IMPLEMENTATION AND PERFORMANCE TESTING OF A SINGLE COPY DEVICE DRIVER FOR THE IBM APE25

---

by

Sripriya Bhikshesvaran

Submitted to

the graduate faculty

of the Department of Computer Science

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in

Computer Science

September 15, 1998

Clemson University

Clemson, SC 29634-1906

Abstract

The motivation for Asynchronous Transfer Mode (ATM) lies in the high bandwidth requirement for multimedia traffic. The bandwidth available may be reduced by factors that include network delays, limits imposed on the number of connections allowed on a particular link, and header overhead in each ATM cell. Additionally, if the software pathway from the application through the operating system to the ATM network interface is not carefully optimized, the bandwidth available may further decrease.

This project is an attempt to decrease software overhead introduced by the operating system during transmission or reception, in order to achieve a higher throughput and a lower load on the end-system CPU. This is accomplished by eliminating the quantity of time spent by the CPU in copying data between kernel and user address spaces. A description of the techniques attempted and the problems encountered is provided. A brief analysis of the performance results obtained is also furnished.

Acknowledgments

I thank my advisor, Dr Westall, and Dr Geist for providing me the opportunity to work on this project. Dr Westall was very patient through all our rambling discussions on the topic, and provided invaluable guidance when I reached the point of "what next ?".

I am grateful to my parents for allowing me the freedom to pursue my goals. My special thanks to Anu, Arun, and Ravi for their constant support. I wish to thank Deepak, and Kiran for standing beside me through the last few vital months. Special thanks to Karthick for the Latex templates as also the umpteen emails that goaded me on. Finally, a heartfelt thanks to all my friends at Oak Crest for the cheerful respite at the end of the day.

Contents

List of Figures

Chapter 1
Introduction

Asynchronous Transfer Mode (ATM) was motivated by the need for high bandwidth and low end-to-end delay demanded by multimedia traffic. It guarantees a fixed Quality of Service (QoS) to applications during connection establishment. The QoS defines the connection's throughput and delay characteristics. Although the network hardware guarantees a certain QoS, the design of the network adapter, the architecture of the end-system, and processing by the operating system reduce the actual throughput and increase the transfer delays observed at the application layer.

Our experiments use the IBM APE25 ATM adapter to perform transmission and reception at the ATM Layer and the ATM Adaptation Layer (AAL)[19]. The interface between transmitting and receiving applications, and the APE25 is a device driver, which is implemented as a loadable Linux module. On initialization, it allocates buffers in kernel memory to store data frames for transmission or reception. During transmission, frames are copied from the application's address space to these buffers by the driver, and subsequently transferred to adapter memory through a DMA transfer conducted by the adapter. For reception, the frames are transferred from adapter memory through the kernel buffers to the application's address space.

Throughput values observed are frequently much lower than that guaranteed by the network hardware. In order to reduce the processing by the operating system and thereby achieve a higher end-to-end throughput and a lower load on the end-system CPU, we eliminate the copy to/from kernel buffers, and transfer data directly between user space and adapter memory. We refer to this technique as "single copy", and study its effects on system performance.

The measures of performance examined are network throughput, system time, and elapsed time. The network throughput signifies the number of bits transmitted

in unit time. The system time is the quantity of CPU time spent by the application in kernel mode during transmission, while the elapsed time is the total time required to transmit the entire benchmark workload.

The remainder of this paper is organized as follows: Chapter 2 provides a background for the later chapters and discusses relevant work that has been conducted. Chapter 3 outlines the implementation of the two-copy driver. Chapter 4 describes the modifications and techniques to implement single-copy. Chapter 5 furnishes the performance results and their graphical representations. The conclusions are presented in Chapter 6. Chapter 7 provides a list of acronyms used in the paper.

Chapter 2
ATM and High Speed Networking

2.1 Overview

ATM is designed to provide high speed multimedia data transfer. It is a connection oriented, cell switching technology that supports a guaranteed Quality of Service (QoS).

Synchronous Transfer Modes (STM) divide the available bandwidth into time slots. As with ATM, a connection has to be established before any data transfer can occur. When a connection is established, a fixed time slot is assigned to the connection which remains reserved until the connection is broken down. A host that does not have any data to send during its time slot wastes that period of time since, no other host is allowed to transmit during that interval. These wasted slots reduce the available bandwidth during periods of peak bursty traffic.

In the Asynchronous Transfer Mode (ATM), a connection is known as a virtual circuit. The virtual circuit can be a manually configured Permanent Virtual Circuit (PVC), or a Switched Virtual Circuit (SVC) that is set up dynamically by an application through signaling. Each virtual circuit is identified by a Virtual Circuit Identifier (VCI). A group of virtual circuits forms a virtual path identified by a Virtual Path Identifier (VPI). Data is transmitted in fixed byte cells that consist of a 48 byte payload and a 5 byte header. The header contains the VPI and VCI values of the connection on which the cell is sent. Since each cell can be switched independently of other cells using these identifiers, there is no reserved slot for a connection as with STM[15].

An important factor that makes ATM most suitable for high speed networks is its guarantee for the Quality of Service. The QoS specifies the cell rate, cell loss ratio,

cell transfer delay, and cell delay variation[17]. Applications request a certain QoS during connection establishment. If the requested QoS cannot be guaranteed, the connection request may be refused. In some cases, 'option negotiation'[19] can occur whereby, the communicating hosts agree to support a lower QoS than that requested. Once the connection is set up, the QoS assigned is assured throughout the lifetime of the connection, notwithstanding fluctuations in network load. Such a guaranteed QoS is vital for real time multimedia traffic which can suffer some loss of data but cannot tolerate delay.

The use of ATM on low-end platforms such as personal computers has been investigated by Almesberger[3]. The design of the PC hardware, the ATM network adapter and the operating system introduce bottlenecks that increase transfer delays and reduce network throughput. The following sections discuss some of these problems and the solutions that have been proposed.

2.2 Adapter Design Issues

An ATM adapter typically implements the functions of the ATM layer and the ATM Adaptation Layer of the ATM protocol stack[19]. The ATM layer is responsible for the transmission and reception of cells. It generates cell headers at the transmitting end and extracts them at the receiving end. It establishes and releases virtual circuits and exercises flow control.

The ATM Adaptation Layer (AAL) consists of two sublayers: the Convergence Sublayer(CS) and the Segmentation and Reassembly (SAR) sublayer. The CS performs message framing and error detection. The SAR segments protocol data units (PDU) into cells at the transmitting end and reassembles them at the receiving end. Five standards for AAL have been defined to support the different traffic classes[19], depending on the timing (real-time / non-real-time), and bit rate (constant / variable) required. AALs 1-4 proposed by the International Telecommunications Union (ITU)

were found to be extremely complex. Hence the computer industry designed the Simple and Efficient Adaptation Layer (SEAL) which was standardized into AAL5[19]. It is now considered the *de facto* standard for Variable Bit Rate (VBR) traffic.

Early adapters performed the ATM layer and AAL functions either in software which significantly increased end-to-end delays, or in hardware which engendered complexity in the adapters. Modern adapters implement these functions in hardware while preserving simplicity. ATM adapter designs mainly fall in two categories[3]:

1. Buffering adapters

2. Just-in-time DMA Adapters

Buffering adapters store PDUs in onboard buffers. These PDUs are copied twice during the course of transmission or reception; once between user space and kernel buffers, and again between kernel buffers and adapter memory. Interrupts are generated during transmission initially, when the PDU has been copied to an onboard PDU buffer and later, after the last cell of that PDU has been sent. The host memory occupied by the PDU is released when the first interrupt is serviced. The second interrupt service routine wakes up processes that may be waiting for empty onboard PDU buffers. At the receiving end, the first interrupt is generated when the reassembled PDU is available in an onboard buffer. The interrupt service routine determines the address of the buffer in host memory to which the reassembled PDU should be transferred. The second interrupt is generated after the PDU has been copied to that buffer.

Disadvantages of these adapters include the need for expensive buffers onboard to store PDUs and the time consumed in servicing interrupts. Druschel[13] describes strategies to reduce the number of interrupts generated to less than one per PDU.

