Design, Implementation and Evaluation of a High performance I/O subsystem in Linux

by

Pramodh Mallipatna

B.E. (Electronics and Communication), Bangalore University, Bangalore, India, 1997

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

_____

Professor in Charge

_____

_____

Committee Members

_____

Date Thesis Accepted

To my family

Acknowledgments

I would like to thank Dr. Douglas Niehaus, my advisor and committee chairman, for his guidance, advice, and support throughout this research and my stay at KU. I thank him for giving me the opportunity to work on various interesting things. It was a great learning experience under him.

I would like to thank Dr. Gary Minden and Dr. Joseph Evans for consenting to be on my committee. I would also like to thank the people involved in the ACS project, for their support during my stay.

Thanks to Peter Whiting who started this project for his help in understanding the code in the early stages and helping me all through. Thanks to Sean House for helping me all through my stay here. It was great learning experience with him. I would also like to thank Roel Jonkman for helping me with many networking doubts during my thesis.

Thanks to Sarin, Deepak, Dhananjay, Harish and Deepu for all the help during my thesis. Thanks to all the Team Niehaus members and many others who have made my stay and experience in Lawrence a memorable one.

Abstract

Improvements in computational science has led to the usage of faster CPUs, more main memory, larger disks and fast network devices. Though the industry is moving towards ever more capable hardware, some of the operations are still done in software, which might become a bottle-neck for fast I/O applications. Conventional software I/O APIs are not suited for all kinds of applications. Hence, fast I/O applications can benifit from re-designing the I/O subsystem. This work presents an approach called *FlexIO driver*, which allows users to define and control data flows in the kernel. FlexIO driver can perform disk and network I/O and I/O with a specialized PCI card housing FPGAs.

Contents

List of Figures

List of Programs

Chapter 1

Introduction

1.1 Motivation

Rapid improvements in computational science, processing capability, main memory sizes, data-collection devices, multimedia capabilities, and integration of enterprise data are producing very large datasets ranging from megabytes up to terabytes. We can expect such large data sets to be common in many scientific and commercial applications. The cost of data access increases rapidly with data size. Several classes of applications involve large and frequent access to data stored in memory, disk and across the network and that many of these operations are done in software. Hence, the performance of the I/O could be constrained. So, more efficient ways of performing I/O need to be investigated and incorporated in the I/O subsystems. This has been the primary motivation of this thesis.

CPU speed is doubling once every 18 months, and this is expected to continue for few more years to come. But, the disk access speed is not improving at the same rate. Hence, there is an on-going research to improve the I/O subsystem. Some of the techniques which are being used to improve the I/O performance include

- Overlapping disk (secondary memory) accesses with computations, which can optimize the I/O by making many of the operations asynchronous and thus parallel. Disk I/O is considerably slower when compared to primary memory access. If a process needs to do disk I/O and if the CPU has to wait for the I/O to

be completed, a significant amount of time is wasted during the I/O. So, the CPU cannot do any useful work while it is waiting for the I/O to complete. Hence overlapping disk accesses with computations is a good approach.

- Direct memory access (DMA) and use of better and faster hardware: When there is a data transfer to be performed, the CPU can give control to the DMA hardware which directly accesses the memory for the I/O,leaving the CPU free to perform other operations. The use of hardware for the operation makes it faster because operations done in hardware are generally faster than software.

- Efficient management of buffer caches, which makes I/O faster: The data accessed from the disk is held in a cache known as buffer cache (Filesystem cache). So, if the same data is to be accessed again and if it is cached, the I/O will be faster as a disk access is not necessary. But, a system has a small cache compared to the main memory and the disk. So, a small amount of data can be cached at any point of time. Efficient management of the cache improves the performance of the I/O.

All these methods improve I/O performance in different ways. An I/O operation can be considered to consist of various component operations, involving both software and hardware. The above techniques optimize different component operations of the I/O. Certain applications such as disk-network I/O, that involve operations to fetch the data from the disk to memory do not use DMA hardware in its entirety. Many applications make use of the conventional APIs for I/O, such as the read, write and ioctl. These applications copy the data from one memory location to another within the same address space or across address spaces. These APIs would suffice for the applications which need such a functionality, but not all. For example, streaming audio, video and some customized applications may not require the traditional API elements. If we were to perform I/O using these APIs in such cases, the performance of the I/O would not be optimized for two reasons. First, the control flow unnecessarily crosses the user-kernel boundaries many times and second, a copy operation takes a lot of CPU cycles and hence the I/O operation might be time consuming as there are lot of

copies involved. Hence, if we could minimize the copy operations, the performance of I/O dependent applications may improve as a result. This is the focus of this thesis.

1.2 Flexible I/O (FlexIO)

FlexIO is a pseudo device driver in the Linux kernel used for high performance I/O applications. We call our driver as FlexIO driver, as it provides a significant amount of flexibility in performing I/O. Chapter 3 explains FlexIO implementation in detail. It lets user processes define and control data flows within the kernel.

The choice of a pseudo device driver was made as a device driver provides flexibility and is a well defined interface, common to most UNIX based operating systems. This makes the basic system easily portable across platforms. Presently, the FlexIO driver is targeted to support disk-disk I/O, disk-network I/O and I/O with a specialized PCI card housing FPGAs, called WildForce board, the details of which are explained in Chapter 3. This project was carried out as a part of the Adaptive Computing Systems project at the Information and Telecommunication Technology Center, University of Kansas[2].

1.3 Organization of This Report

Chapter 2 discusses some of the related work in the area of High-Performance I/O. Chapter 3 is a detailed look at the design and implementation of the FlexIO driver. Chapter 4 presents an evaluation of the FlexIO driver. Finally, Chapter 5 provides some conclusions reached and some possible extensions to the system.

Chapter 2

Related Work

This chapter addresses several efforts being made to achieve high performance I/O and how our approach compares with these efforts. Though some of the efforts discussed here are not for high performance I/O in Linux, they have been included because they try to solve the general problem of improving the performance of I/O.

2.1 The Hardware Approach

As discussed in Chapter 1, the use of hardware is a good approach for high performance I/O. Many applications can be identified, where software is being replaced partially or completely by hardware, for example, consider modern network routers. A simple PC running UNIX based operating system, Linux for example, can be configured to work as a router. The Linux box in this case can forward packets to other routers based on the routes it learns. Functionally, this works just fine. But when we consider a big network where there are thousands of routers and there is a lot of traffic in the network, the routing table lookup is time consuming. The Linux based router does not scale well because it has inherent limitations in performance, such as, the PC has much less memory, has a limitation on the number of Network Interface Cards it can host and the operating system it is running is a generic multi-user, multi-tasking operating system which routes in software. Since the operating system is not just dedicated for routing, it has to spare time for other processes running on the system. Hence

the performance will be lower than that of a more specialized device. Even if we try to minimize the limitations by using a fast processor, more memory and a real-time operating system, the performance is still poor when scalability is concerned.

Considering all these issues, router manufacturers like Cisco Systems, Juniper Networks and others came up with special hardware platforms running specialized code to route the packets. The specialized code is normally a trimmed down version of a UNIX based real-time operating system. Such a system provides high performance for large networks. The high performance is mainly because routing table lookup is done in hardware here and also because it uses a specialized operating system.

Many other applications which used to be in software make use of hardware these days, Application Specific Integrated Circuit (ASIC) being an example. Internet Processor II [7] is an example of an ASIC [5] used in the Juniper routers for high-speed packet processing. It provides the unique ability to combine the IP intelligence for rich packet processing as well as wire-rate packet forwarding and high-speed performance. ASICs are also used for Distributed Buffer Management and I/O Management in the Juniper routers [7].

2.2 Direct Memory Access

Using interrupt driven device drivers to transfer data to or from hardware devices works well when the amount of data is reasonably low. For high speed devices, such as hard disk controllers or Ethernet devices the data transfer rate is a lot higher. A SCSI device can transfer up to 40 Mbytes of information per second. In such cases, interrupt driven device drivers will not be able to handle the data rates. Direct Memory Access (DMA) was invented to solve this problem [6][1]. A DMA controller allows devices to transfer data to or from the system's memory without the intervention of the processor. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. It then tells the device that it may start the DMA when it wishes. When the transfer is complete the device interrupts the PC. While the transfer is taking place,

the CPU is free to do other things. The PC DMA subsystem is based on the Intel 8237 DMA controller [11].

2.3 Efficient Memory Management

For fast I/O, apart from making the processors, memory and so on faster, the best approach is to maintain caches of useful information and data that make some operations faster. Most operating systems maintain several caches both hardware and software, such as, Buffer Cache, Page Cache and Swap Cache for this purpose. Efficient management of the caches becomes important. Second chance, Least Recently/Frequently Used (LRU/LFU) [6][1] are some of the algorithms used for efficient management of these caches. Research has been going on in this area to improve the LRU algorithm in-order to make it more efficient. Early Eviction LRU (EELRU) [15], an adaptive page replacement algorithm, is an effort in this regard.

The problem addressed in this thesis is the general problem of improving the I/O performance and particularly *streaming I/O* where blocks are not used more than once. Thus, caching is of limited utility. The approach adopted here is different than those described in the previous sections. This thesis focuses on improving I/O in software by minimizing the number of data copies and by letting the control flow remain in the Operating System rather than moving in and out of user context.

Chapter 3

Implementation

3.1 Overview

A simple definition of throughput of an I/O system is

$$\text{Throughput} = \frac{\text{Amount of data transferred}}{\text{Time for the transfer}}$$

The amount of data transferred and the time for the data transfer need not always have a linear relationship because, the data transfer is affected by several components in the system, like, available memory, the way the data is cached and related computations which might have different effects on different data sets. Looking at the equation, if we would like to improve the performance (throughput) of an I/O subsystem, we can achieve it if we minimize the time elapsed for the I/O.

Time for the I/O depends on a lot of factors. It depends on several operations including buffer allocation and management, software computations for the I/O and data copies between memory locations. A copy operation takes significant number of CPU cycles and would affect the throughput for large data sets if done multiple times. Also, it would make sense for a lot of I/O applications to have the data flow to remain in the kernel as opposed to a flow crossing user and kernel address spaces.

The focus of this thesis has been to let the data flow remain in the kernel and minimize the number of data copies during the I/O in order to improve I/O throughput. We have implemented a pseudo device driver which lets us define I/O flows in the

kernel. The software is in the form of a pseudo device driver in the Linux kernel.

This chapter is structured as follows. Section 3.2 explains the FlexIO driver, its structure and its entry points. Sections 3.3, 3.4 and 3.5 then explain how the FlexIO driver can be used to perform Disk-Disk I/O, Disk-Network I/O and I/O with the WildForce[4] board supporting FPGAs respectively.

Field Programmable Gate Arrays (FPGA) are a recent kind of programmable logic devices. An FPGA is a re-configurable hardware, which can be configured to perform a particular task by programming with some digital logic. It can then be re-programmed to perform a different task. WildForce is a PCI add-on card housing FPGA chips which can be used in a lot of applications. In our project, it is used to perform signal processing[2]. Digital logic developed for Digital Radio and Synthetic Aperture RADAR processing functions are downloaded onto the FPGA chips residing on the WildForce board.

3.2 FlexIO

FlexIO is a pseudo device driver which can be built into the kernel. The driver is loaded at boot time and is never unloaded. It can also be implemented as a Linux loadable module. A loadable module can be loaded into and unloaded from the kernel after the kernel is booted. It lets user processes define and control data flows within the kernel and is implemented as a character device driver. The structure of the driver is explained by discussing the important functions and entry points in the driver. A beginner's HOWTO to write a device driver is provided in Appendix A.

3.2.1 flexio_init() and init_module()

flexio_init is the name of the routine if the driver is built into the kernel and *init_module* is the routine name if it is a loadable module. It registers the driver with the character device table with a major number of 30 and does the other initializations. Flexio driver has *open()*, *close()*, *ioctl()* and *mmap()* entry points defined.

3.2.2 cleanup_module()

This is used to unregister the driver from the character device table and un-initialize the things that were initialized when loaded. This is used in case the driver is a module. If it is built into the kernel, the driver remains in the kernel till the system is rebooted.

3.2.3 flexio_open() and flexio_close()

They are the open and close entry points, called when *open()* and *close()* are called respectively by a user process. *flexio_open()* is used to initialize the private data structure for the open instance of the driver and *flexio_close()* is used to un-initialize it. The private data structure has the details of the I/O. For example, if the operation is a file I/O, it has details about the input and output files. If the operation is a network I/O, the private data structure has details about the interface the packets travel over, such as Ethernet and ATM, and the receiver information. In case of WildForce I/O, the data structure has details about the WildForce driver.

3.2.4 flexio_mmap()

This is used to map memory from FlexIO space to the user space. This is called when a user process makes an *mmap()* [13] system call to memory map to pages in FlexIO space. This routine then maps the pages in FlexIO space to the virtual pages in user space. So, the user process now maps to the same physical pages the driver uses. This operation is shown in Program 3.1.

Memory mapping is an important feature in the operating system. In UNIX based operating systems, the user and kernel have different address spaces for obvious reasons. So, in the case of a normal system call execution, the data is copied between the address spaces. But, for applications like real-time audio or video, if the data is streamed in real-time, the latency due to such copies across address space boundaries may not be acceptable. So, in such cases, the application would map to the same I/O buffer memory as that of the driver. By memory mapping, neither the application nor the driver need to copy the data across address space boundaries. They will be ac-

Program 3.1 flexio_mmap: Kernel-User memory mapping function

```
static int flexio_mmap(struct file *file, struct vm_area_struct *vma)
{
        unsigned long page, start, size;
        volatile void *pos;

        start = (unsigned long)((char *)vma->vm_start);
        pos = flexio_mmap_buffer;
        size = (unsigned long)(vma->vm_end-vma->vm_start);

        while (size > 0) {
                page = fvirt_to_phys((unsigned long)pos);
                if (remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED))
                        return -EAGAIN;
                start += PAGE_SIZE;
                pos += PAGE_SIZE;
                size -= PAGE_SIZE;
        }

        return 0;
}
```

cessing the same physical pages in memory, even though they have different virtual addresses in different address spaces.

3.2.5 flexio_ioctl()

An ioctl() is a function used to communicate with and manipulate the underlying device specific parameters of special files which represent devices. *flexio_ioctl()* is invoked when a user process makes an *ioctl()* call. A number of ioctls are defined in the FlexIO driver, explained later, which can be used to configure the parameters of the data flow, including input file descriptor, output file descriptor, size of the data transfer, Ethernet or ATM interface details for network transfer and so on. To map a user memory segment to kernel space, a new ioctl has also been defined. The difference between this ioctl and the mmap interface is that, mmap maps kernel memory to user space and using this ioctl, we can map a user memory segment to kernel space, which is helpful in WildForce I/O. The use of these ioctls are explained as we consider different scenarios. User code for performing I/O is provided in Appendix B.

Pseudo code for user-kernel memory mapping is shown in Program 3.2. The user does the memory mapping by executing MAP_USER_MEMORY ioctl. The user sends a pointer to the memory to be mapped to the FlexIO driver. This is *useraddr* as seen in the code. The physical page or pages pointing to this can be obtained using the *flexio_usr_to_phys()* routine, which gets the physical page by looking at the page tables *pgd*, *pmd* and *pte*. The *phys_addr* so obtained is converted to a kernel virtual address using *phys_to_virt()* routine. Now, the *useraddr* and *kernaddr* point to the same physical page.

Program 3.2 flexio_map_memory: User-Kernel memory mapping function

```
int flexio_map_memory(unsigned long useraddr)
{
        char *kernaddr=NULL;
        unsigned long phys_addr=0;

        /* Get the physical address of the virtual user address */
        phys_addr = flexio_usr_to_phys((volatile void *)useraddr);

        /* Get a kernel virtual address for the physical address */
        kernaddr = phys_to_virt((unsigned long)phys_addr);
}

/* Page table lookup to get the the physical address */
static inline void *flexio_usr_to_phys(volatile void *address)
{
        pgd_t * pgd;
        pmd_t * pmd;
        pte_t * pte;

        pgd = pgd_offset(current->mm, (unsigned long)address);
        pmd = pmd_offset(pgd, (unsigned long)address);
        pte = pte_offset(pmd, (unsigned long)address);

        return((void *) ( ( pte_page(*pte) +
                (((unsigned long)address)&(PAGE_SIZE-1)))
                 & ~PAGE_OFFSET));
}
```

3.3 Disk-Disk I/O

FlexIO presently supports I/O involving raw disk partitions and Ext2 * filesystem files. In this section, the concepts of file I/O as done by a user process is explained first before explaining how FlexIO does the I/O. This will provide an introduction to file I/O which will be helpful in understanding the implementation better.

Reading from a disk is very slow compared to accessing memory. In addition, it is common for general applications to read the same part of a disk several times during relatively short periods of time, though it does not apply to the streaming I/O implemented using FlexIO. By reading the information from disk only once and then keeping it in memory until no longer needed, one can speed up all but the first read. This is called disk buffering, and the memory used for the purpose is called the *buffer cache* or *filesystem cache*[6][9]. Since memory is a finite and scarce resource, the buffer cache usually cannot be big enough to hold all the data one ever wants to use. When the cache fills up, the data that has been unused for the longest time is discarded and the memory thus freed is used for the new data.

Whenever a user process requests one or more blocks from a file, the filesystem tries to get a block from the buffer cache. If it does not find the buffer in the buffer cache, the filesystem requests the underlying device driver, SCSI or IDE driver, to read the appropriate block of data from the disk. When the block is read, it is cached in the buffer cache.

3.3.1 User Process doing Disk I/O

A user process can perform file I/O in two ways.

1. Using a user buffer
2. Memory mapped I/O

*Ext2 (Second Extended File System) is a filesystem supported in Linux

3.3.1.1 I/O using a User Buffer

If a normal user process were to read from an input file and write to an output file, it would use the *read()* and *write()* system calls to do so. The user process would read chunks of data to its buffers and write those buffers onto output file using the write() system call. A simple program for this is shown in Section B.2 of Appendix B. A file (disk) read, as done by the standard read() system call, can be considered to consist of the following two operations as shown in Figure 3.1. First, reading data from the disk to the buffer cache or filesystem cache. Note: The terms Buffer cache and Filesystem cache are used interchangeably in this report. Second, copying the data from the buffer cache to the local buffers. Similarly, a file (disk) write, as done by a standard write() system call, can be considered to consist of the following operations. First, copying the data from the local buffers to the buffer cache and second, writing the data from the buffer cache to the disk by syncing the buffers.

