the sender retransmits the whole packet, and the receiver discards the first half. While this costs the sender very little, it increases the amount of traffic and worst of all, makes more work for the receiver, which presumably is busy already.

The ideal would be for **sosend** to know in advance how large the receiver's window will be when TCP sends the packet being built. This is not necessarily the same as the contemporary window size; several packets may be transmitted before this one, and more significantly, there is no way to tell what acknowledgements the sender will receive in the interim. One possible approach is to estimate the future window size, for example, by using a weighted average of the last few known window sizes.

Delayed checksum calculation and acknowledgements

We have already described how the single-copy implementation of TCP postpones calculating the checksum of incoming data until some program actually asks to receive it. There is a further issue: how do we ensure that the receiver acknowledges data reliably?

A TCP sender expects to receive an acknowledgement for data it sends within a reasonable time, typically a small multiple of the time it would take a packet to travel from sender to receiver and back again. If it has not received an acknowledgement, the sender assumes the data were lost in transit and retransmits them. If this happens repeatedly, the sender may give up and close the connection. The conventional implementation can verify and acknowledge packets shortly after they arrive, regardless of what the receiving program is doing. In the single-copy stack, this doesn't happen until that program actually calls **recv**. Should the program be blocked for some reason - waiting for a lock, or suspended by a user - it can't send acknowledgements, and the connection is in trouble.

We have resolved this by adapting a mechanism which, paradoxically, the conventional implementation uses to delay sending acknowledgements. To avoid flooding the network and the sender with an acknowledgment for every incoming segment, TCP maintains an interval timer, sending one acknowledgment for all data received during the period. The single-copy implementation uses the same timer to acknowledge data on behalf of sluggish programs. On each tick of the clock, if a receive socket buffer contains unacknowledged data and **recv** has not been called since the previous tick, TCP converts the first single-copy packet in the socket buffer into normal mbufs, verifies its checksum, and sends the appropriate acknowledgement.

On-card buffers are a limited resource

The current design of Afterburner has one megabyte of buffer space, which the software divides almost equally between transmitter and receiver. From our experience so far, we believe this amount is adequate for most "normal" use, but shortages can happen.

The obvious problem is heavy loading: a large number of connections, each with a few tens of kilobytes of outstanding data could fill the card. In practice so far, this situation has been rare. The most likely explanation is that real programs take time to process data, and don't just send or receive it. It also seems that on workstations, normally only one or two programs are active at any one time.

More serious problems come from unusual programs or situations. Two sending programs, each with quarter-megabyte send socket buffers, can usually operate smoothly in a single Afterburner. If something causes their receivers to slow down or stop, however, they will fill all the space available. Similarly, a single program which sends data in small messages using the TCP_NODELAY socket option can quickly monopolise the buffer space.

There are several strategies the single-copy implementation can use to reduce the problem. Generally, these include prevention, such as attempting to detect and handle situations such as the TCP_NODELAY user, and repair, such as making buffer space available by moving data from single-copy clusters into normal ones. Work continues to find the best mix of strategies to use.

Performance

In this section we present measurements of the end-to-end performance of Afterburner and our single-copy implementation of TCP/IP.

The tests were performed on HP 9000/730 workstations, each with 32 Mbytes of main store and one 420 Mbyte disk. Two workstations were connected together via ribbon cables; no link adapter was present. The benchmarks were the only active 'user' processes, but the systems had the usual assortment of daemons in the background, and were attached to the lab Ethernet LAN.

We used a tool called *netperf* [6] to perform the measurements. As we used it, the test measures the flow rate between a producer program on one machine and a consumer on the other. The producer repeatedly calls **send** with a fixed amount of data; the consumer continuously calls **recv**, requesting all available data. Neither program accesses the data being sent or received. Netperf was written within HP, but is available from a variety of sources, including several network archive sites.

Single-copy compared with the normal implementation

Earlier, we estimated the one-way stream throughput of three implementations of TCP/IP: conventional, two-copy (without a separate checksum calculation), and single-copy. Figure 4 shows the results of measuring their performance. The tests used 4 Kbyte packets and 56 Kbyte socket buffers, and vary the amount of data per **send** from 1 to 50 Kbytes.

The results are roughly as predicted by our analysis. With the application sending at least 4 Kbytes at a time, the conventional stack delivered an average throughput of 8.4 Mbyte/s (71 Mbit/s). Without the separate checksum, the average throughput was 11.1 Mbyte/s (93 Mbit/s). The single-copy implementation achieved an average of 16.6 Mbyte/s (140 Mbit/s).

Throughput and packet size

With 4 Kbyte packets, the time the system spends handling the packet is comparable to the time it spends moving the data. Some recent work [5] has reduced these overheads considerably, but unfortunately is not yet widely available. However, a simple way to reduce the effect of the per-packet costs is to send larger packets.

