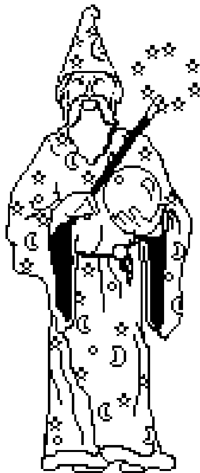
HP–UX Release 9.0 Networking Performance Enhancements

We Go Fast With a Little Help From Your Apps

Rick Jones, Information Networks Division
Sandy Greer, Information Networks Division

With the release of HP–UX 9.0, HP 9000 systems truly enter the realm of high–speed networking. The application of experience gained through years of experimentation and prototyping have come together to produce a networking offering that provides exceptional levels of throughput with minimal CPU overhead. This level of performance is achieved through close cooperation between the application and the transport.



1 Introduction

This report is intended to give you a focused look at the performance enhancements introduced with HP–UX Release 9.0 and how your application might use them. A comprehensive look at networking performance for the HP–UX Release 9.0 and HP 9000 Systems in general is beyond the scope of this report.

This report assumes that you are familiar with the basics of BSD sockets programming as well as the concepts of virtual memory, and data caches.

These performance enhancements rely on features offered by the NIO FDDI Product (pn J2157A). Future link products may include features to support some, all, or none of these enhancements.

Two performance enhancements were added to HP–UX Release 9.0. They are support for Internet checksum offloading by network interfaces, and copy avoidance for user data. Checksum offloading is a feature of the attached network interface(s) benefiting all applications. Copy avoidance also relies on the presence of a feature in the network interface. Further, copy avoidance requires assistance from the application.

The Copy Avoidance features, as implemented in HP–UX Release 9.0, and the NIO FDDI Network Interface, are geared to a "server solution." This is to say that it is most effective in a server environment, where data is moved from one medium to another (eg memory to network) without being manipulated. This is described in greater detail in Section 3.1.

In addition to presenting results, this report will present techniques – in the form of C–style code fragments – that will allow you to code your own networking applications to take advantage of the performance enhancements described in this document.

For the first time in an HP–UX networking brief, CPU utilization information will be presented along with the throughput numbers. However, the specific techniques used to measure this CPU utilization may not be available to you as they are based on options not available with production kernels.

 Revision 1.1, 02/09/93, Copyright Hewlett–Packard Company

2 Checksum Offload

HP–UX Release 9.0 was modified to take advantage of network interfaces offering Internet checksum offloading. When a connection runs over such an interface, the Transport will let the interface calculate the Internet checksum for inbound and outbound packets.

Your applications do not need to be modified to take advantage of checksum offload. This feature is completely contained within the Transport.

The NIO FDDI Network Interface Card (pn J2157A) is the first network interface from Hewlett–Packard to provide hardware support for TCP and UDP Internet checksums. This support is bi–directional and takes place within the card itself. It is a necessary first step to providing complete copy avoidance. (See Section 3)

It is not necessary that both sides of the connection be running over NIO FDDI cards. The feature is completely local to the card and is indistinguishable on the wire from host checksumming. So, you can migrate a server solution from Ethernet (or 802.5) to FDDI and see the server–side CPU utilization gains – even if you leave your clients on the Ethernet!

The loopback interface also offers checksum offload. This is used when a connection goes directly to the loopback interface using the 127.0.0.1 address, and not when using one of the host's other IP addresses (eg the connected Ethernet). To insure that all your loopback connections take advantage of loopback checksum offload you should configure host routes pointing at 127.0.0.1 for each of your connected interfaces. See Appendix D for more information.