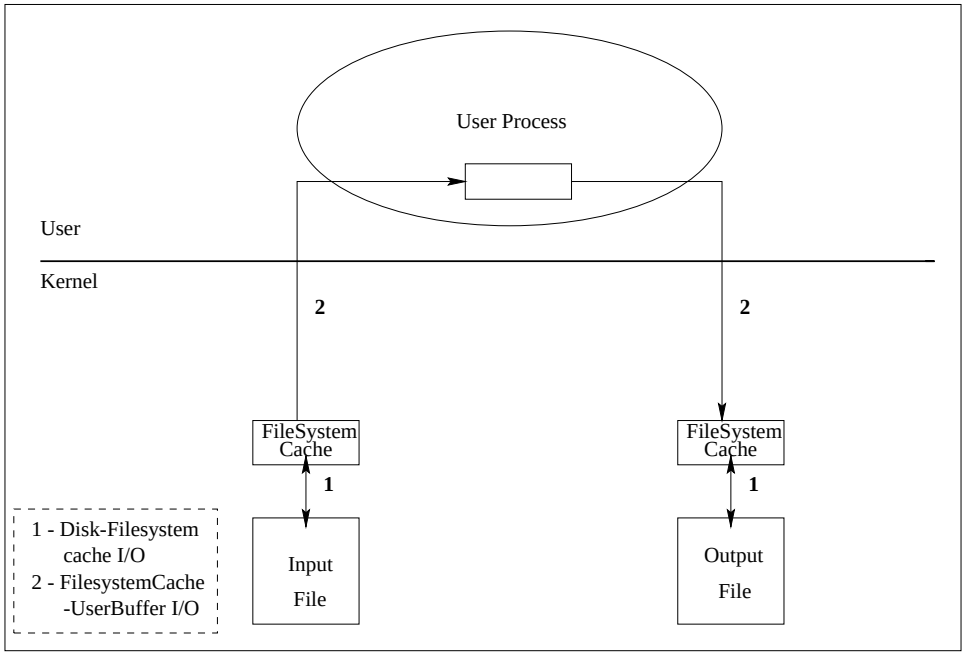

Figure 3.1: User process doing File I/O

Whenever a file system needs to read a block from a file, it tries to get the block

from the buffer cache. If it cannot get the buffer from the buffer cache or if the buffer requested is not up-to-date with the actual block on the disk, the file system must request the SCSI or IDE device driver to read the appropriate block of data from the disk. This block will now be held in the buffer cache for future reference. Since the buffer cache is finite, if free buffers are not available at this point to cache the block to be read from the disk, some of the buffers have to be written back to the disk in order to free some buffers. This process of writing to the disk is called a *sync* operation. Buffers can be synced to the disk on a regular basis or can be explicitly synced when there is a shortage of buffers.

3.3.1.2 Memory Mapped I/O

The other way of doing file I/O in user space is to map input and output files into memory, using mmap() system call, and then just copying the two files as if they were memory locations, using memcpy() system call. This is shown in Figure 3.2. A simple program for this is shown in Section B.3 of Appendix B.

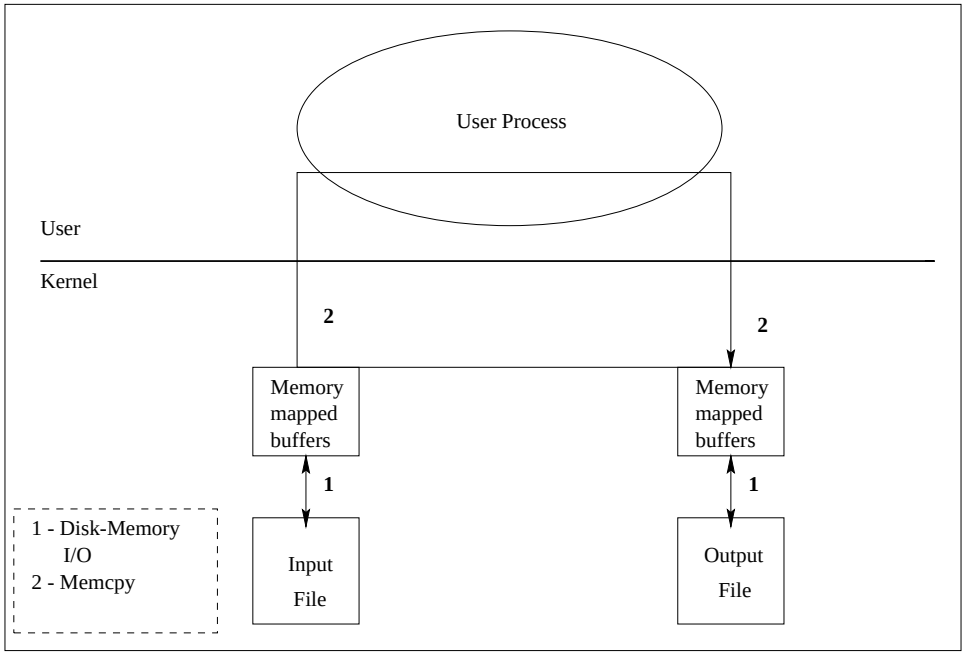

Figure 3.2: User process doing File I/O using mmap()

Memory-mapped files (MMFs) offer a unique memory management feature that allows applications to access files on disk in the same way they access dynamic memory-through pointers[12][13]. With this capability, we can map all or part of a file on disk to a specific range of addresses within the process's address space. And once that is done, accessing the content of a memory-mapped file is as simple as dereferencing a pointer in the designated range of addresses. So, writing data to a file can be as simple as assigning a value to an address in memory.

3.3.2 FlexIO doing Disk I/O

As seen from Section 3.3.1, the data flow goes between user and kernel space and in the first case, there are two extra copies other than the disk reads and writes. As mentioned before, the idea of using FlexIO is to keep the data flow within the kernel and to investigate the effect of data copying on the I/O. So, FlexIO has been designed to perform the I/O in three ways

1. Using kernel buffers
2. I/O at the buffer cache level
3. Memory mapped I/O

3.3.2.1 I/O using Kernel Buffers

This is similar to a user process doing the I/O using an intermediate user buffer. In this case, data is read in chunks from the input file to the FlexIO buffer and written to the output file. The implementation is similar to *read()* and *write()* system calls with the difference being that kernel buffers are used instead of user buffers. Data is read to the buffer cache and then copied onto the FlexIO buffer. It is then written onto the buffers associated with the output file, which is later written onto the disk (synced). The whole operation is presented in Figure 3.3. The data flow in this case remains in the kernel and the number of copies is the same as the user process doing the I/O with a user buffer, but the difference being it copies to a kernel buffer. This is implement-

ed as FILE_KERNEL_READ_WRITE ioctl. Pseudo code for this approach is shown in Program 3.3.

Program 3.3 FILE_KERNEL_READ_WRITE ioctl: I/O using FlexIO kernel buffer

```
file_copy_remaining = input_file_size;

/* Read the input file in chunks of DISK_BLOCK_SIZE
   and write to output file */
while (file_copy_remaining > 0) {
    flexio_read(infile_ptr, flexio_kernel_buffer, DISK_BLOCK_SIZE);

    flexio_write(outfile_ptr, flexio_kernel_buffer, DISK_BLOCK_SIZE);

    file_copy_remaining -= DISK_BLOCK_SIZE;
}
```

3.3.2.2 I/O at the Buffer Cache level

The operation is presented in Figure 3.4. In this, the input file is read from the disk to the buffer cache. These buffers are then copied to the buffers associated with the output file, which is later written onto the disk. The data flow remains in the kernel and it avoids two copies. This is implemented as the BUFCACHE_FILE_KERNEL_READ_WRITE ioctl. The same principle is used for raw disk partitions also, which is implemented as the RAW_KERNEL_READ_WRITE ioctl. Pseudo code for RAW_KERNEL_READ_WRITE is shown in Program 3.4. The basic idea of BUFCACHE_FILE_KERNEL_READ_WRITE is similar to this except for the fact that it is done for Ext2 files.

3.3.2.3 Memory Mapped I/O

The operation is described in Figure 3.5. The input and output files are mapped into memory in the FlexIO driver. The I/O is then done using memcpy(). As explained before, a file is treated as a memory location here. The data flow remains in the kernel and it avoids two copies. This is implemented as the FILE_MMAP_KERNEL_READ_WRITE ioctl, pseudo code for which is shown in Program 3.5.

FLEXIO_SET_INPUT and FLEXIO_SET_OUTPUT ioctls provide the FlexIO driver

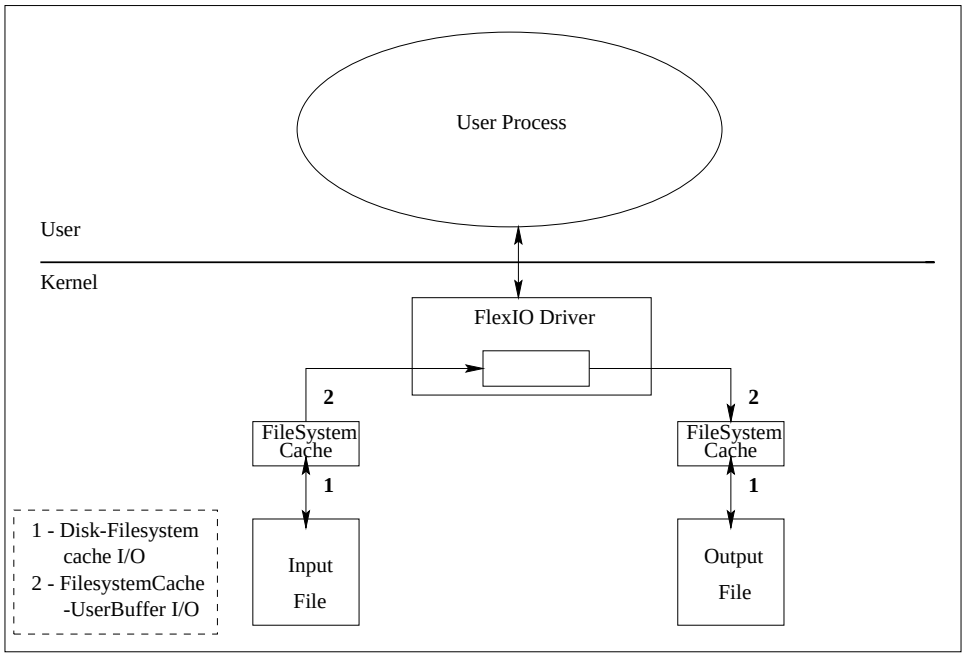

Figure 3.3: FlexIO doing File I/O using a kernel buffer

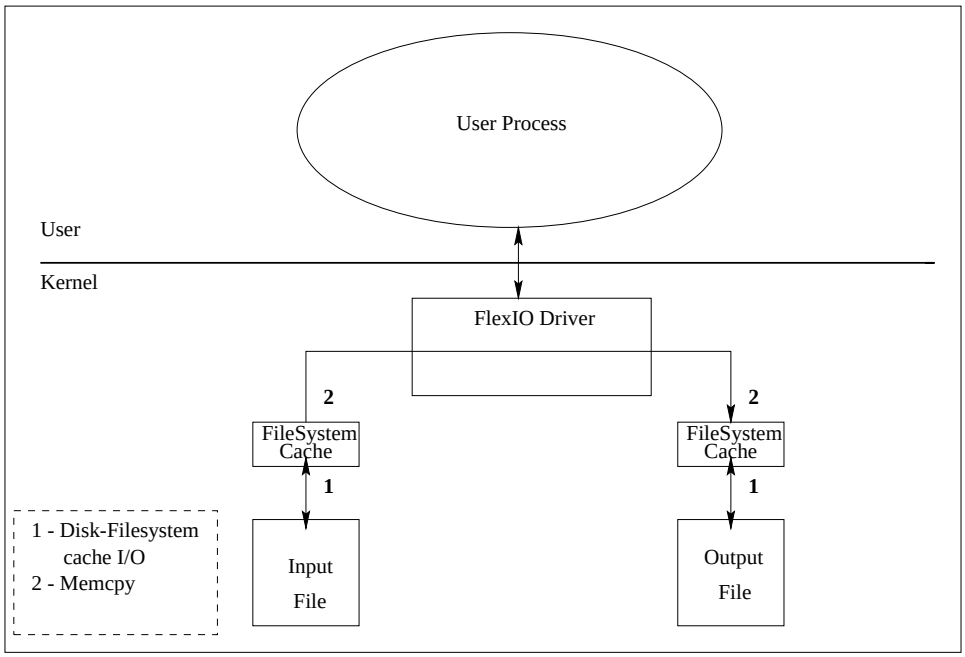

Figure 3.4: FlexIO doing File I/O at the Buffer cache level

Program 3.4 RAW_KERNEL_READ_WRITE ioctl: FlexIO Buffer Cache I/O

```
while (blocks_to_move > 0) {
    /* Read input file block from the Buffer Cache or Disk */
    input_bh = bread(input_dev, curr_input_block, DISK_BLOCK_SIZE);

    /* Get an empty Buffer for the output file */
    output_bh = getblk(output_dev, curr_output_block, DISK_BLOCK_SIZE);

    /* Copy the input file block contents to output file block */
    memcpy(output_bh->b_data, input_bh->b_data, DISK_BLOCK_SIZE);

    /* mark the output file block uptodate and dirty so that it gets
     * written to the disk when synced (flushed)
     */
    mark_buffer_uptodate(output_bh, 1);
    mark_buffer_dirty(output_bh, 0);

    blocks_to_move--;
}
```

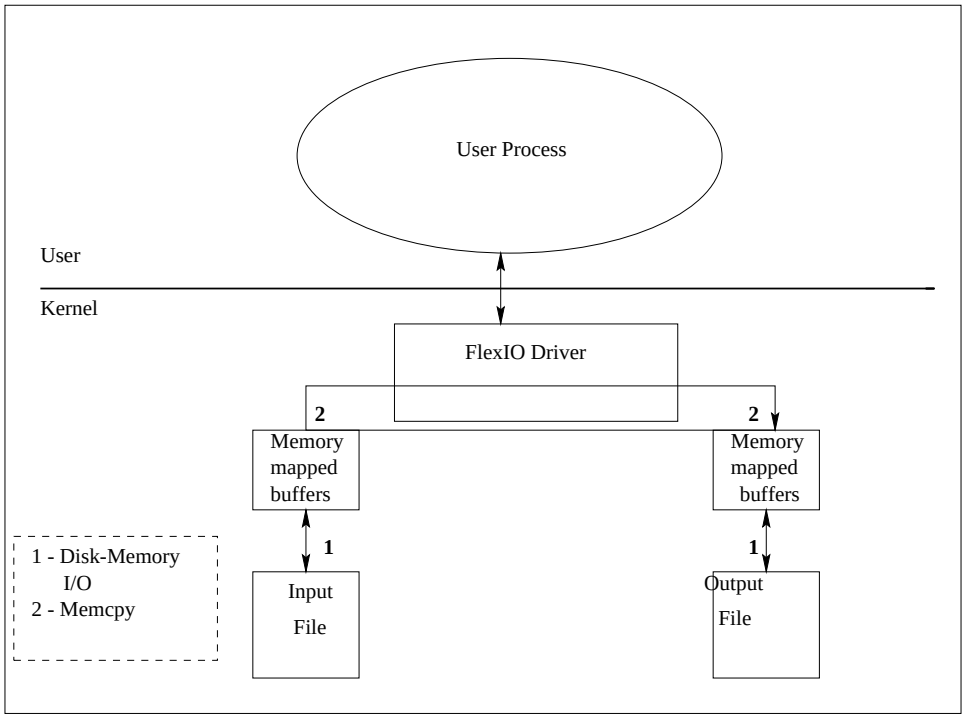

Figure 3.5: FlexIO doing Memory Mapped I/O

Program 3.5 FILE_MMAP_KERNEL_READ_WRITE ioctl: I/O using FlexIO, with memory mapping the files

```
inaddr = NULL;
outaddr = NULL;

/* Memory map the input file to inaddr */
inaddr = do_mmap(infile_ptr, inaddr, infile_size,
                 PROT_READ, MAP_FILE|MAP_SHARED, 0);
/* Memory map the output file to outaddr */
outaddr = do_mmap(outfilp, outaddr, infile_size,
                  PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, 0);

memcpy(outaddr, inaddr, infile_size);
```

with the information about the input and output files needed for the I/O. These ioctls check for the validity of the files, including permissions and size of the file. The user programs for the above scenarios are shown in Sections B.4 and B.5 of Appendix B.

3.4 Disk-Network I/O

FlexIO presently supports I/O between the network and Ext2 files or raw disk partitions. In this section, the concepts of Network I/O is explained first before explaining how FlexIO does the I/O. This will provide an introduction to network I/O which will be helpful in understanding the implementation better

3.4.1 Network I/O Overview

To transfer data over the network, we need to send it in a way the receiver can understand the data it has received. To incorporate the heterogeneity of the Internet, networking protocols have been designed using a layered approach. The TCP/IP protocol suite has become the de-facto standard for the Internet. In UNIX based operating systems, an application (user process) views a network through sockets. Sockets are abstractions of the network connection between processes to the user. So, if a user process were to perform network I/O, it would *open*, *read* and *write* to sockets and em close them. The various kinds of sockets used for network I/O are TCP, UDP and raw

sockets. The packet and the protocol stack would look as shown in the Figure 3.6.

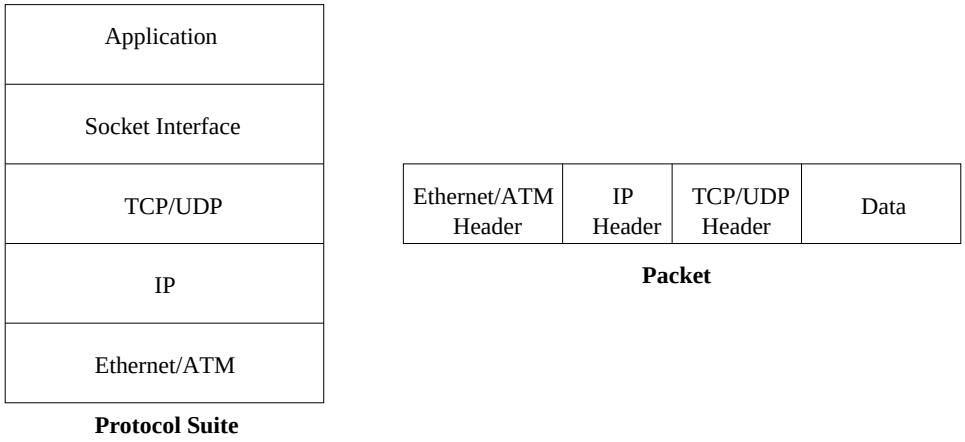

Figure 3.6: The Protocol suite and Packet for TCP/IP based applications

Such a setup need not offer the best performance for all kinds of applications. This is because every layer does some processing on the packet being sent or received and adds its own header. This is required for Internet based applications, but, if we consider some high-performance applications, some or most of these might turn out to be unnecessary overheads. If we were to have some applications running on the local network, where congestion, packet drop and re-transmission are not an issue, we can do without some of these protocol layer functions. Distributed or multi-processor computations on a local ATM network could be an example for this scenario. Network I/O using FlexIO is an effort to support I/O efficiently in such situations.

