MULTIPORTED STORAGE DEVICES

A Thesis

by

MARCUS BRYAN GRANDE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2000

Major Subject: Computer Engineering

MULTIPORTED STORAGE DEVICES

A Thesis

by

MARCUS BRYAN GRANDE

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

A. L. Narasimha Reddy (Chair of Committee)	Ricardo Bettati (Member)
Gwan Choi (Member)	Chanan Singh (Head of Department)

May 2000

Major Subject: Computer Engineering

ABSTRACT

Multiported Storage Devices. (May 2000)

Marcus Bryan Grande, B.S., Texas A&M University

Chair of Advisory Committee: Dr. A. L. Narasimha Reddy

In the past decade the demand for systems that can process and deliver massive amounts of storage has increased. Traditionally, large disk farms have been deployed by connecting several disks to a single server. A problem with this configuration is that a given storage server can become a bottleneck, as all disk activity for a particular server has to be processed by that server.

Increasing performance and decreasing cost of microprocessors are making it feasible to move more processing power to the data source. This allows us to investigate new methods of storage delivery that were not plausible in the past.

In this thesis we present a new "enhanced" storage device called a *Multiported Storage Device* which is endowed with more processing power and intelligence than the traditional block storage device. A multiported storage device allows application-specific code that we call *filter applets* to be downloaded to the device while still maintaining the simple block-level interface. The device contains several software *ports* through which read and write requests pass. Each filter applet is associated with a particular software port in the device.

We present a prototype implementation of a multiported storage device using the Linux operating system. We implement our device using kernel driver modules and by

performing several modifications to the Linux kernel to accommodate the multiported storage device architecture. We also present experimental results showing the impact of our layered drivers on I/O performance.

ACKNOWLEDGMENTS

I would like to give thanks to my advisor, Dr. A. L. Narasimha Reddy, for his help and guidance throughout my academic career. For the past four years he has worked with me and given me opportunities to grow and progress under his leadership. His genuine interest in the success of his students has been a major factor in my success, both academic and professional.

I would also like to thank Ravi Wijayaratne, my fellow lab mate. He was the true Linux kernel pioneer of the research group and spent many months alone poring over books and kernel source code. I thank him for his kindness in readily sharing with me the knowledge he has gained. He was responsible for much of the investigative work in adding layered driver support to the Linux kernel. Without his assistance, my success in this work would have not been possible.

I would like to thank my parents, Bryan and Rachel Grande, for their guidance and encouragement throughout my life. The teachings and principles they instilled in me were key in insuring my success in school, work, marriage and as a person. Although it has been over a decade since I have shared the same roof with them, they will continue to influence my daily life now and forever.

Finally, I would like to thank my wife, Adriana Grande, for her sacrifice and patience throughout my academic career. The rigors of academia have often infringed on my ability to assist in things of home and family, including the raising of our daughter. Time and time again Adriana willingly carried out important tasks that I could not. Without her constant support, I could not have accomplished the things that I have

accomplished. To her I say thank you. I will forever be in your debt and hope to ever spend my days in an effort to repay you.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

II BACKGROUND AND RELATED WORK 3

A. Network-Attached Storage 3
B. Active Disks 5

III MULTIPORTED STORAGE DEVICES 6

IV IMPLEMENTATION BACKGROUND 10

A. Linux File I/O Architecture 10
B. Linux Kernel Modules 12
C. The Windows NT Layered Driver Model 13

V IMPLEMENTATION DETAILS OF MULTIPORTED STORAGE DEVICE 16

A. Linux Kernel Modifications to Support Layered Driver Model . . . 17
B. Buffer Cache and I/O Request Formats 19
C. Implementation Details of Filter Driver 21
1. Port specification 22
2. Read/write determination 24
3. Registering filter applets at the device 25
4. Preservation of original data 27

VI EXPERIMENTATION AND MEASUREMENTS 30

A. Kernel Overhead Measurements of Filter Driver 30
B. Application-Level Measurements 34
C. Overhead Measurements for Periodic Requests and Multiple Processes/Ports 38
D. General Observations and Analysis of Experimental Results 43

VII SUMMARY AND CONCLUSIONS 45

REFERENCES 49

APPENDIX A 51

Page

VITA 54

LIST OF FIGURES

FIGURE		Page
1	Network-attached storage architecture	4
2	Logical structure of multiported storage device	7
3	Linux file I/O subsystem	10
4	Windows NT layered I/O driver model	15
5	Location of multiported module in I/O stack	17
6	The buffer cache	20
7	Queuing of I/O requests	21
8	Processing of I/O requests by smart_blkfilter	22
9	Registering of filter applet via Linux stacked module mechanism	26
10	Table of registered filter applets (functions)	27
11	Overhead due to presence of smart_blkfilter alone	31
12	Overhead of smart_blkfilter using rot13_filter port	33
13	Application-level read response times	35
14	Application-level write response times	37
15	Processes reading through no-op port - read ahead enabled	39
16	Processes reading through no-op port - read ahead disabled	40
17	Processes reading through rot13 port - read ahead enabled	41
18	Processes reading through rot13 port - read ahead disabled	42

CHAPTER I

INTRODUCTION

In the past decade the demand for systems that can process and deliver massive amounts of storage has increased. Multimedia applications such as streaming audio and video require large amounts of data to be read from disk and processed for timely delivery to the client. Large decision support databases need not only to read large amounts of data from the storage media but also to scale in processing power as the data set sizes grow. Data mining requires large amounts of data to be read from the storage media, analyzed, and processed to find trends in the database that can serve as a basis for future decisions, such as what products are likely to be bought in groups.

Traditionally, large disk farms have been deployed by connecting several disks to a single server. Several servers, each with several disks, can be deployed to create a distributed storage system. One problem with this configuration is that a given storage server can become a bottleneck, as all disk activity for a particular server has to be processed by that server. If lots of processing is required on the retrieved data, this can exacerbate the problem. If the data processing occurs at the server, the server becomes even more of a bottleneck. If the data is processed at the client, all of the data has to be sent to the client from the server, even if the processing causes some of the retrieved data to be discarded, as in a SELECT operation on a database. This results in wasted interconnect bandwidth.