Just-in-time DMA adapters do not buffer PDUs in adapter memory. They are segmented from or reassembled into host memory directly. However, they are copied

twice as do buffering adapters. The number of interrupts is reduced to one per PDU. These are generated once the last cell of a PDU has been sent or received. These adapters must handle buffer queues that are allocated in host memory to store PDUs.

With DMA, large blocks of data can be transferred in a single bus transaction, thereby achieving transfer rates comparable to main memory and I/O bus bandwidths. Data transfer proceeds concurrently with processor activity, although contention for main memory access can induce processor stalls during periods of peak DMA traffic. DMA also introduces some complexity in adapter design. A scatter-gather capability in DMA-based I/O adapters is desirable for reducing memory traffic. It allows DMA transfer from/to non-contiguous physical page frames, thereby simplifying physical memory management.

The IBM APE25 ATM adapter that we use in our experiments, can be classified as a just-in-time DMA adapter. It implements AAL1 and AAL5 and also supports scatter-gather DMA transfers.

2.3 Host Architecture and Operating System Issues

Almesberger[3] identifies memory bandwidth as a major bottleneck for network throughput when using ATM on low-end architectures such as personal computers. PCs, as compared to workstations, have lesser CPU speeds and lower memory and I/O bus bandwidths[3]. Also, CPU speeds have increased at a higher rate than memory bandwidths. Hennessy and Patterson[14] report a 50 - 100% improvement in processor performance annually since 1985 as compared to only a 7% improvement in memory performance. This disparity can lead to memory bottlenecks while transferring data[12].

Transfer of data requires a memory-to-memory as well as a memory-to-adapter copy. Druschel[13] states that "software data copying as a means of transferring data across domain boundaries exacerbates the memory bottleneck problem". The

domains refer to user and kernel address spaces. Druschel explains that these "cross-domain" transfers, which constitute the memory-to-memory copy across memory protection barriers are time consuming.

It is therefore desirable to avoid multiple copying of data over the memory bus to reduce the memory bottleneck. A number of techniques that use virtual memory have been proposed to avoid the memory-to-memory copy[13].

2.4 Single Copy

Single copy avoids the memory-to-memory copy between kernel buffers and user space, by transferring data directly between user space and adapter memory. However, it is essential to determine whether the single-copy procedures that substitute the memory-to-memory transfer will actually reduce the copy overhead.

Single copy requires locking the process pages of the data to be transmitted in physical memory, translating their virtual addresses to physical addresses, forming a scatter-gather vector of these addresses, conducting the data transfer and unlocking the pages when the transfer is complete[4].

At the receiving end, queues of buffers are maintained in the kernel memory of the host to store the reassembled PDUs. The buffers are mapped to the process space of the receiving application, and should be page-aligned to reduce the overhead of complex address translations that need to be performed each time a PDU is received[3]. The receiving application reads the PDUs directly from the mapped buffers since they lie within its address space.

Almesberger[3] remarks that single-copy may offer only marginal improvements in performance; in some cases, the performance may even deteriorate. His experiments show that single-copy gives best results for frames of size larger than 4K. For lesser frame sizes, the locking of pages in physical memory required for single-copy, consumes more time than copying data. This paper presents experiments with

single-copy that were conducted with the Linux device driver for the IBM APE25 and the reports on the performance results observed.

Chapter 3
The Two-Copy Driver

The two-copy driver has been developed as a module in Linux operating system version 2.0.25[20] by Drs James Westall and Robert Geist of Clemson University. A character device interface to the driver using *ioctl* function calls is used in our experiments.

Two Gateway 2000 personal computers - a 180MHz Pentium Pro and a 100 MHz Pentium - connected to the IBM 8285 Turboways ATM switch constitute the hardware platform for development and testing. The interface to the network is provided by the IBM ATM APE25 25Mbps cards. The Peripheral Component Interconnect (PCI) bus is used for DMA transfers to and from the APE25.

The device driver acts as an interface between transmitting or receiving applications and the APE25. A description of the two-copy version is provided below as an aid to understanding the modifications that were made to implement single copy.

3.1 The Transmit Mechanism

The transmitting application selects a specific logical channel (LC) identified by a (VPI, VCI) descriptor to send data. The descriptor and the virtual address of the transmitting application's data are passed to the device driver through an *ioctl()* call to transmit the data.

The driver maintains a free descriptor list (FDL) consisting of transmit frame descriptors (TFD). Each TFD is bound to a transmit buffer that is allocated in kernel memory during driver initialization. Every TFD also has a data buffer descriptor (DBD) that contains the physical address and the length of the data to be transmitted *(Fig 3.1)*.

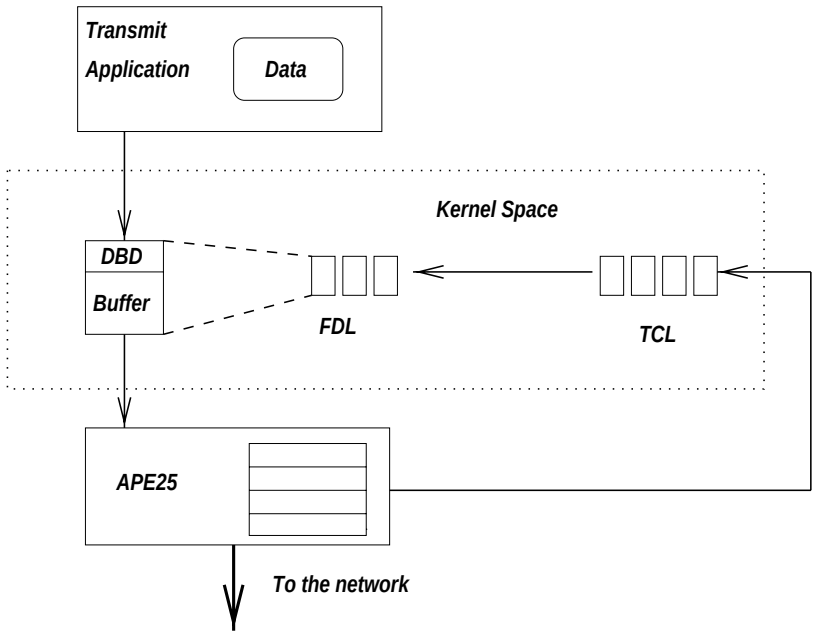

Figure 3.1 Data Flow during Transmission

When the driver receives a request to send data, it removes a TFD from the FDL and copies the data from the virtual address specified by the application to the transmit buffer bound to the TFD. This copy from user space to kernel memory is performed using the *memcpy_fromfs()*[8] kernel function. The physical address of the transmit buffer and the length of the data are stored in the DBD. The address of the TFD is written to APE25 specific registers[20]. The APE25 transfers the data frame from the transmit buffer to cell buffers in its own memory through DMA.

Once the frame has been transmitted, the APE25 adds the TFD to the transmit complete list (TCL), and generates a transmit complete interrupt. The interrupt service routine removes the TFD from the TCL, and adds it to the FDL . The TFD can now be reused for subsequent transmit requests.

3.2 The Receive Mechanism

The device driver maintains a list of receive buffer headers (RBHs) called the free RBH list (FRBHL). Each RBH is bound to a receive buffer allocated in kernel

memory during driver initialization. After the APE25 receives cells from the network and reassembles them into a frame, it removes an RBH from the FRBHL, copies the reassembled frame to the receive buffer bound to it, adds the RBH to the Receive Ready List (RRL) for that LC, and generates a receive interrupt[20]. The receive interrupt service routine removes the RBH from the RRL, and adds it to the LC receive buffer list (LCRBL) for that LC *(Fig 3.2)*.