3.4.2 User Process doing Disk-Network I/O

As discussed in Section 3.4.1, a user process accesses a network through sockets. A user process can perform network I/O using TCP, UDP or raw sockets. As network transfer using FlexIO is connectionless, with no end-to-end assurance like UDP, only UDP sockets are considered in this discussion. File I/O using UDP sockets is as explained below and shown in Figure 3.7.

The process would read the file using the *read()* system call into a user buffer and write the contents to a socket using the *write()* or *sendto()* system calls. The data is now

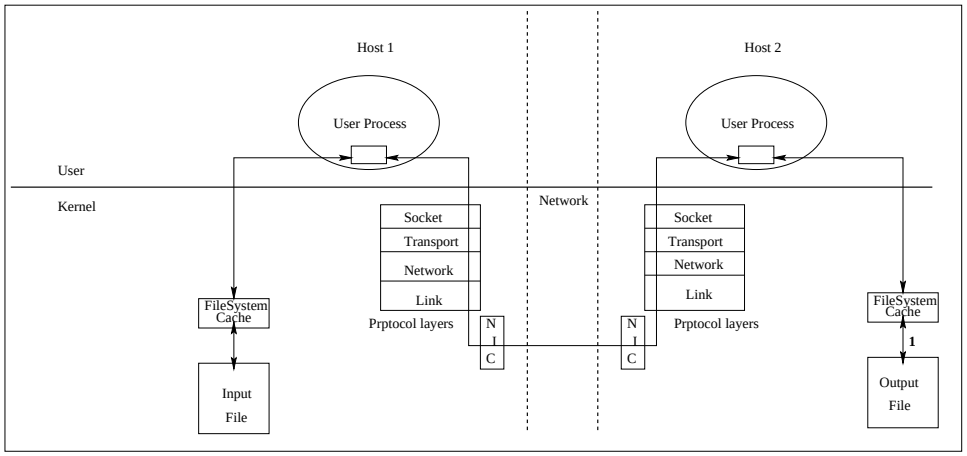

Figure 3.7: Network I/O using UDP sockets

copied to the UDP packet and sent over the network. The receiver at the other end would read from the socket into a user buffer using a *read()* or *recvfrom()* system call and write to the file using the *write()* system call. As can be seen, the control passes across the user-kernel boundaries twice just for the data to be copied from the buffer cache to the actual packet at the sender and from the actual packet to the buffer cache at the receiver. A detailed example of this is shown in the Sections B.8 and B.9 of Appendix B and pseudo code is shown in Program 3.6.

Program 3.6 Pseudo code of network I/O done by a user process

```
/* SENDER code */
        while (file_size > 0) {
                ret = read(fdin, buffer, chunk_size);
                sendto(sockfd, buffer, ret, 0x40, &receiver,
                                sizeof(receiver));
                file_size -= chunk_size;
        }
/* RECEIVER code */
        while (file_size > 0) {
                ret = recvfrom(sockfd, buffer, chunk_size, 0,
                                (struct sockaddr *)&sender, &len);
                write(fdout, buf, ret);
                file_size -= chunk_size;
        }
```

3.4.3 FlexIO doing Disk-Network I/O

As discussed in Section 3.4.2, the data flow goes between user and kernel and there are two extra copies other than the disk reads and writes, and DMA to the Network Interface Card (NIC). We try to eliminate these copies and keep the data flow in the kernel, using the FlexIO Disk-Network I/O implementation.

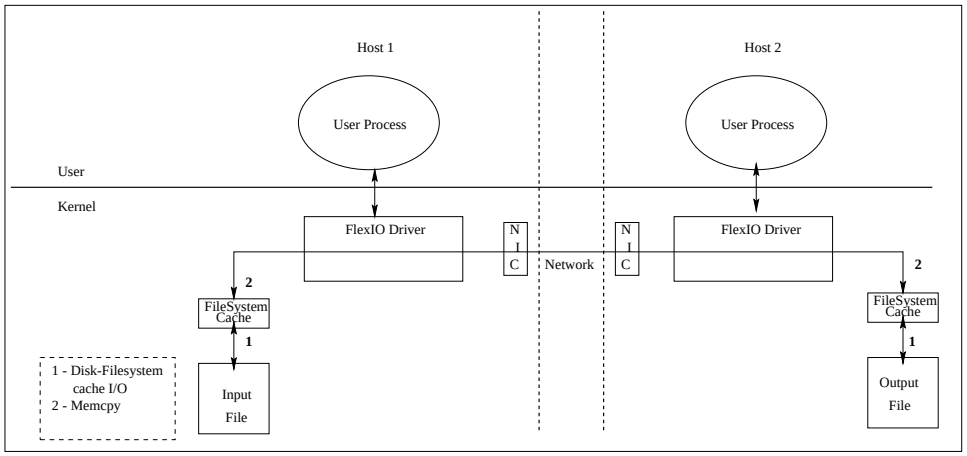

Figure 3.8: Network I/O using FlexIO

The input file is read from the disk to the buffer cache, packetized and sent over the network. At the receiver, the received data is written to the buffer cache, which is later written to the disk (synced). This is shown in Figure 3.8. The packet and the protocol stack would look as shown in Figure 3.9

To support and identify various flows, the notion of a port, as in the case of TCP and UDP socket implementation, is used in the driver. A port uniquely identifies the instance of the driver associated with the transfer. This is important because the flow specific data is held in the driver private data structures.

3.4.3.1 Transmitter

The sending process configures the input file details using the FLEXIO_SET_INPUT ioctl. It then executes FLEXIO_SET_NET_DEV ioctl, which has the details of the network, such as, the interface over which a packet has to be sent, the MAC address of the

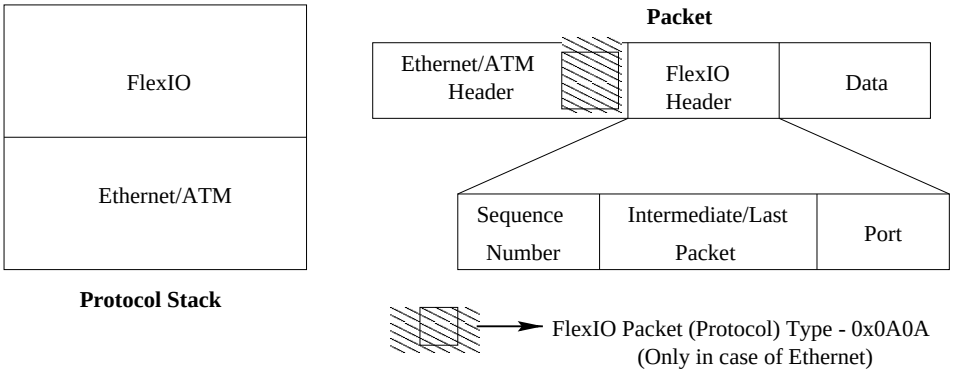

Figure 3.9: FlexIO Protocol stack and Packet

receiver and so on. Note that ARP is not used for knowing the MAC addresses because ARP is for IP-MAC address translation and FlexIO has a non-IP stack. It then executes the FLEXIO_TX ioctl, which is a blocking call that returns only after the file has been sent completely to the network. This works as follows. The kernel thread for the ioctl reads the file in chunks to the buffer cache. It then allocates an *sk_buff* and copies the data from the file (buffer cache) to the data portion of the packet. An sk_buff is the data structure for a packet in the Linux kernel which has the details about the various headers, size of the packet, the interface the packet is to be sent over or received from and so on. After copying the data, it frames the FlexIO header, which has details about the packet sequence number, port and other information as shown in Figure 3.9. It then fills in the Ethernet header and hands the packet over to the NIC driver which then DMA's the packet to the network card. FlexIO uses its own packet type or protocol type, which is 0x0A0A, shown in Figure 3.9. This is something similar to what IP, ARP and other protocols using Ethernet do. The Ethernet header has a field called protocol type which holds the information about the immediate higher layer protocol. This information is used at the receiver for de-multiplexing purposes. Every FlexIO packet will be tagged with this value. In case of ATM, FlexIO driver sets up its own virtual circuits and sends the packets over that virtual circuit.

3.4.3.2 Receiver

The functionality of the receiver is explained in Figure 3.10. The packets at the receiver are DMA'd from the NIC to the Ethernet driver. This is done by interrupting the Ethernet driver. To keep the interrupt service routine short, the packet is queued and returned. This queue, Common network queue, is serviced by the network bottom half. The network bottom half de-multiplexes the packets based on the protocol type in the Ethernet header, as explained in Section 3.4.3.1, and hands the packets to the relevant protocol layers. So, when the network bottom half finds a FlexIO packet, it hands the packet over to *flexio_rcv()*, which then queues the packet in another queue called *flexio_backlog*. This is done in-order not to waste network bottom half's time. This is because network bottom half is a common function which services all kinds of packets, including IP, ARP and other protocol packets coming to all the Ethernet interfaces on the system. Hence, when the network bottom half services our packets, if we consume a considerable amount of time to service our packets, the Ethernet driver may lose incoming packets as the common network queue will be filled. Hence we have our own queue, *flexio_backlog*, which is then serviced by the FlexIO task queue. The FlexIO task queue finds the instance of the FlexIO driver associated with the transfer by looking at the port information in the FlexIO header. With this, it can get hold of the private data structure of the driver instance, which has the information about the output file. It then writes the data portion of the packet to the buffers of the output file, which is later synced to the disk. FlexIO also does the disk syncing periodically to make sure free buffers are available when needed. The receiver is invoked by FLEXIO_RX ioctl. FLEXIO_SET_OUTPUT ioctl is invoked before this to configure the output file details. In the case of ATM, *flexio_push()* does the job of the *flexio_rcv()*. In essence, both these functions act as ISRs for the FlexIO driver, though the driver does not directly receive any interrupt. Arrival of a packet from network bottom half to flexio_rcv() is analogous to an interrupt.

Interrupts are given special priority by the operating system. Hence, when an interrupt occurs, the currently running process is stopped and the interrupt is serviced. To ensure other processes are not starved because of this, the Interrupt Service Routine

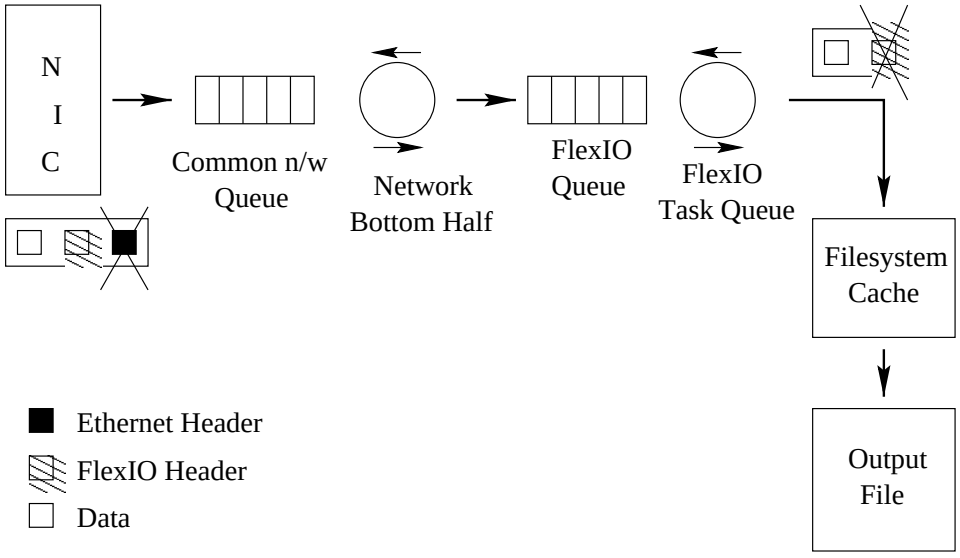

Figure 3.10: FlexIO Receiver Implementation

(ISR) should be as short as possible. The kernel has to do the bare minimum processing in the ISR and the remaining processing is done by *bottom halves*. In the Linux kernel, the pending bottom halves are scheduled at every scheduling instance.

A task queue is a thread in the kernel which can be invoked whenever necessary, either at every scheduler instance, as in the case of bottom halves or only when needed. It has a function associated with it which is used to do the necessary processing in the task queue's context.

3.5 WildForce I/O

WildForce is a PCI add-on card that houses FPGA chips which can be used for performing signal processing or any other data processing functions. In our case, they are used to perform signal processing. For example, Digital Radio and Synthetic Aperture RADAR processing functions can be implemented on the FPGA chip residing on the WildForce board. The WildForce board has on-board memory which is accessed by both the FPGAs and the applications.

The WildForce board comes along with a device driver and APIs to interact with

this device driver. The device driver is used to interact with the board and the APIs are for the user level programs to perform operations like downloading digital data onto the board, perform I/O and so on. Performing I/O with the board using WildForce APIs is discussed first, which should help in understanding how FlexIO performs the I/O.

3.5.1 I/O using WildForce APIs

User level code is written to interact with the board using the WildForce APIs. The various APIs help in configuring stuff on the board such as the clock frequency, the FPGAs that are being used and so on. They are also used to read and write to the memory on the board.

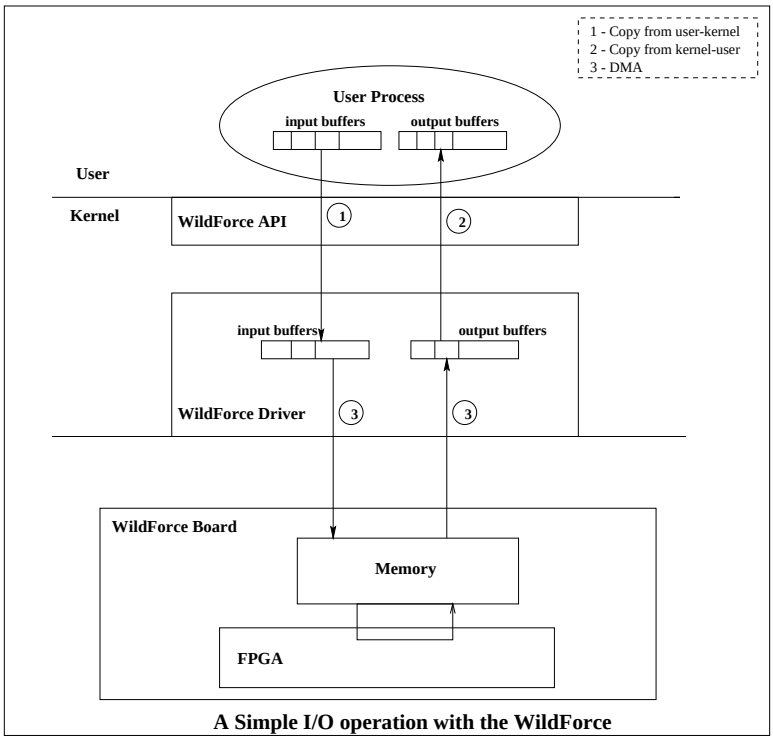

Figure 3.11: A simple I/O using WildForce APIs

A simple flow describing a simple I/O operation is shown in Figure 3.11. Only the I/O is described here and not the configuration of the board. The data from user space

is copied to the WildForce driver buffers in kernel space as the user and kernel have different address spaces. Then the data is DMA'd onto the board. The FPGAs on the board perform the computation for which they have been programmed and the data is DMA'd back to the WildForce driver. The data is then copied back to the user buffers.

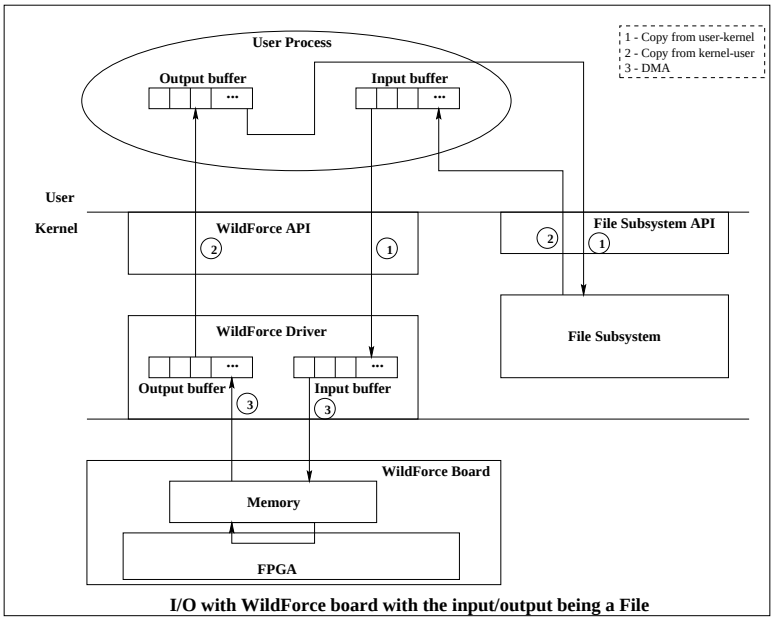

Figure 3.12: I/O using WildForce APIs when data is stored in files

When a significant amount of data is to be processed, the data is normally stored in local files or accessed over the network. In this case, the file is read and then the APIs are used to perform the I/O and the FPGAs do the necessary computation. It is also advantageous to store the processed data in files or over the network for access to the user. This is described in the Figure 3.12. In this case, as we can see from the figure, the data is first transferred to the input buffers in the user space and then the I/O to the WildForce board is done. So, the data is copied twice for the input and twice for the output. So, there are several copies involved. Hence the I/O might be slower than necessary and hence the throughput performance might be lower.

3.5.2 I/O using FlexIO

As seen from Section 3.5.1, the control flow crosses the kernel-user boundary many times and there are several copies involved. We do not have access to the source for the WildForce driver and hence we did not know how the WildForce driver is written. So, when FlexIO is used to perform the I/O, WildForce user level APIs are still used to configure the WildForce board and driver. The FlexIO driver is used to perform only the I/O. The reason for this is explained next.

Any character or a block device driver has the same structure, in the sense that they have standard entry points, although not all may be defined for any particular driver. We can get to any driver's entry points if we look at the character or block device table in the kernel. WildForce driver is a character device driver. Hence we look at the character device table. A pseudo code for this operation is shown in Program 3.7.

Program 3.7 Get the WildForce driver information

```
int get_info_from_chrdev_table(unsigned int major,
                               struct wildforce_info *info)
{
    /* chrdevs        - Character Device table
     * major          - Major number of WildForce driver
     * wildforce_info - FlexIO structure to hold WildForce
     *                  driver info
     * fops           - Has the entry point information
     */
    if (!chrdevs[major].fops)
        return -ENODEV;

    info->fops = chrdevs[major].fops;
    info->name = chrdevs[major].name;

    return 0;
}
```

In UNIX based operating systems, a device is treated as a file. So, we can perform operations such as open(), close(), mmap(), read() and write() on the device just as we do for files. These are called *entry points* as this is how we can interact with the driver. In addition, devices have an entry point called ioctl(), which is used to configure the device or driver.