The journal model is *IEEE Transactions on Automatic Control.*

Several trends are making it feasible to move more processing power to the data source. Disk array controllers today have central processors that are at most one generation behind state-of-the-art workstations. Individual disk controllers are further behind, but are catching up fast. Chip technology is to the point where it is feasible to combine the drive control microprocessor with the specialized ASIC responsible for the drive's processing into a single chip. This makes it possible to integrate a 200 MHz RISC core in the same die space as the ASIC with room to spare [1].

In this thesis we present a new "enhanced" storage device that is endowed with more processing power and intelligence than the traditional block storage device, which usually exports a simple block-level interface to the operating system for reading and writing of the device with little to no extra functionality. We call these new devices *Multiported Storage Devices* for reasons which will be explained later in this thesis. These enhanced devices will be capable of performing advanced operations on the data being read from and written to the disk *at the block level*.

The rest of this thesis is organized as follows. Chapter II investigates the present status of research in enhanced storage devices. Chapter III explains the Multiported Storage Device architecture. Chapter IV presents background information necessary for understanding of the implementation details. Chapter V discusses the implementation details. Chapter VI outlines performance experiments for the prototype multiported storage device implementation and presents the results of these experiments. Finally, Chapter VII presents our summary and conclusions of the research.

CHAPTER II

BACKGROUND AND RELATED WORK

A. Network-Attached Storage

In [2, 3] Gibson et al. discuss the previously-mentioned problems with conventional file servers and distributed file systems. They define a taxonomy of network-attached storage architectures consisting of four configurations: (1) Server-Attached Disks (SAD), (2) Server Integrated Disks (SID), (3) Network SCSI (NetSCSI), and (4) Network-Attached Secure Disks (NASD).

With server-attached disks, clients and servers share a network, and storage is attached directly to general-purpose workstations that provide distributed file services. This is the common configuration in office and campus LANs. Server-integrated disks are not fundamentally different from server-attached disks. Data must still move through the server machine before it reaches the network, but these server machines are dedicated file servers which can move data through the network more efficiently than general-purpose machines. An NFS file server is a typical SID.

NetSCSI uses a different approach than SAD and SID. NetSCSI tries to retain as much as possible of the current dominant storage device protocol, SCSI. As far back as 1996, Seagate's Barracuda FC hard drive was already providing packetized SCSI through Fibre Channel network ports to directly attached hosts [2]. NetSCSI is a storage-attached architecture that makes minimal changes to the hardware and software of SCSI disks. A file manager translates file system requests from the client into SCSI commands

for its disks. However, rather than returning data to the file manager to be forwarded to the client, NetSCSI disks send data directly to the client [2].

With network-attached secure disks, the goal of minimal change from the SCSI interface is relaxed. In this architecture network-attached disks are connected directly to the network (see Fig. 1). The focus is on selecting a command interface that reduces the number of client storage interactions that must be relayed through the file manager, thus offloading more of the file manager's work. Common, data-intensive operations, such as reads and writes, go straight to the disk, while less-common operations, including namespace and access control manipulations, go to the file manager. This removes the file manager as the "middleman" in the data transfer. The client can be given tokens, or capabilities, from the file manager so that all subsequent communication between the client and storage server go directly to the disk. This architecture remedies the problem of the file server becoming a bottleneck and allows more flexibility than the NetSCSI solution. For more on network-attached secure disks, see [2, 3, 4, 5, 6].

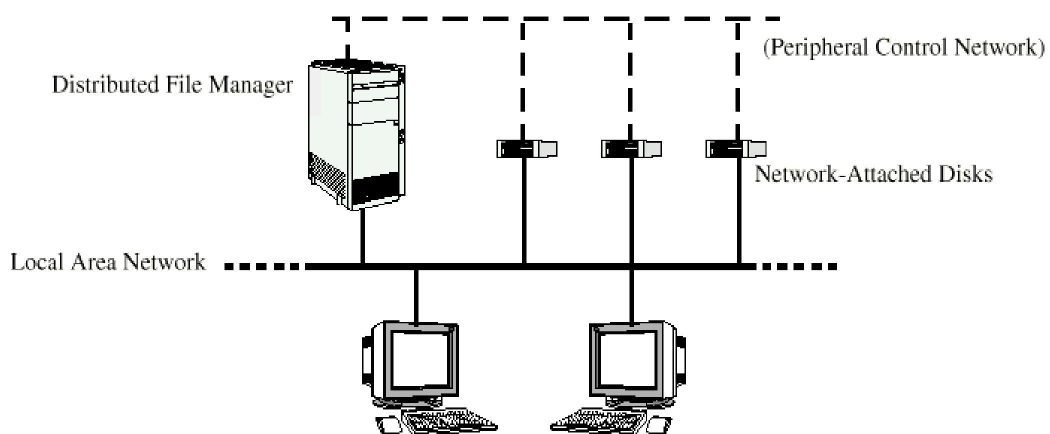


Fig. 1. Network-attached storage architecture [2]

B. Active Disks

It has become feasible, due to the increasing performance and falling price of microprocessors, to move system intelligence into peripherals from the CPU [7]. Papers [1, 7] propose a system called *Active Disks* – next-generation disk drives that provide an environment for executing application code directly at individual drives. Partitioning processing across clients, servers and storage devices can reduce the amount of data crossing the network and exploit the processing cycles available at the storage device. Specifically, applications that apply simple filters or statistics to stored data or that make disk or network scheduling decisions based on current information at the storage device can improve the efficiency of an application's use of network and host resources, resulting in more scalable and higher performance storage systems [1].