Effect of Checksum Offload on TCP Stream Throughput

32KB Socket Buffers, 4KB Sends

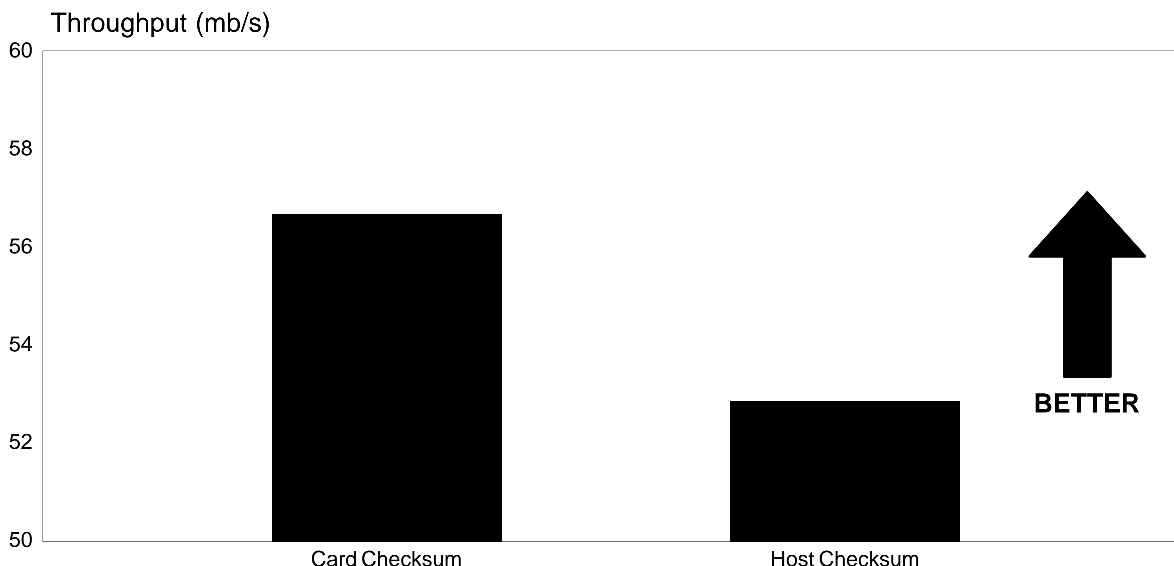


Figure 1: Card vs. Host Checksums

3 Copy Avoidance

One of the biggest bottlenecks to networking performance is the copying and accessing of the user's data. In a classic (BSD) TCP implementation, the user's data is either moved or examined three times. There is the copy from user space into kernel space, the TCP checksum calculation, and finally the movement of the data between host memory and the interface card.

If one can find ways to avoid copying data, then it should be possible to greatly increase the efficiency of sending or receiving data. In the absence of other bottlenecks, one should also see an increase in throughput.

There are many ways to avoid copying data. The two which will be discussed in this document are copy–on–write, and page remapping. Briefly, copy–on–write is the setting–up of a duplicate reference to a buffer. If someone tries to write to that buffer while the reference is active, a copy will be made. Page remapping is the transfer of data from one virtual address to another by swapping their physical page mappings.

Figure 2 presents the results of two TCP unidirectional stream tests between a pair of HP 9000 Series X30 class machines (a.k.a. 837/847/857). In one test, labeled “Copies,” the only optimization present was Internet checksum offload. In the other test, labeled “Copy Avoidance,” conditions were such that the copying of data between user space and kernel space could be forgone.

As Figure 2 shows, enabling the copy avoidance features available with HP–UX Release 9.0 and NIO FDDI does indeed improve the efficiency of networked data transfer.