The Character or Block device table is the data structure which has the information of all the character and block device drivers in the kernel. It is normally implemented as an array with the index being the major number of the device.

The FlexIO driver accesses the information about the WildForce driver by indexing into the character device table and thus gets the entry points to the WildForce driver. This is done by the GET_WFDEV_INFO ioctl. The I/O is done by FILE_READ_WF_WRITE and WF_READ_FILE_WRITE ioctls. These ioctls read from the file and are write to the WildForce board using WildForce write() entry point and vice versa.

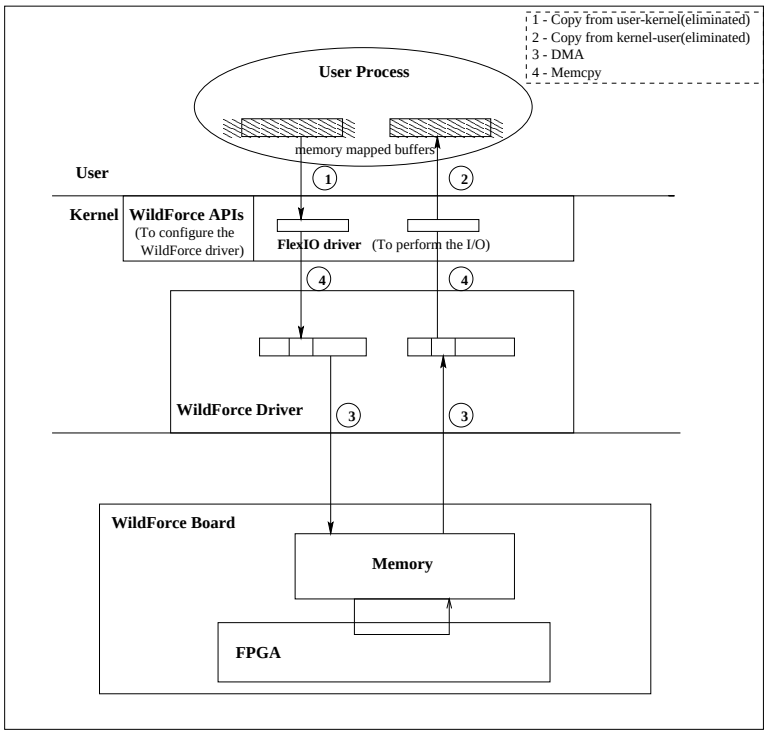

Figure 3.13: A simple WildForce I/O using FlexIO

As mentioned earlier, WildForce user level APIs are still used to configure the WildForce board and driver. This is because, these are implemented as ioctls in the driver. For ioctls, we need to know the structure it expects and the command for the ioctl. As we do not have access to the driver source code, we cannot call the ioctl from within the kernel. Hence, we still use WildForce user level APIs. A User program still uses

the WildForce APIs to configure the WildForce board and driver. The *WF_Memwrite()* and *WF_Memread()* APIs are now replaced by FlexIO ioctls [4].

The scenarios shown in Figures 3.11 and 3.12 now are implemented as shown in Figures 3.13, 3.14 and 3.15 respectively using FlexIO.

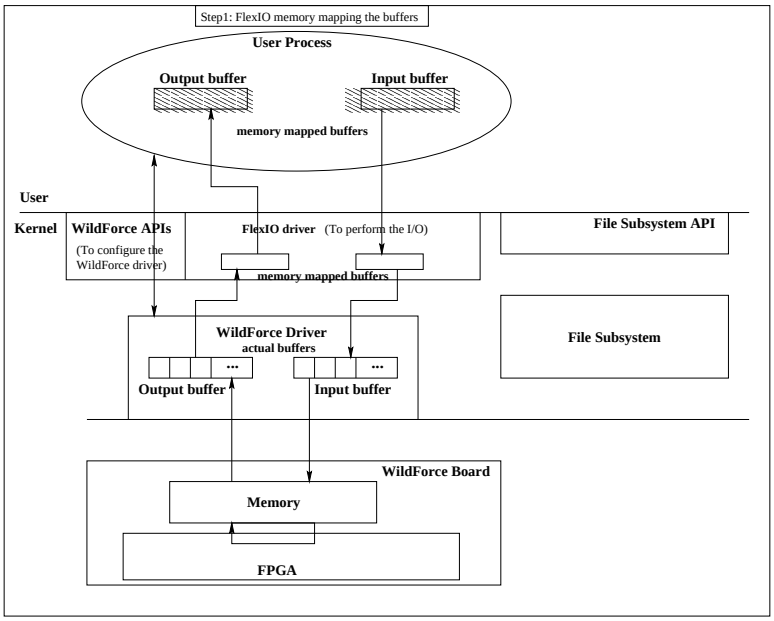

Figure 3.14: I/O using FlexIO when data is stored in files, Step 1 - FlexIO Memory Mapping the buffers

As seen from the figures, WildForce APIs are used to configure the WildForce driver and FlexIO is used for the I/O. FlexIO memory maps user buffers to its space before performing the I/O and when the read and write calls are invoked, it calls the WildForce driver's read and write entry points in the kernel, which then does the I/O. The idea here is to show that the data flow can remain in the kernel, as in Figure 3.15. Even though we memory map the buffers to FlexIO space to avoid copies, we are not achieving what we want, because, the WildForce driver's read and write still do the copy from and to the user space buffers. We have done that in order to show it is possible. If the interface in the WildForce driver is changed so that it does not copy to and from the user buffers, we can use kernel buffers for the whole I/O and eliminate some of the copies.

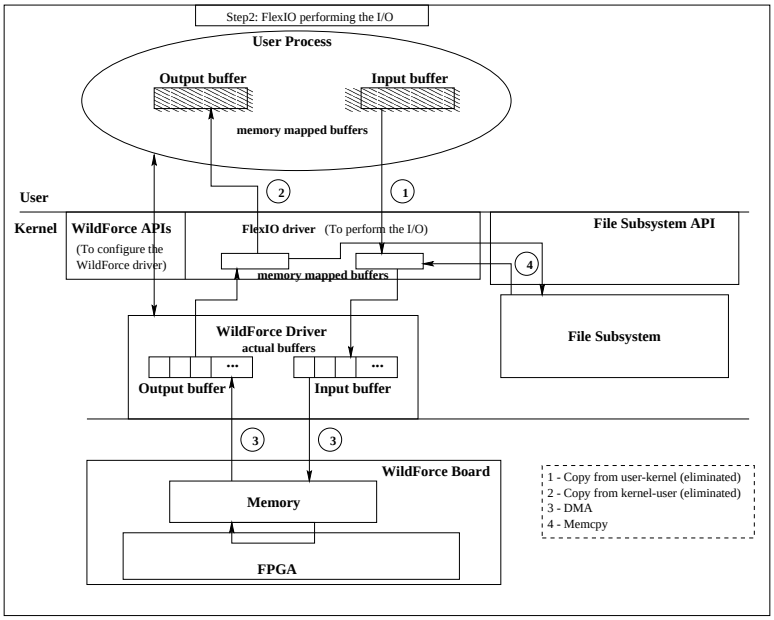

Figure 3.15: I/O using FlexIO when data is stored in files, Step 2 - FlexIO performing the I/O

Chapter 4

Evaluation

The focus of this thesis has been design and development of a high-performance I/O subsystem in general and an approach to the specific case of I/O with the WildForce in particular. The FlexIO driver was evaluated by transferring various sets of data and by comparing it with other modes of transfer. FlexIO driver was tested for the following cases.

- Disk-Disk I/O
- Network-Network I/O
- Disk-Network I/O
- WildForce I/O

4.1 Disk-Disk I/O

FlexIO was used to copy an input file to an output file. It was used to perform I/O in different ways and was compared to a user process performing I/O. The different scenarios are explained as we proceed.

To evaluate the performance of the driver for different hard disk interfaces, the following tests have been done for four different scenarios

1. File transfer to the same partition of an IDE disk

2. File transfer to a different partition of an IDE disk

3. File transfer to the same partition of a SCSI disk

4. File transfer to a different partition of a SCSI disk

The results for the tests are provided after the scenarios have been explained and then the results are discussed.

4.1.1 Test Setup

The test setup that was used for the tests is shown in Figure 4.1. The test machine housing the IDE disks was a Linux Box with a Pentium-II 400MHz processor with 128MB RAM and the test machine housing the SCSI disks was a Linux Box with a Pentium Pro 200 MHz processor and 128MB RAM.

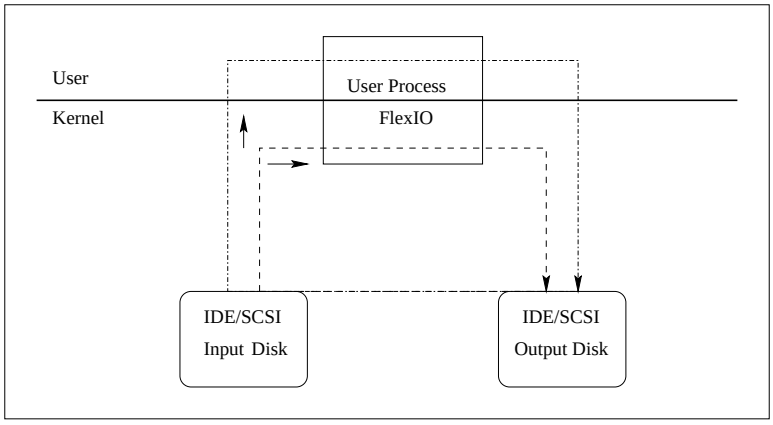

Figure 4.1: Test Setup for Disk-Disk I/O

4.1.2 Comparison between using User Process and FlexIO for Disk-Disk I/O

To test and compare the ability of the FlexIO driver to perform Disk-Disk I/O with the conventional Disk-Disk I/O methods, I/O using FlexIO was compared with a user process performing Disk-Disk I/O. To test the effect of various component operations on the I/O throughput, I/O was performed in several ways.

FlexIO was used to perform the I/O in the following ways.

1. *Using a kernel buffer* as shown in Figure 4.2. FlexIO performs the I/O with the control being in the kernel. It reads 1MB at a time from the input file and writes 1MB at a time to the output file. The reason for choosing 1MB is explained in Section 4.1.4.

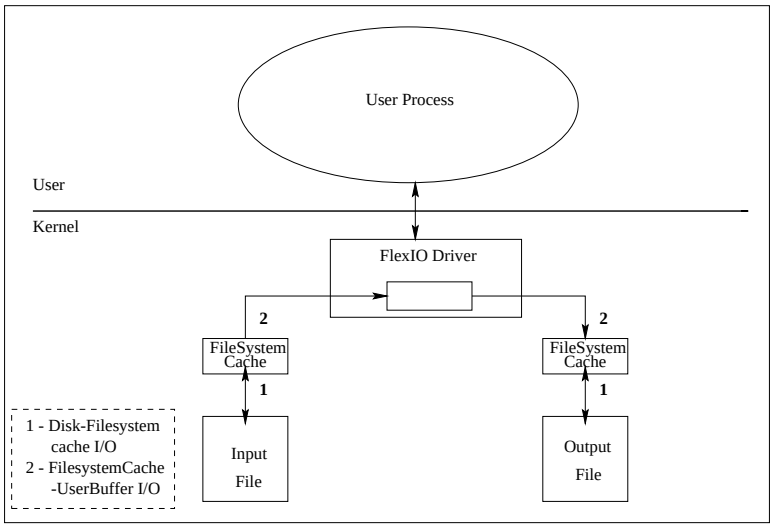

Figure 4.2: FlexIO, Using a kernel buffer

2. *Performing I/O at the buffer cache or filesystem cache level* as shown in the Figure 4.3. FlexIO performs the I/O with the control being in the kernel. It reads blocks from the disk onto the buffer cache, copies the data to the buffers pertaining to the output file which will be written onto the disk.

3. *Performing I/O memory mapping the files*, as shown in the Figure 4.4. FlexIO performs the I/O with the control being in the kernel. It maps the input and output files into kernel memory. Hence, we can copy the files as though we copy memory locations, as a file is treated as memory here. This again copies the data at the buffer cache level, with the difference from that of I/O at the buffer cache level being this uses read-ahead schemes.

A user process was used to perform the I/O in the following ways.

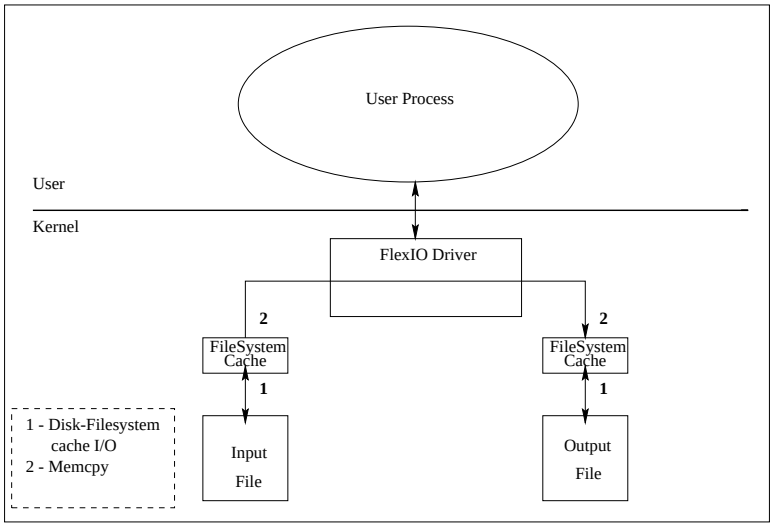

Figure 4.3: FlexIO, I/O at the buffer cache level

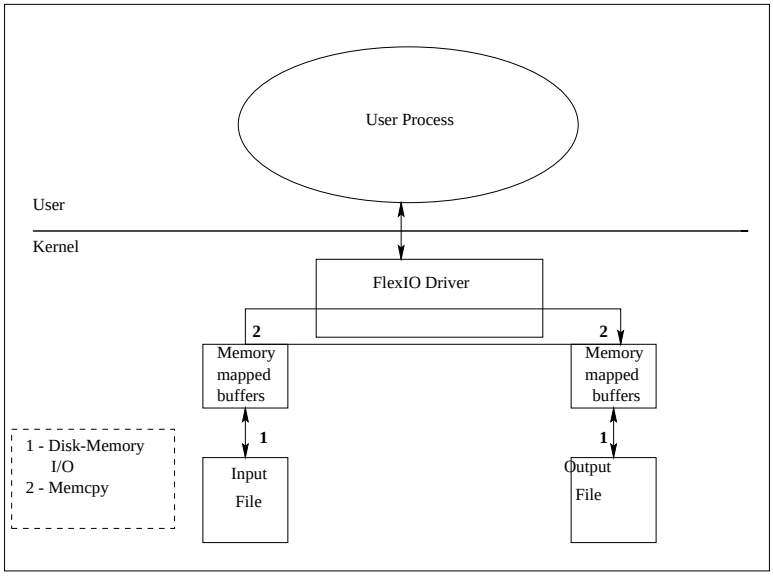

Figure 4.4: FlexIO, Memory mapping the files

1. *Using a user buffer* as shown in the Figure 4.5. The data flow is not in the kernel here. The user process reads 1MB of data from the input file and writes 1MB of data to the output file. The reason for choosing 1MB is explained in Section 4.1.4. It uses the standard file entry points such as read(), write() system calls for the I/O.

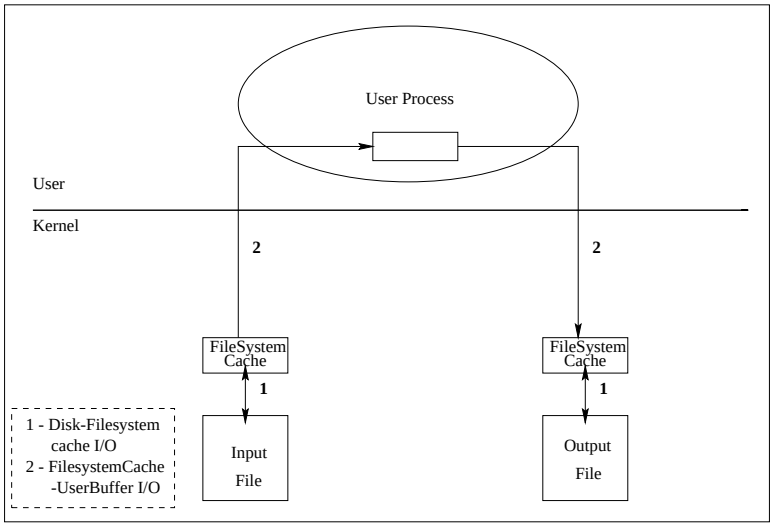

Figure 4.5: User Process, Using a user buffer

2. *Memory mapping the files* as shown in Figure 4.6. This is something similar to FlexIO performing the memory mapping. It uses the mmap() system call to map the input and output file into user memory. Hence, we can copy the files as though we copy memory locations, as a file is treated as memory here. This again copies the data at the buffer cache level with some read-ahead schemes, but the only difference here being the copy is done by a user process.

4.1.2.1 Results for the comparison

The results for file transfer to the same IDE disk partition, different IDE disk partition, same SCSI disk partition and different SCSI disk partition are respectively shown in Figures 4.7, 4.8, 4.9 and 4.10. As seen from the results, the disk I/O throughput for SCSI is better than that of IDE. Memory mapping the files improves the throughput

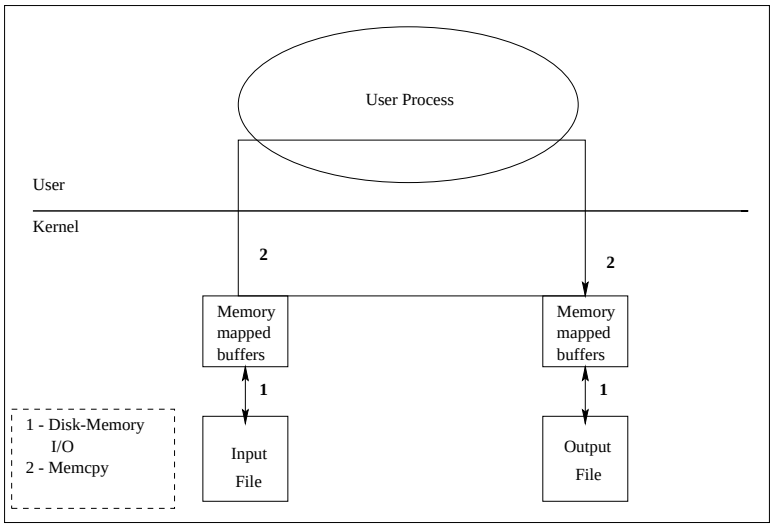

Figure 4.6: User Process, Memory mapping the files

for files as large as 200MB. The results are explained in detail in Section 4.1.4.