Several candidate applications for active disks are (1) filters that remove data that would be discarded anyway at the client, such as text searches, database SELECT, image processing, etc., (2) real-time applications, such as multimedia, that may benefit from performing resource and scheduling decisions at the disk, (3) batching, where requests require large amounts of work but are not time critical, such as drive-to-drive copying, (4) self-management, such as defragmentation and drive layout optimization, and a myriad of other applications [1]. [8, 9] further investigate active disks, proposing a stream-based programming model with programs called *disklets* running at the disk. They also investigate the performance of several applications, comparing active disk systems to conventional systems via their simulator *ADsim*.

CHAPTER III

MULTIPORTED STORAGE DEVICES

We propose an enhanced storage device called a *Multiported Storage Device*. A multiported storage device is much like an active disk, which exploits general-purpose processing power at the device itself. However, the previously-mentioned proposals of active disks and network-attached disks involve changing the interface to the device. A multiported storage device allows application-specific code to be downloaded to the device while still maintaining the simple block-level interface. Multiported storage devices offer many of the same benefits as systems based on NASD and active disks while providing additional benefits.

Fig. 2 shows the logical structure of a multiported storage device. The inner cylinder (solid line) is the physical hard drive. The outer cylinder (dotted line) is the image of the disk that the outside world sees. The box in the middle represents the various software “ports” through which reads and writes to the disk pass. We call the device “multiported” because we borrow the concept of ports from the world of networking. A port is a logical channel in a communications system. The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) transport-layer protocols use port numbers to distinguish between (demultiplex) different logical channels on the same network interface (such as an Ethernet card) on a computer. Each network application program has a unique port number associated with it. A network server may

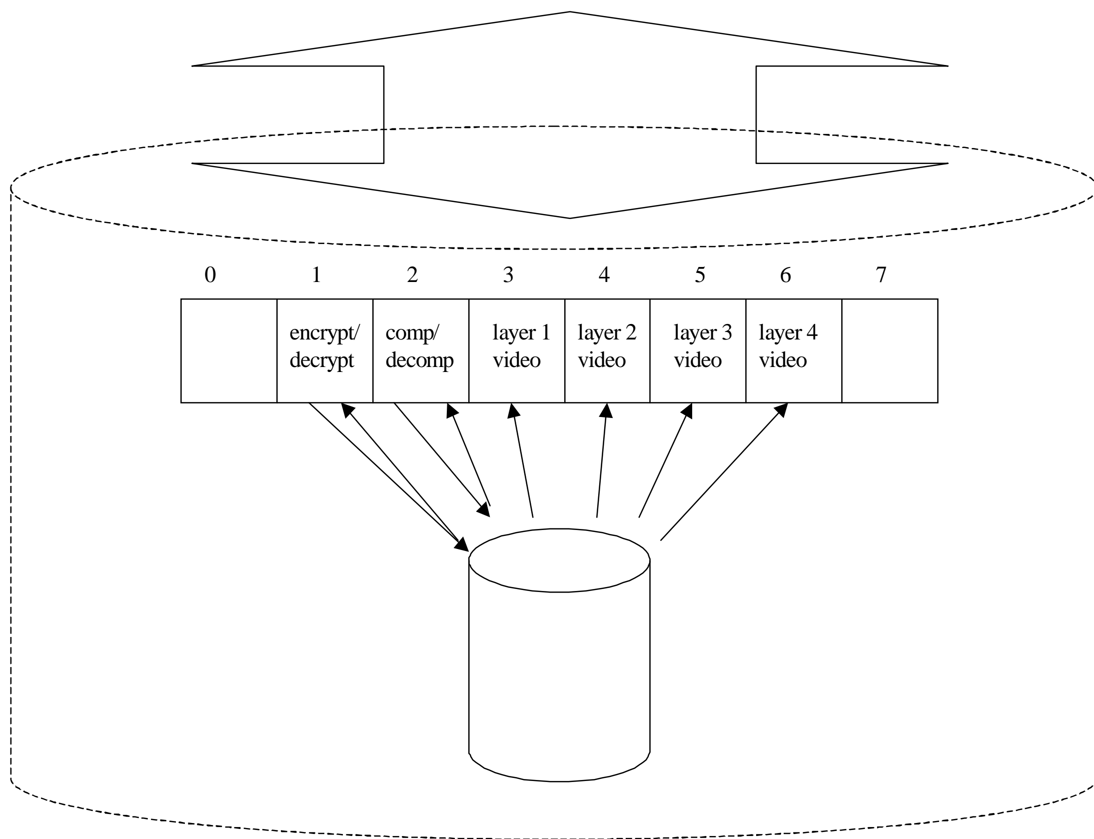


Fig. 2. Logical structure of multiported storage device

receive several different types of networking requests. Some requests may be telnet requests, some FTP requests, some http requests, etc. The way the server differentiates these requests is by listening on specific "ports" for a given application. The telnet daemon, for example, usually listens on port 23. The http daemon (web server) usually listens on port 80. The mail server listens on port 25 (see definition of "port" at [10]). Thus, the port defines the behavior of the network server. The above-mentioned port numbers are called *well-known ports* because most network client applications assume that the server is listening to these specified ports for these applications. A network server can also allow user-defined ports where applications that do not fall into the well-

known category can listen. These ports are not well known, and the client and server have to come to an agreement beforehand as to the port through which the client will talk to the server. Many times these ports are used during the testing of new applications or for a specific application between a particular client and server.

In a multiported storage device, there are several ports through which read and write requests pass. Similar to networking ports, a particular behavior is associated with each port in a multiported storage device. In essence, a port acts like a filter, intercepting requests and performing some operation on the data blocks as they enter (on a write) or leave (on a read) the disk. Since there are several ports, several different pieces of application-level code, completely unrelated, can co-exist at the disk. This allows greater flexibility than the active disk model described in [8] where only a single disklet can reside at the disk. Several applications can use the same multiported disk, each going through a different port, resulting in a different view of the storage device for each application.

