A Faster UDP

Craig Partridge and Stephen Pink*

Swedish Institute of Computer Science

Abstract

As an experiment in protocol optimizations, the authors undertook to improve the performance of a stateless protocol, the User Datagram Protocol (UDP) in the 4.3 BSD UNIX† kernel. This paper describes the successful optimizations that were done along with measurements that show a UDP performance improvement of between 25% and 35% on CISC and RISC systems, and overall kernel improvement of between 12% and 18%.

1. Introduction

Recent years have seen an impressive series of algorithms and implementation techniques that have sharply improved software performance of implementations of the Transmission Control Protocol (TCP). However, when we started the work described in this paper in 1991, none of these algorithms and techniques had been applied to User Datagram Protocol (UDP).

Applying the optimizations to UDP is interesting for two reasons. First, unlike TCP, UDP is stateless. Defined in RFC-768 [1], UDP simply adds transport level addressing and an optional checksum to the Internet Protocol (IP) service of best-effort datagram delivery. Some of work on optimizing protocols has suggested that more stateful protocols can achieve better performance. For example, Srinivasan and Mogul showed that by adding state to a Network File System (NFS) implementation they could achieve a modest performance improvement [2]. Jacobson's Header prediction algorithm [3] uses information that TCP must store about the connection state to predict the next segment that will arrive and optimize the processing of the expected segment. One interesting research question is whether the optimizations that work for stateful protocols like TCP also work for stateless protocols like UDP.

Second, UDP is very heavily used in the Internet protocol suite. Applications that use UDP include the Domain Name System (DNS) [4], the NFS [5], and the Network Time Protocol [6]. Any information about how to improve UDP performance is therefore likely to pay in improved performance for several important applications.

To experiment with UDP performance, we applied various optimizations to the implementation of UDP in the 4.3BSD Tahoe release, as ported to the Mach 2.5 operating system running on a 386 processor. We then applied these optimizations to the UDP implementation in SUN OS release 4.1.1 running on a Sparc processor. These two systems allowed us to test the impact of the optimizations on both CISC (386) and RISC (Sparc) processors.

2. An Overview of TCP Performance Optimizations

The first problem in applying the TCP performance optimizations to UDP is determining which optimizations are TCP-specific and which optimizations could fruitfully be applied to UDP. This section briefly surveys the work on TCP performance optimization (including work done concurrently with the work described in this paper) and discusses which optimizations appear applicable to UDP.

---

* The work described in this paper was done while Dr. Partridge was on sabbatical from his regular employer, Bolt Beranek and Newman. † UNIX is a trademark of Bell Laboratories.

From *IEEE/ACM Trans. on Networking* (August 1993)

The cost of processing a TCP segment has two parts: (1) a per-segment overhead which does not depend on the size of the data in the segment; and (2) costs incurred in handling the data in the segment on transmission and reception.

In 1989, Clark, Romkey, Salwen and Jacobson examined a common TCP implementation and found the per-segment overhead was about 200 instructions for both the sending and receiving TCP, but noted that Jacobson had an experimental implementation that reduced the receiving overhead [7]. In subsequent lectures, Jacobson has explained the techniques he used to reduce the per-segment overhead. First, he observed that, in most circumstances, TCP segments arrive in order and do not require special handling (e.g., they have no out of band data). Based on this observation, he developed a technique known as as header prediction, which, on TCP input, tests to see if the inbound segment is the expected segment and, if so, uses an optimized processing path of less than a dozen lines of code [3]. Jacobson also observed that TCP traffic exhibits considerable locality. The next TCP segment received is highly likely to be destined for the same application as the last segment. Caching information about the last application (in the form of a protocol control block or PCB) that received a segment often makes it possible to skip doing an expensive search or PCB lookup to find out who is to receive the segment. A subsequent study by Mogul confirmed that there is substantial locality in TCP and UDP traffic [8]. Furthermore, McKenney and Dove have done an extensive study of algorithms for PCB lookups [9].

The reductions in TCP overhead have made data handling the major cost in TCP processing, particularly for larger packets. (See, for example, the data in [10]). At minimum, the data in each TCP segment must be copied between application buffers and the network interface (or vice-versa) and must be checksummed. However, most implementations do several copies. In 1990, Clark challenged protocol implementers to reduce the data copies in their code [11], and Jacobson suggested that building memory-mapped network interfaces would make it possible to reduce the number of data copies to a single copy [12]. Jacobson further suggested that on RISC processors it might be possible to effectively eliminate the cost of doing the TCP checksum by fitting the checksum into the code that copied data between application buffers and the network interface buffers. Very recently, work by a team at Hewlett-Packard's Bristol Laboratory has demonstrated that an interface based on Jacobson's ideas can achieve extraordinary TCP performance [13,14].

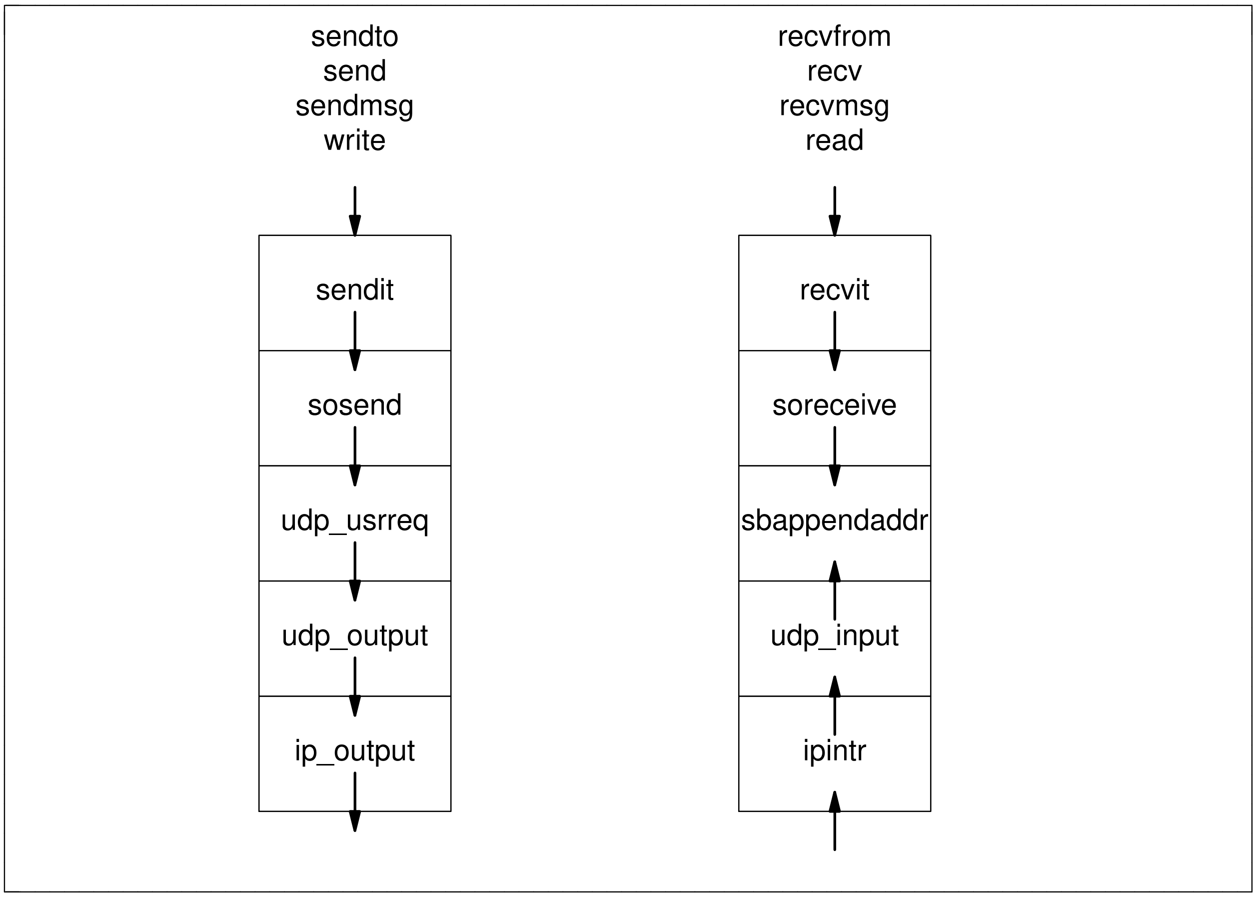
Examining these TCP optimizations from the perspective of trying to apply them to UDP, we concluded that some optimizations were clearly more promising than others.