4.1.3 Effect of I/O Buffer Size used for the Data Transfer on the Throughput

To test the effect of the size of the I/O buffer used for the data transfer on the I/O throughput, a user process was used to perform the I/O with different user buffer sizes starting from 1KB till 1MB. This was compared with FlexIO performing I/O at the buffer cache level as it does not use any intermediate buffers and hence eliminates two copies.

4.1.3.1 Results for Effect of I/O Buffer Size on the Throughput

The results for file transfer to the same IDE disk partition, different IDE disk partition, same SCSI disk partition and different SCSI disk partition are respectively shown in Figures 4.11, 4.12, 4.13 and 4.14. As seen from the results, the throughput increases as the size of the buffer increases, but the increase is not very significant when compared to the actual throughput. The results are explained in detail in Section 4.1.4.

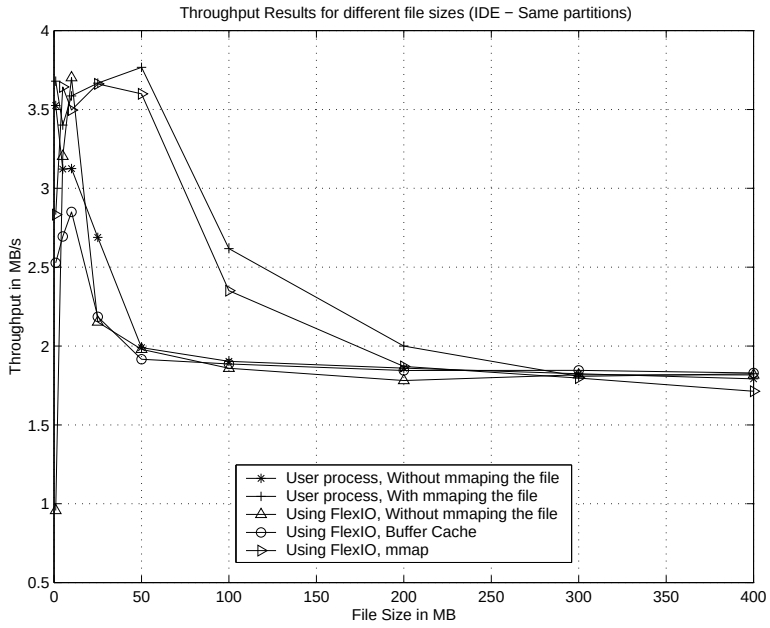

Figure 4.7: Throughput for I/O to the same IDE disk partition

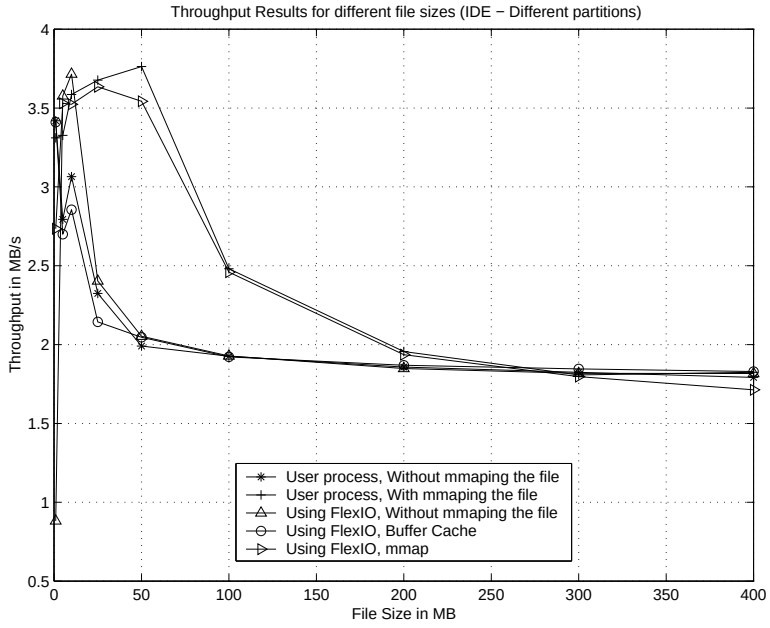

Figure 4.8: Throughput for I/O to a different IDE disk partition

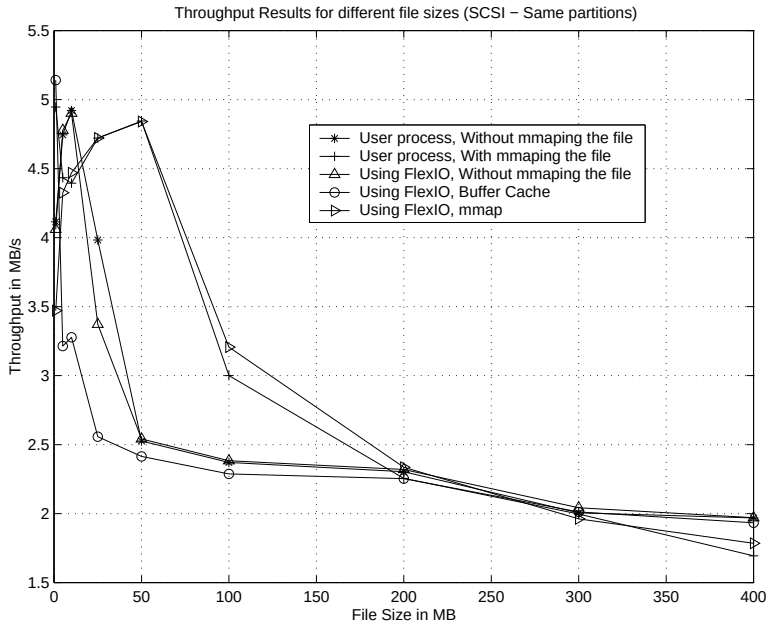

Figure 4.9: Throughput for I/O to the same SCSI disk partition

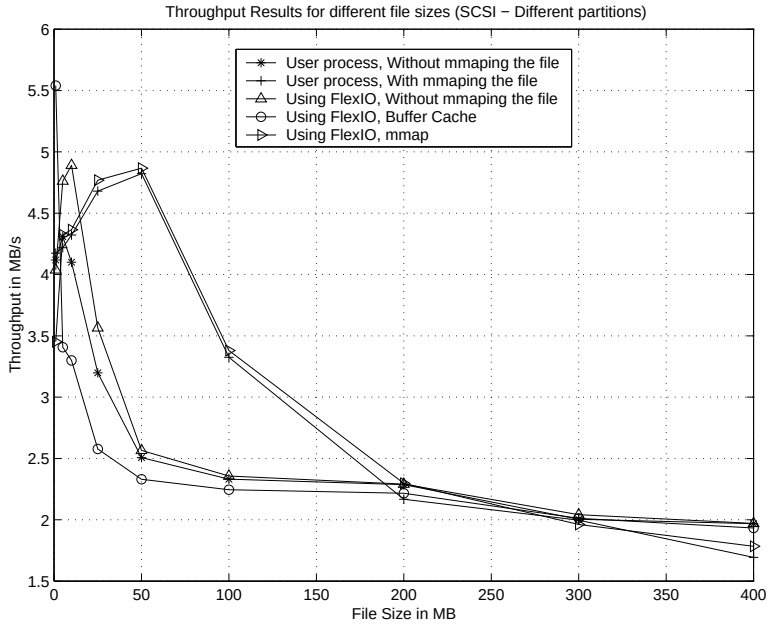

Figure 4.10: Throughput for I/O to a different SCSI disk partition

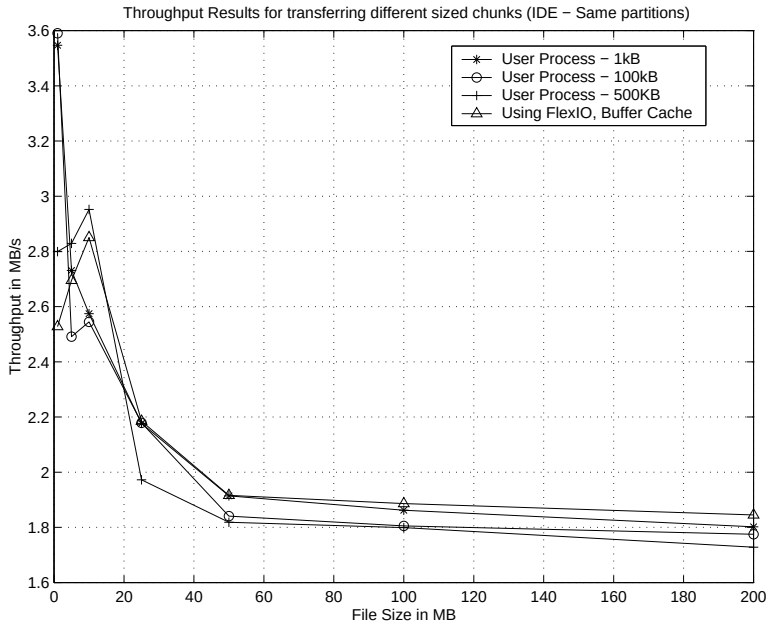

Figure 4.11: Effect of size of the data transfer, same IDE disk partition

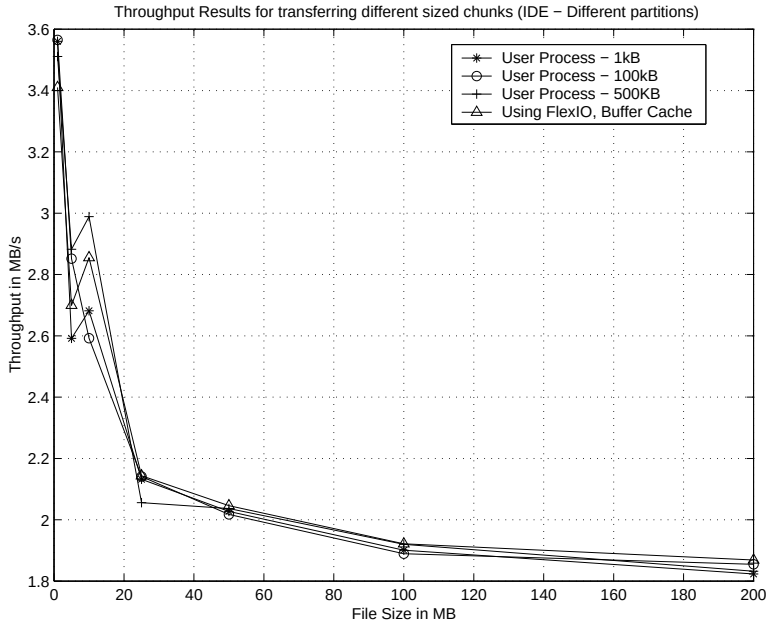

Figure 4.12: Effect of size of the data transfer, different IDE disk partition

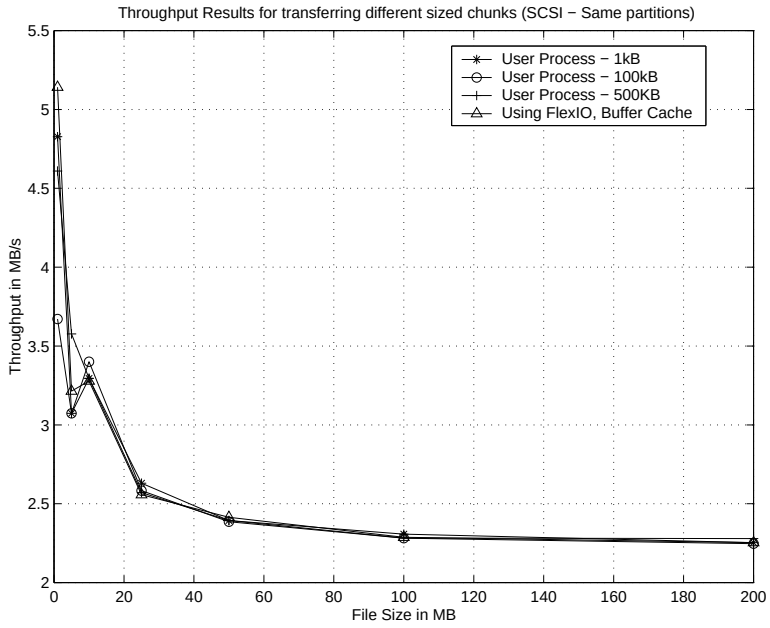

Figure 4.13: Effect of size of the data transfer, same SCSI disk partition

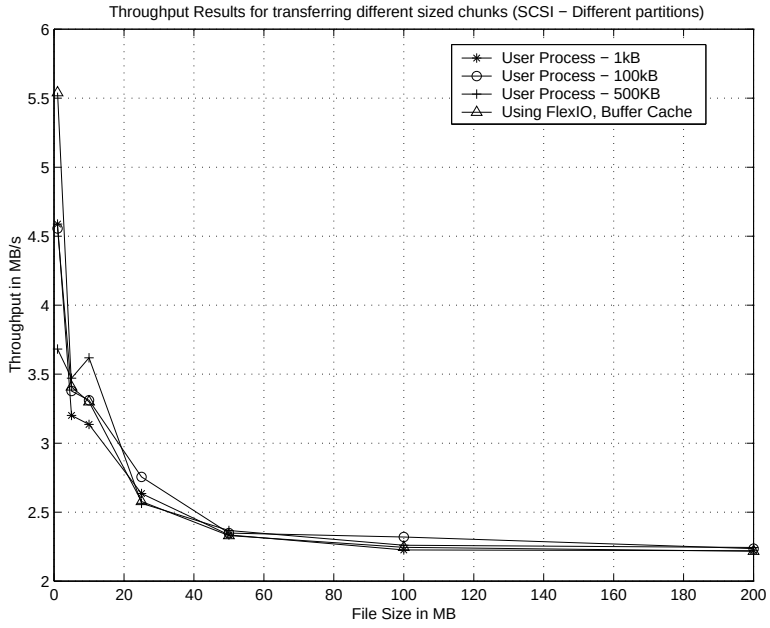

Figure 4.14: Effect of size of the data transfer, different SCSI disk partition

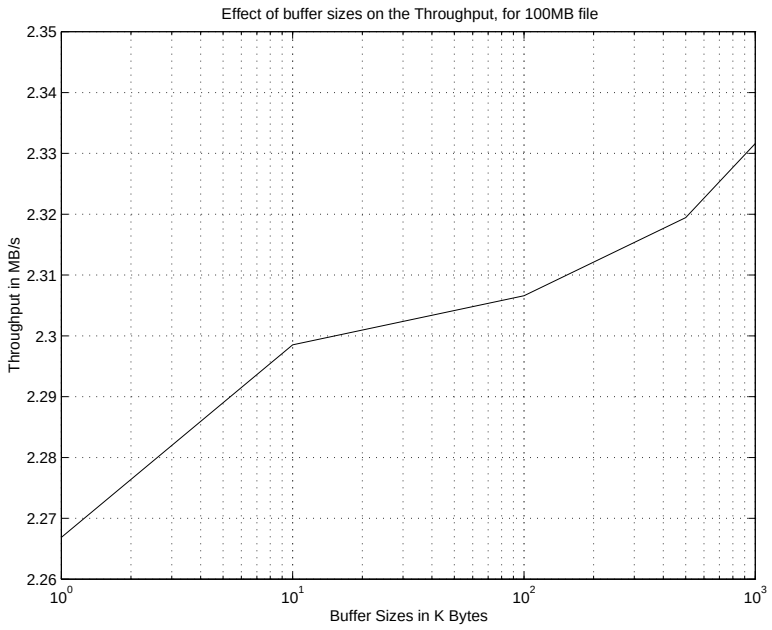

Figure 4.15: Effect of size of the I/O buffer on throughput for a 100MB file transfer

4.1.4 Inference

As seen from the results, the following can be inferred.

The throughput for small file transfers and the sustained throughput for large file transfers is more in the case of disk I/O with SCSI when compared to IDE. The performance of the system is almost similar when performing I/O with the same disk partition and when performing I/O with a different disk partition

Memory mapping the files improves the performance of the I/O for transfers as large as 50MB. This is because memory mapping the file uses read-ahead schemes which makes the I/O faster by reading ahead the file into the filesystem cache. But for files that are larger than 50MB, the throughput comes down. Even though, the same read-ahead scheme is used here, since the file is very large and the filesystem cache is limited, the read-ahead scheme does not help after a stage when the buffers have to be written to the disk and the write operation is time consuming as disk I/O is generally slower compared to the speed at which memory is accessed and hence the throughput reduces. For files as large as than 200MB, the throughput using the

memory mapping case and other modes of performing I/O is almost the same. For files larger than 200MB, the throughput of non-memory mapping case is better than that of memory mapping case. This is more significant in case of SCSI than IDE. So, for maximizing the sustained throughput, the data needs to be synced to the disks on a regular basis.

The throughput in the case of FlexIO performing I/O at the buffer cache level and that of FlexIO and the user process using an intermediate buffer is almost the same. This is because, even though in the FlexIO buffer cache case, it avoids copying the data back and forth to an intermediate buffer and hence we would expect it to be faster, the data here is transferred in smaller chunks and for each transfer, i.e. each time a read or a write happens, lot of computations are done in-order to get the details of the block to be read from the filesystem, which is time consuming. Whereas in the case of using an intermediate buffer, the number of copies are increased but the number of filesystem computations are reduced as it is done a fewer number of times for larger buffers. As can be seen from the Figures 4.11, 4.12, 4.13 and 4.14, the throughput in both the cases have almost identical behaviour.

As seen from Figure 4.15, the throughput increases as the size of the intermediate buffer increases. But, the difference between transferring 1KB at a time and transferring 1MB at a time is about 100KB/s, which is not very significant when compared to the actual throughput. In the tests that were conducted, the intermediate buffer size was chosen to be 1MB as the lowest sized file that was transferred was 1MB and the other files were multiple of 1MB and also that in case of WildForce I/O, a buffer of 1MB was used because WildForce board has an on-board memory of 1MB and FlexIO was used to perform I/O with WildForce board too. Hence, we wanted to maintain the same test scenario for the Disk-Disk transfer too.

4.2 Disk-Network I/O

To test and compare the ability of the FlexIO driver to perform Disk-Network I/O with the conventional I/O methods, I/O using FlexIO was compared with a user process

performing file transfer over UDP.

4.2.1 Comparison between using UDP and FlexIO for Disk-Network I/O

FlexIO was used to perform the I/O in the following way. Read the data from the buffer cache, packetize it and transfer over the network and write it to the buffer cache as shown in Figure 4.16. FlexIO reads the input file in chunks from the buffer cache and then packetizes it and sends it across the network. The FlexIO driver at the receiver end then receives the packet, and writes the information to the file in chunks through the buffer cache.