Fig. 2 shows some examples of different applications that can benefit from a multiported storage device. Possible applications that can benefit from multiported storage devices are many of those mentioned when discussing active disks. Data can be compressed/decompressed at the disk. Perhaps encryption can be associated with a different port. An interesting use of multiported storage devices is for layered video, where different bit-rate streams are available via different ports. On many multimedia servers, separate files encoded at different bit rates have to be stored on the media for conventional, non-layered video. For a large movie, this can add up to a substantial amount of disk space. Video can be layered, however, so that different bit-rate video

streams can be encoded using a single file [11]. Nevertheless, the entire file still has to be read from the disk to be processed, which means that the file transfer between the data source and the processing machine can still consume a substantial amount of bandwidth. Using a multiported storage device, the video file can be encoded as a layered video file. The extracting of the correct bit rate for the layered video could happen *at the device itself*, thus reducing network traffic between the disk and the file server. In fact, combined with the network-attached disk concept, the layered data could even be transferred from the disk directly to the client, thus decreasing the CPU cycles spent at the server for I/O as well as reducing the amount of network traffic.

Similar to the networking model, some ports can be defined as well-known ports. Perhaps some of the well-known ports can be ports that provide standard compression or encryption services to applications. These can be hard-coded into the multiported storage device or simply installed into particular ports at set up time. Conversely, users can dynamically register “filter applets” (or disklets, using the parlance of [8, 9]) at the device as needed. Multiported storage devices provide a mechanism by which users can define their own “filter applets” and assign them to a port at the device. For example, if a different compression algorithm or encryption algorithm is desired, another port can be allocated for this purpose. Perhaps several layered video ports can be registered by the user who wants to provide several bit-rates of video playback. This allows a great deal of flexibility by providing a mechanism for adding functionality to a multiported storage device already deployed in the field.

CHAPTER IV

IMPLEMENTATION BACKGROUND

We implemented the multiported storage device code as Linux kernel driver modules. In order to understand the implementation details, we must first explain some of the basics of Linux file I/O, Linux kernel modules, and the Windows NT layered driver model.

A. Linux File I/O Architecture

Fig. 3 gives a logical overview of the Linux file I/O architecture. Linux has the ability to understand and access partitions on the hard disk that are formatted for different types of file systems. The ext2 file system is the native Linux file system. Linux also has

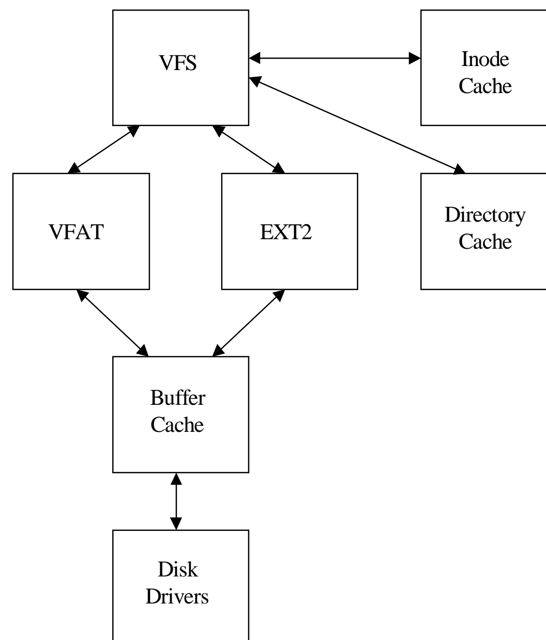


Fig. 3. Linux file I/O subsystem

the ability to read the VFAT file system, which is the native file system of Windows 95, as well as many other file systems. In order to enable processes to access files in a uniform manner without having to worry about file-system-specific interfaces, the kernel has a software layer called the *Virtual File System (VFS)* that sits between the system calls and the actual file system code layers. The VFS is responsible for performing operations which are independent of the format of the file system concerned. The VFS redirects the request to the file system module actually managing the file, such as ext2 or VFAT [12].

As files are used, they generate many read and write requests to the block devices (disks). To speed up access to the physical block devices, Linux maintains a cache of block buffers called the *buffer cache*. This cache is shared between all of the physical block devices (all of the disks/partitions in the system). At any given time there are many disk block buffers in the cache, belonging to any one of the system's block devices. If valid data is available from the buffer cache during a file read, this saves the system an access to the physical device, thus speeding up file accesses [13, 14]. On a file write, the data is written into a buffer in the buffer cache and marked dirty. Dirty buffers are flushed to the disk at a later time by default. This allows quick return from a write operation since the application does not have to wait for the media to receive the write.

When a desired block is not in the cache on a read, or when dirty buffers are flushed on a write, a low-level read/write command is issued to the device below through what is called a *strategy routine*. The strategy routine, called ll_rw_block() in Linux, is a generic interface to all of the block devices below. The strategy routine figures out to

which physical device the request should go and then queues the request on the appropriate device driver queue, such as IDE or SCSI. The device driver then identifies the actual block device that should receive the request (for example, which disk/partition on which adapter card) [12, 14].

B. Linux Kernel Modules

Linux is mainly a monolithic kernel, i.e., it is a single, large program where all the functional components of the kernel have access to all of its internal data structures and routines. The alternative to a monolithic kernel is a microkernel structure where the functional pieces of the kernel are broken out into separate units with strict communication mechanisms and interfaces between them [13]. Windows NT uses this structure [15]. In a monolithic kernel, adding new components into the kernel can be rather time consuming. If you wanted a new driver for a new network card out on the market that you had not built it into the kernel, you would have to configure and then build a new kernel before you could use the network card.

Recognizing this limitation, Linux now allows a user to dynamically load and unload components of the operating system as needed. Linux *kernel modules* are lumps of code that can be dynamically linked into the kernel at any point after the system has booted. They can be unlinked from the kernel and removed when they are no longer needed. Most Linux kernel modules are device drivers, pseudo-device drivers (such as network protocol drivers), or file systems [13].