Header prediction, which tries to predict the next segment, was clearly not relevant to UDP. Each UDP datagram stands alone, independent of the ones before or after it. But the general principle of locality seemed likely to have some relevance. Indeed, Mogul's work showed that UDP demonstrated locality similar to that of TCP. In the optimizations described below, we successfully made use of locality to improve UDP performance.

Regarding data related costs, the cost of a data copy would appear to be largely independent of the protocol used to encapsulate the data. So reducing data copies appeared likely to yield UDP improvements similar to those reported for TCP. Rolling the checksum into the copy code was a problematic performance improvement, since UDP permits applications to turn off the checksum if they wish. In our work, we experimented with rolling the checksum and copy loops together. At the end of this paper some of the issues related to the checksum are discussed.

3. A Sketch of the 4.3 Tahoe UDP Code

We started by optimizing the 4.3 BSD Tahoe release code, as ported to the Mach 2.5 operating system. We later applied the optimizations to SUN OS release 4.1.1. The differences between the original Tahoe release and the Mach release are sufficiently minor that they need not be discussed. What follows is a sketch of the Tahoe UDP code. The important differences between the Tahoe code and SUN code are described at the end of this section. Descriptions of special features of the BSD code, which were not essential to the optimization, have been left out. See [15] for a more detailed discussion.



3.1. Sending Code

An application that wishes to send a UDP datagram can use of one of four system calls:

- sendto(s,msg,msglen,flags,to,tolen). Sendto sends the given message (msg) to the specified destination address (to) on a pre-allocated UDP socket (s).
- send(s,msg,msglen,flags). In the BSD system, it is possible to fix the remote address to which all UDP datagrams will be sent, using the connect system call. For UDP sockets which have been connected, the send call is simply a variant of the sendto call, without the destination parameters.
- sendmsg(s,msg_struct,flags). Sendmsg is another variant of the sendto call, which provides a scatter-gather IO interface. The msg_struct parameter contains an array specifying where the various pieces of the message lie in the application's memory. The kernel gathers the various pieces into a single message and sends it. Sendmsg can be used with either connected or unconnected sockets. When sending over an unconnected socket, the destination address also must be placed in msg_struct.
- write(s,msg,msglen). Write is the standard UNIX® system call for writing to a file or socket and, for sockets, behaves just like send.

Once in the kernel, all four system calls package their arguments into a standard form and call a common sending routine, sendit. Sendit confirms that the arguments are valid (for example, confirming that the memory addresses to copy are within the application's memory space), and then calls sosend.

Sosend is the start of real protocol processing. Sosend copies the user data into a buffer chain in kernel memory and passes the data down to the appropriate transport protocol. If the transport protocol is not ready to accept the data due to flow control, sosend suspends, waiting for space to become available. Note that for a protocol like UDP, which has no flow control, all sosend does is copy the user data into the kernel and call the transport protocol.

Sosend calls the transport protocol using a protocol switch table. Each transport protocol has a protocol switch entry which includes basic information about the protocol (such as whether a connection must be established before sending), and definitions for nine protocol entry points. All communication with the transport protocol is made by calling one of the nine entry points.

Sosend always calls the usrreq entry point, the entry point that handles most application system calls (''user requests''), with the instruction to send a certain amount of data (the data is passed as an argument). Internally, a protocol's usrreq routine typically does some very basic parameter checking and then calls an appropriate subroutine to handle the system call. For UDP, the routine to handle sending UDP datagrams is udp_output.

Udp_output is passed a socket-specific protocol control block (PCB), a buffer chain containing the data to be sent, and the destination address. The protocol control block contains the UDP-specific information for the socket on which the data is being sent. Udp_output prepends the UDP header and a template of the IP header, performs the UDP checksum, and calls IP, via ip_output to send the datagram.

Ip_output fills in the IP header left unfilled by udp_output, looks up a route for the datagram, fragments the IP datagram if it is too large for the outbound interface, checksums the IP header, and calls the appropriate interface device driver to send the IP datagram.

3.2. Receiving Code

The receiving code has a somewhat simpler structure than the sending code.

Upon receiving an IP datagram, a network interface places the datagram on the IP input queue and sets a software interrupt for the IP input routine, ipintr. When interrupted, ipintr removes datagrams from the input queue, checks the IP header and datagram length for validity, and, if the datagram is destined for one of the host's addresses, does IP reassembly (if required), and passes the IP datagram up to the transport protocol, using the pr_input routine in the transport protocol's protocol switch entry.

UDP's input routine is udp_input. Udp_input checks the UDP header for errors and if the datagram is valid, uses the UDP header and the IP header information to locate the appropriate socket to receive the UDP data. If a socket exists to receive the data, udp_input calls sbappendaddr (socket buffer append, with address) to append the data in the UDP datagram, plus the source address of the application that sent the datagram, onto the socket's read buffer.

Applications read from a socket's read buffer using one of four system calls:

- recvfrom(s,buf,buflen,flags,from,fromlen). Recvfrom reads from the given socket into the buffer (buf) until either the buffer has been filled, or an entire datagram has been read. If a partial datagram is read, the kernel discards the remainder of the datagram. The address of the sending application is placed in the from structure.
- recv(s,buf,len,flags). Recv is a variant of recvfrom in which the the address of the sending application is not returned.
- recvmsg(s,msg_struct,flags). Recvmsg is the scatter-gather version of recvfrom. The message structure is the same as the one used for sendmsg.