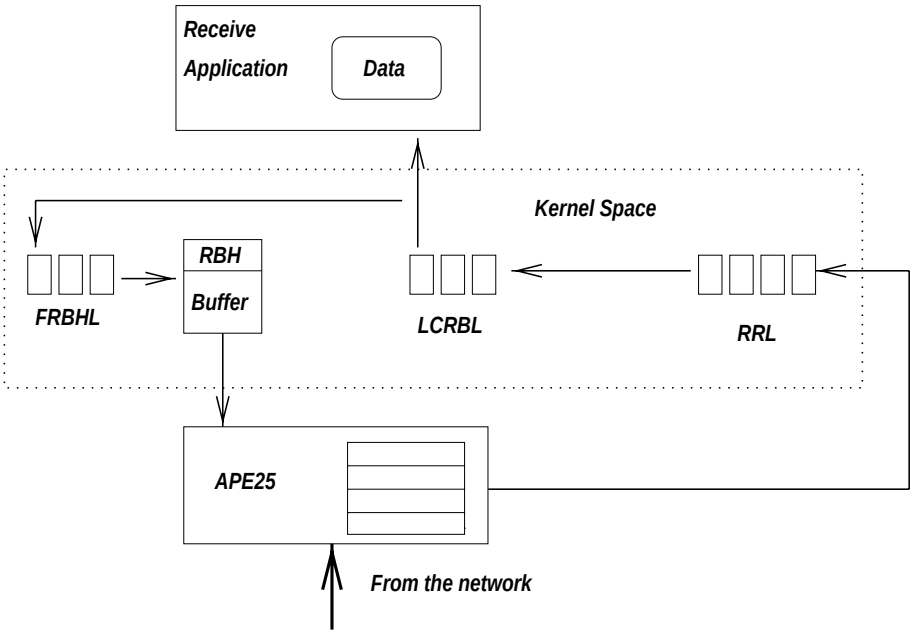

Figure 3.2 Data Flow during Reception

When the driver receives a request from an application to read data from a specific LC, it checks the corresponding LCRBL to determine whether any frames have been received. If there are any received frames, these are copied from the LCRBL to a buffer specified by the application. This is a kernel memory to user space copy performed using the *memcpy_tofs()* kernel function[8]. The RBHs and the corresponding buffers that have been read are removed from the LCRBL, and returned to the FRBHL to be reused by the APE25.

The transfers between user space and kernel memory that are performed by the two-copy driver during transmission and reception are avoided in single-copy.

Chapter 4
The Single Copy Driver

Single copy aims to reduce the processing performed by the operating system during transmission and reception by eliminating the data copy between user space and kernel buffers. The techniques implemented are described in this section. Applications can access the single-copy driver only through the character device interface.

4.1 The Transmit Mechanisms

4.1.1 Transmission using no pre-allocated buffers

With this technique, the driver performs two important functions when it gets a request from an application to transmit data - locking the data in physical memory[3][4], and binding the locked data to a TFD. The transmit buffers allocated in kernel memory during initialization of the two-copy driver are not used.

Locking Data in Physical Memory

The kernel function *lock_user()*[6], defined in */usr/src/linux/net/atm/mmuio.c*, is used by the driver to lock the data that has to be transmitted. The pseudo code for *lock_user()* is given below:

```
int lock_user(unsigned long start, unsigned long size, int iov_max,
struct iovec *iov)
{
 'end' = 'start' + 'size' ;
 find vm_area_struct for 'current' process for 'start' ;

 get page directory offset for 'start';
 use page directory offset, get page mid directory ;
 use page mid-directory, get page table entry ;
```

```
do
{
 if ( no page table entry for ‘start’ )
 {
   page fault for the page with ‘start’ ;
 }
 use page table entry, get page for ‘start’ ;
 reserve page ;
 write protect the page ;
 calculate the physical address of ‘start’ ;
 record the physical address and length in ‘iov’ ;
 modify ‘start’ to next virtual address to be locked ;
 resolve page directory, page mid-directory and page
 table entry for ‘start’;

} while ‘start’ < ‘end’;
}
```

The parameter ‘start’ specifies the virtual address of the data within the application's address space. The length of the data is specified by ‘size’ while ‘iov_max’ indicates the number of pages in physical memory over which the data can extend. This is calculated as follows:

$$iov\_max = size/4096 + 2$$

The two is added to allow for extra pages that may be occupied by a few bytes at the beginning or the end. The array of *iovec* structures[18], pointed to by ‘iov’ is used to record the *(physical address, length)* pairs for each locked range.

In the architecture independent memory management model adopted in Linux, the linear address is split into four parts[8]. The first part is an offset into the page directory table and locates the page mid-directory entry. The second part is an offset into the page mid-directory and retrieves the page table entry. The third part is the offset into the page table and gives the appropriate page. The fourth part is an offset into the page and identifies the address from which the data begins.

In the *lock_user()* function, the page directory, the page mid-directory, the page table and the offset within the specific page for ‘start’ are resolved. The physical

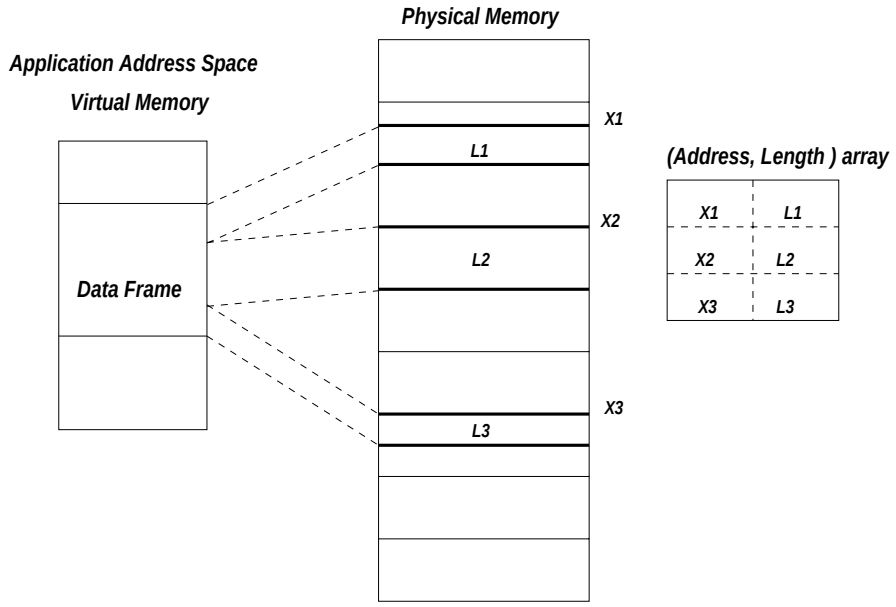

Figure 4.1 Locking and Generating a Frame's Physical Address Ranges

address of the location specified by 'start' is calculated. The page is locked in physical memory by write-protecting it using the kernel function *pte_wrprotect()*[8]. The address and the length of the data locked in that page are recorded in the *iovec* structure pointed to by 'iov'. The data can spread over multiple pages, in which case, 'start' is modified in each iteration through the *while* loop to point to the start of the next page that has to be locked. The *while* loop is executed until all the data has been locked and the *(address, length)* pairs have been recorded *(Fig 4.1)*. The array of *iovec* structures now contains all the locked physical ranges for the frame. The *lock_user()* function returns the number of locked ranges for the frame.

Binding the locked data to a Transmit Frame Descriptor (TFD)

The TFD is a standard header recognized by the APE25 *(Fig 4.2)*. The *TCL Next TFD* field is used by the driver to traverse the transmit complete list (TCL). The *LCI* indicates the logical channel over which the frame should be transmitted. The *BufCount* field indicates the number of buffers bound to this TFD. In the two-copy driver, there is one buffer bound to each TFD during driver initialization. In

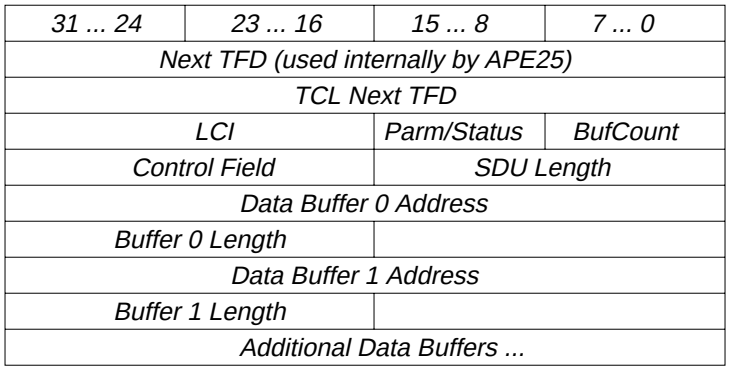

Figure 4.2 Layout of Transmit Frame Descriptor

the single copy driver, *BufCount* indicates the number of ranges locked in physical memory for that frame. The *SDU length* field specifies the length of the frame.