Once a Linux module has been loaded it is as much a part of the kernel as any normal kernel code. A kernel module can export services for use by other modules, thus

making possible *module stacking*. Module stacking occurs when one module uses the exported services of another module, creating a dependency. For example, the VFAT file system module requires the services of the FAT file system module, as the VFAT file system is essentially an extension of the FAT file system. VFAT uses services provided by FAT so that it does not have to reimplement them.

As each module is loaded, it can export services (functions and/or variables). The kernel modifies the kernel symbol table, adding to it all of the resources or symbols exported by the module. This means that when another module is loaded, it has access to the exported services (functions and variables) of the previously loaded modules. Linux can even be configured to detect module dependencies and load the other necessary modules when a particular module is loaded [13, 16].

C. The Windows NT Layered Driver Model

Although both Linux and Windows NT allow drivers to be “stacked” on top of each other, the way Linux and Windows NT accomplish this is quite different, which has several implications regarding the semantics and capabilities of stacked or layered drivers in Linux vs. Windows NT. We will briefly explain the Windows NT layered driver model at this point in the thesis for reasons which will become clear when explaining the implementation details.

In the Windows NT I/O model, a “driver” is a somewhat general classification. Although most of the time a driver is associated with a piece of code that controls a device, in Windows NT a driver can sit at different levels of the I/O call stack. A driver may not control any hardware at all. In this respect, a Windows NT driver is much like a

Linux kernel module.

Fig. 4 shows the Windows NT I/O architecture. The layered driver model allows drivers to be inserted at nearly any level of the I/O call stack. The dotted lines indicate that the driver layers communicate with each other implicitly. In Windows NT, all communication between driver layers is centralized at the I/O manager, while in Linux stacked drivers communicate with each other directly through exported services. When a read/write request is issued, an *I/O Request Packet (IRP)* is created which contains the request information and data. The I/O manager hands the IRP to the driver at the top of the stack. When the driver is finished processing the data, it returns the IRP to the I/O manager, which then invokes the next driver in the stack [15, 17]. Since driver layers do not communicate directly, drivers can be easily inserted and deleted into the call stack.

In Fig. 4 we see that below the file system is a volume manager, which can be used to group several disks into a single logical disk. Below this is a driver that simply gathers performance statistics such as I/O completion time, throughput, etc. These layers could easily be removed from the I/O call stack without affecting the reading and writing of disk blocks. They could even be swapped in their positions in the call stack. Windows NT goes even further in defining what are known as *filter drivers*. Filter drivers are drivers that transparently intercept requests intended for some other driver [17]. Filter drivers "attach" themselves to other drivers to accomplish this. For example, the disk performance driver could transparently attach itself to the device driver itself as opposed to being a separate layer in the I/O call stack as depicted in the figure (the performance driver *diskperf.c* actually does just that). This model provides a flexible way of adding

functionality to the I/O stack.

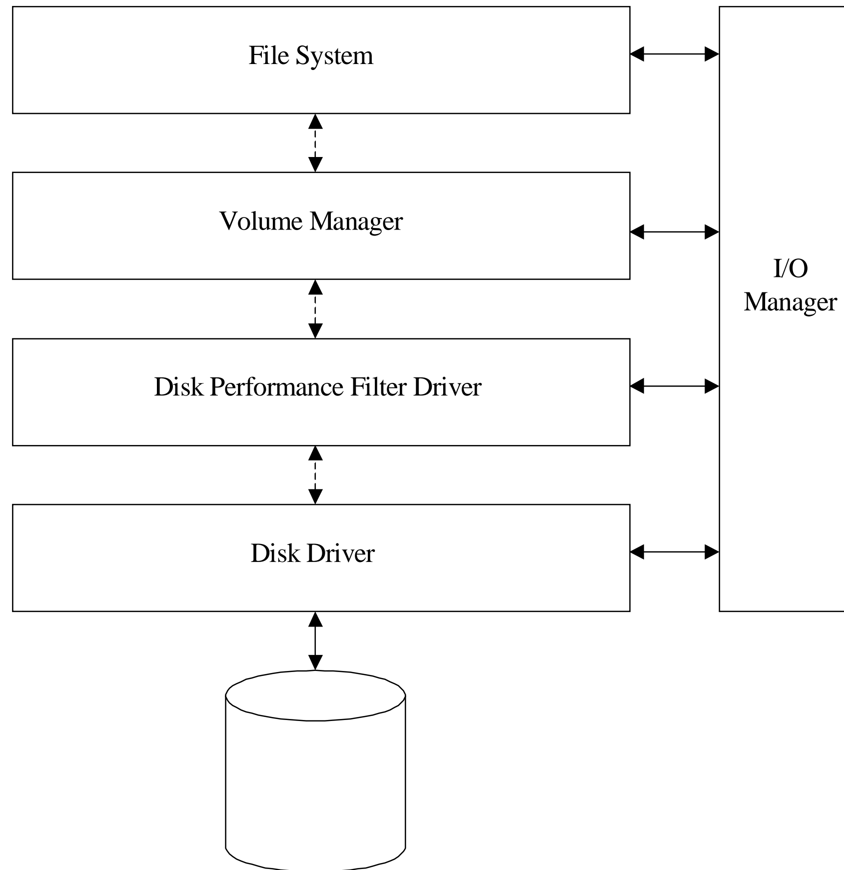


Fig. 4. Windows NT layered I/O driver model

CHAPTER V

IMPLEMENTATION DETAILS OF MULTIPORTED STORAGE DEVICE

We implemented this prototype multiported storage device on a Pentium 166MHz PC with 48MB of RAM, a Western Digital AC26400 6.4GB IDE hard drive formatted with a 1KB block size (2 sectors per block), and the Debian Linux 2.1 operating system (2.0.36 kernel). The entire implementation required 2316 lines of C code. We implemented the multiported functionality as a kernel module driver which we call *smart_blkfilter*. We call it smart_blkfilter to denote that we are endowing the device with increased functionality. We will refer to the driver as a "filter driver" from this point on for several reasons. First, the module acts like a type of filter, intercepting all requests to the device and inspecting them to decide if any operations need to be performed on the data. In this way, each request is "filtered" through our driver, even if the driver does not have any work to do. We also call it a filter driver after the filter driver concept used in Windows NT as explained previously.

