Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000*

Jonathan Kay (jkay@cs.ucsd.edu)
Joseph Pasquale† (pasquale@cs.ucsd.edu)

Computer Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract

Networking software is a growing bottleneck in modern workstations, particularly for high throughput applications such as networked digital video. We measure various components of the UDP/IP protocol stack in a DECstation 5000/200 running Ultrix 4.2a, and quantify the way in which checksumming and copying dominate the processing time for high throughput applications. This paper describes network software measurements and substantial performance improvements which derive from a faster checksum implementation.

1.0 Introduction

Network software of most modern workstation operating systems is not able to take full advantage of hardware speeds. With emerging high-bandwidth network hardware and network-I/O intensive applications, operating system network software is increasingly a bottleneck. The highest-bandwidth network widely supported by workstation vendors is FDDI, which operates at 100 megabits per second or 12.5 megabytes per second (MB/s). Measuring the performance of DEC's DEFZA FDDI controller for a DECstation 5000/200 at the level of the device-driver, we find that the send rate is 7-8 MB/s (the receive rate is even higher) using maximum size FDDI packets. However, when sending from user level using UDP/IP over an FDDI network, the maximum sustained throughput measures at only 2.4 MB/s. The network software is reducing throughput from user processes to the device by a factor of three.

The goal of this study is to determine how various components of network software contribute to this bottleneck. We instrument the UDP/IP protocol stack since most of the traffic on our departmental networks is comprised of NFS-generated UDP packets. We analyze network software component processing time by layer and by operation. By layer, we measure the individual processing times of the socket, UDP, IP, link, and device driver software. By operation, we measure the processing times for various copying, checksumming, and malloc/free operations. Other studies have shown these operations to be expensive [3-5, 7]. Cabrera et al. discuss a number of network software bottlenecks in the 4.2 release of Berkeley Unix [3]. Clark notes that checksumming and to a lesser extent copying were the dominant costs in the Multics TCP/IP implementation [4]. Clark et al. discuss copy and checksum costs versus other protocol costs in the context of a precursor to the 4.3 Reno release of Berkeley Unix and an experiment that

*. An earlier version of this paper appeared in the 1993 Winter USENIX Conference.

†. This work was supported in part by grants from DEC, IBM, NCR, NSF, TRW, and UC MICRO.

attempted to isolate protocol processing costs [5]. Jacobson discusses a number of costs in the 4.3 Tahoe release of Berkeley Unix and their improvements in the Reno precursor [7].

This work is part of Project Sequoia 2000 [9], a project which brings together computer scientists and global change scientists from all over the University of California, including from UC San Diego and UC Berkeley, to improve network, storage, database, and visualization support for global change research. An important goal of the Sequoia 2000 network [6] which we are constructing is to support high throughput for the large volumes of data required for global change applications. The research described here furthers that goal by improving the effective networking bandwidth available to Sequoia 2000 researchers.

2.0 Experiments

We connect an HP 1652B Logic Analyzer to the DECstation TURBOchannel to obtain software processing time measurements with a resolution of 40 nanoseconds (which is the DECstation clock cycle time). Special cpp macros are placed at the beginning and end of the source code of each operation of interest. Each time a macro executes, a pattern plus an event identifier is sent over the DECstation's I/O bus. The logic analyzer is programmed to recognize the pattern and store the event identifier, along with a timestamp. The measurement software causes minimal interference, generating overhead of less than 1% of the total network software processing time. Statistical analysis is done off line.

The experimental system consists of two workstations connected by an FDDI network with no other workstations and no network traffic other than that generated by the experiment. An experiment consists of one workstation sending a message to the *system under test* , which then sends the same message back. All measurements are made on the system under test, which is executing a probed kernel and is hooked up to the logic analyzer. The experiments are repeated 100 times for each message size to obtain statistical significance in the results. That experiment, in turn, is repeated for 40 message sizes in the range between 1 and 8192 bytes. It is worth noting that NFS typically transfers large files in units of 8192 byte blocks.

3.0 Measurements

We present the total message processing time for various components of network software when receiving and then sending the same-sized message, by layer and by operation. For example, a checksum time of 359 microseconds for a 1024 byte message refers to the total time spent in the six calls to the checksum routine: two header and one data checksum computed while receiving a 1024 byte message and sending the same message back. Since the maximum FDDI frame size is 4352 bytes, IP must break larger messages into two FDDI frames on the sending side and reassemble them on the receiving end (fragmentation/defragmentation).

3.1 Analysis by Operation

We analyze operation overheads in two different ways. The first is to look at the overall operation times over both send and receive sides. This is relevant to total processing overhead and latency. However, since a major goal of this paper is to understand bottlenecks, and throughput is limited by max(send time, receive time), we also analyze operation times by send side vs. receive side.

3.1.1 Total Operation Costs

Figure 1 shows the per-message combined send and receive processing times for various operations: CHECKSUM, COPY, MALLOC/FREE, and OTHER (i.e. the combined times for all other operations). Figure 2 shows cumulative times for each operation (so that the top line is total processing time for messages of given sizes). Figure 3 is another cumulative graph of components of COPY.