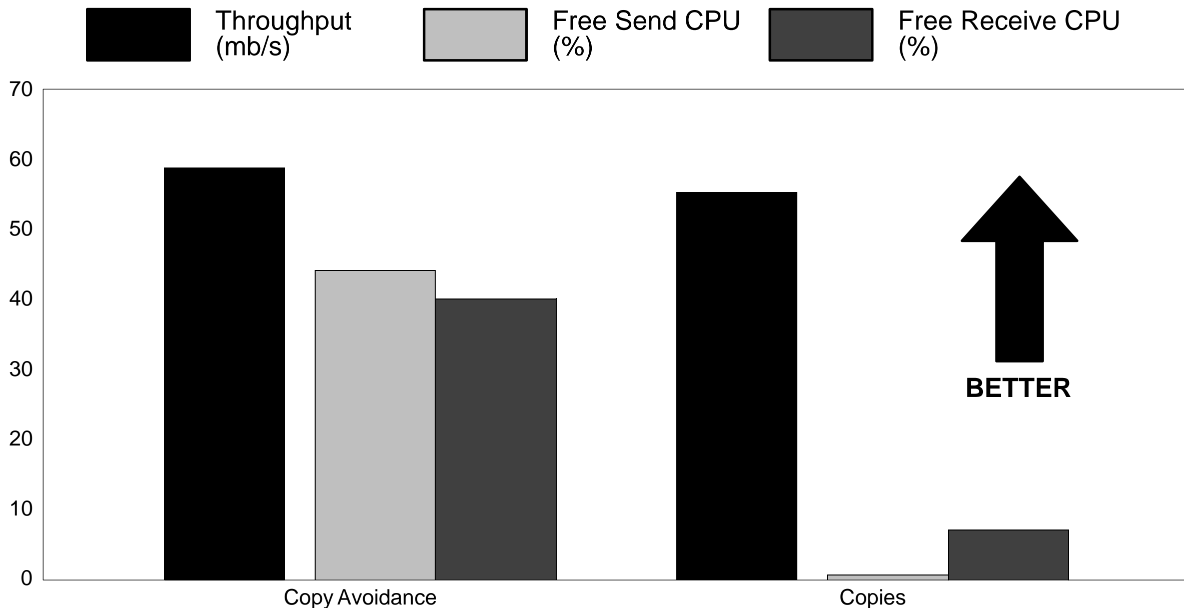


Figure 2: Copies vs. No Copies

 Revision 1.1, 02/09/93, Copyright Hewlett–Packard Company

Figure 2 shows a considerable improvement in efficiency, but not a considerable increase in throughput. This is the unfortunate result of other bottlenecks in the systems under test. These bottlenecks are the topic of current research and development for future releases and platforms.

A description of the application code changes necessary for copy avoidance can be found in Section 4.

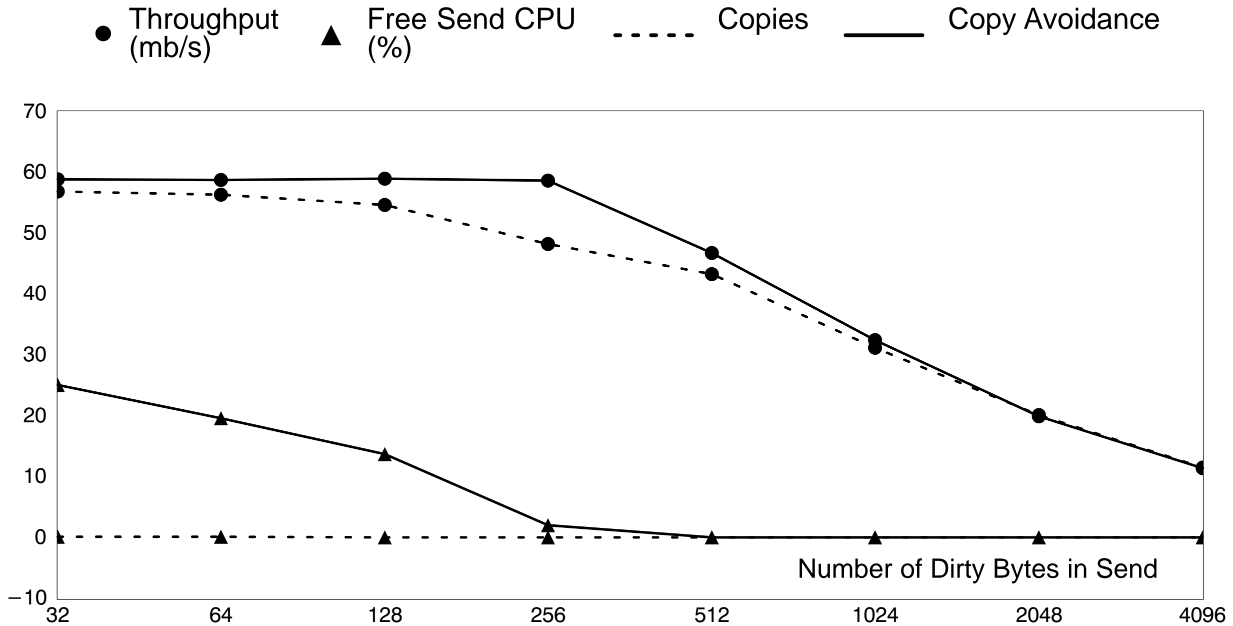


Figure 3: Effect of Dirtiness on Send Performance

3.1 The Importance of Staying Clean

Earlier, we said that the HP–UX 9.0/NIO FDDI copy avoidance features were "server oriented." This stems from their sensitivity to the state of buffers in the processor's data cache. In the context of this discussion, the terms "dirty" and "clean" will refer to the state of the user's buffers in the processor's data cache. To say that a location is dirty in cache implies that the application has written to that location. A location is clean in cache, or "cache–clean," if it has been read, but not written. It is also possible for a location to be marked as invalid.