- read(s,buf,len). Read is the standard UNIX® system call for reading from a file or socket. Like recv it does not return the address of the sending application.

Each of these routines calls a common routine, recvit which checks the validity of the applications buffers and then calls soreceive to actually copy the UDP data from the inbound socket queue into the application's space. If the application tried to read from a socket which had no data waiting, soreceive will block, waiting for data.

3.3. Differences in SUN 4.1.1

The networking portion of the SUN 4.1.1 Operating System has almost the same structure as the BSD code. There were only two differences of note.

First, the SUN code has a special version of the UDP sending code for use by the kernel resident part of the Network File System (NFS). This code has been optimized to make the types of UDP calls made by NFS slightly faster. Some of these optimizations turned out to be similar to those we made to the UDP code in general and where we concluded the results were close enough as to make no difference, we left the SUN code as it was.

Second, the SUN code normally does not checksum UDP datagrams when sending. However, many system managers insist on turning the checksum on to better protect against corrupted data packets in applications like the NFS. Furthermore, since we were interested in studying combined checksum and copy loops, it made sense to start with software that did the checksum. So we changed the code to always perform the checksum.

4. Experimental Approach

We chose to try to optimize UDP while retaining the basic structure of the BSD networking code. We took this approach for two reasons. Pragmatically, substantial modifications to the BSD structure would force us to change code for protocols other than UDP and we needed to bound our effort. From an experimental perspective, large scale changes to the code would make it more difficult to identify the effects of particular optimizations.

A more radical restructuring of the code has been done at Univ. California Berkeley for the 4.4 BSD release. As part of the Berkeley restructuring, some optimizations were made to the UDP code by Van Jacobson and Mike Karels. Many of their optimizations are similar to ours. (We were not aware of each other's work until both projects were largely done). Where the Berkeley work is particularly insightful or different from our work, we occasionally comment on the 4.4 BSD work in footnotes.

4.1. Optimizing with Gprof

The initial step in the process was to profile the kernel to get a sense of where the protocol bottlenecks appeared to be. We used the gprof profiling application that both tracks the amount of time spent in each routine and also produces call graphs which show which routines called other routines and how much time in each routine is the result of a call from a particular parent routine.

Gprof displays its results in two forms. First, it produces a chart displaying the call graph. Functions are listed in order of how much time they took (including calls to subroutines), along with a list of the routines that called them and the subroutines called. Second, gprof lists the total time spent in each function.

In general, gprof is a useful tool for profiling systems. It does, however, have at least one serious flaw: it assumes that all calls to a function take the same time. Thus when it builds the call graphs, the percentage of time spent in a subroutine is simply estimated by dividing the total time spent in the subroutine by the number of times the subroutine was called by the parent function. This problem is well-documented in the gprof manual pages, however we did forget it more than once in the heat of analysis (sometimes with beneficial results – see the discussion of in-line IP checksums below).

4.2. Test Cases

To generate profiles, we used a test application that sent UDP datagrams to the discard server [16] on the same machine over a software loopback using the sendto system call. (The discard server simply throws away any data sent to it). Profiles over the loopback showed both sending and receiving costs. The two basic tests were to send 200,000 datagrams, each containing 64-bytes of data, and 200,000 datagrams containing 512-bytes of data. The two datagram sizes were chosen to represent common transfer extremes: 64-bytes for small datagrams such as those used for RPC calls and 512-bytes to represent larger datagrams such as those generated by distributed file systems when they transfer file blocks. Because 64 and 512 are both powers of two, and may therefore benefit from various memory handling routines that might favor such sizes, we also tested our slowest and fastest implementations with data sizes of 59 and 509 bytes.

Initial experimentation with the test application showed that there were two minor problems (from an experimenter's point of view) with the standard discard daemon supplied with both the Mach and the SUN code. First, the receiver buffer space was not large enough. Some inbound UDP datagrams were being dropped without being fully processed. Second, the discard program was implemented as part of a larger program that provided several services at once. To figure out which service had data waiting, the application used the select system call to find out which sockets had received data. The result was that a considerable amount of time was spent in the select system call, which was not of interest. To avoid these issues, the standard discard server was replaced with a version that allocated extra receiver space and did not call select.

4.3. General Approach

The optimizations were done in two stages. In the first stage, we looked for ways to reduce the per-packet overhead. These changes were generally simple to make, a matter of changing a few lines of code. In the second stage we tried to improve data handling (in particular, combine checksum and copy loops). Changing the data handling code required considerably more effort, although the payoffs proved dramatic.

Table 1a: Initial Performance (386)

	64-byte datagrams		512-byte datagrams	
Code Section	**Time (seconds)**	**% of Total**	**Time (seconds)**	**% of Total**
sosend	25.94	2.7%	26.33	2.6%
udp_usrreq & output	29.17	4.3%	32.17	3.2%
ip_output	14.46	1.5%	13.93	1.4%
ipintr	18.99	2.0%	18.49	1.8%
udp_input	32.29	3.4%	34.99	3.4%
in_cksum	47.64	5.0%	98.82	9.7%
soreceive	28.88	3.1%	28.73	2.8%
Total UDP/IP/Socket	193.37	20.6%	253.45	24.9%
Total Kernel Time	936.03	100.0%	1018.36	100.0%

5. Initial System

The initial profile of both systems is shown in Tables 1a and 1b below. For each major routine, the table shows the total number of seconds spent in that routine and its local subroutines. (By local subroutines, we mean those routines called by the listed procedures which are not themselves included in the table). The listing for the ip_output routine does not include the cost of calls to the loopback interface's output routine.

Table 1b: Initial Performance (Sparc)				
	64-byte datagrams		512-byte datagrams	
Code Section	Time (seconds)	% of Total	Time (seconds)	% of Total
sosend	10.48	5.8%	12.02	5.7%
udp_usrreq & output	23.24	12.8	20.65	9.8%
ip_output	6.56	3.6%	6.35	2.0%
ipintr	5.30	2.9%	6.38	3.0%
udp_input	15.37	8.5%	16.83	8.0%
in_cksum	12.92	7.1%	17.84	8.4%
soreceive[1]	9.24	5.1%	11.41	5.4%
Total UDP/IP/Socket	83.11	45.8%	91.48	43.3%
Total Kernel Time	181.42	100.0%	211.33	100.0%

Probably the most significant observation that can be made from these tables is that the total time spent in the socket and UDP and IP code is rather small, even though the test systems were just running the UDP test applications. On the 386 processor, gprof revealed that a *majority* of the kernel's time was spent simply handling interrupts and managing changes in processor priority to protect critical code regions. On the Sparc, less than 50% of the time in the kernel is actually in UDP or IP code. It is interesting to observe that the time spent executing UDP/IP and socket code in both 386 and Sparc is only a factor of two apart, while the overall kernel times differ by a factor of four. A major difference between the processors is in how they handle overhead activities.