CHECKSUM refers to computation of the Internet checksum in UDP and IP. Processing time for CHECKSUM increases with message byte size at a rate of approximately .330 usec per byte. For large messages, CHECK-

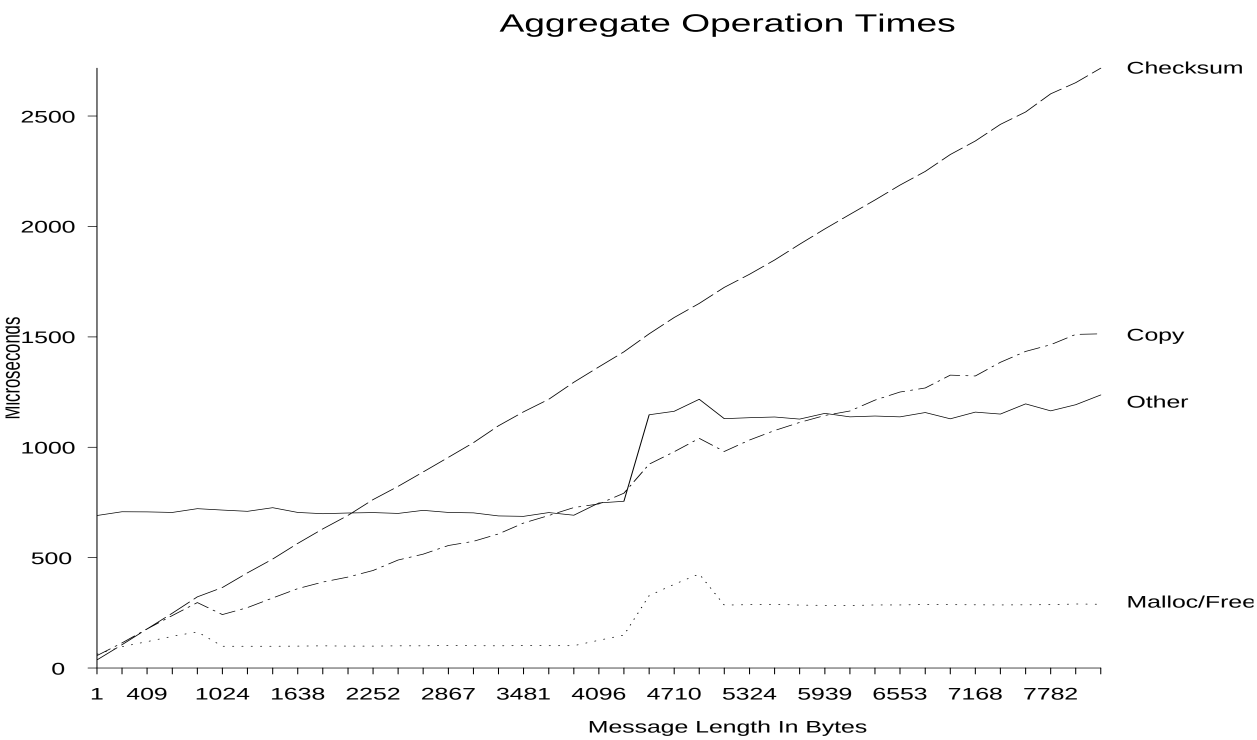

Figure 1

SUM processing time dominates that of all other operations; as can be seen in Figure 2, it is responsible for almost 50% of the total processing time for 8192 byte messages.