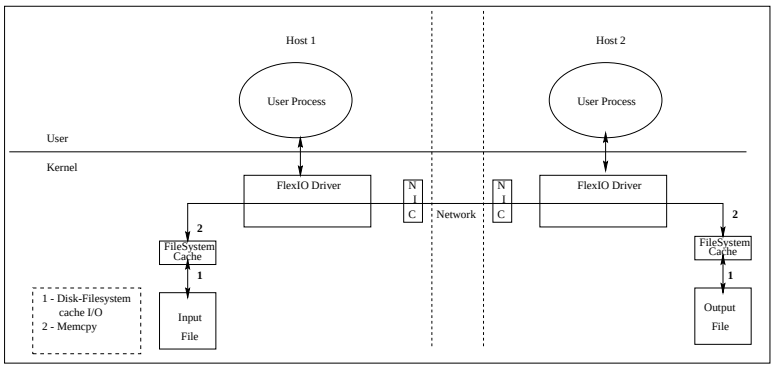

Figure 4.16: FlexIO, I/O at the buffer cache level

This was compared with a user process performing I/O over UDP in the following way. Read data from the file, transfer over UDP and write it to the file as shown in Figure 4.17. A user process reads the input file using read() system call. It then writes the data over a UDP socket. The receiver then writes the data to the file using write() system call.

4.2.2 Effect of Send and Receive Buffers on the I/O

To test the effect of send and receive buffers on the I/O throughput, a user process was used to perform the I/O with different send and receive buffer sizes, starting from 1KB till 50KB. This was compared with FlexIO performing the I/O.

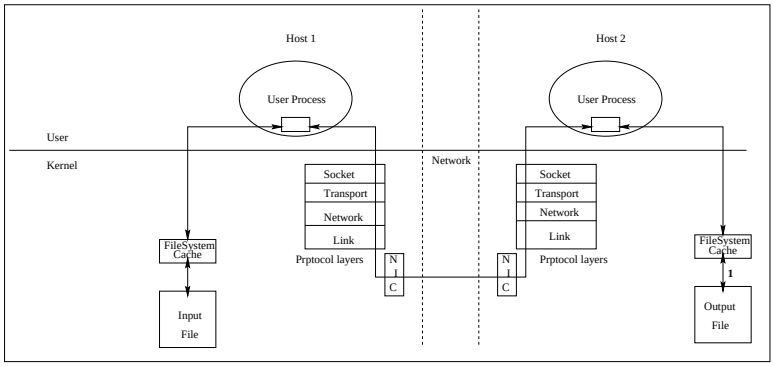

Figure 4.17: User Process, over UDP, using regular Filesystem APIs

4.2.3 Results and Inference

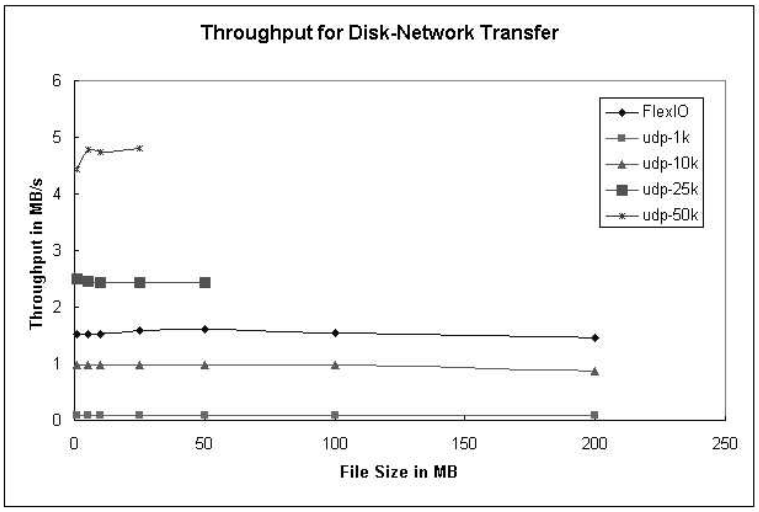

Figure 4.18: Throughput results for Disk-Network I/O

The results are shown in Figure 4.18. The data flow remains in the kernel in case of FlexIO as opposed to user-kernel transition in case of UDP transfer. Throughput in the case of FlexIO is better than that of UDP for large file transfers. The throughput in the case of UDP depends on the send and receive buffer size. The larger the buffer size, the higher the throughput. For buffer sizes of 50KB and 25KB, the throughput is greater than FlexIO. But, as seen from the figure, only up to 25MB can be transferred

using 50KB buffer size and up to 50MB can be transferred using 25KB buffer size. This is because, the data is not read by the user process as fast as it received and hence some of the packets are lost when the buffer is full. Hence these throughputs are thus not sustainable. Using FlexIO, a constant throughput of around 1.55MB/s can be obtained for files as large as 200MB.

4.3 Network-Network I/O Results and Inference

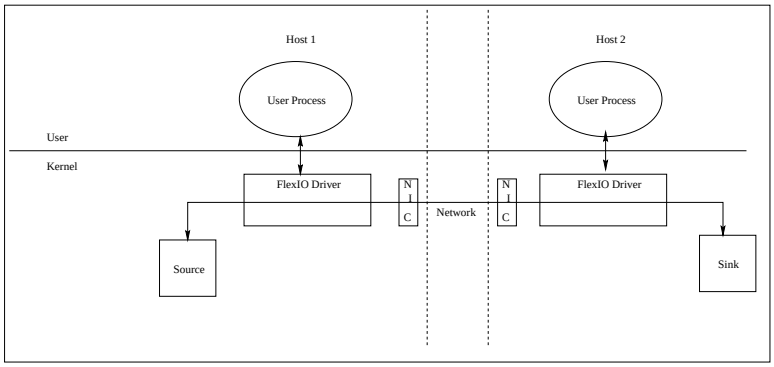

Figure 4.19: Network-Network I/O between two hosts

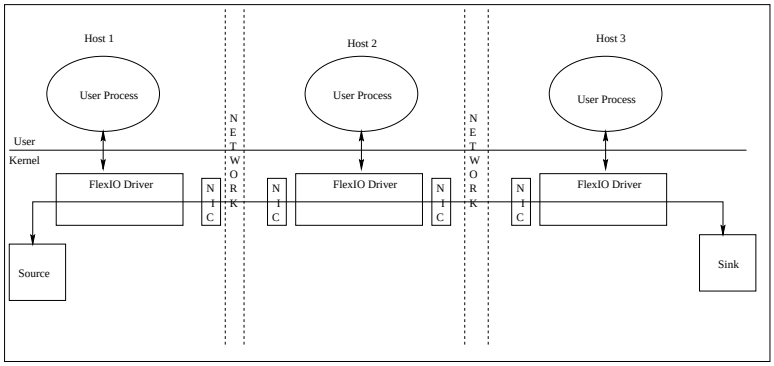

Figure 4.20: Network-Network I/O between three hosts

To test the sustained throughput for the network I/O using FlexIO, dummy packets with sequence numbers were sent over the network using FlexIO. This did not involve

reading and writing to the disk. The network I/O was performed between two machines as shown in Figure 4.19 and between three machines as shown in Figure 4.20. The intermediate host in the three machine network case was just used to forward the packets from source to the sink. The respective results are shown in Figures 4.21 and 4.22. As seen from the results, a sustained throughput of around 2.3MB/s is obtained for transferring 400MB. Higher throughput can be obtained for smaller data sets. This is because, for smaller data sets, the Ethernet driver will be able to find buffers when packets arrive at the card. If large data sets arrive at the same rate, freeing used buffers and allocating new buffers takes time and hence the Ethernet card starts dropping the packets. So, we chose lower data rates for sustainability.

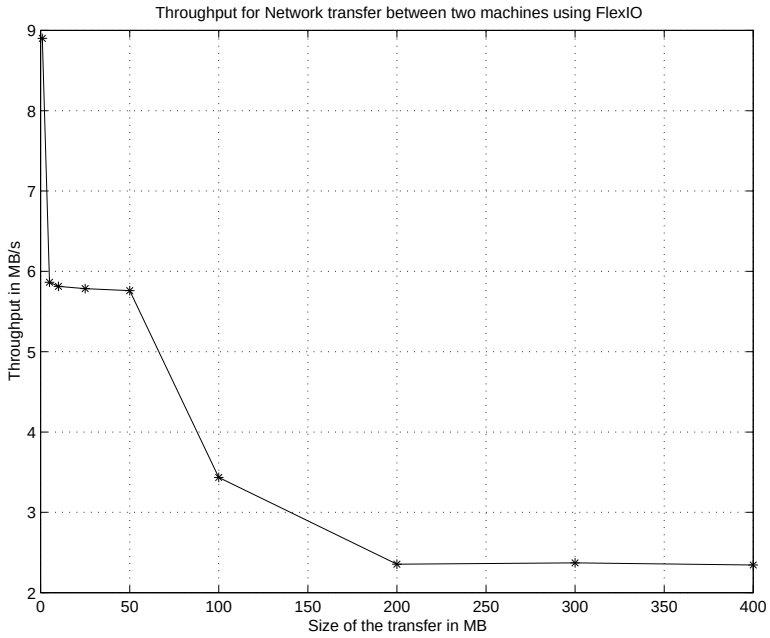

Figure 4.21: Throughput results for Network-Network I/O between two hosts

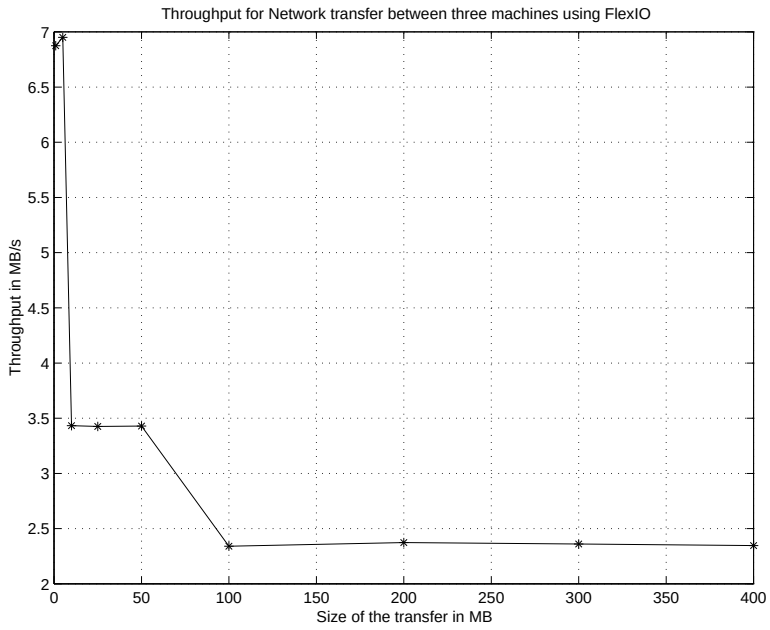

Figure 4.22: Throughput results for Network-Network I/O between three hosts

4.4 WildForce I/O

4.4.1 FlexIO vs WildForce APIs

FlexIO was used to perform I/O with the WildForce board and compared against WildForce APIs used for the I/O

FlexIO was used for the I/O as shown in Figure 4.23. In this case, WildForce user APIs were still used to configure the WildForce board and the WildForce driver. Operations such as downloading digital code onto the FPGAs on the board, setting the clock and so on were done by the WildForce APIs. FlexIO was used to perform the I/O instead of the user level WildForce APIs.

A user process doing the I/O used WildForce APIs as shown in Figure 4.24. In this case, WildForce APIs were used for the whole operation, both for the I/O and for the other configuring stuff as-well.

FlexIO was also used to perform the I/O for SAR processing on the WildForce board. This involved performing I/O with all the FPGAs on the board.

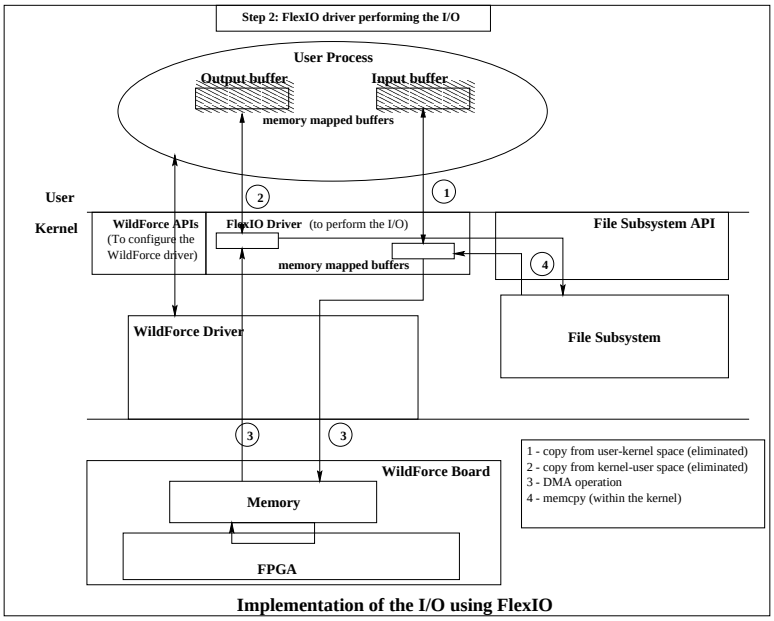

Figure 4.23: FlexIO performing the I/O

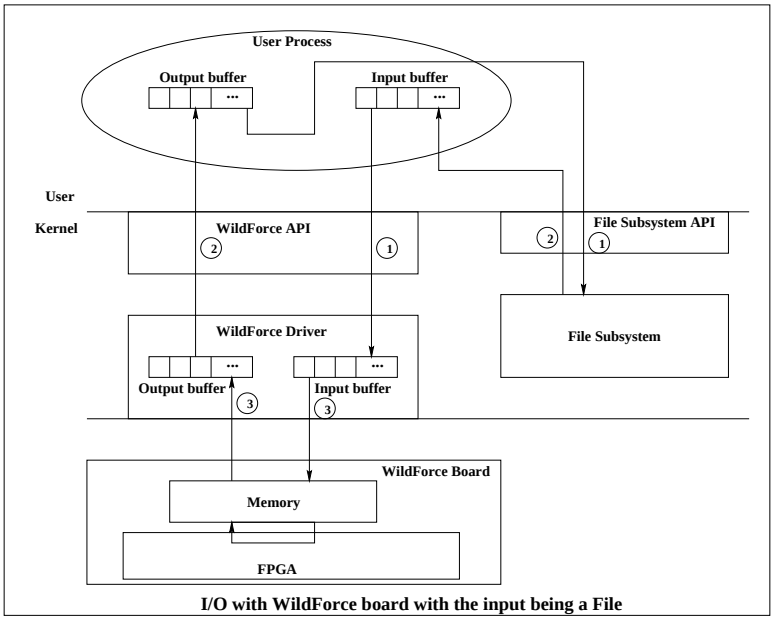

Figure 4.24: I/O using WildForce APIs

4.4.2 WildForce I/O Results and Inference

The results are shown in Figure 4.25. The results shown in the figure are for a scenario where the input and output data are in files. The graph shows a comparison of FlexIO performing I/O with a user process performing the I/O.

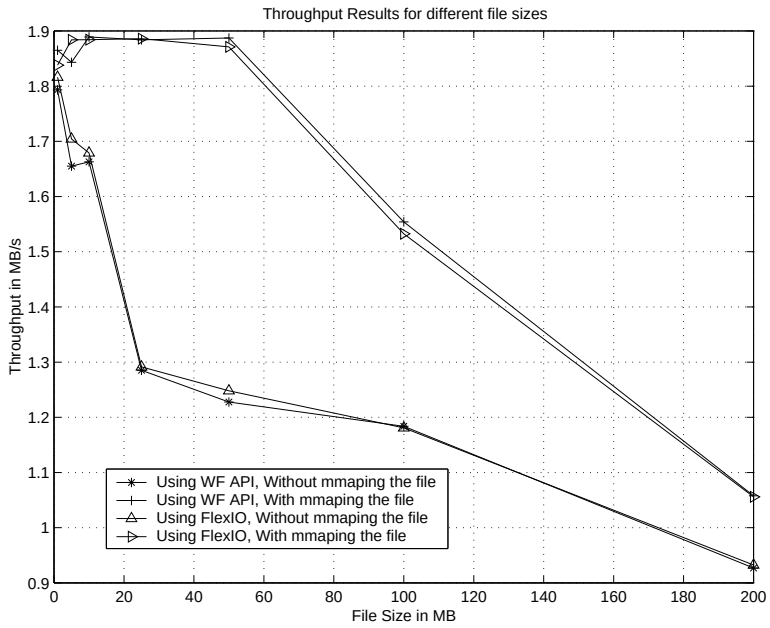

Figure 4.25: Throughput results for I/O with the WildForce board

As seen from the figure, Memory-mapping the files improves the throughput for files as large as 200MB. The difference in throughput is more for files as big as 50MB and the difference narrows down for larger files. The reason for this being, by memory mapping the files, the operating system uses read-ahead schemes to read from the file. But, since the buffer cache is much smaller when compared to the actual memory and disk size, smaller files can take advantage of the read-ahead scheme. However for large files, even though the same read-ahead scheme is used, the buffer cache fills quickly and time is wasted in managing the buffer cache. Hence large files cannot make the best use of the read-ahead scheme beyond a certain point and hence the throughput drops.

The data flow remains in the kernel in case of FlexIO as opposed to user-kernel

transition in case of a user process using WildForce APIs. FlexIO can be used for performing I/O with the WildForce board for real-time applications. This is demonstrated by FlexIO performing I/O with the WildForce board for SAR processing.

Chapter 5

Conclusions and Future Work

5.1 What has been achieved?

As discussed earlier in this report, improvements in computational science has led to the usage of faster CPUs, more main memory, larger disks and fast network devices. Though the industry is moving towards hardware, some of the operations are still done in software, which might become a bottle-neck for fast I/O applications. *FlexIO driver* is an effort in designing a high-performance I/O subsystem. The following conclusions can be made from this work.

- Conventional software I/O APIs are not suited for all kinds of applications. Hence the I/O subsystem needs to be re-designed in such cases.
- It makes sense for the I/O flow to remain in the kernel for some of the applications and software architecture should be changed for such applications.

We have developed an application for high-performance I/O in the kernel. This application in the form of a pseudo device driver provides the following functionality.

- It lets a user process define and control I/O flows in the kernel.
- User-kernel and kernel-user memory mapping is supported which are useful for various kinds of applications.
- It supports the following kinds of I/O

  - Disk-Disk I/O (among raw disk partitions and Ext2 filesystem)
  - Network-Network I/O
  - Disk-Network I/O (Ethernet and ATM)
  - I/O with a specialized PCI card called WildForce housing FPGAs

- It can be used to perform disk I/O within the kernel. However, it offers the same performance as that of disk I/O using the filesystem APIs. The advantages FlexIO offers here is that, the I/O flow remains in the kernel and I/O can be performed in three different ways; using an intermediate kernel buffer, I/O at the buffer cache level and memory mapped I/O.