Figure 3 shows the relationship between the "dirtiness" of the buffer being sent, and the sending CPU utilization. For the copy avoidance case, the throughput does not degrade until CPU saturation is reached (somewhere after 256 bytes with the processor measured). From that point on, the copy avoidance case, and the default case perform similarly.

Figure 4 shows a similar relationship for the receiving side.Again, the throughput holds steady until CPU saturation is achieved. However, in the receive case, notice that the copy avoidance throughput stays above the throughput in the copy case.

The difference in semantics between sending and receiving data dictate different implementations of copy avoidance. When an application makes a call to send(2), it must be able to assume that the data presented will be preserved after the syscall. The copy avoidance mechanism used in this context is copy–on–write. When an application makes a call to recv(2) it expects the data in its buffer will be replaced with new data. The copy avoidance mechanism used in this context is page remapping.

Revision 1.1, 02/09/93, Copyright Hewlett–Packard Company

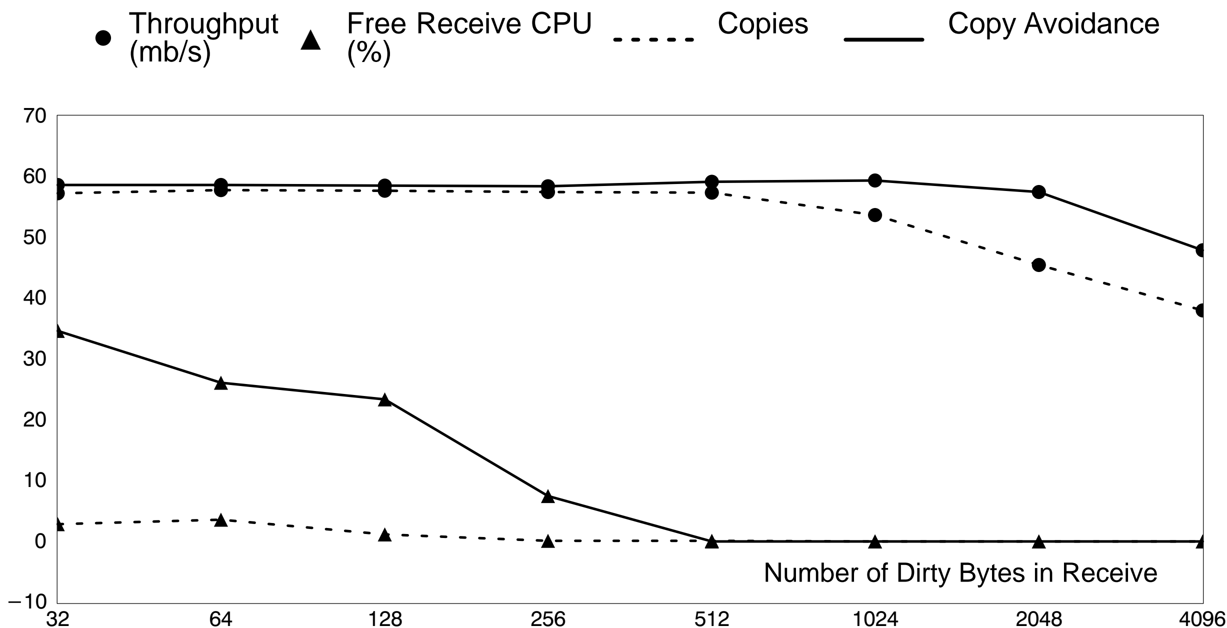


Figure 4: Effect of Dirtiness on Receive Performance

When an application presents data in a send(2) call that is dirty in the data cache, it must be moved from the cache into memory before the packet(s) can be sent. This is called “flushing” the cache. However, when an application presents a dirty buffer in a recv(2) call, that data does not need to be moved to memory because it is going to be overwritten; instead, the dirty buffer is simply marked as invalid. This is referred to as “purging’’ the cache. The replacing of data is why receive copy avoidance is less sensitive to buffer dirtiness than send copy avoidance.

3.2 Send Buffer Strategy

In Section 3.1, we saw the importance of providing “clean” buffers to efficient send performance. The point at which an application touches a send buffer can have as considerable an impact on performance. In this section, we examine the effect that different application buffering strategies can have on that efficiency, with particular emphasis on TCP bulk–data connections.