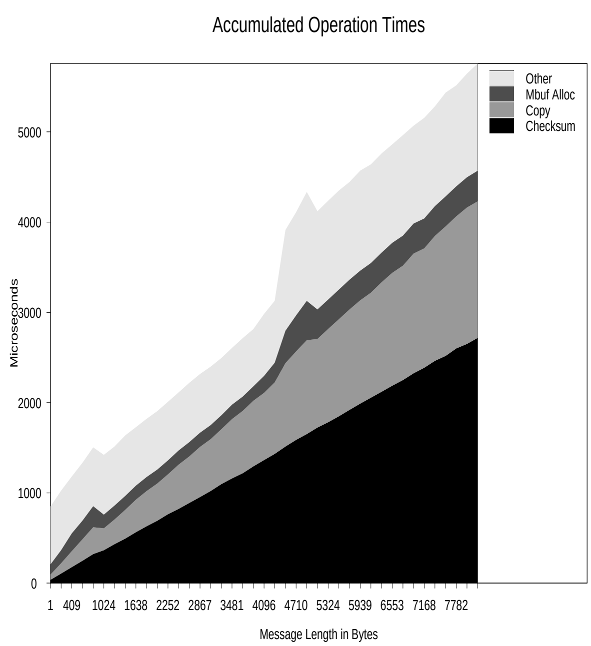

Figure 2

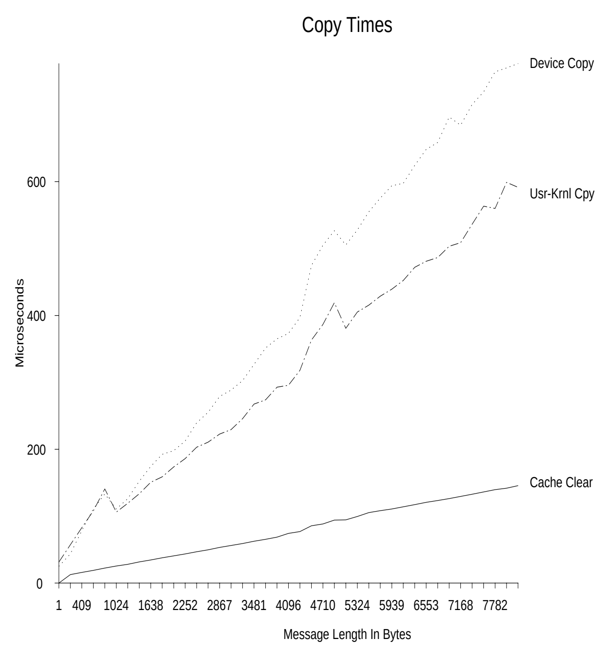

Figure 3

COPY refers to various bulk data copying operations. Figure 3 contains a breakdown of the various components of COPY: copying between user and kernel memories (35%), copying from the kernel to the FDDI controller across the relatively slow I/O bus (50%), cache clearing in preparation for DMA from the FDDI controller to kernel memory (10%). The values in parentheses refer to the percentage of total COPY time for which the operation is responsible for all but the smallest message sizes. Total processing time for COPY increases with message byte size at a rate of approximately .175 usec per byte, or about half the rate of CHECKSUM.

MALLOC/FREE refers to allocation and freeing of memory buffers. Berkeley UNIX, of which Ultrix is a derivative, stores messages in data structures called mbufs. An mbuf comprises a header, which can directly store up to 108 bytes of data, and possibly a page (4096 bytes) pointed to by the header, which can be used to store data of size greater than 108 bytes; if the data page is used, no data is stored in the header. To reduce internal fragmentation, the socket sending code allocates a string of up to 10 small mbufs if the message size is less than 1024 bytes. For messages larger than 1024 bytes but less than a 4096 byte page, an mbuf with one data page is allocated. A number of operations, including both malloc and free, must be performed on each mbuf. Consequently, the processing time for MALLOC/FREE increases with the number of 108 byte size blocks up to 1024 bytes, and then is constant for messages sizes which fit in one page. In a similar fashion, for messages larger than a page, another string of mbufs are allocated for the data between 4096 bytes and 5120 bytes, whereas for messages larger than 5120 bytes, two data pages are allocated. That is the reason for the humps, both in MALLOC/FREE and elsewhere, for sizes just less than 1024 bytes and 5120 bytes, and the constant times for messages with sizes between 1024 and 4096 bytes and between 5120 and 8192 bytes. Relative to total processing time, MALLOC/FREE accounts for 3-7%, but can go higher than 10% at the humps. However, this time is the aggregate of many mbuf allocation times; individual malloc operations are very cheap. Upon inspection of the Ultrix source code, it is obvious that a great deal of effort went into optimizing the speed of malloc and free.

OTHER refers to all other processing not accounted for by CHECKSUM, COPY, and MALLOC/FREE. This includes protocol processing, various mbuf operations (e.g. header prepending, queueing/dequeueing), fragmentation/defragmentation, software interrupts, checking system call parameters, etc. As can be seen from Figures 1 and 2, OTHER appears to increase with the number of mbufs in a similar fashion as MALLOC/FREE, except that it accounts for a much greater percentage of the total processing time than MALLOC/FREE.

3.1.2 Costs of Sending vs. Receiving

This subsection analyzes the breakdown of operation times into times spent to send messages vs. times spent to receive messages. Figures 4 and 5 show the per-message send and receive processing times, respectively. Figures 6 and 7 show the send and receive cumulative times. Figures 8 and 9 contain graphs of components of COPY. One can see from Figures 6 and 7 that the send-side and receive-side times are fairly well balanced, except at the humps caused by the mbuf allocation algorithms, where the receive side is noticeably faster than the send side.

.