Fig. 5 shows where in the Linux I/O stack that we have inserted our driver module. The way block I/O normally works in Linux is shown in Fig. 3 on page 10. If the buffer cache is not able to satisfy a read request, or if a dirty buffer is being flushed to the disk, the buffer cache code calls ll_rw_block() to issue a request to the device. One problem is that the ll_rw_block() strategy routine directly queues a request onto the device driver queue of the device that will eventually perform the read or write. Linux does not implement the layered driver I/O model as Windows NT does. This makes the

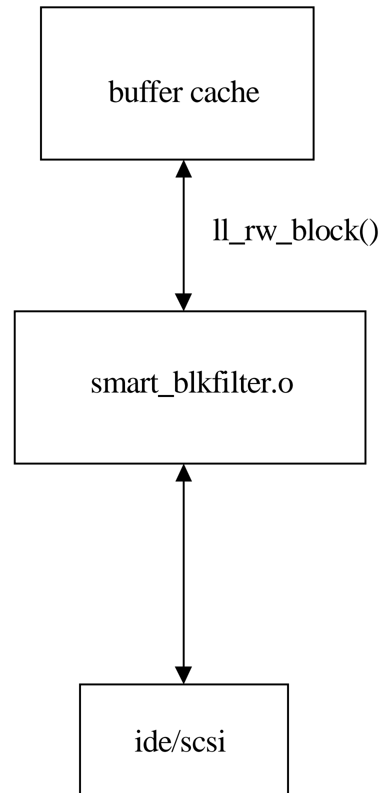


Fig. 5. Location of multiported module in I/O stack

inserting of our filter module into the driver stack at the desired position impossible. In order to make this possible, we had to modify the kernel to support the layered driver concept.

A. Linux Kernel Modifications to Support Layered Driver Model

Most of the modifications to support the layered driver architecture were made in the strategy routine ll_rw_block() found in the drivers/block/ll_rw_blk.c file, the ide_end_request() routine in include/linux/blk.h, and in the blk_dev_struct structure in include/linux/blkdev.h in the Linux source tree (all relative path names from this point on in the thesis are with respect to /usr/src/linux). In Linux, each block device in the kernel

is described by an entry in a table called blk_dev, which is a an array of blk_dev_struct structures. This array is indexed by what is called the *major number* of the block device. We modified the blk_dev_struct structure as follows:

```
struct blk_dev_struct {
        void (*request_fn)(void);
        struct request * current_request;
        struct request   plug;
        struct tq_struct plug_tq;
/************filter driver support***********/
        int next_dev; /*block device (module) *below* of this one*/
        int prev_dev; /*block device (module) *on top* of this one*/
        void (*callback_fn)(void);
/***********end filter driver support*********/
};
```

The next_dev and prev_dev fields allow a device to have another device "layered on top" of it. For example, the major number for the IDE drive we used in our experiments was 22, which is the major number of the second IDE controller in the machine. When the smart_blkfilter module is loaded, it is assigned a major number such as 127, for example. The prev_dev value for the IDE device would then be 127. The next_dev field for smart_blkfilter would be 22.

Within ll_rw_block() we check the value of the prev_dev field. If it is zero, no layered driver is loaded. If it is non-zero, then a layered driver is loaded "on top" of the intended driver. The number stored in this prev_dev field is the major number of the device on top (which also gives its index in the block device table). Recognizing this, ll_rw_block() sends the request to smart_blkfilter instead of the IDE device driver. It is then up to smart_blkfilter to invoke the next device when it is finished. Thus we accomplish "transparent layering" by having ll_rw_block() deal with the layered driver

issues. Neither the buffer cache above nor the block device driver below is aware of the smart_blkfilter layer.

We implemented a registry function, register_filter_blkdev(), to register a layered driver. Although Linux has a register_blkdev() function, it is unaware of the added fields used for layering. Instead of changing the interface and implementation of the register_blkdev() function (which could affect the functionality of the kernel as a whole), we implemented a new kernel function. Our new register_filter_blkdev() allows the programmer to specify the device to attach to by providing its major number. Note that we are limited to attaching to block devices (disk drivers) at this point. However, this is sufficient for our purposes.

B. Buffer Cache and I/O Request Formats

In order to understand the implementation details of the filter driver we must first explain some of the details of the buffer cache and I/O request semantics *before the kernel modifications we discussed in the previous section*. The buffer cache is essentially a hash table of what are called *buffer headers*, or buffer_head structures, as shown in Fig. 6. Each buffer_head structure points to a data area in memory. The size of the data area corresponds to the block size of the underlying device. The buffer_head structure contains several fields of metadata, including the size of the cached block and its location on the block device, the state of the buffer in memory (up-to-date, dirty, etc.), and other

