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.

Internal Accession Date Only

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.

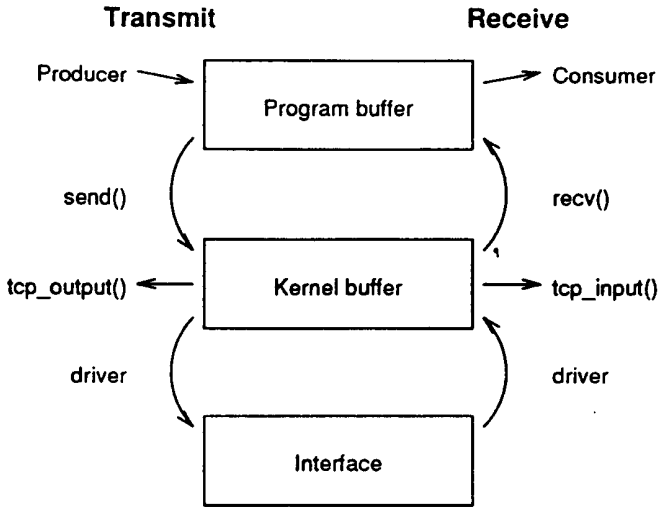

Figure 1: *Data movements in a typical TCP/IP implementation*

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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.