One common practice (among TCP benchmarks at least) is to allocate a single buffer, and then send that same buffer repeatedly. On classic BSD TCP implementations, that one buffer will be copied over and over into buffers held by TCP. If a 32KB send socket buffer is used, and a 4KB application send buffer, then a maximum of 36KB of memory will be used to hold data to be transferred (32KB in the Transport, and 4KB in the application)

By requesting send copy avoidance (See Section 4), an application is advising the Transport that it will try not to re–use that buffer until the Transport is finished with it. Given that advice, the Transport will employ the copy–on–write optimization. If the application does touch a buffer that is being used by the Transport, everything will still “work,” but the application will force a copy of the buffer and thus incur a performance penalty. Those are the semantics of a copy–on–write scheme.

Figure 5 shows how the number of application buffers can affect TCP send performance and efficiency when copy avoidance is enabled. In this test, the application opens a 32KB socket, and is making 4KB sends in a round–robin fashion among its send buffers. Before calling send(2), the application writes into the first word of the buffer.

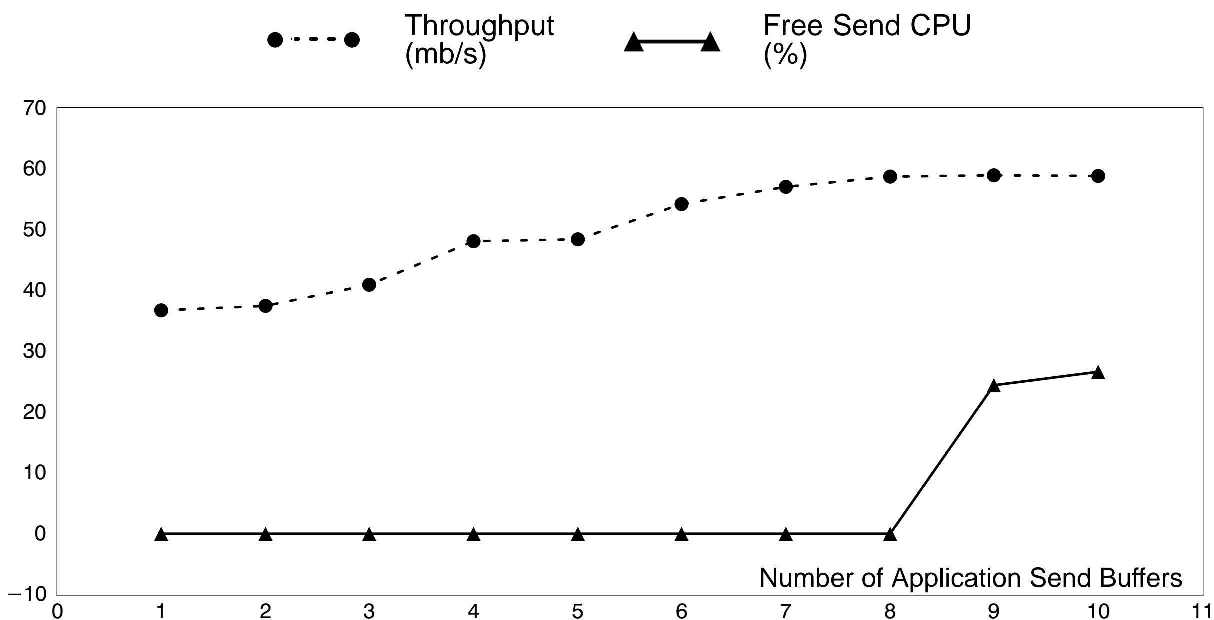


Figure 5: Effect of “Width” on Dirty Send Performance

Revision 1.1, 02/09/93, Copyright Hewlett–Packard Company

Notice that there is a considerable increase in free CPU when the number of send buffers increases from 8 to 9. This corresponds to an increase from 32KB of application buffer space to 36KB, which is larger than the size of the send socket buffer.

With a TCP connection, one buffering strategy is to allocate a quantity of bytes equal to the size of the send socket buffer plus one send. (Ie SO_SNDBUF + send_size). You can see this being used in the example code in Appendix A. This strategy insures that the application does not try to re–use a buffer before its time. Returning to the test described in Figure 5, when 9 application send buffers are used the first 8 sends will proceed, but the 9th send will block until first send is ACKnowledged by the remote. This will release the first buffer and unblock the application. The application can now re–use the first buffer without additional performance penalties. It all works something like a circular buffer.

The total number of bytes being used to hold data will be the same in this case as it was in the copy case described at the beginning of this section. While the application allocates 36KB of space, the Transport will not allocate any extra space. Copy–on–write allows the Transport to effectively “borrow” the buffer from the application, so long as the application does not try to write to it while the Transport is using it.