6. Reducing Per-Packet Overhead

The first step was to try to reduce the per-packet overhead. The improvements involved either exploiting locality, or replacing excessively general purpose code with simpler and faster special purpose code.

6.1. In-Line Checksum of IP Header

Although Table 1 does not show it, the initial profile showed that a surprisingly large amount of time on the 386 processor was spent calling the IP checksum routine from ipintr and ip_output to checksum the 20-byte IP header. The call trace showed that about 25% of the time in ip_input and over 50% of the time in ip_output were actually spent in the checksum routine. In reality, this was a case of gprof amortizing the cost of calling the checksum routine over all calls and averaging the cost of doing IP header checksums with the cost of doing checksums over entire datagrams. So the header checksum costs were somewhat lower than gprof suggested. But the observation still lead to a potential optimization.

A quick investigation of the checksum routine showed that it had been optimized for checksumming long packets. Clearly this was not optimal for the short IP header checksum (which must be done on every IP datagram). There was a choice of approaches: the checksum routine could be reoptimized, or, given that the IP header checksum was almost always done on a 20-byte header (adding options to IP datagrams is not common), the IP header checksum could be put in-line.

[1] Time spent in soreceive does not include time spent in *sbwait*, waiting for packets to arrive.

For the immediate problem of the IP header checksum, the in-line option seemed best. Most systems can checksum 20 bytes in about 6 to 8 assembler instructions, which is less than the cost of making a procedure call (much less doing anything in the procedure). So we installed an in-line checksum. The code implemented a portable version of the checksum in both ipintr and ip_output which required about 20 instructions to compute the checksum for 20-byte headers. Headers larger than 20-bytes (i.e. IP headers with options) were still checksummed by the regular checksum routine.[2] The 386 results are shown in Table 2a.

Table 2a – In-Line IP Header Checksum (386)

Code Section	64-byte datagrams old time	64-byte datagrams new time	512-byte datagrams old time	512-byte datagrams new time
in_cksum	47.64	30.29	98.82	85.46
ip_output	14.46	16.27	13.93	15.60
ipintr	18.99	21.26	18.49	20.93
Time Improvement	13.27		9.25	
% UDP/IP Improvement	7.9%		4.0%	

The results appear to confirm the utility of the in-line checksum. The cost of doing the in-line checksum caused a modest increase in the cost of ip_output and ipintr but produced a sharp reduction in checksumming costs.

When the SUN code was examined it proved to already have an in-line IP checksum implemented in ipintr and in the NFS version of ip_output. However, the regular ip_output routine did not have an in-line checksum, so we added it. Table 2b shows the improvement.

Table 2b – In-Line IP Header Checksum (Sparc)

Code Section	64-byte datagrams old time	64-byte datagrams new time	512-byte datagrams old time	512-byte datagrams new time
in_cksum	12.92	10.05	17.84	17.20
ip_output	6.56	4.94	6.35	5.49
Time Improvement	4.49		1.50	
% UDP/IP Improvement	4.4%		1.4%	

One oddity of both Tables 2a and 2b is that larger datagrams benefit less in absolute time from the elimination of the call to the checksum routine. On the 386, large packets save a little over 9 seconds, while small packets save over 13 seconds. Similarly on the Sparc, large packets save only 1.5 seconds while small packets save nearly 4.5 seconds. This difference is consistent across several profiles. The new code contains no dependency on the length of the packet, so we believe (but have been unable to prove) that this difference represents a kernel effect probably related to interactions between interrupts (which are more likely to interrupt processing on large packets) and instruction and data cache management.

6.2. Deleting Pseudo-Connect

In the BSD system, UDP datagrams can be sent in one of two ways. A socket can fix its remote UDP address by using the connect system call, after which the socket can only be used for sending to the address it is connected to. Or, a socket can use sendto to individually set the remote address for each UDP datagram sent.

[2] No IP headers with options were sent by the test application.

In the 4.3 BSD code, using sendto with an address is treated as a special case of sending over a connected socket. The code in udp_output goes through the standard procedure for internally connecting a remote address to the UDP socket's protocol control block (PCB), sends the datagram, and then disconnects the remote address.

No packets go over the network for the connect procedure, but the operations of connecting an address are still very expensive and consume nearly a third of the cost of each UDP transmission. There are several reasons for this expense.

First, binding creates a race condition: during the time the remote address is set in the PCB, the socket cannot receive inbound datagrams from any system other than the one it is sending to. This race condition occurs because on the inbound side, udp_input scans the list of UDP PCBs looking for a PCB that matches the source and destination address(es) of the inbound datagram. There is a chance that if an inbound datagram arrives while a socket has temporarily bound its destination address during a sendto call, a mistaken match (or failure to match) may occur. To avoid this race condition, udp_usrreq must switch to a higher processor priority level to block out input interrupts. On the 386 changing the processor priority level is expensive.

Second, the routine for binding to a remote address is a general purpose routine, intended to be called at connection setup time, and is not optimized to be called for every datagram. Calling it repeatedly is expensive.

To fix this problem, udp_output was revised to send to a specified remote address without changing the socket PCB. This eliminated the race condition and got rid of the call to the general purpose connecting routine, although some code from this routine had to be copied into udp_output.[3]

The performance improvements are in the sending side and in the priority management code, and are shown in Tables 3a and 3b. The improvement in performance is significant.

Table 3a– Deleting Pseudo-Connect (386)

	64-byte		512-byte	
Code Section	**Time**	**% of Total**	**Time**	**% of Total**
Priority Mgmt/Interrupts (old)	543.32	58%	572.14	55.7%
Priority Mgmt/Interrupts (new)	541.30	60.2%	550.55	56.1%
udp_usrreq & udp_output (old)	29.17	4.3%	31.17	3.1%
udp_usrreq & udp_output (new)	17.95	1.9%	17.87	1.8%
Total UDP/IP/Socket Improvement[4]	11.22	6.6%	13.30	5.5%

Table 3b– Deleting Pseudo-Connect (Sparc)

	64-byte		512-byte	
Code Section	**Time**	**% of Total**	**Time**	**% of Total**
Priority Mgmt (old)[5]	1.53	0.8%	2.07	1.0%
Priority Mgmt (new)	1.58	0.9%	2.15	1.0%
udp_usrreq & udp_output (old)	23.24	12.4%	20.65	9.6%
udp_usrreq & udp_output (new)	9.22	5.1%	12.78	6.0%
Total UDP/IP/Socket Improvement	14.02	7.5%	7.87	3.7%

[3] The 4.4 BSD implementation has gone further and actually restructured the UDP data structures so that a connected socket is an exceptional case of a unconnected socket rather than the other way around.

[4] Improvement percentages are measured against the time spent in UDP/IP code in the original code.

[5] For the Sparc profile, it was possible to partially isolate the calls to particular spl routines by the UDP

6.3. Improved One-Behind Cache(s)

A *one-behind cache* contains the PCB of the last UDP socket to receive a datagram. The reason for keeping the one-behind cache is that there is a good chance that the next datagram received will be destined for the same socket as the previous datagram received. Keeping the cache makes it possible to avoid a more expensive search of all the UDP PCBs.