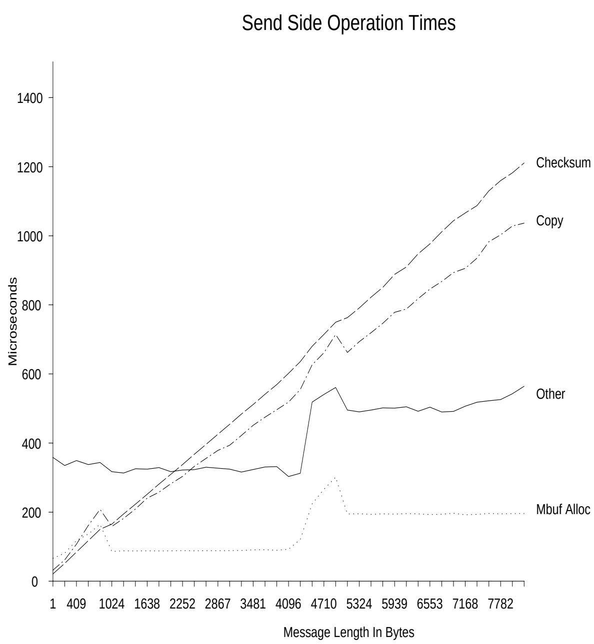

Figure 4

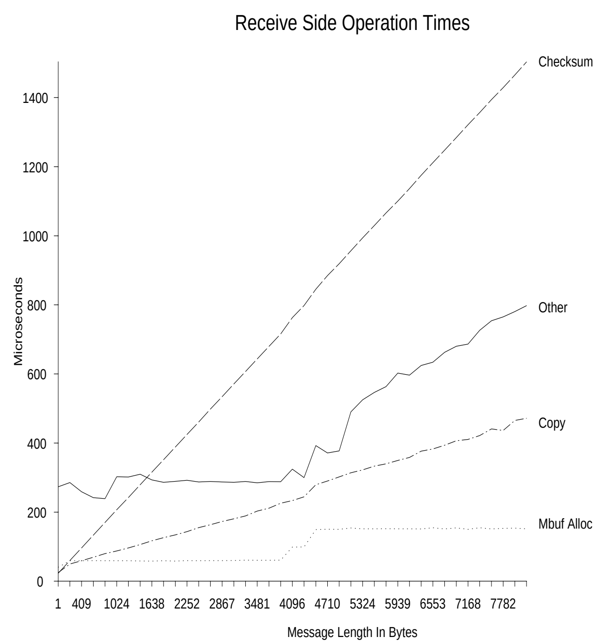

Figure 5

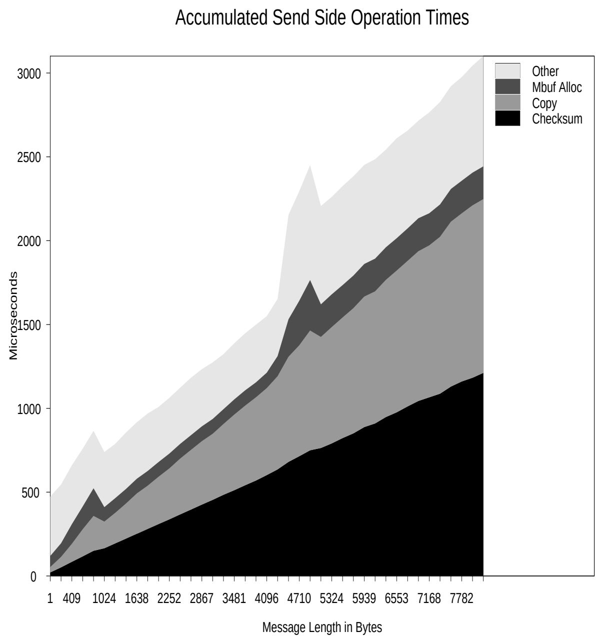

Figure 6

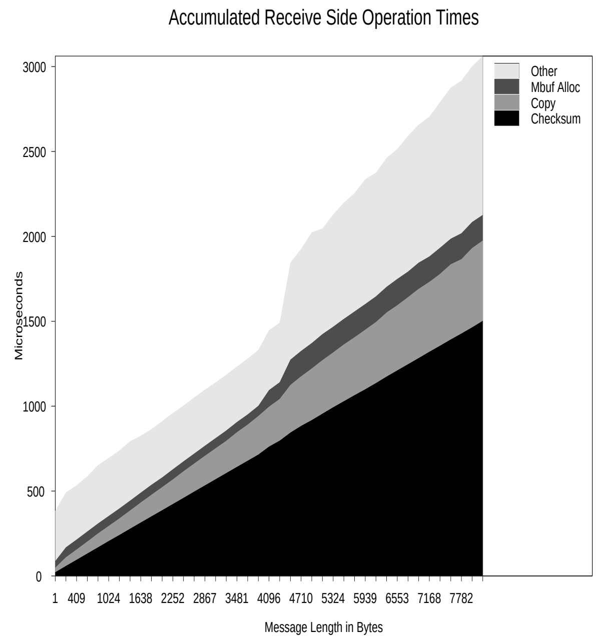

Figure 7

CHECKSUM is the dominant operation for large messages on both send and receive sides. The receive side costs rise more steeply with message size (0.180 usec/byte) than the send side costs (0.150 usec/byte). This is because messages being sent are already in the cache, whereas incoming messages are put into memory (and not in the cache) by DMA, and are first touched by the checksum code. Thus, the checksumming time shown includes the time of moving data from memory into the cache.

COPY time is much larger on the sending side than on the receiving side. Figures 8 and 9 contain breakdowns of COPY times on each side. Time spent copying between user and kernel space is approximately the same for each side, because in each case the message is already cached. However, on the sending side the dominant time is due to copying data to the DEFZA interface, since the DEFZA does not support DMA on the send side. On the other hand, the receive side must perform a cache clear in preparation for incoming data transferred by DMA (which is supported by the interface on the receive side). Send-side and receive-side times remain balanced largely because the time to copy to the device is offset by the receive-side checksumming time and the receive-side cache clear operation time.

The effects of the complicated mbuf MALLOC/FREE described above are easy to see here. Only the send side features the humps caused by the send-side mbuf allocation algorithm. On the receive side, the shape of the MALLOC/FREE curve is caused by the FDDI driver allocation policy. The FDDI driver allocates a single small mbuf for incoming frames less than 108 bytes in length, a cluster mbuf for frames between 108 and 4096 bytes, a cluster mbuf and a small mbuf for incoming frames between 4096 and 4204 bytes in length, and two cluster mbufs for larger frames. Since messages larger than the FDDI MTU size of 4324 bytes are fragmented into two FDDI frames, the algorithm repeats with additional mbufs for messages larger than 4324 bytes.

As in the combined case analyzed above, OTHER largely shows the same profile as MALLOC/FREE, but on a larger scale. When the receive-side times are isolated from the send-side times, it becomes possible to see that OTHER is climbing with message size above the FDDI MTU. Since we know little about OTHER, we cannot account for that phenomenon.

3.2 Analysis by Layer

Figure 10 shows the per-message combined send and receive processing times for various layers: SOCKET, UDP, IP, LINK, and DRIVER.