- It can be used to perform network I/O and disk-network I/O with higher sustained throughput for large data sets when compared to UDP based transfers. Hence, FlexIO can be useful to get better performance in distributed applications on a local network where congestion, drop and re-transmission are not an issue.

- It provides the framework for generalized high-performance I/O subsystem in the kernel by demonstrating the ability to perform I/O with a specialized PCI card. It also demonstrates how I/O among dissimilar input and output entities can be controlled from within the kernel, Disk-Network I/O and File-WildForce I/O being the examples.

5.2 What can be done?

Some of the possible future work is discussed here.

- FlexIO can be used for streaming real-time audio and video data. It will be interesting to extend the functionality of the driver to stream audio and video data and compare the performance against some of the conventional streaming methods.

- Though memory mapping functionality is supported in the driver, it was not used for WildForce I/O in the true sense because the WildForce APIs still copy

the data across address spaces. It will be interesting to use FlexIO to perform the I/O for a modified WildForce driver which does not copy the data multiple times.

- FlexIO does not use TCP/IP protocols for network I/O. FlexIO can be suitably modified and used for testing the performance of some of the emerging protocols as it provides high performance compared to TCP/UDP based applications.

Bibliography

[1] Abraham Silberschatz, Peter Galvin, *Operating System Concepts*, Addison Wesley, 5th edition, 1998

[2] *Adaptive Computing Systems @ KU*, *A Functional Programming Environment for Design and Implementation of High Performance Radio and Synthetic Aperture Radar Processing Functions*, http://www.ittc.ukans.edu/ACS/

[3] Alessandro Rubini, *Linux Device Drivers*, O'Reilly, 1998

[4] *Annapolis Micro Systems Inc.*, *WildForce reference Manual*, 1999, revision 3.4.

[5] David Darus, *Network Processor Architecture Design Trends*, http://user.gru.net/darusdp/cda5155paper.html

[6] David Rusling, *The Linux Kernel*, Version 0.8-3, http://www.linuxdoc.org/LDP/tlk/tlk.html

[7] Internet Processor II http://www.juniper.net/news/features/ipii, http://www.juniper.net/news/features/ipii/faq_asics.html

[8] *Linux Kernel Hackers' Guide*, http://khg.redhat.com/HyperNews/get/khg.html

[9] Maurice J Bach, *The Design of the UNIX Operating System*, Thirteenth Printing, Prentice-Hall, July 1997

[10] M Beck, H Bohme, M Dziadzka, U Kunitz, R Magnus, D Verworner, *Linux Kernel Internals*, Addison-Wesley, 2nd edition, 1998

[11] *PCI/ISA Bridge Functional Description*,
http://developer.intel.com/design/intarch/techinfo/430tx/brdgfunc.htm,
DMA Controller Unit,
http://developer.intel.com/design/iio/manuals/techinfo/80960rmrn/19_DMA.htm

[12] Randy Kath et al, *Managing Memory-Mapped Files in Win32*,
http://dept-info.labri.u-bordeaux.fr/ betrema/winnt/manamemo.html

[13] W. Richard Stevens, *Advanced Programming in the UNIX environment*, Addison Wesley, First ISE reprint, 1998

[14] W. Richard Stevens, *UNIX Network Programming*, Prentice-Hall, 1997

[15] Yannis Smaragdakis, Scott F. Kaplan, and Paul R. Wilson, *EELRU: Simple and Effective Adaptive Page Replacement*, presented at SIGMETRICS '99,
http://www.cs.amherst.edu/ sfkaplan/papers/

Appendix A

A HOWTO to write a simple device driver in Linux

Typically a device driver is a piece of software to interact with the hardware. A pseudo device driver (like the one explained in this report) does not directly control any particular piece of hardware, but uses the device driver like interface as it is very flexible and useful for certain kinds of applications. DataStream Kernel Interface driver is one other example.

The structure of a character device driver is explained here by explaining the entry points to the driver. The entry points are determined by the entries in the *file_operations* structure. The driver init routine registers itself with the character device table at boot time. The character device table holds the information about the major number, entry points and so on. of the driver. The rest of the driver is explained in the comments section of the code.

A.1 my_driver.c: A simple character driver skeleton

```
/* Major number of the device. We could choose any number that is not
 * already existing, in this we have just chosen 30 for eg.
 */
#define MYDRIVER_MAJOR 30

/* File operations we define for the driver. This is what defines what
 * entry points the driver is going to have.
 * In this case, we only have open(), close() and ioctl()
 * We could have any other if there is a need.
 */
static struct file_operations my_driver_fops = {
        NULL,   /* loff_t lseek(struct file *,loff_t,int) */
        NULL,   /* ssize_t read(struct file *,char *,size_t,loff_t *) */
        NULL,   /* ssize_t write(struct file *, const char *, size_t, loff_t *) */
        NULL,   /* int readdir(struct file *,void *,filldir_t) */
        NULL,   /* u_int poll(struct file *, struct poll_table_struct *) */
        my_driver_ioctl,   /* int ioctl(struct inode *,struct file *,u_int,u_long) */
        NULL,   /* int mmap(struct file *, strcut vm_armea_struct *) */
        my_driver_open,    /* int open(struct inode *, struct file *) */
        NULL,   /* int flush(struct file *); */
        my_driver_close,   /* int release(struct inode *,struct file *) */
        NULL,   /* int fsync(struct file *,struct dentry *) */
        NULL,   /* int fasync(int, struct file *, int) */
        NULL,   /* int check_media_change(kdev_t dev) */
        NULL,   /* int revalidate(kdev_t dev) */
        NULL    /* int lock(struct file*, int, struct file_lock *) */
};

/* The driver init fn. One of the most important things to do here is
 * register the driver as a character/block device driver. It makes an
 * entry in the character/block device table (this is the case if the
 * driver is built into the kernel...driver is up when the kernel boots)
 * This fn will be called by some other kernel routine where all the
 * drivers are initialized
 * If we do not want the driver to be built into the kernel, we can make it
 * as a module (load/unload the driver after the kernel boots). In this case,
 * the module init will have to register the driver as a character/block
 * device driver.
 */
int my_driver_init(void)
{
        /* MYDRIVER_MAJOR will be the same number with which an entry
         * in /dev will be made using mknod...for eg.
         * mknod /dev/my_driver c 30 0
         * will create a device file called my_driver in /dev....which
         * will be used by the user code to open/ioctl/close the driver
         * as done for a normal file
         */
        register_chrdev(MYDRIVER_MAJOR, "mydriver", &my_driver_fops);
}

/* This is the fn which opens the driver and does some data structure
 * initializations.....ultimately this is what returns the file descriptor
 * (file handle) to the user process which opened the driver. This file
 * descriptor is used for accessing the driver from then on in the user code.
 * But, for that we need not return the file descriptor here. The file
 * subsystem takes care of that
 */
static int my_driver_open(struct inode *inode, struct file *filp)
{
        /* We can also initialize the driver private data structures here
         * and have the pointer (if needed) in the filp->private_data,
```

```
         * so that it can be used anywhere in the driver as all the
         * entry points have the struct file *filp as an argument.
         */
}

/* fn to close the driver */
int my_driver_close(struct inode *inode, struct file *filp)
{
        /* We can also un-initialize the driver private data structures here
         * and free all the memory what we allocated
         */
}

/* ioctl interface for the driver. Can be used to configure the driver options
 * using this. As discussed before, it gets passed the filp pointer. So, we
 * have access to the private data structures of the driver, which can be
 * modified here based on the user input. For eg. if it a driver for I/O
 * and the private data structures holds information related to the I/O,
 * it can be modified/configured here based on the user input
 */
static int my_driver_ioctl(struct inode *inode, struct file *filp,
                           unsigned int cmd, unsigned long val)
{
        switch (cmd) {
                case: IOCTL_DEFINITIONS
                /* related code for the ioctl */
        }
}
```

Appendix B

User programs

B.1 tsc.h: Timer related stuff

This is a header file which has the assembly routines to read the Time Stamp Counter (TSC). *cycles_to_usec()* converts the CPU ticks into micro seconds. *read_proc()* reads the */proc/flexio/timer* file to get the values of *cycles_per_seconds, cycles_per_usec* and *cycles_per_jiffies* values calibrated by the FlexIO driver at boot time. These are used to calculate the elapsed time.

```
#ifndef __TSC_H
#define __TSC_H

#include <stdio.h>

/* Macros to read TimeStamp Counter */

#define rdtscl(low) \
     __asm__ __volatile__ ("rdtsc" : "=a" (low) : : "edx")

#define rdtscll(val) \
     __asm__ __volatile__ ("rdtsc" : "=A" (val))

#define HZ 100
#define USEC_PER_SEC 1000000L
#define USEC_PER_JIFFIES (1000020/HZ)

unsigned long cps = 0;
unsigned long cpus = 0;
unsigned long cpj = 0;

/* Convert ticks to micro seconds */
inline unsigned long long cycles_to_usec(unsigned long long update)
{
        return ((unsigned long long)(((double)update*(double)USEC_PER_JIFFIES)/(double)cpj));
}

/* Read /proc/flexio/timer for the cycles per jiffies value
```

```
 * calibrated by FlexIO driver
 */
int read_proc(void)
{
        FILE *fp;
        char buff[200], a[3][16];

        memset(buff, 0, 200);
        fp = fopen("/proc/flexio/timer", "r");
        if (!fp) {
                perror("fopen");
                exit(1);
        }

        fgets(buff, sizeof(buff), fp);
        fgets(buff, sizeof(buff), fp);
        sscanf(buff, "%s\t%s\t%s\n", a[0], a[1], a[2]);

        cps = (unsigned long)atoi(a[0]);
        cpus = (unsigned long)atoi(a[1]);
        cpj = (unsigned long)atoi(a[2]);

        fclose(fp);
}

#endif // __TSC_H
```

B.2 fileIO.c: A simple file I/O by a user process

This is a simple user program used to perform file I/O. It copies *inputfile,* passed from the command line to *outputfile* using *read()* and *write()* system calls.

```
USAGE - ./fileIO inputfile sizeof_individual_transfer

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>

#include "tsc.h"

unsigned long long startTicks=0, endTicks=0;

int main(int argc, char **argv)
{
        int file_size, size, ret;
        int fdin, fdout;
        struct stat statbuf;
        char *buf;

        read_proc();

        /* Input file */
        fdin = open(argv[1], O_RDONLY);
```

```
        if (fdin < 0) {
                perror("open");
                exit(1);
        }
        if (fstat(fdin, &statbuf) < 0) {
                perror("fstat");
                close(fdin);
                exit(1);
        }
        file_size = statbuf.st_size;

        /* Output file */
        fdout = open("./outputfile", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU|S_IRWXG|S_IRWXO);
        if (fdout < 0) {
                perror("open");
                close(fdin);
                exit(1);
        }
        /* Size of the individual data transfer */
        size = atoi(argv[2]);
        buf = (char *)malloc(size);
        memset(buf, 0, size);

        if (file_size < size)
                size = file_size;

        rdtscll(startTicks);
        while (file_size > 0) {
                ret = read(fdin, buf, size);
                if (ret < 0) {
                        perror("read");
                        close(fdin); close(fdout);
                        free(buf);
                        exit(1);
                }
                ret = write(fdout, buf, size);
                if (ret < 0) {
                        perror("write");
                        close(fdin); close(fdout);
                        free(buf);
                        exit(1);
                }
                file_size -= size;
                if (file_size < size)
                        size = file_size;
        }
        rdtscll(endTicks);
        fprintf(stderr, "Time for the operation was: %Lu
                micro seconds\n", cycles_to_usec(endTicks-startTicks));

        free(buf);
        fsync(fdin); fsync(fdout);
        close(fdin); close(fdout);
}
```

B.3 fileIO_mmap.c: A simple file I/O by a user process using mmap

This is a user program used to perform file I/O for memory mapped files. It copies *inputfile*, passed from the command line to *outputfile*. It maps the files using *mmap()*. It then copies the files using *memcpy()* system call. For details, refer Richard Stevens[13].

```
USAGE - ./fileIO_mmap inputfile

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/time.h>

#include "tsc.h"

unsigned long long startTicks=0, endTicks=0;

int main(int argc, char **argv)
{
        char *ifsptr=NULL, *ofsptr=NULL;
        int fdin, fdout;
        struct stat statbuf;

        read_proc();
        /* Input file */
        fdin = open(argv[1], O_RDONLY);
        if (fdin < 0) {
                perror("open");
                exit(1);
        }
        if (fstat(fdin, &statbuf) < 0) {
                perror("fstat");
                close(fdin);
                exit(1);
        }
        if ((ifsptr = mmap(0, statbuf.st_size, PROT_READ,
                           MAP_FILE|MAP_SHARED, fdin, 0)) == NULL) {
                perror("mmap");
                close(fdin);
                exit(1);
        }
        /* Output file */
        fdout = open("./outputfile", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU|S_IRWXG|S_IRWXO);
        if (fdout < 0) {
                perror("open");
                close(fdin);
                exit(1);
        }
        /* Setting the size of the output file */
        if (lseek(fdout, statbuf.st_size-1, SEEK_SET) < 0) {
                perror("lseek");
                close(fdin); close(fdout);
                exit(1);
```

```
        }
        /* IF you do not do this, memcpy will result in a bus error */
        /* What it does is something like validating the address for the memcpy */
        if (write(fdout, "", 1) != 1) {
                perror("write");
                close(fdin); close(fdout);
                exit(1);
        }
        if ((ofsptr = mmap(0, statbuf.st_size, PROT_READ|PROT_WRITE,
                        MAP_FILE|MAP_SHARED, fdout, 0)) == NULL) {
                perror("mmap");
                close(fdin); close(fdout);
                exit(1);
        }

        rdtscll(startTicks);
        /* Actual transfer */
        memcpy(ofsptr, ifsptr, statbuf.st_size);
        rdtscll(endTicks);
        fprintf(stderr, "Time for the operation was: %Lu
                micro seconds\n", cycles_to_usec(endTicks-startTicks));

        fsync(fdin); fsync(fdout);
        close(fdin); close(fdout);
}
```

B.4 fl_fileIO.c, fl_bufcache_fileIO.c: File I/O using FlexIO

This is a user program used to perform file I/O using FlexIO. It opens the *inputfile* and *outputfile* and passes the information to FlexIO using FLEXIO_SET_INPUT and FLEXIO_SET_OUTPUT ioctls. The I/O is performed using FILE_KERNEL_READ_WRITE ioctl in case of file I/O using an intermediate kernel buffer or in case of file I/O at the buffer cache level, BUFCACHE_FILE_KERNEL_READ_WRITE ioctl is used.

```
USAGE - ./fl_fileIO inputfile

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/time.h>

#include <linux/flexio_ioctls.h>

#include "tsc.h"

#define PAGE_SIZE (1UL << 12)
#define PAGE_MASK (~(PAGE_SIZE-1))

unsigned long long startTicks=0, endTicks=0;
```

```
int main(int argc, char **argv)
{
        int fd, rval, *filedata, file_size;
        int fdin, fdout;
        struct stat statbuf;

        read_proc();
        filedata = (int *)malloc(sizeof(int)*2);

        /* Input file */
        fdin = open(argv[1], O_RDONLY);
        if (fdin < 0) {
                perror("open");
                exit(1);
        }
        if (fstat(fdin, &statbuf) < 0) {
                perror("fstat");
                close(fdin);
                exit(1);
        }
        /* Output file */
        fdout = open("./outputfile", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU|S_IRWXG|S_IRWXO);
        if (fdout < 0) {
                perror("open");
                close(fdin);
                exit(1);
        }
        /* Open the FlexIO driver */
        fd = open("/dev/flexio", 0);
        if (fd < 0) {
                perror("open");
                close(fdin); close(fdout);
                exit(1);
        }
        file_size = statbuf.st_size;
        filedata[0] = fdin;
        filedata[1] = file_size;
        rval=ioctl(fd, FLEXIO_SET_INPUT, (void *)filedata);
        if(rval!=0) {
                perror("FLEXIO_SET_INPUT");
                close(fd); close(fdin); close(fdout);
                exit(1);
        }
        filedata[0] = fdout;
        filedata[1] = file_size;
        rval=ioctl(fd, FLEXIO_SET_OUTPUT, (void *)filedata);
        if(rval!=0) {
                perror("FLEXIO_SET_OUTPUT");
                close(fd); close(fdin); close(fdout);
                exit(1);
        }
        rdtscll(startTicks);
        /* use FILE_KERNEL_READ_WRITE instead of BUFCACHE_FILE_KERNEL_READ_WRITE
         * for FlexIO to use a kernel buffer */
        rval=ioctl(fd, BUFCACHE_FILE_KERNEL_READ_WRITE, file_size);
        if(rval!=0) {
                perror("BUFCACHE_FILE_KERNEL_READ_WRITE");
                close(fd); close(fdin); close(fdout);
                exit(1);
        }
        rdtscll(endTicks);
        fprintf(stderr, "Time for the operation was: %Lu
                        micro seconds\n", cycles_to_usec(endTicks-startTicks));
```

```
        free(filedata);
        close(fd);
        fsync(fdin); fsync(fdout);
        close(fdin); close(fdout);
}
```