The strategy above was geared towards a TCP unidirectional data stream, but will also work with a TCP requests/response paradigm. However, when the application is written along a request/response paradigm, then the receipt of a reply can serve as the indication on the request side that the Transport has finished with the buffer(s) containing the request. Similarly for the other side the receipt of a request can indicate that the previous response buffer is no longer being held by the Transport. This allows the application to allocate less buffer space.

The request/reply strategy is the only strategy that can be used with UDP applications. UDP, unlike TCP, lacks a local indication (eg blocking on send(2)) that the Transport is finished wit the buffer. The only indication that the buffer has been freed is a subsequent response arriving from the remote system.

3.3 Receive Buffer Strategy

As was seen in Section 3.1, page remapping, the optimization used for receive copy avoidance, is less sensitive to the state of the buffers in the processor’s data cache. This is also the case with the number of buffers actually used. Since the pages are being replaced entirely, rather than borrowed, there is essentially no time that the Transport is holding a buffer that the application will try to access. So, an application does not need to alter its buffer strategy (eg number of buffers) to effectively utilize receive copy avoidance.

4 That's Great, But How Can I Use It?

The copy avoidance features of HP–UX 9.0 and NIO FDDI do need a little help from your apps. If you can code your application(s) such that the following conditions are true, then you can expect that your application(s) will run with maximum efficiency, both on HP–UX Release 9.0/NIO FDDI, and on future releases and network interfaces:

1. the application has made the appropriate setsockopt(2) calls to enable copy–avoidance (described later in this section)
2. packets are transferred using calls to send(2) and recv(2)/read(2)
3. buffers passed to send(2) and recv(2)/read(2) are page–aligned
4. buffers passed to send(2) and recv(2)/read(2) are integer multiples of page size in length
5. the application tries to avoid touching or otherwise accessing the buffers while they are in use by the Transport.
6. the application tries to pass buffers that are "cache–clean" to send(2) and recv(2)

Some of these limitations may be relaxed or eliminated in future releases, however, if you consider these to be the "least common denominator," and code accordingly, you should receive the greatest benefit.

The appendices on the following pages contain code fragments from the netperf benchmark used to generate the data presented in this report. They are provided as example code only, and may or may not be suitable to your task(s). Further, they represent neither complete, nor compilable code.

It is assumed that you are already familiar with the concepts of BSD sockets programming. If you are not, we suggest that you consult the *Berkeley IPC Programmer's Guide* before you proceed.

Finally, while these examples show one way to follow the rules listed above, they are not the only way those rules could be followed.

For a look at a complete program using these features, consult the source to the netperf benchmark. Netperf should be available from several archives on the Internet.

To enable copy avoidance, ie request it of the Transport, a new pair of socket options have been added to the setsockopt(2). They are SO_SND_COPYAVOID and SO_RCV_COPYAVOID. Here is a brief description of the setsockopt(2) interface (see the manpage for a more complete general description of setsockopt(2)):

```
setsockopt(
        int s,
        int level,
        int optname,
        const void *optval,
        int optlen);
```

The parameters for the new options are as follows:

s:	the socket descriptor
level:	SOL_SOCKET
optname:	SO_SND_COPYAVOID –or– SO_RCV_COPYAVOID
optval:	a pointer to an integer whose value is 1 to enable, or 0 to disable the feature
optlen:	sizeof(int)

A Code Fragments for send(2)

Your socket create, connect, and other code might go here...

First of all, tell the Transport that you wish to use the outbound copy avoidance features...

```
#ifdef  SO_SND_COPYAVOID
send_avoidance = 1;
if  (setsockopt(send_socket,
             SOL_SOCKET,
             SO_SND_COPYAVOID,
             &send_avoidance,
             sizeof(int)) < 0)
                perror("Could not enable send copy avoidance");
#endif
```

Here we allocate our send buffers. It is important to make sure that they are page aligned, and there are enough of them to have one more send's worth than will "fit" with the size specified for SO_SNDBUF with the setsockopt(2) call. If you do not set SO_SNDBUF explicitly, it's current value can be retrieved with the getsockopt(2) call. In this example, we are assuming that the send_size is an integral multiple of the page size. It does not have to be, but the example would be more complicated.

```
num_send_buffers = (so_sndbuf_size/send_size) + 1;
if (num_send_buffers == 1) num_send_buffers = 2;
```

We do not know what sort of alignment malloc will provide, so we must align the buffer(s) ourselves. To make sure that we have enough space in the buffer after we shift the "real" pointer, we should add the alignment value to the size of the buffer we are allocating... The constant NBPG comes from the include of <param.h>...

```
message_base = (char *)malloc((num_send_buffers * send_size) + NBPG);
message_free_ptr = message_base; /* have to retain for the free() */
message_ptr = (char *)(((int)message_base + NBPG - 1) & ~(NBPG - 1));
message_base = message_ptr;
message_end = message_base + (num_send_buffers * send_size);
```

Everything is set–up...