The *Data Buffer Descriptor* (DBD) structure contains the *Data Buffer Address* and *Buffer Length* fields. The two-copy driver has one DBD that holds the address and length of the buffer that is bound to the TFD. The number of DBDs in the single-copy driver is fixed during driver initialization. But only as many DBDs as the number of locked ranges for a frame are actually filled with the *(address, length)* pairs. A DBD can hold at most the address of one physical page. Hence, the number of DBDs determines the maximum frame size that can be sent.

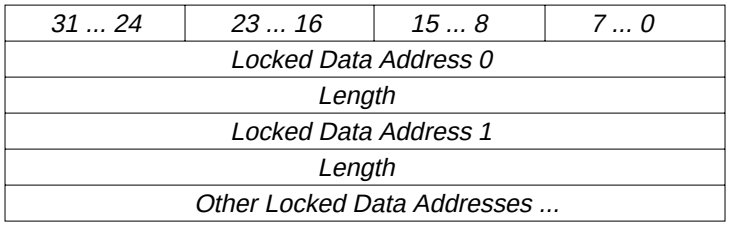

Figure 4.3 Addition to Transmit Frame Descriptor

The array of *iovec* structures shown in *(Fig 4.3)* was added to the TFD. The *lock_user()* function fills the array with the locked *(address, length)* ranges. The driver also stores the locked ranges in the DBDs of the TFD. This information is

redundant. But the APE25 requires the locked ranges to be stored in the DBDs while the *unlock_user()* function needs the information to be stored in an *iovec* array. Hence the duplication cannot be avoided.

After these two operations are performed, the driver writes the address of the TFD to APE25 specific device registers. The APE25 notices that there is a frame to be transmitted and retrieves the data from the locked memory.

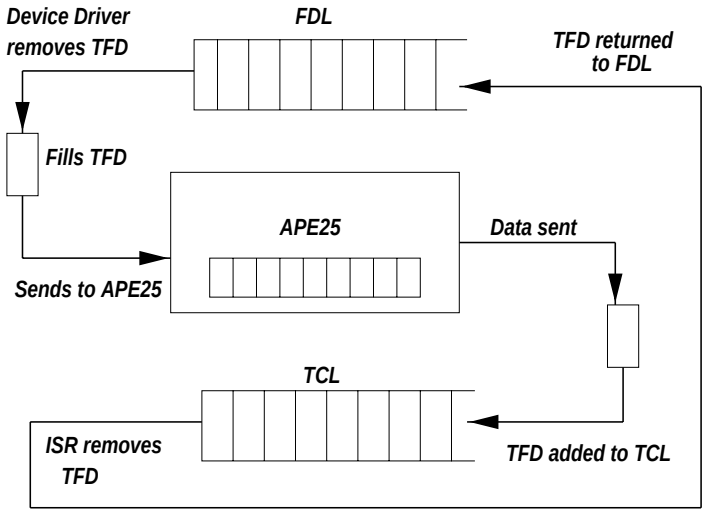

Figure 4.4 Handling Transmit File Descriptors

After the APE25 transmits the frame, it adds the TFD to the TCL and generates a transmit complete interrupt. The interrupt service routine removes the TFD from the TCL *(Fig 4.4)* and calls the kernel function *unlock_user()*. This function retrieves the locked ranges from the array of *iovec* structures in the TFD to unlock the memory reserved for that frame. The TFD is then returned to the FDL for reuse.

4.1.2 Transmission using pre-allocated locked buffers

This technique avoids the overhead of locking and unlocking data in physical memory for every transmitted frame. The transmitting application allocates a buffer within its address space, and makes an *ioctl* call to the driver to lock the buffer.

The driver locks the buffer in physical memory using the *lock_user()* kernel function. The array of *(address, length)* pairs returned by the *lock_user()* function is stored in the (*Physical Data Address, Data length*) array in the *Virtual Address to Physical Address Ranges Map (Fig 4.5)*. The virtual address of the locked buffer is stored in the *Virtual Address* field. The *Size* field indicates the length of the buffer while the *Vector Length* is the number of locked ranges. A table of these maps is maintained with an entry for each locked buffer.

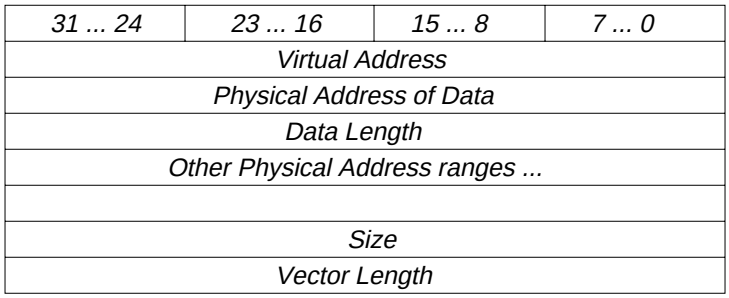

Figure 4.5 The Virtual Address to Physical Address Ranges Map

When an application needs to transmit data from its locked buffer, it passes the buffer's virtual address, the offset of the data within the buffer, and the length of the data to the driver through an *ioctl* call. The driver searches the table of maps for the entry corresponding to the virtual address given, in order to retrieve the physical address ranges locked for that buffer. The physical address of the start of the frame is found using the physical address ranges and the offset. Using this address and the length, the locked address ranges enclosed by the data are identified, and recorded in the DBDs of a TFD. The address of the TFD is then written to the APE25's registers for the frame to be transmitted.

When the frame has been transmitted, the TFD is removed from the TCL and added to the FDL *(Fig 4.4)*, but the memory locked for that frame is not unlocked since it is still a part of the application's buffer. Once the application has sent all the data, it requests the driver to unlock the entire buffer.

4.2 The Receive Mechanism

The receive mechanism consists of two operations; mapping receive buffers allocated during driver initialization to the receiving application's address space, and reading data from the mapped buffers.

4.2.1 Mapping receive buffers to application address space

The kernel function *do_mmap()*[8] is used by the driver to map the receive buffer pool to the address space of the receiving application that requests the mapping. The function requires the buffer pool address to be page-aligned. Hence from the memory allocated for the buffer pool using *kmalloc()*, the first page is discarded to make the address page-aligned. *do_mmap()* returns a virtual address that marks the beginning of the buffer pool. This address acts as a handle to the kernel buffer pool from which the application can directly read data received from the network.

4.2.2 Reading data from the mapped buffers

The application reads data from the network by specifying the virtual address of the beginning of the buffer pool. This virtual address is used to calculate the virtual address of the actual buffer within the pool that contains the incoming data. This calculated virtual address is returned to the application which directly reads the data from the kernel buffers.

The APE25 reads cells from the network and reassembles them into frames. There exists a receive free list (RFL) consisting of receive buffer headers (RBH). Each RBH is bound to a receive buffer from the allocated pool, by storing the physical address of the buffer in the *Data Buffer Address* field in the RBH *(Fig 4.6)*. The *Buffer Length* field indicates the size of the buffer. An RBH does not remain bound to the same buffer throughout its lifetime but cycles through the buffer pool, binding with subsequent buffers after each read.

31 ... 24	23 ... 16	15 ... 8	7 ... 0
Data Buffer Address			
Next RBH Address			
Buffer Length		Status	
LCI		Control	
SDU Length		OAM / CLP0 / TUC	

Figure 4.6 Layout of the Receive Buffer Header

The APE25 removes an RBH from the RFL and copies a reassembled frame to the receive buffer that is bound to it *(Fig 4.7)*. The length of the frame is stored in the *SDU Length* field of the RBH. The *LCI* field is filled with the number of the LC from which the frame was received. The RBH is added to the receive ready list (RRL) of this LC, and a receive interrupt is generated.