6.3.1. Fixing the Reno Cache: Wildcard Support

On the 4.3BSD Tahoe release, UDP does not have a one-behind cache of the UDP PCBs. However, the 4.3 BSD Reno release does and we incorporated the Reno cache into our UDP code.

But when we tested the cache performance, the Reno cache had no effect. The problem turned out to be that the caching code did not support wildcarding. Wildcarding is the practice of specifying only part of the remote or local address and accepting any value for the remaining fields. In the BSD code, UDP addresses can be classified into three types:

(1) [<laddr,lport><faddr,fport>]

(2) [<laddr,lport><*,*>]

(3) [<*,lport><*,*>]

where [lf]port is the local or foreign UDP port, [lf]addr is the local or foreign IP address, and * is a wildcard (i.e. ‘‘don’t care’’). When an inbound UDP datagram is received, the UDP code looks for the most complete (least wildcarded) address match it can find and delivers the data to that socket.

One problem with wildcarding is that it makes caching difficult. Consider for example, a datagram that arrives and matches an address of type (3). Now suppose that the next arriving datagram would match both the same type (3) address, but also a more complete address of type (1). If the PCB of the first datagram is placed in a cache, the second datagram will make a false cache hit. The cache hit is false because while the second datagram matches the type (3) address, it should be delivered to the type (1) address it matches more completely. We call this problem *cache hiding*, because the type (3) address in the cache hides the existence of the preferred type (1) address.

Unfortunately, wildcarding is very common in BSD applications. Most UDP servers do not fix their remote address or local IP address, but simply accept all inbound UDP datagrams sent to their reserved UDP port. By simply binding to the local port, but not an address, the servers on multi-homed systems (systems with more than one IP address) are able to serve datagrams regardless of the interface they arrived on. The Reno cache will not cache wildcard addresses (to avoid cache hiding) and therefore fails to have an effect on most applications (including our tests).

We enhanced the Reno cache code to allow the cache to contain wildcard addresses. To solve the problem of cache hiding, the routines that add and remove addresses from a PCB were modified to detect potential cases of cache hiding and to mark as uncacheable those PCBs that could cause cache hiding.

6.3.2. Mogul’s Suggestion: Caching the Sender’s PCB

Mogul [8] has recently done a study of the locality of network traffic that shows that half of all UDP datagrams received are replies to the last UDP datagram sent. This suggests that a one-behind cache of the *sending* PCB might have an effect.

---

code, so only this code is measured (splnet, spl3 and splx called by UDP output routines).

As with the one-behind receiving cache, wildcarding is a concern, but the same marking algorithm works. Sending PCBs that would cause *cache hiding* are not cached.

6.3.3. SUN Caching and Wildcards

SUN OS 4.1.1 turned out to have a broken one-behind cache of receiving PCBs which could cause data to be misdelivered to the wrong application, due to a failure to detect cache hiding. We replaced the SUN cache code with ours.

6.3.4. Caching Improvements

Testing the effects of the caching improvements was difficult. Running the tests on a ‘‘quiet’’ machine (a machine receiving no traffic other than our tests) would give reproduceable results. But a quiet machine would not give much indication of how a real workload would interact with the caching algorithms. Running on an active machine, however, meant the results might not be reproduceable.

In the end we compromised. The 386 test machine was already an isolated machine, so it was tested using the standard program sending 200,000 datagrams. These results are shown in Table 4a.

Table 4a – Improved Caches (386)

	64-byte		512-byte	
Code Section	**Time**	**% of Total**	**Time**	**% of Total**
udp_input (old)	32.29	3.4%	34.99	3.4%
udp_input (new)	24.03	2.7%	23.63	2.4%
Total UDP/IP/Socket Improvement	8.26	4.8%	11.36	4.7%

It should be noted that these results are probably still misleading, for two conflicting reasons. First, the simple test application achieved a 100% cache hit rate. Second the test system was running a minimum of user applications, so the UDP PCB cache was somewhat smaller than normal. Fewer applications makes a normal PCB lookup less expensive and tends to understate the benefits of caching.

In the second test, the caching code was run on a SUN workstation that served both as a client and a file server and profiled the code with the original SUN cache, the improved cache code and with the cache turned off. By picking a workstation that did some work as both file server and client we hoped to get a reasonable mix of traffic. Furthermore, the SUN had a more normal number of active UDP applications: between 60 and 65 UDP sockets open at any given time. Each test lasted for about 2-1/2 hours on a normal business day during which roughly 100,000 UDP datagrams were received. The results of these tests are shown in Table 4b. All time values have been normalized to 100,000 packets so they are comparable.

Table 4b – Improved Caches with Traffic (SUN)

	no-cache	**SUN cache**	**new cache**
Time in udp_input	17.44	14.16	11.37
udp_input without socket code[6]	8.82	5.07	3.44
% of hits in receiver cache	NA	59%	57%
% of hits in sender cache	NA	NA	30%

[6] In other words, udp_input without the calls to the socket layer buffering routines.

The most important observations on the SUN are that caching is clearly beneficial and the sender cache pays off splendidly, improving performance by about 50% more than just the receiver cache alone. (Recall that the SUN cache scheme suffers from cache hiding and can mis-deliver data; we include it to illustrate the effects of receiver-only caching). A thorough analysis of these issues by McKenney and Dove [9] has shown that even under demanding traffic loads, this caching scheme performs well. (McKenney and Dove also identified a more sophisticated algorithm that performs better under high loads with low locality).

6.3.5. Other Potential Cache-Related Improvements

The cache hit rates on the sender and receiver caches were so high that we decided that trying to optimize the PCB lookup code further was not fruitful. However, we had prepared two additional optimizations and we discuss those optimizations here on the grounds that they may be interesting in other situations.

First, the original PCB lookup routine is rather inefficient. It scans a linked list of UDP PCBs looking for a PCB that exactly matches the incoming datagram (on both source and destination address). While scanning the list, the routine also notes the best wildcard match for the datagram. If the routine finds an exact match, it immediately returns the exact match. If the routine fails to find an exact match after scanning the entire list, it returns the best wildcard match, if any. The inefficiency comes in having to scan the entire list before returning a wildcard. In the original 4.3BSD code, the lookup routine had to check every PCB before returning a wildcard to protect against cache hiding. But in our code, wildcard PCBs that may cause cache hiding are marked. Thus if a wildcard that does not hide another PCB is found, the lookup routine can return the wildcard match immediately. Since most applications use wildcarded PCBs, this enhancement should shorten the average PCB search time by roughly 50% (half the length of the search list).