To investigate the effect of using larger packets, we ran the same tests as before on the single-copy implementation varying the packet size from 4 to 14 Kbytes. This socket buffer size was 192 Kbytes; this was mainly to eliminate effects seen when the socket buffer is a

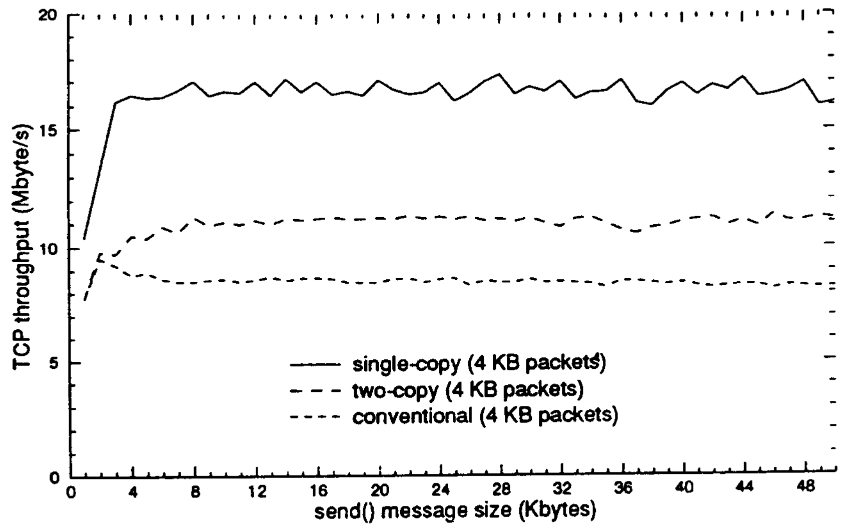

Figure 4: *Stream throughput of three implementations of TCP/IP*

small multiple of the packet size. Figure 5 shows the results.

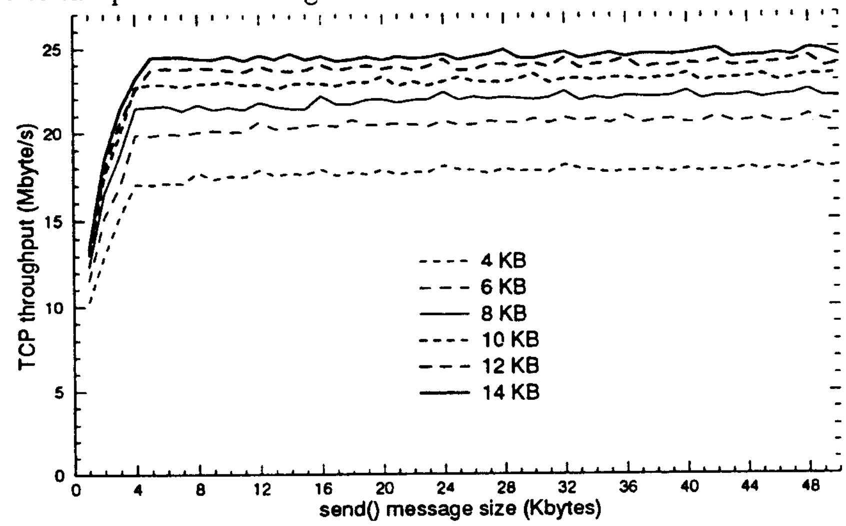

Figure 5: *Single-copy throughput for different packet sizes*

As expected, an increase in the packet size results in greater overall throughput. It is also clear that increasing the packet size yields diminishing returns in terms of performance, and the throughput is tending towards a limit which in this case is the interface-to-memory copy rate of 32 Mbyte/s (267 Mbit/s). It is very pleasing to see that excellent throughput- 25 Mbyte/s (210 Mbit/s)- is achieved with relatively small 14 Kbyte packets. This suggests that 64 Kbyte and even 32 Kbyte packets may prove to be unnecessary in order to achieve Gbit/s performance in the future.

Conclusions

Many current implementations of network protocols such as TCP/IP are inefficient because data are often accessed more frequently then necessary. We have described three techniques which have been proposed to reduce the need for memory bandwidth. Of these three, we have implemented the single-copy approach, and we have measured the performance of our implementation.

Afterburner is a network-independent card that provides the services that are necessary for a single-copy protocol stack. The card has 1 Mbyte of local buffers, and provides a simple interface to a variety of network link adapters including HIPPI and ATM. Afterburner can support transfers to and from the link adapter card at rates up to 1 Gbit/s.

While the Afterburner model is quite general, our implementation is very specific to the HP series 700 workstations. Data transfers are achieved by programmed I/O for outbound data and by block move hardware for inbound. If we were to design Afterburner for a different workstation, DMA might prove to be the most effective mechanism – there is no single approach that will be best for all workstations. What is clear to us is that this decision can be quantified and so the best mechanism can be determined by simply counting the cycles required to move data by each of the possible methods.

Experiments with HP series 700 workstations have shown that applications can communicate at more than 200 Mbit/s using a single-copy implementation of TCP and the Afterburner network interface. We believe this architecture will scale to future workstations to yield throughputs of 1 Gbit/s.

1 References

[1] D. Banks, M. Prudence, "A High Performance Network Architecture for a PA-RISC Workstation", *IEEE Jnl. Sel. Areas in Comms*, Vol. 11, No. 2, Feb 1993.

[2] D.D. Clark, Van Jacobson, J. Romkey and H. Salwen, "An Analysis of TCP Processing Overhead", *IEEE Commun. Mag.*, pp. 23-29, June 1989.

[3] P. Druschel, M.B. Abbott, M.A. Pagels and L.L. Peterson, "Network Subsystem Design: A Case for an Integrated Data Path" , to appear in *IEEE Network Mag.*, July 1993.

[4] Van Jacobson, "Efficient Protocol Implementation", ACM SIGCOMM '90 Tutorial, September 1990.

[5] Van Jacobson, "A High-performance TCP/IP Implementation", Gigabit/s TCP Workshop, Center for National Research Initiatives, Reston VA, March 1993.

[6] R.A. Jones, "Netperf: A Network Performance Benchmark", Revision 1.7, Hewlett-Packard Co., March 1993. The netperf utility can be obtained via anonymous ftp from ftp.csc.liv.ac.uk in /hpux/Networking.