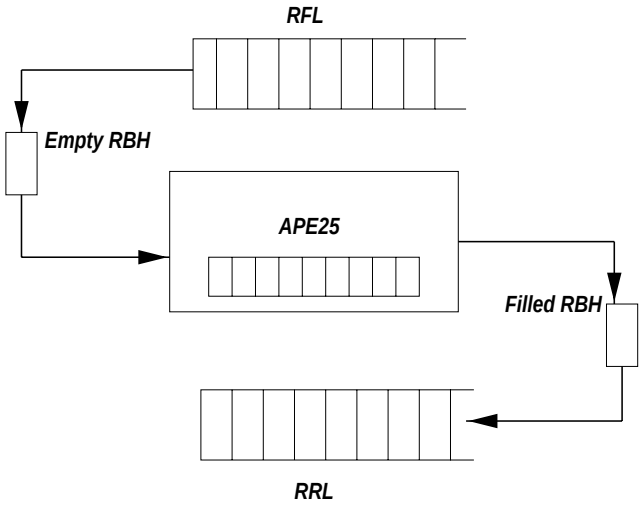

Figure 4.7 Handling Receive Buffer Headers

The interrupt service routine removes all RBHs from the RRL and adds them to the LC receive buffer list (LCRBL) for the appropriate LC. The first RBH on the RRL is a dummy that does not have any buffer attached to it. The buffer that is attached to the last RBH is bound to the dummy. The dummy is removed from the RRL and added to the LCRBL. The last RBH on the RRL does not have any buffer bound to it, and now becomes the dummy.

When a receiving application requests the driver to read data from a particular LC, the driver checks the LCRBL of that LC for any received data. The LCRBL can be in two states. If this is the first time the driver is reading data after it has been installed, the LCRBL contains a dummy RBH and possibly, one or more RBHs that have data frames to be read *(Fig 4.8)*.

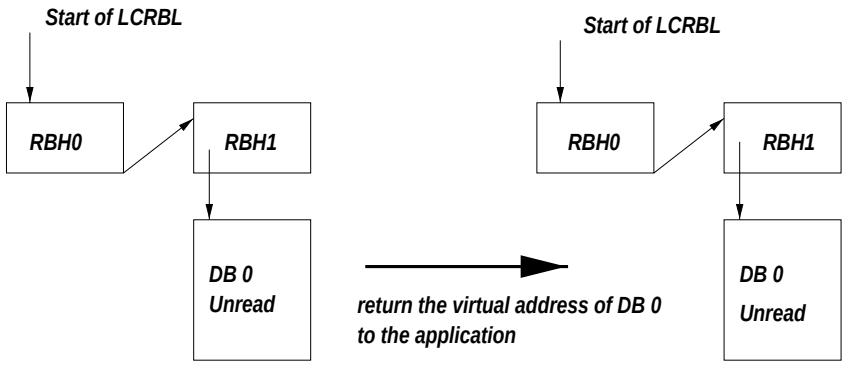

Figure 4.8 State of LCRBL when reading for the first time

The offset within the receive buffer pool, of the buffer bound to the first of these RBHs, is calculated. This offset is added to the virtual address provided by the application. The result is a virtual address within the application's address space. This virtual address is a mapping of the kernel buffer that contains the data. This address is returned to the application. The RBH along with the data buffer remains on the LCRBL since the application has not yet read the data.

The second state occurs when subsequent requests to read data are made by the application *(Fig 4.9)*. The LCRBL now has a dummy, an RBH with a data buffer whose address was returned during the previous request, and maybe another RBH with data to be read. The RBH with the buffer that has been read is made the dummy. Its receive buffer is bound to the dummy at the front of the queue and the *(RBH, buffer)* pair is returned to the receive free list (RFL). This procedure occurs on any other read request except the first. The virtual address of the buffer

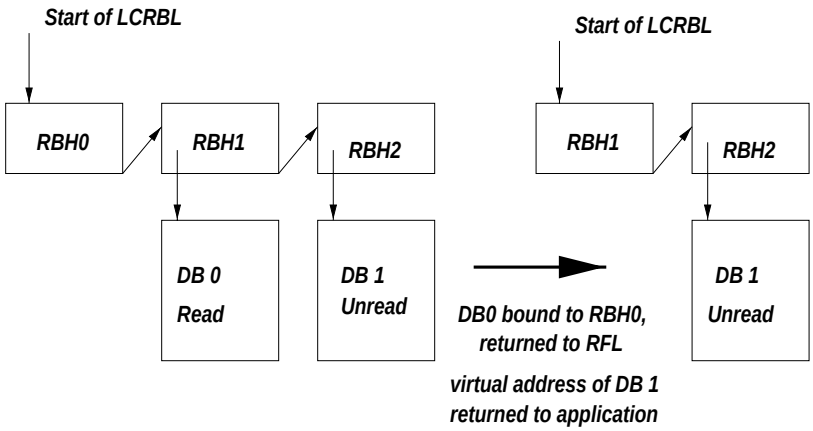

Figure 4.9 State of LCRBL after the first read

that actually has data to be read is calculated as in the first case. This address is returned to the receiving application.

There is a drawback with this receive technique. The kernel buffers are mapped to an application's address space before the application attempts to read any data. If there is more than one application that performs this mapping, then the buffers are visible to all these applications. Hence an application can read any data; even those that were received for another application.

The receive technique described here was however, implemented for the sake of completeness. Performance was not measured at the receiving end.

Chapter 5
Performance Measures and Results

The tests conducted to obtain measures of performance focused on throughput (in Mbps), elapsed time (the total time spent in transmitting the workload, in seconds), and system time (the time spent by the CPU in kernel mode during transmission, in seconds). Two Gateway 2000 PCs, *atm* and *chattooga* were used to perform the experiments. *atm* was used as the sender while *chattooga* was used as the receiver. Measurements were made on *atm* which is the faster machine. A 50 Mbyte load was sent in frame sizes ranging from 136 to 4036 bytes with 100 byte increments for all tests. Tests to measure the number of cycles that the functions *lock_user()* and *memcpy_fromfs()* consume were conducted, with frame sizes ranging from 520 to 7520 bytes with increments of 500 bytes.

5.1 Comparison of Two Copy and Single Copy with no pre-allocated buffers (SCDA)

The single copy driver (with no pre-allocated buffers), called SCDA in this discussion, was expected to perform better than the two-copy driver in terms of throughput and system time. Since the CPU does not have to perform time-consuming data copying between user space and kernel buffers, the amount of system time was expected to be reduced. The elapsed time was also expected to decrease, resulting in higher throughput.

The tests conducted gave results to the contrary. The system time actually increased for SCDA *(Fig 5.1, Table 5.1)*. The elapsed time for SCDA was much greater than that of the two-copy driver for small frames of sizes, 136, 236, and 336 bytes. As frame size increases, the elapsed time for SCDA reduces considerably but is still larger than that of the two-copy driver *(Fig 5.2, Table 5.2)*. Due to the greater

elapsed time for small frames, the throughput for these frames sizes (136, 236, and 336 bytes) is less than two-copy *(Fig 5.3, Table 5.3)*. As the elapsed time reduces for larger frames, the throughput is comparable, yet slightly less than two-copy.

The single-copy driver calls the *lock_user()* function instead of the *memcpy_fromfs()* function used by the two-copy driver. These functions are invoked each time a frame is transmitted. Hence, for single-copy to be an effective optimization, the locking must consume less time than copying.

The Pentium processor's Read Time Stamp Counter (RDTSC) instruction[16], was used to measure the actual number of clock cycles that these function calls consumed *(Fig 5.4, Table 5.4)*. Understandably, the *memcpy_fromfs()* kernel function itself consumes more cycles as the size of the frame increases. The *lock_user()* function though, displays a smaller increase in the number of cycles consumed with increase in frame size. SCDA then should perform better than the two-copy driver in terms of system time and throughput. However, with SCDA the memory that is locked for each transmit is unlocked using the *unlock_user()* function when transmission is completed. The service routine for the transmit complete interrupt which performs this unlocking may be executed when processing returns to user mode after a transmission. Hence the number of cycles that *unlock_user()* consumes also gets accumulated to the system time and elapsed time measurements. Tests showed that unlocking consumes as many cycles as locking *(Fig 5.4, Table 5.4)*. Locking and unlocking together consume more time than copying, even for large frames.