Another optimization (suggested to us by Gary Delp of IBM) is to move frequently accessed PCBs to the front of the PCB list. An easy implementation of this idea is to move a PCB to the front of the PCB list whenever it is accessed. The PCB list is doubly linked so moving is an easy operation and the algorithm will tend to cluster heavily used PCBs at the front of the list. Observe that this algorithm works best if the PCB lookup routine has already been modified to immediately return a wildcarded PCB that does not cause cache hiding. The expected result of this optimization would be to shorten the average PCB search time to just two or three comparisons.

6.4. Improving Sendit Performance

Recall that the 386 processor spent a majority of its time in the routines to manage changes of processor priority. This observation led us to try to find ways to reduce the calls to these routines. The goal was to find improvements that would significantly help the 386 and also improve performance (though perhaps only a little) on the Sparc.

The BSD networking code changes (or at least, checks) the process priority quite a bit. For example, when a packet arrives, the processor priority must change at least five times before data reaches the user:

(1) First, the network interface must interrupt and after reading the packet, the network driver routine schedules a software interrupt at a lower priority for ipintr;

(2) ipintr in turn calls the transport level routine which places the data on the application's input queue

(3-5) after ipintr completes, the processor priority can be reduced to the lowest level and the application will try to read the data from the queue. However, to protect against race conditions in the input queue, the processor priority must be raised to the level used by ipintr while the queue is being manipulated, then reduced again.

In fact, there are even more priority changes than these, as the networking code often must briefly lock out higher priorities while manipulating shared data structures.

One should observe that the intermediate priority level for ipintr is superfluous and could be removed (all inbound data could be processed at application priority). This would improve performance of both the input and output code, as the output code would no longer have to raise its priority level to avoid race conditions with the input code.[7] However to make that change would have required us to restructure all the BSD networking code.

We were, however, able to identify a more modest improvement in the sendit routine.

When an application calls the sendto system call, it passes down the address to which the datagram is to be sent. Sendit, in turn, allocates a kernel memory buffer, copies the address into the memory buffer, and passes the buffer containing the address to sosend and udp_usrreq. After the datagram has been sent, sendit frees the memory buffer.

There is a small inefficiency in this process. Sendto is a frequently-used system call, so sendit is constantly calling the buffer management routines to allocate and free a buffer. These routines are not very expensive themselves, but they do require a change in processor priority, because memory buffers sometimes have to be allocated by device drivers (which run at a higher interrupt level than the networking code). Thus, if we could reduce the calls to the buffer management routines, the time spent managing processor priority changes should go down.

So we changed sendit to try to avoid freeing memory buffers. The routine keeps a one pointer cache. If sendit is about to free a memory buffer, it first checks the cache. If the pointer is null, it saves the memory buffer in the cache, otherwise it frees the buffer. Before allocating a memory buffer, sendit first checks to see if there is a buffer in the cache; if so, it uses the cached buffer. Note that this scheme works because the memory buffers in the BSD system are a fixed size.

The results for the 386 processor were encouraging and are shown in Table 5a.

Table 5a – Improved Sendit (386)

	64-byte		**512-byte**	
Code Section	**old**	**new**	**old**	**new**
Priority Mgmt/Interrupts	541.30	511.37	550.55	529.25
sendit	53.28	47.76	51.75	53.76
Total Improvement	35.45		19.31	
% Improvement	3.8%		1.9%	

The 386 improvements represent about a 2% to 4% improvement in kernel performance and the reduction in time in priority management code for 64-byte packets is equal to nearly a third of the time spent in UDP and socket code.

We expected improvements on the Sparc to be less notable, largely because changing processor priority levels on the Sparc is much cheaper. However the Sparc improvements were useful. The results are shown in Table 5b. (The Sparc costs for sendit include the costs of calls to priority routines).

[7] The latest 4.4BSD release will eliminate this priority level.

Table 5b – Improved Sendit (Sparc)				
	64-byte		512-byte	
Code Section	**old**	**new**	**old**	**new**
sendit	22.30	19.27	24.26	22.54
Total Improvement	3.03		1.72	
% Improvement	1.6%		0.8%	

The improvement is equal to about 1% to 2% of total kernel time.

7. Reducing Data Handling Costs

After reducing the per-packet overhead, we worked on reducing the data handling costs. The structure of the BSD code largely dictated that there be two data copies on both transmission and receipt of data plus a checksum. Since neither copy could be eliminated without completely rewriting the BSD code, we focussed on trying to reduce costs by combining the checksum with one of the copy loops.

7.1. Combining the Checksum and Copy Loops

Over the past few years, there has been much discussion of the possible merits of reducing the number of times that the data in a datagram must be scanned by reducing the number of data copies between different memory and buffer spaces and by combining the checksum and data copy loops. (This idea is believed to have originated with Van Jacobson).

The standard BSD code scans data three times on both input and output. On output, the data is copied from user space to kernel buffers, checksummed, and then copied from kernel buffers to interface memory. The same operations are done on input, but in reverse order. The copies between kernel buffers and interface memory are often done with some hardware assist, such as DMA, but the copies between user and kernel space and the checksum are all done in software. We changed the UDP code to do the copy between user and kernel space and the checksum in a single loop.

7.1.1. Changes to Code

Since checksums can be protocol specific, the code to do copies and checksums had to be done in a protocol specific way. The logical approach, therefore, was to add some new entries to the protocol switch table. Initially, we tried simply adding protocol specific data copy routines to the protocol switch table but this turned out to make the already complicated socket routines even more complex.[8] Making code more complex did not seem desirable, so we tried another approach and wrote protocol-specific versions of sosend and soreceive and placed them in the protocol switch table.

In this new version of the code, when a datagram is sent, sendit calls udp_sosend. Udp_sosend does some error checking, then copies and checksums the data from user space, adds the UDP protocol header, and calls ip_output with the UDP datagram. Thus a chain of three procedure calls (sosend, udp_usrreq, udp_output) was reduced to a single call to

[8] The list of added complexities is long and not very edifying. For those who are curious, here are a two examples of the problems encountered. First, having protocol-specific data copy routines meant that the socket code had to be recoded to either use the new copy routines, if they existed, or the old copy routines if there were no protocol specific routines. So already complex code to manage data copies was made even harder to read. Another problem was communication with lower layers. How does the socket layer pass the partial checksums it computes down to the UDP layer on output? How does the UDP input routines pass up partial checksum information to socket routines on input?

udp_sosend. Furthermore, because udp_sosend is protocol-specific, we were able to eliminate a lot of unnecessary tests and reduce the number of critical regions that had to be protected.

On the receiving side, the code is a bit more complex. Udp_input was changed to compute the checksum on the UDP header (but not the UDP data) and then remove the UDP header from the datagram. The remainder of the datagram, along with the partial checksum and the length of the UDP datagram are then passed to sbappendaddr, which queues the datagram to be read by the application. When the application does the read, recvit calls a UDP-specific version of soreceive, which checksums the data while copying the data into application memory. If the checksum is correct, then the read completes. If the checksum fails, one of two things happens. If the application is not non-blocking, the read routine simply waits for the next datagram to arrive. If the application is non-blocking, and there are no additional datagrams immediately waiting to be processed, the routine returns an error indicating the read would block (EWOULDBLOCK).