It is obvious that UDP is the most time-consuming layer. This is mainly due to its checksum computation; UDP is the only layer that computes a checksum for the data portion of its messages. The next most time-consuming layer is the DRIVER. This is largely due to its copy operation from the kernel to the controller, which is the most sig-

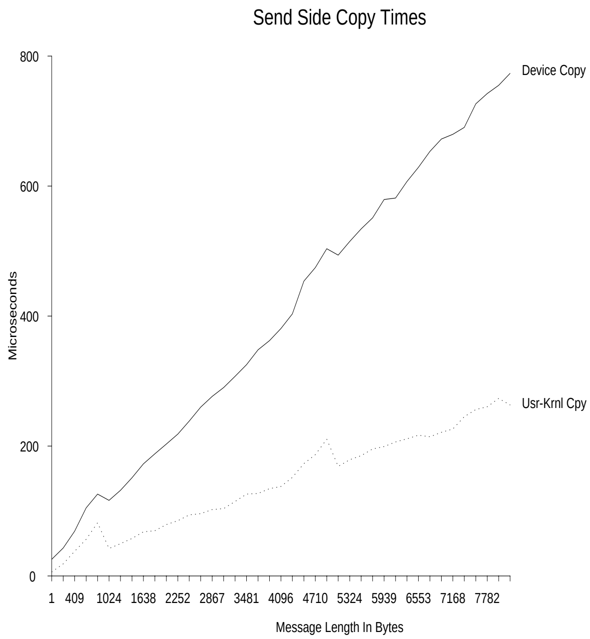

Figure 8

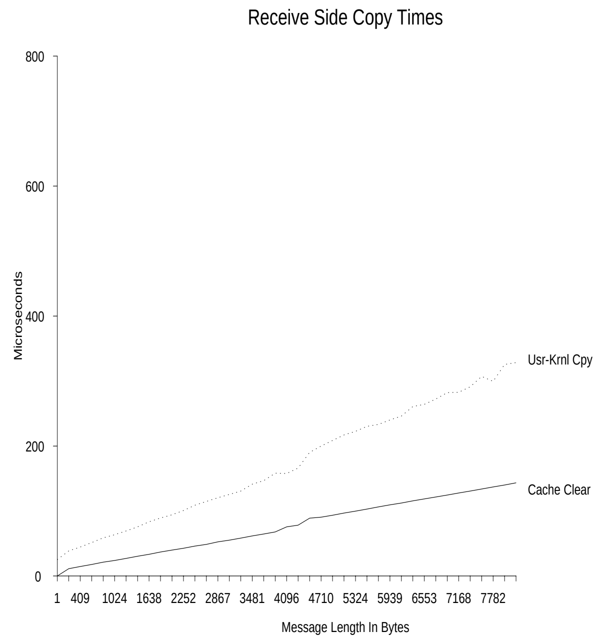

Figure 9

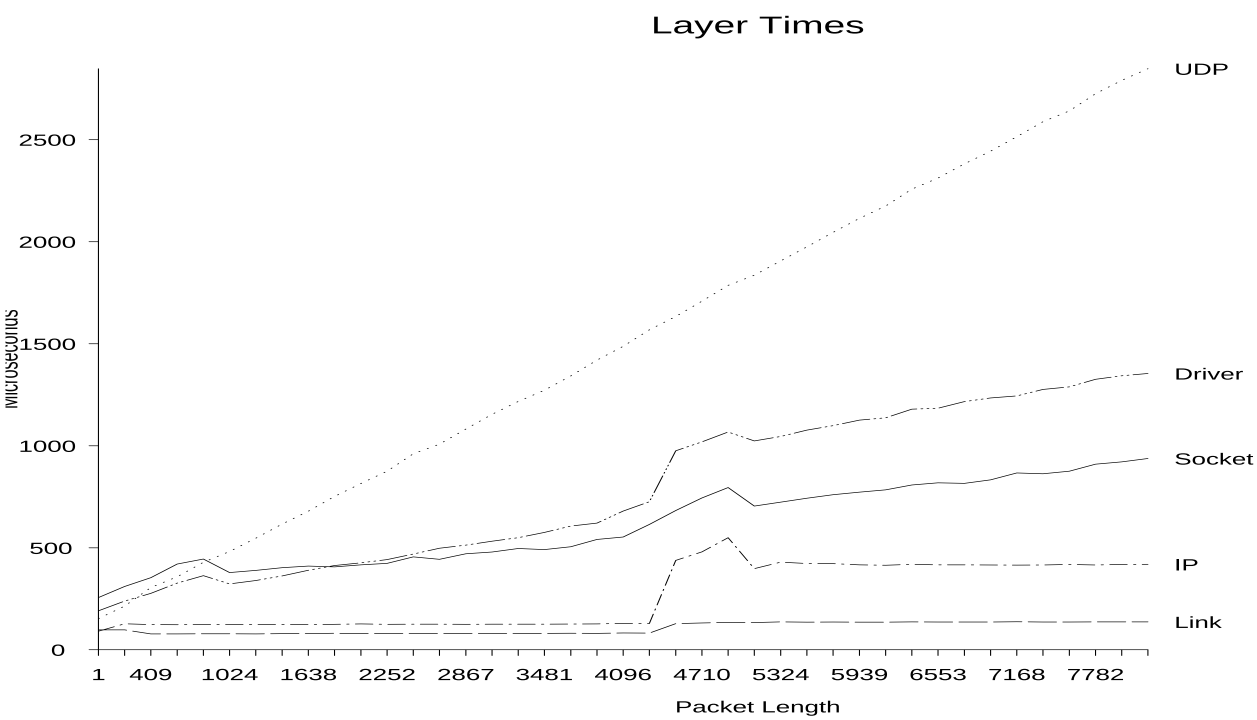

Figure 10

nificant of all the copy operations. The SOCKET layer processing time is due to copy operations between user and kernel memories, as well as OTHER operations. By contrast, the IP layer does not contain significant checksum or copy operations, and therefore its processing time does not increase linearly with message size like the previous layers. However, since fragmentation/defragmentation occur in the IP layer, we see a significant jump in processing time when a message requires more than one packet. Finally, the LINK layer refers to code lying between the IP and DRIVER layers. This includes data-link encapsulation code common to all Ethernet and FDDI devices, and software interrupt code. Like the IP layer, its processing time increases with number of fragments per message rather than message size. This is because the number of times it is called corresponds to the number of FDDI frames which must be sent or received. However, this time is relatively small compared to that of other layers.

4.0 Checksum Improvement

A goal of this study is to improve throughput. Since CHECKSUM dominates the processing time for large messages, such as those produced by network-I/O intensive applications, this is clearly an operation worth improving. Thus, we concentrated on reducing, and where it is reasonable to do so, eliminating, checksum computation time. The first step was to optimize the checksum code itself; we found that the checksum code used in DECstations and indeed most MIPS processor-based workstations was poorly tuned for RISC processors. Below we describe our improved checksum algorithm for RISC processors, which cuts the checksum computation time in half. Since checksumming still consumed a large portion of processing time, we developed a proposal for eliminating checksumming entirely over most messages.

4.1 A RISC Checksum Algorithm

The Internet Request for Comments describing the Internet checksum [2] includes a generic checksum algorithm written in ‘C’ and optimized algorithms written in assembly language for three different machine architectures: a CISC microprocessor (the Motorola 68020), a vector supercomputer (the Cray), and a CISC mainframe (the IBM 3090). None of the algorithms described are well suited to RISC processors. The 68020-based algorithm relies on extended integer arithmetic support not usually found in RISC processors: a carry bit and an instruction that adds the carry bit to two operands are required. The Cray algorithm uses vector operations, an even less common feature of RISC microprocessors. The IBM algorithm uses a number of branches which causes inefficient operation for pipelined processors such as RISC microprocessors. It is not surprising that the checksum routine provided by MIPS is effectively a hand translation of the ‘C’ algorithm into MIPS assembly language.