5.2 Comparison of Two Copy and the Single Copy with pre-allocated locked buffers (SCDB & SCDC)

The single copy driver with pre-allocated locked buffers (SCDB) avoids the overhead of locking and unlocking memory for each data frame that is transmitted. A single large buffer allocated by the application is locked in memory before any data can be transmitted, and unlocked only after all the data has been sent. SCDB gave

smaller system time than the two-copy driver since it reduces the locking and unlocking overhead in addition to avoiding the copy *(Fig 5.1, Table 5.1)*. Elapsed time remained higher than the two-copy driver *(Fig 5.2, Table 5.2)*, resulting in lower throughput *(Fig 5.3, Table 5.3)*.

With SCDB, the physical address of each frame that is transmitted from the locked buffer has to be calculated from the offset and length passed to the transmit call. We further modified the driver so that it avoids this address calculation overhead by always sending frames from the start of a 4Kbyte page. The transmitting application program allocates a buffer of size 8Kbytes. When this buffer is locked in physical memory, it extends over three physical pages, of which the first has a few bytes, the second has 4K bytes and the third contains the remaining bytes. An offset of zero is passed to the transmit call. The modified driver called SCDC here, uses this zero offset to transmit all frames from the beginning of the second page. System time reduced further as compared to two-copy *(Fig 5.5, Table 5.5)*. Elapsed time and throughput were the same as those of the two-copy driver. This is understandable as SCDC does not differ from the two-copy driver except that the latter performs the crucial user space to kernel copy.

5.3 Performance of the Scatter-Gather feature of the APE25

The large elapsed time and low throughput demonstrated by SCDA and SCDB could not be attributed to the single-copy driver after the optimizations made. The other factor that could have an effect on these measures was the functioning of the network adapter itself. Hence, tests were conducted to observe the performance of the scatter-gather facility provided by the APE25. Each frame was divided into equal chunks distributed over the 4Kbyte page that SCDC uses. The number of chunks was varied over the range 1, 2, 4 and 8. For frames of the same size, the elapsed time increased as the number of chunks increased from 1 to 8 *(Fig 5.6, Table 5.6)*, which indicated that the APE25 actually becomes slower as the number of *(address, length)*

pairs given to it increases. Since the elapsed time increases, the throughput becomes lower for frames of the same size which only differ in that they are distributed over the 4Kbyte page.

Hence, we finally conclude that the APE25 consumes more time for scatter-gather transmissions. The single-copy optimizations give larger elapsed time and smaller throughput values while performing such transmissions, although system load reduces significantly.

Frame Size	Two Copy	SCDA	SCDB
136	8.79	25.30	7.88
336	3.40	6.88	2.88
536	2.13	4.27	1.99
736	1.68	3.22	1.30
936	1.42	2.47	1.10
1136	1.16	2.07	1.02
1336	1.03	1.81	0.86
1536	1.00	1.51	0.63
1736	0.93	1.38	0.59
1936	0.85	1.29	0.60
2136	0.87	1.25	0.46
2336	0.75	1.07	0.38
2536	0.71	1.02	0.40
2736	0.69	0.86	0.41
2936	0.62	0.91	0.35
3136	0.59	0.87	0.39
3336	0.56	0.75	0.35
3536	0.57	0.75	0.31
3736	0.53	0.76	0.27
3936	0.58	0.71	0.39

Table 5.1 System times for Two Copy, SCDA and SCDB

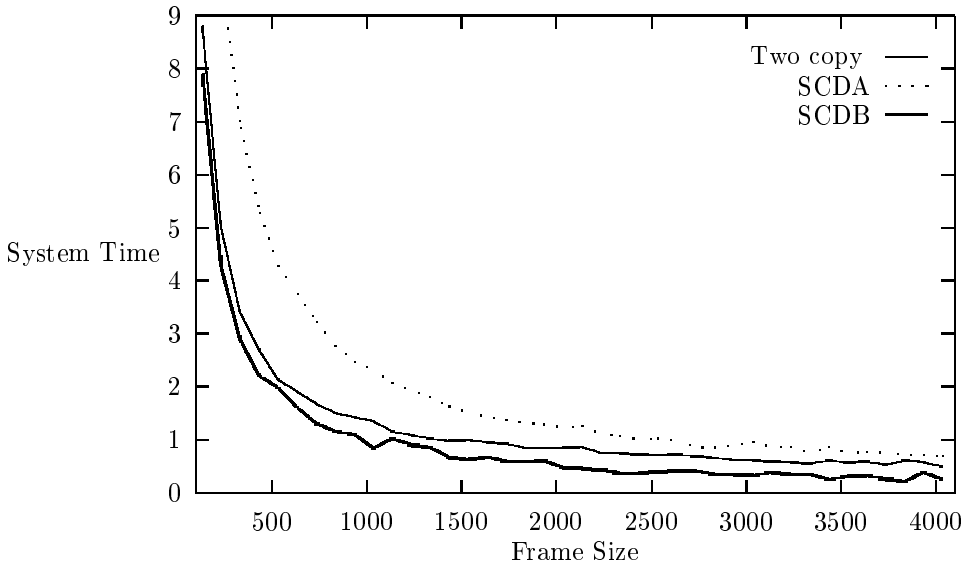

Figure 5.1 Comparison of System time - Two copy, SCDA, and SCDB

Frame Size	Two Copy	SCDA	SCDB
136	24.63	26.70	24.73
336	21.09	21.08	21.46
536	19.70	19.70	19.93
736	18.93	18.93	19.12
936	18.50	18.50	18.64
1136	18.25	18.24	18.33
1336	18.03	18.01	18.11
1536	18.36	18.34	18.42
1736	18.24	18.23	18.31
1936	18.12	18.10	18.15
2136	17.99	17.98	18.05
2336	17.90	17.89	17.95
2536	17.80	17.79	17.86
2736	18.00	17.99	18.04
2936	17.95	18.00	17.99
3136	17.87	17.89	17.92
3336	17.79	17.83	17.85
3536	17.75	17.77	17.80
3736	17.69	17.72	17.74
3936	17.83	17.86	17.88

Table 5.2 Elapsed times for Two Copy, SCDA and SCDB

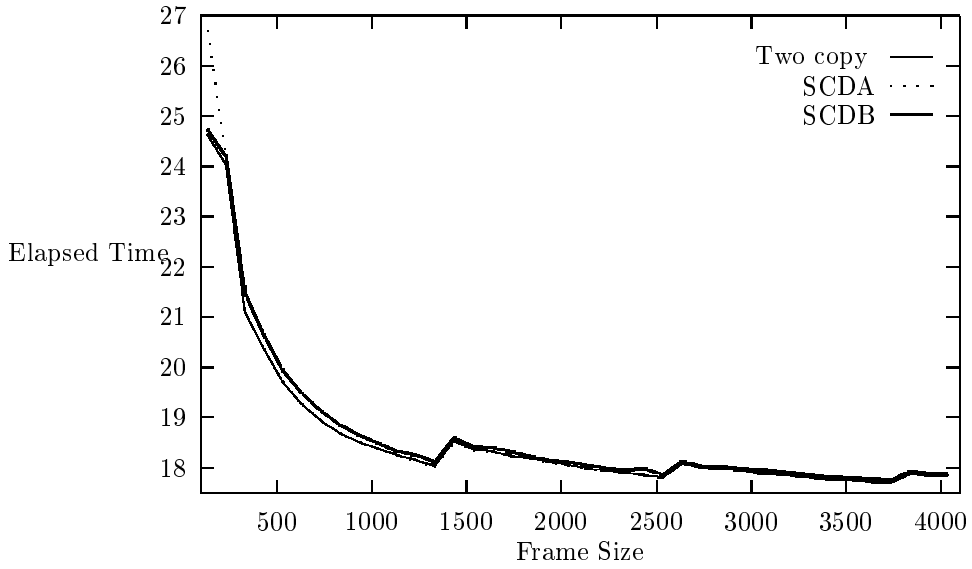

Figure 5.2 Comparison of Elapsed Time - Two Copy, SCDA, SCDB

Frame Size	Two Copy	SCDA	SCDB
136	16.243	14.984	16.172
336	18.964	18.975	18.639
536	20.305	20.305	20.066
736	21.133	21.130	20.921
936	21.620	21.625	21.457
1136	21.915	21.928	21.827
1336	22.181	22.214	22.087
1536	21.786	21.816	21.712
1736	21.932	21.944	21.849
1936	22.074	22.095	22.039
2136	22.228	22.241	22.161
2336	22.344	22.365	22.287
2536	22.470	22.485	22.399
2736	22.226	22.238	22.177
2936	22.286	22.217	22.232
3136	22.387	22.364	22.320
3336	22.479	22.440	22.406
3536	22.538	22.505	22.478
3736	22.618	22.579	22.542
3936	22.434	22.398	22.366