Observe that this new version of the receiving code implies two minor changes to the semantics of reading from a socket. First, a non-blocking application which learns from a select call that data is waiting to be read may get an unexpected error (EWOULDBLOCK) if the data fails to checksum. Second, because the checksum can only be confirmed after the data has been copied, an application's buffers will be changed even if the read call fails.

The key piece of code that both copies and checksums the data was written in assembler for both the 386 and Sparc systems. (The authors would like to thank Van Jacobson for providing the assembly code for the Sparc).

7.1.2. Performance Improvements

Combining the copy and checksum loops required extensive changes to the UDP implementation. In addition to combining the two loops, calling paths have been changed, and old code is often executed in new places and at different priority levels. As a result, direct comparisons of old and new code are somewhat difficult. However, there were dramatic improvements.

The effect of combining the checksum and copy loops is to reduce three memory accesses to two. The copy loop had one read and one write for each word of data, and the checksum had to read the data again, for a total of three accesses. The combined loop reads each word into a register, adds the word to the running checksum, and then writes the word, thus giving two accesses plus an addition. Intuitively, one would look for about a one third improvement in speed.

In fact, the improvement was substantially more, as can be seen in Tables 6a and 6b, which compare the time spent in the checksum and copy routines before and after the changes.[9] These results are shown in Tables 6a and 6b.

Table 6a: Combined Checksum and Copy (386)

data size	before	after	improvement	%
64 bytes	51.57	27.56	24.01	46.6%
512 bytes	108.59	65.57	43.02	39.6%

[9] The "before" numbers are the costs of calling both uiomove (the copy routine) and in_cksum. They are from a profile taken after the ip header checksum code was optimized. The "after" numbers are the cost of calling in_uiomove which both moves and checksums in one loop.

Table 6b: Combined Checksum and Copy (Sparc)				
data size	**before**	**after**	**improvement**	**%**
64	12.40	4.40	8.00	64.5%
512	20.20	5.0	15.2	75.2%

For both 386 and the Sparc, tests of the copy-checksum routines suggest that, for some inputs, the combined routines are faster than the original copy routines they replaced! The 386 code was written by the authors and was carefully designed to fit inside the small cache of the 386 processor we were using. (Tests of the same code with other processors with different cache sizes gave smaller performance improvements). The Sparc code provided Van Jacobson was modified by the authors to fit into the kernel. These modifications made Jacobson's routine slightly slower but still substantially faster than the existing copy code in the Sparc.

8. Summary of Improvements

The incremental discussions of improvements above may make it difficult to assess the total performance improvement. This section summarizes the improvements and discusses the significance of the results.

8.1. Overall Improvements

Tables 7a and 7b compares the total improvement between the intial (unoptimized) system and the final (most optimized) system. In addition to the regular data sizes of 64 and 512 bytes, the sizes of 59 and 509 were tested to show that the performance improvements are not restricted to data sizes which are powers of two.

Table 7a: Total Performance Improvement (386)				
	59 bytes	**64 bytes**	**509 bytes**	**512 bytes**
initial UDP/IP & socket	202.19	193.37	264.21	253.45
optimized UDP/IP & socket	134.91	140.88	181.37	177.05
UDP/IP & socket improvement	74.86	66.31	90.46	85.57
initial total kernel	927.12	936.03	1034.35	1018.36
optimized total kernel	808.63	811.50	869.76	854.58
total kernel improvement	118.49	124.53	164.59	163.78
% UDP improvement	36%	32%	33%	32.6%
% Kernel improvement	12.8%	13.3%	15.9%	16.1%

Table 7b: Total Performance Improvement (Sparc)				
	59 bytes	**64 bytes**	**509 bytes**	**512 bytes**
initial UDP/IP & socket	92.32	83.11	93.00	91.48
optimized UDP/IP & socket	55.57	58.53	68.95	64.17
UDP/IP & socket improvement	36.75	24.58	24.05	27.31
initial total kernel	184.40	181.42	207.06	211.33
optimized total kernel	151.00	147.77	178.25	175.15
% UDP improvement	39.8%	29.6%	25.9%	29.9%
% Kernel improvement	18.11%	18.54%	13.9%	17.1%

The results are generally very good. UDP performance across both CISC and RISC systems improved about 30% and overall kernel performance improved between 12% and 18%

8.2. Significance of Results

We set out to optimize UDP as an exercise in applying optimizations used on TCP to other protocols. Broadly speaking, the optimizations that worked well took one of three forms:

1. Caches were used to exploit locality. A one-behind sender and receiver cache was added. Mbufs were cached in sendit.
2. Expensive general purpose code was replaced with code tuned to the particular protocol. Sosend was replaced with udp_sosend and the IP header checksum was computed in-line rather than in a general purpose subroutine.
3. Memory accesses were reduced. In this experiment, the checksum and copy loops were combined.

These all are general techniques. They worked well when applied to TCP and have worked equally well when applied to UDP. Recent work by Sample and Neufeld has shown that these same techniques can also be applied to making external data format conversion run fast [17]. The implication is that these techniques should be routinely applied to protocol implementations.

8.3. Optimizations Not Tested

There are at least three optimizations that are believed to improve protocol performance that could not be tested in our test environment.

The first optimization is to build a better network interface. As protocol processing costs have been reduced, it has become increasingly clear that network device drivers are often both performance pigs (requiring lots of code to manage) and manage data buffers poorly (causing increased overhead in data handling). Recently there has been consider success in building better network interfaces, most notably the Medusa and Afterburner interfaces built by a team at Hewlett Packard [13,14]. These interfaces are designed to minimize device driver complexity (it is a matter of a few instructions to send or receive a packet) and provide better buffer management. Unfortunately, an interface built along these lines was not available for either of our test systems.

The second optimization is to try to improve code locality and reduce instruction cache misses in the processor. In RISC processors, it is also possible to improve branch predictions and minimize branch stalls. Promising results in this area have been reported by Speer *et. al.* [18] However, neither of our tests systems had the compiler support necessary to experiment with this type of optimization.

Finally, there has been some interesting work with parallelism. Work by Björkman and Gunningberg with TCP [19] and Asthana *et. al.* on a parallel router [20] suggest that by striping entire datagrams (rather than pieces of the datagrams) across processors, it may be possible to achieve better TCP and IP performance. Unfortunately, we did not have access to a multiprocessor machine to test these ideas.

8.4. Generality of Results to Different Hardware Bases

The performance improvements were done on both a CISC processor and a RISC processor to evaluate the impact of various performance improvements on different systems.

One of the interesting results is that, except for the extraordinary overhead (particularly, priority management and interrupt code) of the 386 processor, each optimization had a similar effect on both processors. This suggests, though by no means proves, that the optimizations are general rather than processor specific.

Recently, there's been some debate about whether combining the checksum and copy loops is a general solution or specific to certain machine architectures. The evidence is somewhat mixed.