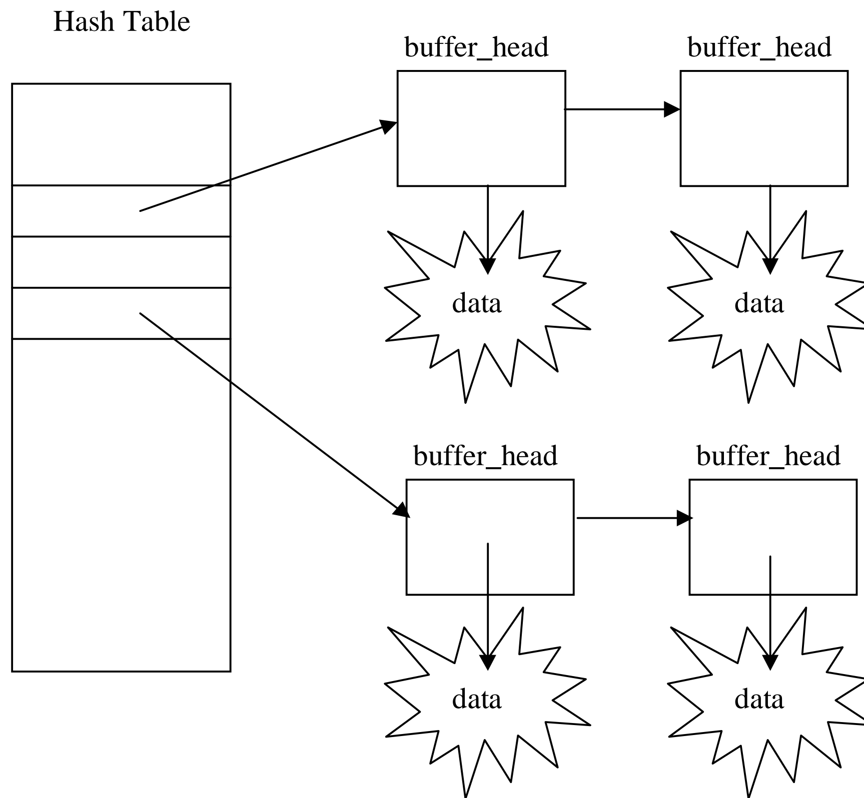


Fig. 6. The buffer cache

information (see Appendix A). Within the buffer_head structure there are many pointers for inclusion on various linked lists, including the free buffer list and the hash queue list of the buffer cache itself. Another important linked-list pointer is the b_reqnext pointer, which is used when making requests to the I/O device.

Whenever the buffer cache code cannot find the desired block in the buffer cache on a read, or when it wants to flush a dirty block back to the disk, it issues a call to ll_rw_block() to read or write the block. The ll_rw_block() function then takes the buffer_head structure passed to it and creates a request to the I/O device. A request to a block device is represented by the *struct request* data type (see Appendix A). The

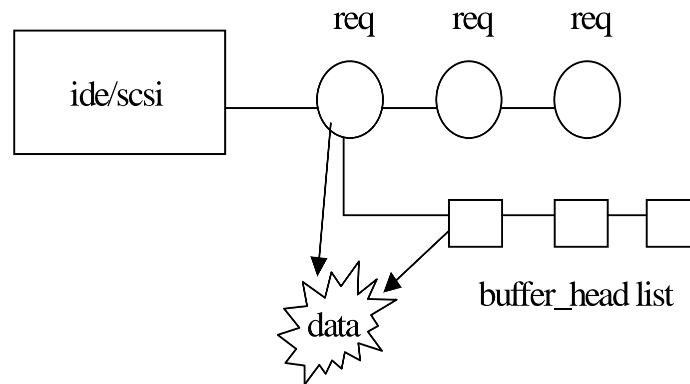


Fig. 7. Queuing of I/O requests

request structures are queued onto the I/O device driver queue using the *next* pointer in the structure. The request structure also has a pointer to the buffer_head and memory area of the data to be written/read.

The ll_rw_block code, upon receiving a request, attempts to perform *request coalescing*. The code inspects the current requests in the request queue to see if the new request and any of the currently outstanding requests are contiguous on the disk. If so, a linked list of buffer_head structs is created on the appropriate request structure. The request->buffer member points to the data area of the first buffer_head in the list. If the new request and the currently outstanding requests are not contiguous, a new request structure is allocated for this buffer_head. Fig. 7 gives a graphical representation of both queued requests and coalesced requests [16].

C. Implementation Details of Filter Driver

The previous section explained the *normal operation* of the Linux kernel. As stated in section A of this chapter, we modified the behavior of ll_rw_block() to check for an installed layered driver. If it finds one, it queues the request on this layered driver

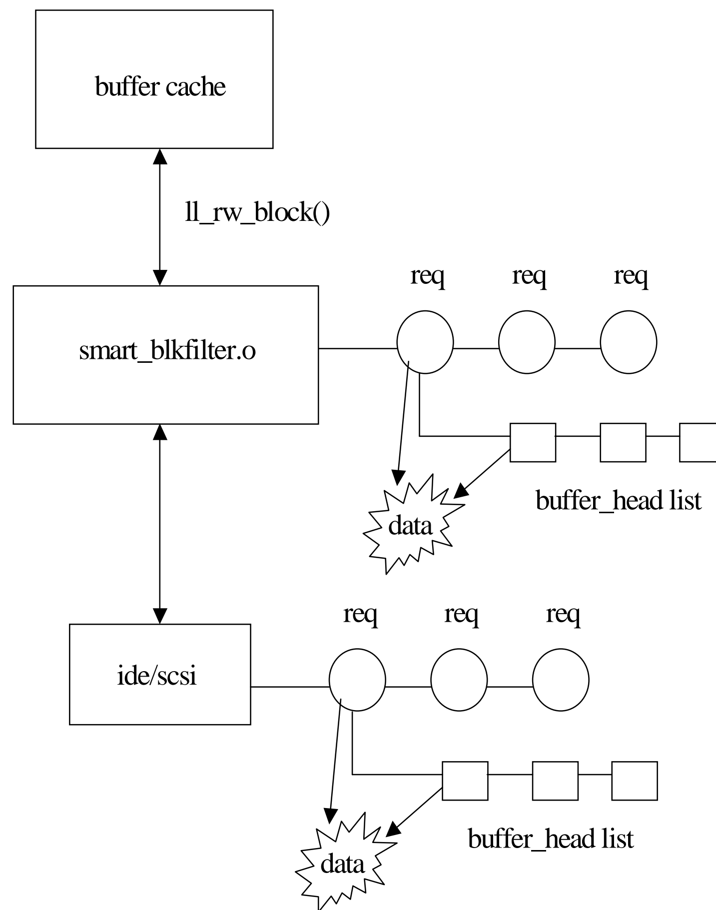


Fig. 8. Processing of I/O requests by smart_blkfilter

instead of the disk device driver. This happens transparently to the buffer cache. The smart_blkfilter filter driver inspects each request and processes all buffer_heads on a given request. These requests are then queued onto the disk device below (ide.c in our case). This process is shown in Fig. 8.