Table 5.3 Throughput for Two Copy, SCDA and SCDB

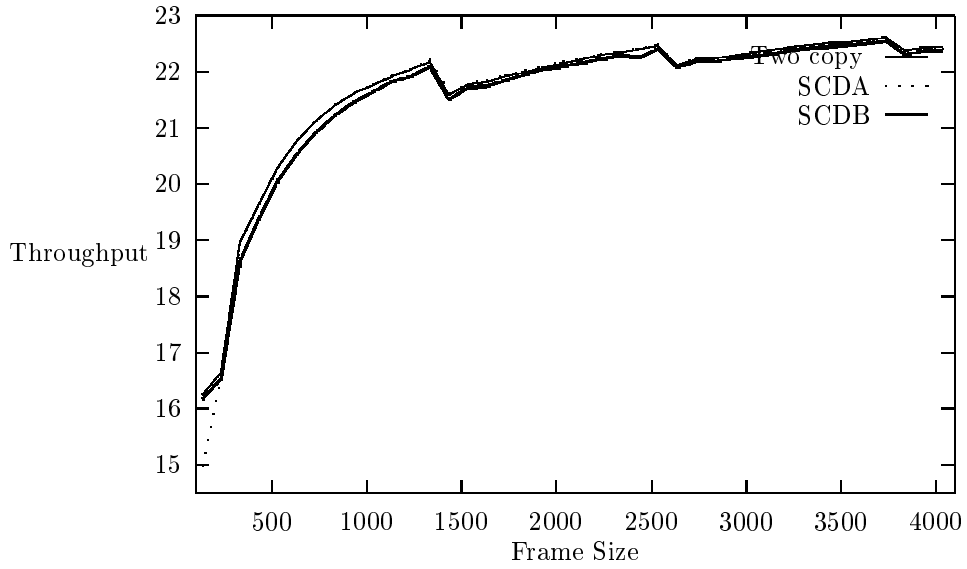

Figure 5.3 Comparison of Throughput - Two Copy, SCDA, and SCDB

Frame Size	lock_user()	memcpy_fromfs()	unlock_user()
520	12398	12122	11762
1020	12537	12502	12317
1520	12676	13107	12458
2020	12813	13818	12550
2520	12467	14342	12720
3020	12893	14635	12808
3520	12485	15288	13078
4020	12667	16113	13270
4520	12712	16498	13314
5020	12993	17079	13429
5520	12941	17444	13500
6020	13007	17842	13574
6520	13063	18475	13687
7020	13282	19468	13755
7520	13205	22372	13932

Table 5.4 Comparison of the Clock Cycles for lock_user() and memcpy_fromfs()

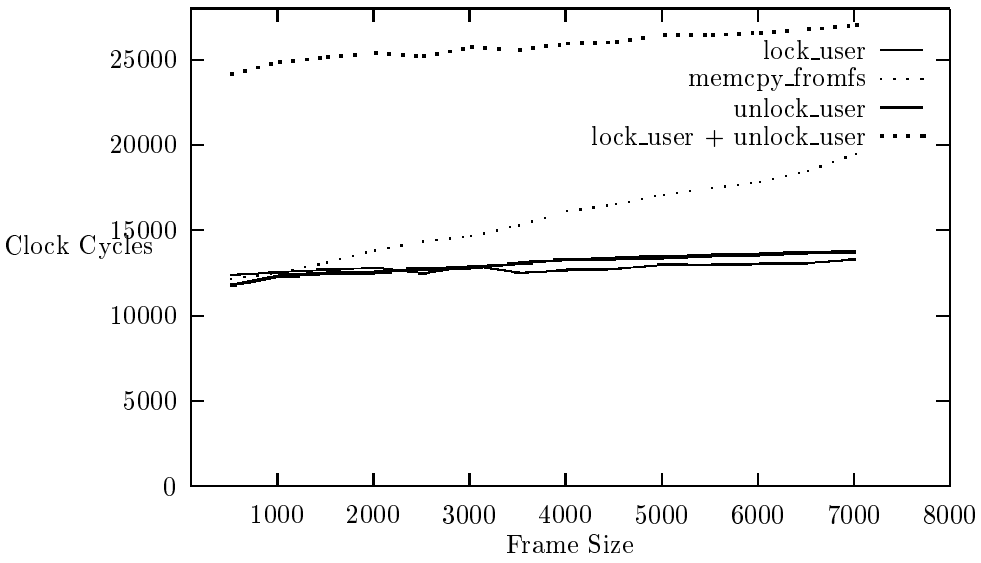

Figure 5.4 Comparison of the Clock Cycles - lock_user() and memcpy_fromfs()

Frame Size	Two Copy	SCDC
136	8.79	8.18
336	3.40	3.57
536	2.13	1.97
736	1.68	1.47
936	1.42	1.15
1136	1.16	1.05
1336	1.03	0.67
1536	1.00	0.65
1736	0.93	0.69
1936	0.85	0.46
2136	0.87	0.40
2336	0.75	0.36
2536	0.71	0.30
2736	0.69	0.33
2936	0.62	0.35
3136	0.59	0.30
3336	0.56	0.23
3536	0.57	0.25
3736	0.53	0.23
3936	0.58	0.22

Table 5.5 System Time for Two Copy, and SCDC (using 4K buffer)

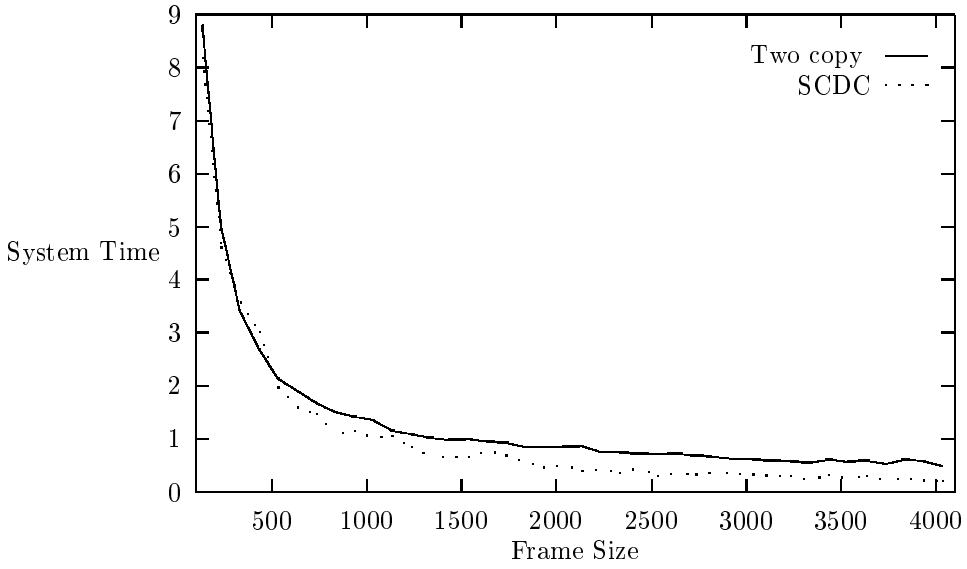

Figure 5.5 Comparison of System Time - Two Copy and SCDC (using 4K buffer)