We were able to make a number of improvements to that algorithm. The most effective improvement resulted from reading memory in units of 32 bit words rather than in the units of 16 bit words used by the ‘C’ algorithm described in [2]. Reading a 32 bit word results in a single memory access, but requires two additional instructions (shift and mask) to unpack the desired two 16 bit quantities. As long as the ratio of memory access time to instruction execution time is more than 2 (generally the case for RISC workstations, and this ratio is expected to increase), a single memory access and two register instructions are executed faster than two direct 16 bit word memory accesses.

The next most useful technique is loop unrolling, used by the Motorola and IBM algorithms [2]. The inner loop of our checksum algorithm is unrolled sixteen times. We chose sixteen because it is the maximum number of unrolls that is both a power of two, and allows the expanded loop to efficiently operate on a 108-byte small mbuf. Both the Motorola and IBM algorithms unrolled their inner loops sixteen times as well.

Some smaller improvements are derived from consideration of pipelining effects. The MIPS processor uses load delay slots to permit the processor to accomplish useful work while bringing in memory contents to a register. Unfortunately, the assembler does a poor job of scheduling the instructions of the checksum computation into those delay slots, so we resorted to hand placement. The key principle is that a register load should begin right after its previous contents are used, and therefore, multiple registers are used to achieve pipelining [8]. We use two registers in our data pipeline which is enough to keep the processor busy. This technique is independent of whatever mechanism is used to wait for memory contents to arrive at the processor (e.g. load delay slots or scoreboarding).

With these modifications, the checksum computation time was reduced by more than a factor of two. Consequently, when sending large messages, we observed a 33% overall improvement in UDP/IP throughput.

Below is the code of the main loop of the improved checksum routine. For brevity, we leave out details such as setup, termination, and code for dealing with small or unaligned memory segments.

```
lw      t5,0(a0)        # load up pipeline
lw      t6,4(a0)        # with 32-bit words
                        # note: t5 and t6 alternate
```

```
1:                              # do {
      srl      v1,t5,16         # get upper half of 32-bit word
      addu     v0,v1            # add to running checksum
      addu     v1,t5,0xffff     # get lower half of 32-bit word
      addu     v0,v1            # add to running checksum
      lw       t5,8(a0)         # immediately start reloading t5

      srl      v1,t6,16         # get upper half of 32-bit word
      addu     v0,v1            # add to running checksum
      addu     v1,t6,0xffff     # get lower half of 32-bit word
      addu     v0,v1            # add to running checksum
      lw       t6,12(a0)        # immediately start reloading t6

      srl      v1,t5,16         # get upper half of 32-bit word
      addu     v0,v1            # add to running checksum
      addu     v1,t5,0xffff     # get lower half of 32-bit word
      addu     v0,v1            # add to running checksum
      lw       t5,16(a0)        # immediately start reloading t5

... 12 repetitions...

      srl      v1,t6,16         # get upper half of 32-bit word
      addu     v0,v1            # add to running checksum
      addu     v1,t6,0xffff     # get lower half of 32-bit word
      addu     v0,v1            # add to running checksum
      lw       t6,72(a0)        # immediately start reloading t6

      addu     a0,64
      subu     a1,64
      bge      a1,72,1b         # } while (len >= 72)
```