... and we can start sending buffers. At some point we will have to figure–out where to stop.

```
while (should_be_sending) {
        if((len=send(send_socket,
                   message_ptr,
                   send_size,
                   0)) != send_size) {
                perror("data send error");
                exit(1);
        }
```

This send has completed, so we move our buffer pointer to the next send buffer. If we have reached the "end" of the set of buffers, we want to wrap around to the beginning again.

```
        if ((message_ptr += send_size) == message_end) {
                message_ptr = message_base;
        }
}
```

We have finished sending all our data, and can now move on to other things...

B Code Fragments using recv(2)

This will also work with read(2) calls against a BSD socket.

Your socket create and other code goes here...

First of all, tell the Transport that you wish to use the outbound copy avoidance features...

```
#ifdef  SO_RCV_COPYAVOID
receive_avoidance = 1;
if  (setsockopt(recv_socket,
               SOL_SOCKET,
               SO_RCV_COPYAVOID,
               &receive_avoid,
               sizeof(int)) < 0)
                  perror(”Could not enable receive copy avoidance”);
#endif
```

Allocate our receive buffer. The receive copy avoidance does not require that we allocate more than one buffer, so for this example, we will just allocate one. We add our alignment to the size passed in to malloc to ensure that there are enough bytes after alignment...

```
message_base = (char *)malloc(receive_size + NBPG)
message_free_ptr = message_base; /* have to retain for the free() */
message_ptr = (char *)(((int)message_base + NBPG – 1) & ~(NBPG – 1));
```

Now we just keep receiving data until there is no more.

```
while (len = recv(recv_socket, message_ptr, recv_size, 0)) {
        if (len == –1) {
                perror(”problem with recv()”)
                exit(1);
        }
}
```

C Tools and Configurations

The data presented in this report was generated with the netperf benchmark. Netperf is a benchmark initially developed by Hewlett–Packard to measure the performance TCP/UDP/IP on HP 9000 computer systems. Around the time of this publication, it is expected that netperf will be made available to the "general public" on an "as–is" basis.

One HP 9000 857 and one HP 9000 847 were used to generate the numbers presented in this report. Apart from the number of NIO slots available, these two machines are identical in performance and no effort has been made in this report to distinguish between the two. After December 1, 1992, the naming scheme and product structure for the HP 9000 8X7 systems changed. The netperf results presented here are expected to be valid for any of the systems in the "30" performance range (Eg F30, G30, H30, and I30).

Each system was configured with at least 32MB of main memory, and one J2157A NIO FDDI card. The systems, along with a single concentrator, formed a completely isolated two node FDDI ring. Neither system was connected to another network.

All systems were running HP–UX Release 9.0 MR kernels with the NIO FDDI software product installed. One non–standard feature was enabled to facilitate the gathering of CPU utilization figures. Briefly, this entails replacing the normal kernel idle loop with one that counts while the system is idle. Such a kernel is not presently available outside of the Hewlett–Packard R&D Labs. If possible, the IND Networking Performance Team may make a .o file available that enables this feature. USE OF THIS .O FILE WOULD BE COMPLETELY UNSUPPORTED.

If you have any questions about this report, the system configurations, or the tools used, please feel free to send Internet mail with a descriptive subject line to:

perfbrief@hptnjar.cup.hp.com

D Routing for Loopback Checksum Offload

In Section 2, we mentioned that to receive the full benefit of checksum offload from the loopback interface (lo0), it was necessary to configure host routes pointing to the loopback interface for each of a host's directly connected IP addresses. This appendix provides a quick example to help clarify the point.

Before proceeding, we suggest that you familiarize yourself with the route(1M) and netstat(1M) commands.