Table 8: Cost of Adding Checksum to Copy Loop			
System	**copy** ns/byte	**copy and checksum** ns/byte	**additional cost** ns/byte
HP9000/370	122	144	18%
Sparcstation1	164	177	8%
Sparcstation2	109	109	0%
HP9000/720	54	54	0%
DECstation 5000/133 (R3000 CPU)	82.5	99.5	20%
DECstation 2100 (R2000 CPU)	288	292	1.4%
Values for HP9000s and Sparcstations from Van Jacobson. Values for DECstations from Peter Desnoyers. All values are for transfering uncached data. Results for cached data typically show slightly higher costs for adding the checksum.			

Tests on a number of machines have shown that a combined copy and checksum loop runs only very slightly slower than the copy loop alone. The results for several different workstations architectures are summarized in Table 8. However, in systems where the only memory copy is between application buffers and the network interface, it may be more efficient to do the data copy using DMA rather than a software loop [21]. The emerging principle seems to be that if one has to do a copy loop using software, then the checksum should be combined with the copy loop because the checksum adds very little to the cost. But if the data copy is best done by hardware, other mechanisms for performing the checksum need to be found.

8.5. Conclusion

Recently Van Jacobson has informally reported improving TCP/IP and UDP/IP performance in BSD by as much as 95%, by doing a complete reimplementation of the protocols that takes greater advantage of many of the performance enhancements mentioned above. The improvements discussed in this paper are much more modest, but in the same vein. Our conclusion is that many problems experienced with protocol performance are more appropriately blamed on implementations than the protocols themselves.

9. Acknowledgements

Thanks to several members of the End-To-End Interest mailing list for several helpful comments and notes while we worked on this effort. Special thanks are due to Van Jacobson, many of whose ideas are reflected in this work, and who graciously volunteered his code for the combined checksum and copy loop on the Sparc.

Appendix I: 386 Copy and Checksum Code

As an example of the combined copy and checksum code, here is an excerpt from the user-space version of the code for the 386. Note that this routine returns the 1's-complement sum over the data – to generate the full IP checksum requires the sum to be complemented.

```
/ C calling convention:
/ in_bcopy(from,to,count,xsum); char *from, *to; int count; unsigned short *xsum
/ copy count bytes between from and to and put sum in xsum
```

```
_in_bcopy:
        pushl %edi                    / save various registers
        pushl %esi
        pushl %ebx
        pushl %ecx
        pushl %edx
        movl 20 +12(%esp),%edx        / 20 for 5 pushes onto stack
        movl 20 +8(%esp),%edi
        movl 20 +4(%esp),%esi

        movl $0,%eax                  / zero the initial xsum
        movl %edx,%ecx  / copy the byte count
        js ibout                      / if byte count negative, quit now

        sarl $5,%ecx                  / start by adding groups of 8 long words
        dec %ecx                      / count down to enter loop
        jl ibL2                       / jump to code to do smaller units of data
ibL1:
        addl (%esi),%eax              / this ordering runs best in our cache
        movsl
        adcl (%esi),%eax
        movsl
        adcl (%esi),%eax
        movsl
        adcl (%esi),%eax
        movsl
        addl (%esi),%eax
        movsl
        adcl (%esi),%eax
        movsl
        adcl (%esi),%eax
        movsl
        adcl (%esi),%eax
        movsl
        adcl $0,%eax                  / fold in carry bit before doing dec
        dec %ecx                      / decrement 8 long word count
        jge ibL1                      / fall through to code to handle
                                      / smaller units of data

ibL2:                                 /do remaining data here
```

References

1. Jon B. Postel, "User Datagram Protocol; RFC-768," *Internet Requests for Comments*, no. 768, August 1980.
2. V. Srinivasan and Jeffrey C. Mogul, "Spritely NFS: Experiments with Cache-Consistency Protocols," *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, December 3-6, 1989.
3. V. Jacobson, "4BSD Header Prediction," *Computer Communication Review*, vol. 20, no. 2, pp. 13-15, ACM SIGCOMM, April 1990.
4. P.V. Mockapetris and K. Dunlap, "Development of the Domain Name System," *Proc. ACM SIGCOMM '88*, pp. 123-133, Stanford, USA, 16-19 August 1988.

5. Sun Microsystems, "NFS: Network File System Protocol specification; RFC-1094," *Internet Requests for Comments*, no. 1094, March 1989.

6. D.L. Mills, "Network Time Protocol (Version 3): Specification, implementation, and analysis; RFC-1305," *Internet Requests for Comments*, no. 1305, DDN Network Information Center, March 1992.

7. D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications*, vol. 27, no. 6, pp. 23-29, June 1989.

8. Jeffrey C. Mogul, "Network Locality at the Scale of Processes," *Proc. SIGCOMM '91*, Zurich, Switzerland, September 3-6, 1991.

9. P.E. McKenney and K.F. Dove, "Efficient Demultiplexing of Incoming TCP Packets," *Proc. ACM SIGCOMM '92*, pp. 269-280, Baltimore, MD, August 1992.

10. J. Kay and J. Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *Proc. ACM SIGCOMM '93*, San Francisco, USA, 15-17 September 1993.

11. D.D. Clark, *SIGCOMM Award Lecture*, 1990.

12. V. Jacobson, *Tutorial Notes from SIGCOMM '90*, Philadelphia, USA, September 1990.

13. D. Banks and M. Prudence, "A High Performance Network Architecture for a PA-RISC Workstation," *IEEE Jour. Selected Areas in Communications*, vol. 10, no. 1, pp. 191-202, February 1993.

14. G. Watson, D. Banks, C. Calamvokis, C. Dalton, A. Edwards, and J. Lumley, "Afterburner: Architectural support for high performance protocols.," *IEEE Network Magazine*, vol. 7, no. 4.

15. S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, pp. Addison-Wesley, 1989.

16. Jon B. Postel, "Discard Protocol; RFC-863," *Internet Requests for Comments*, no. 863, May 1983.

17. M. Sample and G. Neufeld, "Implementing Efficient Encoders and Decoders for Network Data Representations," *Proc. IEEE INFOCOM '93*, pp. 1144-1153, San Francisco, USA, 30 March-1 April 1993.

18. S.E. Speer, R. Kumar, and C. Partridge, *Improving UNIX Kernel and Networking Performance using Profile Based Optimization (Technical Report)*, August 1993.

19. M. Björkman and P. Gunningberg, "Locking Effects in Multiprocessor Implementation of Protocols," *Proc. ACM SIGCOMM '93*, San Francisco, USA.

20. A. Asthana, C. Delph, H.V. Jagadish, and P. Krzyzanowski, "Towards a Gigabit IP Router," *Jour. of High Speed Networks*, vol. 1, no. 4, pp. 281-288, 1992.

21. J.M. Smith and G.Q. Maguire, Jr., "Measured Response Times for Page-Sized Fetches on a Network," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 5, pp. 71-77, September 1989.