4.2 Checksum Elimination

Although we have made considerable gains by carefully optimizing the checksum algorithm, eliminating the checksum altogether would be twice as effective. In fact, options already exist in Berkeley Unix based networking code to completely turn off checksumming whenever interoperability would not be affected. However, the Internet checksum exists for a good reason; very simply, packets are occasionally corrupted during transmission, and the checksum is needed to detect corrupted data. Turning off checksumming by default is specifically forbidden within the Internet [1] .

4.2.1 Eliminating Checksum Redundancy

There is a certain amount of redundant checksumming in the system. Ethernet and FDDI networks implement their own 16-bit cyclic redundancy checksum. Thus, packets sent directly over an Ethernet or FDDI network are already protected from data corruption. It is obvious from this study that this redundancy is expensive. Thus, we suggest that in cases of complete redundancy, the Internet checksum not be performed.One must be careful, though, about deciding when the Internet checksum is in fact redundant. We believe that it is not unreasonable to turn off checksums when crossing a single network which implements its own checksum. Since the destinations of most TCP and UDP messages are within the same LAN on which they are sent, this policy would eliminate checksumming on most TCP and UDP messages.

Such a policy differs somewhat from traditional TCP design in one aspect of protection against corruption. Always performing a checksum in software in host memory protects against errors in data transfer over the I/O bus in addition to the protection between network interfaces given by the Ethernet and FDDI checksums. However, since data transfers over the I/O bus for such common devices such as disks are routinely assumed to be correct and are not checked in software, we conclude that such a reduction in protection against I/O bus transfer errors for network devices is not unreasonable, especially given the heavy cost of checksum computation.

However, turning off checksum protection in any wider area context seems unwise without considerable study. Not all networks are protected by checksums, and it is difficult to see how one might check that an entire routed path is protected by checksums without undue complications involving IP extensions. A more fundamental problem is that network checksums only protect a frame between network interfaces; errors may arise while a frame is in a gateway machine. Although such corruption is unlikely for a single machine, the chance of a packet not being corrupted decreases exponentially with the number of gateways a packet travels through.

In addition to disabling checksumming on local packets, we believe that it should be possible for applications to disable checksumming on any TCP connection on a per-connection basis. While disabling checksums is unwise for many applications such as NFS or FTP, many interactive audio and video applications may be tolerant of some corruptions because of the transient usage of the data.

4.2.2 UDP Implementation

The basic implementation in UDP is to use the existing checksum-nonuse protocol. In UDP, if the checksum field on an incoming message is zero, then it it is assumed that there is no checksum on the message. All that remains, then, is to decide on the details of when the sender should send a zeroed checksum field. Our implementation involves adding a flag to each network interface which supports some form of checksum. Under Berkeley Unix, routing information is already available to UDP, and the route structure includes a flag on whether the next hop in the route is just a gateway or is the destination host. The algorithm checks whether the next hop is a destination, and whether the network interface the packet will travel supports checksumming. Checksumming is skipped if both of those conditions are true. We also implemented a new socket option which lets the application turn off checksumming, even for sessions involving a large number of hops between hosts, useful for audio and video applications.

4.2.3 TCP Implementation

Extending TCP to implement checksum elimination is more difficult, since one is always supposed to checksum TCP data, and zero is specifically a valid checksum. Thus, some additional mechanism is required so that two implementations can agree on turning off checksums. An experimental Alternate Checksum Option already exists [11]. The Alternate Checksum Option provides a generic mechanism for TCP implementations to agree on use of an alternate checksum, as well as mechanism in case an alternate checksum requires more space than is provided by the TCP checksum field (e.g., a 32-bit or 64-bit checksum). Thus, it is straightforward to define an alternate checksum value specifying no data checksum.

Negotiation is started by the side that wishes to create a connection. TCP uses the same series of checks as in UDP to decide whether to send an option to turn off checksums. When server-side TCP responds to a TCP connection request, it will respond with the option if it gets a checksum-elimination request in the connection setup packet. Each side of a conversation only disables checksumming if it gets a checksum elimination option sent by the other host in the conversation.

As with UDP, we added a socket option that causes the implementation to try to negotiate turning off checksums. Notice that since the sender of the option makes the decision as to whether to negotiate checksums, and since receivers knowledgeable of this option automatically agree to any suggestion of turning off checksums, only one side of a conversation need call the option.

4.3 Performance

Table 1 summarizes the throughputs resulting from the improved checksum techniques. The numbers in MB (i.e. 1,000,000 bytes) are absolute throughputs of each configuration, and the percentage values in parenthesis are the percentage improvements relative to unmodified Ultrix 4.2a. The performance improvements are significant.