Let us assume that we have a system with two directly connected Ethernet Interfaces, with IP addresses 15.13.104.244, and 192.2.1.244. Also assume that there is a loopback interface at 127.0.0.1. The output of a 'netstat –r –n' command on a typical system would look something like this:

```
# netstat -r -n
Routing tables
Destination       Gateway           Flags     Refs      Use  Interface
127.0.0.1         127.0.0.1         UH           0   397953  lo0
192.2.1           192.2.1.244       U            0   572167  lan1
15                15.13.104.244     U            2   603013  lan0
```

If one establishes a connection to 15.13.104.244, the routing code on HP–UX will route that connection through the Ethernet interface lan0, which will loop the packets back. This interface does not offer checksum offload, so the TCP code will not try to take advantage of it. However, you can use "route add" commands and get a routing table which looks something like this:

```
# route add host 15.13.104.244 127.0.0.1
add host 15.13.104.244: gateway 127.0.0.1
# route add host 192.2.1.244 127.0.0.1
add host 192.2.1.244: gateway 127.0.0.1
# netstat -r -n
Routing tables
Destination       Gateway           Flags     Refs      Use  Interface
127.0.0.1         127.0.0.1         UH           0   442246  lo0
192.2.1.244       127.0.0.1         UH           0        0  lo0
15.13.104.244     127.0.0.1         UH           0        0  lo0
192.2.1           192.2.1.244       U            0   572167  lan1
15                15.13.104.244     U            2   633081  lan0
```

In this case, connections to 15.13.104.244 will be routed through the loopback interface lo0. This interface does offer checksum offload. A quick test of loopback throughput using an HP 9000 Series 842 shows the performance improving by ~71%, from 28 to 48 mb/s!

E Q & A

Q Which releases of HP–UX support copy–avoidance?

A Copy avoidance requires HP–UX Release 9.0 or greater.

Q Which network interfaces support copy avoidance?

A Presently, the only network interfaces supporting both receive and send copy avoidance is the NIO FDDI (pn J2157A) interface. The Integrated FDDI interface supports receive copy avoidance only. Future network interfaces may support inbound and/or outbound copy avoidance.

Q Which network interfaces support Internet checksum offload?

A Internet checksum offload is supported by the NIO FDDI (pn J2157A), loopback, and Integrated FDDI interfaces. As with copy avoidance, future network interfaces may provide this feature.

Q Are Internet checksum offload and copy–avoidance supported by the EISA FDDI (pn J2156A) Interface?

A No.

Q Is Internet checksum offload supported on HP–UX 8.02 with the NIO FDDI card?

A No. While the NIO FDDI card is available on HP–UX 8.02, the software changes required to use its Internet checksum offload features are not.

Q Why do sending and receiving copy avoidance work differently?

A The semantics of sending and receiving packets require different implementations. The semantics of a send(2) call dictate that the application's data be preserved across the call and the technique employed is *copy–on–write* The semantics of a recv(2) or read(2) call dictate that the application's data be replaced by the call and *page remapping* is used.

Q What is copy–on–write?

A Copy–on–write is a technique whereby a duplicate reference to the page(s) holding the application's data is created for the Transport. If the application attempts to write to that data before the Transport is finished with it, a copy will be made to be used by the application.

 Revision 1.1, 02/09/93, Copyright Hewlett–Packard Company

Q What is page remapping?

A Page remapping is the swapping of pages between the application and the Transport.

Q Can I avoid copies with sends and receives which are smaller than a page?

A No. The implementation in 9.0 requires that only page–sized and larger quantities be used. The concept of sub–page copy avoidance is the topic of further study, and your opinions would be useful. Please feel free to send them to the email address perfbrief@hptnjar.cup.hp.com.

Q Does copy avoidance work with the write(2) system call?

A No. The code implementing the write(2) system call was not modified to include copy avoidance.

Q Does Internet checksum offload work with the write(2) system call?

A Yes. Support for Internet checksum offload did not require changes to any system calls.

Q Does copy avoidance work with UDP sockets?

A Yes, copy avoidance works with UDP, but it is not as easy to use as with TCP. While there is a way to ensure that applications do not prematurely reuse send buffers with TCP, there is no similar method for UDP as UDP sends will not block on windows. If your UDP application operates in a request/reply manner, you can use the receipt of the reply as the implicit signal that your request is no longer being held by the Transport. Receive copy avoidance will work just the same for UDP as for TCP in all cases.

Q Will copy avoidance be available on my 68K box?

A No. Copy avoidance will be available for PA–RISC based systems only, and then only those systems supporting network interfaces providing the necessary features.

Q Does the loopback interface on my 68K box offer checksum offload?

A No.