Frame Size	Vector Size 1	Vector Size 2	Vector Size 3	Vector Size 4
136	24.79	26.85	30.93	37.18
336	21.50	21.98	23.04	26.51
536	20.10	20.02	20.65	22.81
736	19.13	19.18	19.45	20.15
936	18.69	18.74	18.90	19.32
1136	18.37	18.42	18.51	18.91
1336	18.12	18.17	18.27	18.54
1536	18.43	18.43	18.43	18.46
1736	18.31	18.33	18.32	18.40
1936	18.16	18.18	18.21	18.17
2136	18.04	18.06	18.06	18.18
2336	17.94	17.96	17.97	17.99
2536	17.85	17.86	17.87	17.90
2736	18.05	18.05	18.07	18.11
2936	18.00	18.00	18.02	18.07
3136	17.93	17.93	17.93	17.97
3336	17.88	17.89	17.92	17.97
3536	17.82	17.84	17.89	17.85
3736	17.76	17.77	17.80	17.80
3936	17.90	17.90	17.90	17.92

Table 5.6 Comparison of Elapsed Times with various Vector Sizes

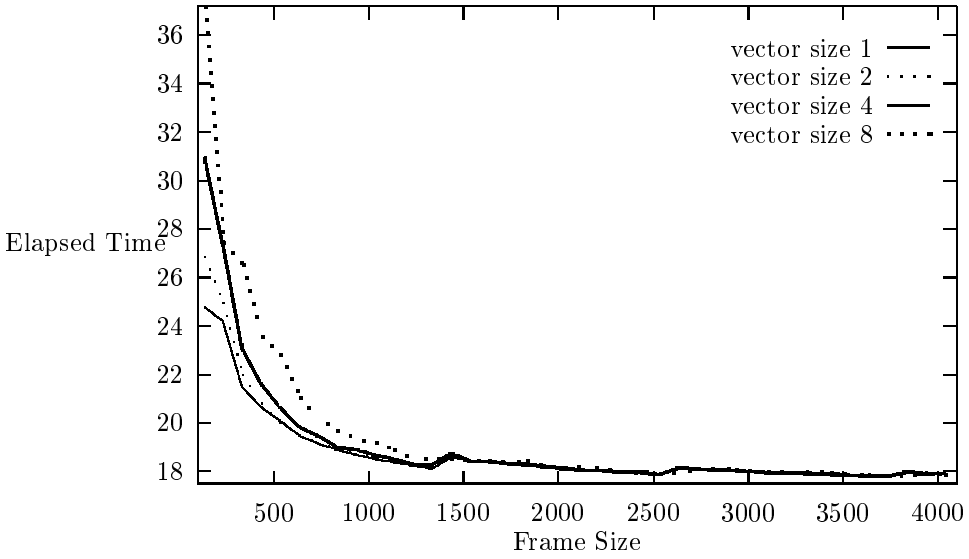

Figure 5.6 Performance of Scatter Gather provided by APE25

Chapter 6
Conclusions

The device driver for the IBM APE25 was modified to implement single copy transmission and reception of data. Tests were conducted to compare the performance of the single-copy driver with the two-copy driver. We observed the following behavior:

1. Single-copy experiments that locked and unlocked physical memory for each frame that was transmitted produced greater system time than two-copy. However, those tests that locked memory prior to transmission and released it after all the frames were transmitted displayed a significant reduction in system time.

2. Elapsed time was much greater for frames of small sizes with the single-copy driver than with the two-copy driver. As the frame size increased, the elapsed time decreased but still remained marginally greater than the two-copy driver. The throughput was, hence always less than the two-copy driver.

3. Scatter-gather performance analysis of the APE25 revealed that elapsed time increased as the size of the vector grew, thereby reducing the throughput.

Finally, we conclude that though the pathway from the application through the operating system to the network interface card was optimized, the characteristics of the adapter itself played a significant role in determining the throughput and delay. However, single-copy might be effective in improving performance if used with higher speed adapters that provide better performance for scatter-gather techniques.

Chapter 7

Acronyms

AAL	ATM Adaptation Layer
APE25	ATM Protocol Engine
ATM	Asynchronous Transfer Mode
CS	Convergence Sublayer
DBD	Data Buffer Descriptor
DMA	Direct Memory Access
FDL	Free Descriptor List
FRBHL	Free Receive Buffer Header List
ITU	International Telecommunications Union
LC	Logical Channel
LCI	Logical Channel Identifier
LCRBL	LC Receive Buffer List
MTB	Maximum Transmit Buffers
NIC	Network Interface Card
PCI	Peripheral Component Interconnect
PDU	Protocol Data Unit
PVC	Permanent Virtual Circuit
QoS	Quality Of Service
RBH	Receive Buffer Header
RDTSC	Read Time Stamp Code
RFL	Receive Free List
RRL	Receive Ready List
SAR	Segmentation and Re-assembly
SCDA	Single Copy Driver A
SCDB	Single Copy Driver B
SDU	Service Data Unit
SEAL	Simple and Efficient Adaptation Layer
STM	Synchronous Transfer Mode
SVC	Switched Virtual Circuit
TCL	Transmit Complete List
TFD	Transmit Frame Descriptor
VBR	Variable Bit Rate
VCI	Virtual Circuit Identifier
VPI	Virtual Path Identifier

References

1. Almesberger, W., ATM on Linux, *Linux/Internet Kongress '95*, 1995
   http://lrcwww.epfl.ch/linux-atm/misc.html

2. Almesberger, Werner., ATM on Linux, *Linux/Internet Kongress '96*, 1996
   ftp://lrcftp.epfl.ch/pub/linux/atm/papers/atm_on_linux.ps.gz

3. Almesberger, W., High Speed ATM networking on low-end computer systems
   ftp://lrcftp.epfl.ch/pub/linux/atm/papers/atm_on_lowend.ps.gz

4. Almesberger, W., ATM on Linux *3rd International Linux Kongress* 1996
   ftp://lrcftp.epfl.ch/pub/linux/atm/berlin96/slides.ps.gz

5. Almesberger, W., Linux ATM API Draft Version 0.4, July, 1996
   ftp://lrcftp.epfl.ch/pub/linux/atm/api/atmapi-0.4.tar.gz

6. Almesberger, W., Patch: Single Copy vs Shared Memory, *The LINUX ATM Mailing list Archives*
   ftp://lrcftp.epfl.ch/pub/linux/atm/mailing-list/archive.6

7. Bach, M.J. *The Design of the UNIX Operating System*, Prentice Hall, 1986

8. Beck, M., H.Bohme, M.Dziadzka, U.Kunitz, R.Magnus, and D.Verworner, *LINUX Kernel Internals*, Addison-Wesley, 1996.

9. Boosten, M., Device Drivers - DMA to User Space, *The LINUX Kernel Hacker's Guide*
   http://www.redhat.com:8080/HyperNews/get/devices/devices/22.html

10. Boosten, M., Device Drivers - How to DMA to User Space, *The LINUX Kernel Hacker's Guide*
    http://www.redhat.com:8080/HyperNews/get/devices/devices/22/1.html

11. Davies, E.B., Re: Memory Allocation for Large Buffers, *The LINUX ATM Mailing list Archives*
    ftp://lrcftp.epfl.ch/pub/linux/atm/mailing-list/archive.8

12. Druschel, P., M. Abbott, M. Pagals, L. Peterson, Network Subsystem Design
    ftp://cs.arizona.edu/xkernel/Papers/analysis.ps

13. Druschel, P., Operating System Support for High-Speed Networking, Ph.D. Dissertation, 1994.

ftp://ftp.cs.arizona.edu/reports/1994/TR94-24.ps

14. Hennessy, J.L., D.A. Patterson, D. Goldberg, *Computer Architecture - A Quantitative Approach*

15. Ibrahim, Z., A Brief Tutorial on ATM (Draft), 1992

http://dirac.py.iup.edu/otherplaces/internet-networking/atm-intro.html

16. Intel Pentium II Processor Application Notes, Using the RDTSC Instruction for Performance Monitoring

http://www2.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM

17. Prycker, M.D., *Asynchronous Transfer Mode - Solution for Broadband ISDN*, Third Edition, Prentice Hall, 1995

18. Stevens, R.W., *UNIX Network Programming*, Prentice Hall, 1997

19. Tanenbaum A.S., *Computer Networks*, Third Edition, Prentice Hall, 1996

20. Westall, J.M., Geist, R.M., Flower, A.L., ATM Device Driver Development in Linux, Clemson University, 1998