Table 1: Throughput improvements from checksum optimization and elimination

	TCP	UDP
Unmodified Ultrix 4.2a	2.1 MB/s	2.4 MB/s
Optimized Checksum	2.7 MB/s (29%)	3.2 MB/s (33%)
Checksum Elimination	3.1 MB/s (48%)	4.1 MB/s (71%)

5.0 A Faster FDDI Adapter

Earlier we noted that the FDDI adapter we are using to perform our experiments, the DEFZA, does not support outgoing DMA, so the processor must copy outgoing frames to the controller. Furthermore, we showed in Figure 8 that copying data out to the device (the DEVICE COPY operation) is time-consuming for large messages. DEC has recently developed a new FDDI controller for the DECstation, the DEFTA, which features send-side DMA, which we obtained after conducting the experiments already described. In running some tests using the new controller, we learned, unsurprisingly, that send-side performance using the new controller is much better than with the old controller. The difference is even more marked once one eliminates the overhead of checksumming. Table 2 lists send-side UDP/IP throughputs using the old and new controllers, without and with checksum elimination. Note that the throughputs given in Table 2 using the old adapter and unmodified kernel are higher than those in Table 1. This is because Table 2 describes only send-side performance, while the experimental configuration used to generate Table 1 involved the receive side which is often the bottleneck.

Table 2: Send-side throughput improvements from the improved adapter

	Old Adapter	New Adapter
Unmodified Ultrix 4.2a	2.6 MB/s	3.3 MB/s
Checksum Elimination	4.1 MB/s	6.3 MB/s

Notice that our checksum elimination technique results in a 58% improvement in throughput (4.1 vs. 2.6 MB/s) using the old adapter, and a 91% improvment (6.3 vs. 3.3 MB/s) using the new adapter. This is because the new adapter increases the speed of data transfer from memory to the network device (due to DMA, which can operate concurrently with other network processing), making the checksum computation time, and therefore its elimination, a more prominent factor.

6.0 Conclusions

We present measurements of DECstation UDP/IP network software, analyzed by operation and by layer. The main time consuming operations, particularly for large messages, are the checksum computation and, to a lesser degree, data copying. The main time-consuming layer is the UDP transport layer, since it contains the checksum computation. We improved the checksum implementation, and suggested a way of eliminating checksum computations which could be considered redundant. Both of these improvements resulted in a noticeable increase in throughput performance.

7.0 Acknowledgements

The section on improved adapter performance was made possible by Fred Templin's tenacity in getting us the new FDDI adapter and a driver for it. Keith Muller was of great assistance in setting up the logic analyzer and designing probe software. Kevin Fall aided in bulletproofing the checksum elimination algorithm.

8.0 Availability

Improved checksum software is available in the form of diffs against Ultrix 4.2a. It can be found on the ftp server *ucsd.edu*.

9.0 References

[1] R. T. Braden, "Requirements for Internet Hosts -- Communication Layers," Internet Request for Comments 1122, October 1989.

[2] R. T. Braden, D. A. Borman, and C. Partridge, "Computing the Internet Checksum," Internet Request for Comments 1071, September 1988.

[3] L.-F. Cabrera, E. Hunter, M. J. Karels, D. A. Mosher, "User-Process Communication Performance in Networks of Computers," *IEEE Transactions on Software Engineering*, 14(1), 38-53, Jan. 1988.

[4] D. D. Clark, "Modularity and Efficiency in Protocol Implementation", Internet RFC, 817, 1982

[5] D. D. Clark, V. Jacobson, J. Romkey, H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, 23-29, June 1989.

[6] D. Ferrari, J. Pasquale, and G. Polyzos, "Network Issues for Sequoia 2000," Proceedings IEEE COMPCON, 401-406, February 1992.

[7] V. Jacobson, "BSD TCP Ethernet Throughput", comp.protocols.tcp-ip, Usenet, 1988.

[8] J. L. Hennessy and D. A. Patterson, **Computer Architecture: A Quantitative Approach**. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[9] M. Stonebraker and J. Dozier, "Sequoia 2000: Large Capacity Object Servers to Support Global Change Research," Sequoia 2000 Technical Report #1, U.C. Berkeley, July 1991.

[10] R. W. Watson, S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems*, 5(2), 97-120, May 1987.

[11] J. Zweig and C. Partridge, "TCP Alternate Checksum Options," Internet Request for Comments 1146, March 1990.,

10.0 Biographies

Jonathan Kay is a Ph.D. student in computer science at the University of California, San Diego. He received both B.S. and M.S. in Computer Science from the Johns Hopkins University in 1989 and 1990, respectively, and hopes to get his Ph.D from UCSD sometime this decade. As you might expect from this paper, he is interested in TCP/IP performance improvements, gigabit networking, network support for multimedia, and data movement issues in general. He works in the UCSD Computer Systems Lab, and is currently involved in making the Sequoia 2000 network faster.

Joseph Pasquale is Assistant Professor of Computer Science and Engineering at the University of California, San Diego, and Senior Fellow at the San Diego Supercomputer Center. He received the BS and MS from MIT in 1982, and the PhD from UC Berkeley in 1988. He is a recipient of the NSF PYI Award in 1989 and an IBM Faculty Development Award in 1991. His current research interests are in the areas of operating systems and networks, particularly the design, implementation, and performance evaluation of I/O system and network software to support distributed multimedia (especially digital video and audio) applications and I/O-intensive scientific applications. Dr. Pasquale co-directs the UCSD Computer Systems Lab, and he is currently involved in building the Sequoia 2000 Network.