B.5 fl_mmap_fileIO.c: File I/O using FlexIO with mmap

This is a user program used to perform file I/O using FlexIO memory mapping the files. It opens the *inputfile* and *outputfile* and passes the information to FlexIO using FLEXIO_SET_INPUT and FLEXIO_SET_OUTPUT ioctls. The I/O is performed using FILE_MMAP_KERNEL_READ_WRITE ioctl.

```
USAGE - ./fl_mmap_fileIO inputfile

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/time.h>

#include <linux/flexio_ioctls.h>

#include "tsc.h"

#define PAGE_SIZE (1UL << 12)
#define PAGE_MASK (~(PAGE_SIZE-1))

unsigned long long startTicks=0, endTicks=0;

int main(int argc, char **argv)
{
        int fd, rval, *filedata, file_size;
        int fdin, fdout;
        struct stat statbuf;

        read_proc();

        filedata = (int *)malloc(sizeof(int)*2);

        /* Input file */
        fdin = open(argv[1], O_RDONLY);
        if (fdin < 0) {
                perror("open");
                exit(1);
        }
        if (fstat(fdin, &statbuf) < 0) {
                perror("fstat");
                close(fdin);
                exit(1);
        }
```

```
        /* Output file */
        fdout = open("./outputfile", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU|S_IRWXG|S_IRWXO);
        if (fdout < 0) {
                perror("open");
                close(fdin);
                exit(1);
        }
        /* Setting the output file size */
        if (lseek(fdout, statbuf.st_size-1, SEEK_SET) < 0) {
                perror("lseek");
                close(fdin); close(fdout);
                exit(1);
        }
        /* IF you do not do this, memcpy will result in a bus error */
        /* What it does is something like validating the address for the memcpy */
        if (write(fdout, "", 1) != 1) {
                fprintf(stderr,"Couldn't write to outputfile");
                perror("");
                close(fdin); close(fdout);
                exit(1);
        }
        fd = open("/dev/flexio", 0);
        if (fd < 0) {
                perror("open");
                close(fdin); close(fdout);
                exit(1);
        }
        file_size = statbuf.st_size;
        filedata[0] = fdin;
        filedata[1] = file_size;
        rval=ioctl(fd, FLEXIO_SET_INPUT, (void *)filedata);
        if(rval!=0) {
                perror("FLEXIO_SET_INPUT");
                close(fd); close(fdin); close(fdout);
                exit(1);
        }
        filedata[0] = fdout;
        filedata[1] = file_size;
        rval=ioctl(fd, FLEXIO_SET_OUTPUT, (void *)filedata);
        if(rval!=0) {
                perror("FLEXIO_SET_OUTPUT");
                close(fd); close(fdin); close(fdout);
                exit(1);
        }
        rdtscll(startTicks);
        rval=ioctl(fd, FILE_MMAP_KERNEL_READ_WRITE, file_size);
        if(rval!=0) {
                perror("FILE_MMAP_KERNEL_READ_WRITE");
                close(fd); close(fdin); close(fdout);
                exit(1);
        }
        rdtscll(endTicks);
        fprintf(stderr, "Time for the operation was: %Lu
                        micro seconds\n", cycles_to_usec(endTicks-startTicks));

        free(filedata);
        close(fd);
        fsync(fdin); fsync(fdout);
        close(fdin); close(fdout);
}
```

B.6 rawIO: Shell script for FlexIO used to perform Raw disk I/O

This is a shell script used for performing I/O among raw disk partitions. */dev/sdc1* is the input partition in this example and */dev/sdc2* is the output partition. The input partition is filled with random data and the output partition is filled with zeroes. I/O is then performed using FlexIO using fl_raw_fileIO shown in Section B.7.

```
USAGE - ./rawIO no_1024_blocks_size

#!/bin/sh

# Change the value of count for different file sizes

# Input partition
in=/dev/sdc1
# Output partition
out=/dev/sdc2

echo in will be $in out will be $out

# Input partition is filled with random data
dd if=/dev/urandom of=/d0/in  bs=1024 count=$1 >/dev/null 2>&1
dd if=/d0/in      of=$in      bs=1024 count=$1 >/dev/null 2>&1

# Output partition is filled with zeroes
dd if=/dev/zero    of=$out     bs=1024 count=$1 >/dev/null 2>&1

echo before running tst
./fl_raw_fileIO -i $in -o $out -d /dev/flexio -m 1 -b $1

echo checking $out against original
dd if=$out         of=/d0/out bs=1024 count=$1 >/dev/null 2>&1
diff /d0/in /d0/out

echo checking $in against original
dd if=$in          of=/d0/out bs=1024 count=$1 >/dev/null 2>&1
diff /d0/in /d0/out
echo diff complete

free
echo
```

B.7 fl_raw_fileIO.c: FlexIO used to perform Raw disk I/O

This is a user program used to perform I/O among raw disk partitions using FlexIO. It opens the input partition and output partition and passes the information to FlexIO using FLEXIO_SET_INPUT_RAW and FLEXIO_SET_OUTPUT_RAW ioctls. The I/O is performed using RAW_KERNEL_READ_WRITE ioctl.

```
USAGE - Called by rawIO script (look at the script).

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/time.h>

#include <linux/flexio_ioctls.h>

#include "tsc.h"

#define PAGE_SIZE (1UL << 12)
#define PAGE_MASK (~(PAGE_SIZE-1))

unsigned long long startTicks=0, endTicks=0;

int main(int argc, char **argv)
{
        int fd, rval, *filedata, file_size;
        FILE *fpin, *fpout;
        int blocks_to_move = atoi(argv[3]);

        read_proc();

        filedata = (int *)malloc(sizeof(int)*2);

        /* Input disk partition */
        fpin = fopen(argv[1], "r");
        if (!fpin) {
                perror("open");
                exit(1);
        }
        /* Output disk partition */
        fpout = fopen(argv[2], "w");
        if (!fpout) {
                perror("open");
                fclose(fpin);
                exit(1);
        }
        /* Opening FlexIO */
        fd = open("/dev/flexio", 0);
        if (fd < 0) {
                perror("open");
                fclose(fpin); fclose(fpout);
                exit(1);
        }
        rval=ioctl(fd, FLEXIO_SET_INPUT_RAW, fileno(fpin));
        if(rval!=0) {
                perror("FLEXIO_SET_INPUT_RAW");
                close(fd); fclose(fpin); fclose(fpout);
                exit(1);
        }
        rval=ioctl(fd, FLEXIO_SET_OUTPUT_RAW, fileno(fpout));
        if(rval!=0) {
                perror("FLEXIO_SET_OUTPUT_RAW");
                close(fd); fclose(fpin); fclose(fpout);
                exit(1);
        }
        rdtscll(startTicks);
        rval=ioctl(fd, RAW_KERNEL_READ_WRITE, blocks_to_move);
```

```
        if(rval!=0) {
                perror("RAW_KERNEL_READ_WRITE");
                close(fd); fclose(fpin); fclose(fpout);
                exit(1);
        }
        rdtscll(endTicks);
        fprintf(stderr, "Time for the operation was: %Lu
                        micro seconds\n", cycles_to_usec(endTicks-startTicks));

        close(fd);
        fclose(fpin); fclose(fpout);
}
```

B.8 sender.c: User program performing Network I/O using UDP sockets, Sender code

This is a user program used to perform network I/O using UDP sockets. This is the sender part of the code. It reads the data from the file and sends it to the network using *sendto* system call. For details about socket programming, refer Richard Stevens [14].

```
USAGE - ./sender inputfile rx_ip_addr sizeof_individual_transfer

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "tsc.h"

#define COMMON_PORT 4444

unsigned long long startTicks=0, endTicks=0;

int fdin, sockfd;

void terminate_sender()
{
        fprintf(stderr, "Terminating sender\n");
        close(sockfd);
        close(fdin);
        exit(1);
}

#define SIZE 1024

int main(int argc, char **argv)
{
        int file_size, size, ret, written;
```

```
struct stat statbuf;
struct sockaddr_in sender, receiver;
char *buf, first[SIZE];

signal(SIGINT, terminate_sender);

read_proc();

size = atoi(argv[3]);

fdin = open(argv[1], O_RDONLY);
if (fdin < 0) {
        perror("open");
        exit(1);
}
if (fstat(fdin, &statbuf) < 0) {
        perror("fstat");
        close(fdin);
        exit(1);
}
file_size = statbuf.st_size;

buf = (char *)malloc(size);
memset(buf, 0, size);
memset(first, 0, SIZE);

if (file_size < size)
        size = file_size;

receiver.sin_family = AF_INET;
receiver.sin_addr.s_addr = inet_addr(argv[2]);
receiver.sin_port = htons(COMMON_PORT);

if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(1);
}
sender.sin_family = AF_INET;
sender.sin_addr.s_addr = htonl(INADDR_ANY);
sender.sin_port = htons(COMMON_PORT);
if (bind(sockfd, (struct sockaddr *)&sender, sizeof(sender)) < 0) {
        perror("bind");
        close(sockfd); close(fdin);
        free(buf);
        exit(1);
}
ret = read(fdin, first, SIZE);
if (ret < 0) {
        perror("read");
        free(buf);
        close(fdin); close(sockfd);
        exit(1);
}
/* send 1024 bytes to start the clock on the receiver side */
if ((written = sendto(sockfd, first, ret, 0x40 /* MSG_DONTWAIT */,
                      (struct sockaddr *)&receiver, sizeof(receiver))) < 0) {
        perror("sendto");
        free(buf);
        close(fdin); close(sockfd);
        exit(1);
}
file_size -= written;
if (file_size < size)
if (file_size < size)
        size = file_size;
```

```
        rdtscll(startTicks);
        while (file_size > 0) {
                ret = read(fdin, buf, size);
                if (ret < 0) {
                        perror("read");
                        free(buf);
                        close(fdin); close(sockfd);
                        exit(1);
                }
                if ((written = sendto(sockfd, buf, ret, 0x40 /* MSG_DONTWAIT */,
                                        (struct sockaddr *)&receiver, sizeof(receiver))) < 0) {
                        perror("sendto");
                        free(buf);
                        close(fdin); close(sockfd);
                        exit(1);
                }
                if (written != ret)
                        fprintf(stderr, "%d %d\n", written, ret);

                file_size -= size;
                if (file_size < size) {
                        size = file_size;
                }

                /* Time to breathe...if not, it will not get memory */
                usleep(1);
        }
        rdtscll(endTicks);
        fprintf(stderr, "Time for the operation was: %Lu
                        micro seconds\n", cycles_to_usec(endTicks-startTicks));

        close(fdin); close(sockfd);
        free(buf);
}
```

B.9 receiver.c: User program performing Network I/O using UDP sockets, Receiver code

This is a user program used to perform network I/O using UDP sockets. This is the receiver part of the code. It reads the data from the socket using *recvfrom* system call and writes the data to the file. For details about socket programming, refer Richard Stevens [14].

```
USAGE - ./receiver sizeoffile sizeof_indivivual_transfer

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>

#include "tsc.h"

#define COMMON_PORT 4444

unsigned long long startTicks=0, endTicks=0;

int sockfd, fdout;

void terminate_client()
{
        fprintf(stderr, "Terminating client\n");
        close(sockfd);
        close(fdout);
        exit(1);
}

#define SIZE 1024

int main(int argc, char **argv)
{
        int size, ret, len, recvbufsize;
        char *buf, first[SIZE];
        struct sockaddr_in receiver, sender;
        int tx_size;

        signal(SIGINT, terminate_client);

        read_proc();

        fdout = open("./outputfile", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU|S_IRWXG|S_IRWXO);
        if (fdout < 0) {
                perror("open");
                exit(1);
        }
        size = atoi(argv[2]);

        buf = (char *)malloc(size);

        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                perror("socket");
                close(fdout);
                free(buf);
                exit(1);
        }
        receiver.sin_family = AF_INET;
        receiver.sin_addr.s_addr = htonl(INADDR_ANY);
        receiver.sin_port = htons(COMMON_PORT);

        if (bind(sockfd, (struct sockaddr *)&receiver, sizeof(receiver)) < 0) {
                perror("bind");
                close(sockfd); close(fdout);
                free(buf);
                exit(1);
        }
        tx_size = atoi(argv[1]);

        ret = recvfrom(sockfd, first, SIZE, 0 /* MSG_WAITALL */,
                                (struct sockaddr *)&sender, &len);
        if (ret < 0) {
                perror("recvfrom");
                free(buf);
                close(fdout); close(sockfd);
```

```
                exit(1);
        }
        ret = write(fdout, first, ret);
        if (ret < 0) {
                perror("write");
                free(buf);
                close(fdout); close(sockfd);
                exit(1);
        }
        tx_size -= ret;
        rdtscll(startTicks);
        while (tx_size > 0) {
                ret = recvfrom(sockfd, buf, size, 0 /* MSG_WAITALL */,
                                (struct sockaddr *)&sender, &len);
                if (ret < 0) {
                        perror("recvfrom");
                        free(buf);
                        close(fdout); close(sockfd);
                        exit(1);
                }
                if (ret != size)
                        fprintf(stderr, "%d %d\n", ret, size);
                if (ret == 0)
                        break;
                tx_size -= ret;
                if (tx_size < size) {
                        size = tx_size;
                }
                ret = write(fdout, buf, ret);
                if (ret < 0) {
                        perror("write");
                        free(buf);
                        close(fdout); close(sockfd);
                        exit(1);
                }
        }
        rdtscll(endTicks);
        fprintf(stderr, "Time for the operation was: %Lu
                       micro seconds\n", cycles_to_usec(endTicks-startTicks));

        close(fdout); close(sockfd);
        free(buf);
}
```

B.10 fl_nwIO_tx.c: FlexIO used to perform Network I/O, Transmitter code

This is a user program used to perform network I/O using FlexIO. This is the sender part of the code. It opens the *inputfile* and passes the details to FlexIO using FLEXIO_SET_INPUT ioctl. Since the FlexIO driver needs to know about the hardware address of the receiver, the receiver is pinged to make sure the hardware address of the receiver is available in the ARP cache. ARP cache is then read using SIOCGARP ioctl. All the information

is passed to FlexIO using FLEXIO_SET_NET_DEV ioctl. The I/O is performed using FLEXIO_TX_ETH or FLEXIO_THROTTLE_TX ioctls.

```
USAGE - ./fl_nwIO_tx inputfile my_ip_addr rx_ip_addr port

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <linux/if_arp.h>
#include <linux/sockios.h>

#include "flexio_ioctls.h"

main(int argc, char **argv)
{
        int fd, rval, fdin, file_size, *filedata;
        struct flexio_eth_info *info;
        struct stat statbuf;
        char *buf;
        unsigned long port;
        struct sockaddr_in dest_addr;
        struct arpreq r;
        int sockfd, aret, i;
        char command[100];

        memset((char *)&statbuf, 0, sizeof(struct stat));
        memset((char *)&r, 0, sizeof(struct arpreq));

        /* Input file */
        fdin = open(argv[1], O_RDONLY);
        if (fdin < 0) {
                perror("open");
                exit(1);
        }
        if (fstat(fdin, &statbuf) < 0) {
                perror("fstat");
                close(fdin);
                exit(1);
        }
        /* Opening FlexIO */
        fd = open("/dev/flexio", 0);
        if (fd < 0) {
                perror("open");
                exit(1);
        }
        filedata = (int *)malloc(sizeof(int)*2);
        memset((char *)filedata, 0, sizeof(int)*2);

        file_size = statbuf.st_size;
        filedata[0] = fdin;
        filedata[1] = file_size;
        rval=ioctl(fd, FLEXIO_SET_INPUT, (void *)filedata);
        if(rval!=0) {
                perror("FLEXIO_SET_INPUT");
                close(fd); close(fdin); free(filedata);
```

```
                exit(1);
        }
        info = (struct flexio_eth_info *)malloc(sizeof(struct flexio_eth_info));
        memset((char *)info, 0, sizeof(struct flexio_eth_info));

        /* ARP for the receiver's hardware address */
        memset(command, 0, 100);
        strcpy(command, "ping -c 1 ");
        strcat(command, argv[3]);
        system(command);

        /* Some time to breathe */
        sleep(2);

        sockfd = socket(AF_INET,SOCK_STREAM,0);
        if (sockfd < 0) {
                perror("socket");
                close(fd); close(fdin); free(filedata); free(info);
                exit(1);
        }
        dest_addr.sin_family =  AF_INET;
        dest_addr.sin_addr.s_addr = inet_addr(argv[3]);
        dest_addr.sin_port = 0;
        memcpy((char *)&r.arp_pa, (char *)&dest_addr, sizeof(struct sockaddr_in));
        r.arp_flags = ATF_PERM;
        /* Change this to the proper interface name if required */
        strcpy(r.arp_dev, "eth0");

        /* Get the  hardware address from the ARP cache */
        aret = ioctl(sockfd, SIOCGARP, &r);
        if (aret < 0) {
                perror("SIOCGARP ioctl error");
                close(fd); close(fdin); free(filedata); free(info);
                close(sockfd);
                exit(1);
        }
        /* Change this to the proper interface name if required */
        strcpy(info->ifname, "eth0");
        info->saddr = inet_addr(argv[2]);
        info->daddr = inet_addr(argv[3]);
        info->nw_element_type = FLEXIO_SRC;
        memcpy(info->h_dest, &r.arp_ha.sa_data, 6);

        rval = ioctl(fd, FLEXIO_SET_NET_DEV, (void *)info);
        if (rval < 0) {
                perror("FLEXIO_SET_NET_DEV");
                close(fd); close(fdin); free(info); free(filedata);
                close(sockfd);
                exit(1);
        }
        port = (unsigned long)atoi(argv[4]);
        /* Note FLEXIO_TX_ETH can be used instead, but only for small files as
         * it is very fast and overruns the receiver
         */
        rval = ioctl(fd, FLEXIO_THROTTLE_TX, port);
        if (rval < 0) {
                perror("FLEXIO_THROTTLE_TX");
                close(fd); close(fdin); free(info); free(filedata);
                close(sockfd);
                exit(1);
        }

        free(info); free(filedata);
        close(fd); close(fdin); close(sockfd);
}
```

B.11 fl_nwIO_rx.c: FlexIO used to perform Network I/O, Receiver code

This is a user program used to perform network I/O using FlexIO. This is the receiver part of the code. It opens the *outputfile*, sets the size of the file using the details provided by the user and passes the details to FlexIO using FLEXIO_SET_OUTPUT ioctl. The I/O is performed using FLEXIO_RX ioctl.

```
USAGE - ./fl_nwIO_rx port sizeoffile sleeptime

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>

#include "flexio_ioctls.h"

main(int argc, char **argv)
{
        int fd, rval, fdout, *filedata, num, size;
        unsigned long port;

        port = (unsigned long)atoi(argv[1]);
        fdout = open("./outputfile", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU|S_IRWXG|S_IRWXO);
        if (fdout < 0) {
                perror("open");
                exit(1);
        }

        num = atoi(argv[2]);
        size = num;
        if (lseek(fdout, size-1, SEEK_SET) < 0) {
                fprintf(stderr,"Couldn't seek outputfile");
                perror("lseek");
                close(fdout);
                exit(1);
        }
        if (write(fdout, "", 1) != 1) {
                perror("write");
                close(fdout);
                exit(1);
        }
        if (lseek(fdout, 0, SEEK_SET) < 0) {
                perror("lseek");
                close(fdout);
                exit(1);
        }
        fd = open("/dev/flexio", 0);
        if (fd < 0) {
                perror("open");
                exit(1);
        }

        filedata = (int *)malloc(sizeof(int)*2);
        memset((char *)filedata, 0, sizeof(int)*2);
```

```
        filedata[0] = fdout;
        filedata[1] = size;
        rval=ioctl(fd, FLEXIO_SET_OUTPUT, (void *)filedata);
        if(rval!=0) {
                perror("FLEXIO_SET_OUTPUT");
                close(fd); close(fdout); free(filedata);
                exit(1);
        }
        rval = ioctl(fd, FLEXIO_RX_ETH, port);
        if (rval < 0) {
                perror("FLEXIO_RX_ETH");
                close(fd); close(fdout); free(filedata);
                exit(1);
        }
        /* Though the receiver is blocking, give it time till
         * it receives all the packets
         */
        sleep(atoi(argv[3]));

        fsync(fd);
        close(fd); close(fdout);
        free(filedata);
}
```