1. Port specification

In order for smart_blkfilter to determine if any processing should be done on the data, it must discover what port the request should go through. In order to support the specifying of ports in read/write requests, we added a *port* field to the buffer_head and

request structures (see Appendix A). The smart_blkfilter inspects the port field value. If the port is zero then this is a normal request, and smart_blkfilter should simply queue the request onto the next device without performing any further processing. If the port field is non-zero then the function registered at that port is invoked on the data. There is a total of 510 ports available to the user, with ports 0 and 1 being unavailable, serving as a flag for a normal request and a flag to go through the "no-op port" (an empty function call for testing and measurement purposes), respectively.

In order for the user program to specify through which port reads and/or writes should go, we implemented a new file open system call, open_smart_blkfilter(). This function call has an extra parameter, the port parameter, to allow a user to specify which port all reads or writes should go through. In order to change the port value or do normal file operations, the file must be closed and re-opened with a different port parameter or with the normal open() function call for normal reads and writes. All normal reads and writes cause the port value to be set to zero. One problem is that by the time our filter driver sees the I/O request, all traces of the file to which this block corresponds are gone. There is no concept of a "file" at the smart_blkfilter level, just disk blocks. In order to propagate the port values down to the buffer_head port fields, we had to trace the function call path of all reads, writes and opens. We added a *port* field to the file structure and page structures for the propagating of the port value down to the filter driver. We also had to modify several functions in the kernel to copy the port values from the file structures to the page structures and from the page structures to the buffer_head structures, where there are finally visible to the filter driver.

2. Read/write determination

One of the things that smart_blkfilter must determine is if the request is a read or a write request. For writes, smart_blkfilter does any necessary processing of the data buffer *before going to the disk*. For example, if we were performing some sort of encryption, the encryption function should be performed on the way down to the device. However, if we were wanting to perform filtering on the reads, then the smart_blkfilter should process the data buffer *after it has been read from the disk*. In this case, smart_blkfilter simply queues an incoming request onto the next device without performing any processing, deferring the processing until the request returns with the data from the disk. One problem is that the ll_rw_block() function, where smart_blkfilter is invoked, is not on the return path of the I/O requests. We had to apply another hook for our filter driver in the ide_end_request() function defined in /usr/src/linux/include/linux/blk.h. This function is invoked by ide.c when the device is finished processing *a particular buffer_head* on a request. The ide_end_request() function then determines if there are more buffer_heads on the request. If there are, the pointers and block counts for the request are adjusted accordingly. If not, the request structure is removed from the queue. We modified ide_end_request() to check for the presence of a layered driver before unlocking the buffer. If ide_end_request() sees a layered driver installed, a *callback function* is invoked. The callback function is implemented in smart_blkfilter. A pointer to the callback function is kept in blk_dev_struct (see page 18). The callback function then checks to see if the request is a read or a write. On a read, it applies the appropriate filter function on the data before

returning if the request has a non-zero port value.

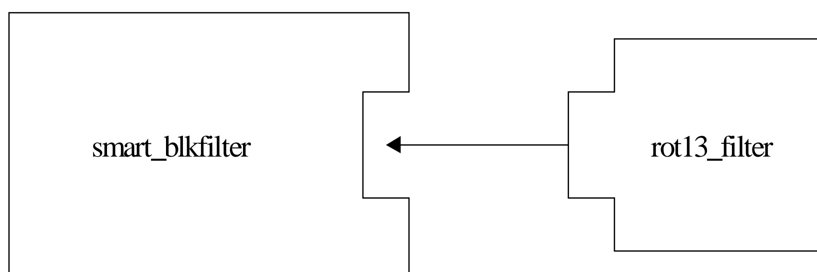
One final problem we had to deal with for file reads was file read ahead. When a file is read from the disk, the kernel, by default, issues read requests for subsequent blocks in the file in anticipation that the application will ask for these blocks in the near future. If a request for these next blocks is issued, the data blocks will already be in memory, and the read request can be satisfied more quickly. Specifically, the function generic_file_read() in mm/filemap.c calls the static function generic_file_readahead() to accomplish this. The function call trace for read ahead is slightly different than for normal reads, which necessitated further modification of the kernel to get the correct port values to the driver for read ahead. Another solution is to disable read ahead for requests that go through non-zero ports, which could present a performance problem. We actually compiled separate versions of the kernel for both of these cases and ran experiments to see the effects. The results of these experiments are presented in the next chapter.

3. Registering filter applets at the device

One goal of the multiported storage device architecture is to allow a user to be able to add functionality to an existing multiported device. In other words, we want to provide a way for users to “plug in” their own applets into the device. We do this by taking advantage of the *stacked module* capability in Linux. Although Linux does not provide transparent layering as Windows NT does, Linux does provide module stacking as explained on page 13. We exploit this stacked-module architecture to implement user-defined filter applets, or filter functions. Fig. 9 shows how we accomplish this.

We implement the filter applet itself as another Linux driver module. In the

smart_blkfilter module we export a smart_blkfilter_register_port() and smart_blkfilter_unregister_port() function to the kernel. This makes these functions available to other modules just as normal kernel symbols are available to modules. When the filter applet module is loaded, the kernel invokes the init_module() function of the driver. It is within the init_module() function that the filter applet driver calls this smart_blkfilter_register_port() registry function to register its applet with the smart_blkfilter. The registry function has as its parameter the port number at which the filter wants to register and a function pointer. If the filter applet calls the registry function with the port parameter set to zero, the smart_blkfilter attempts to dynamically allocate an available port. The filter "applet" interface is actually just a function that is called when the filter is invoked. The second parameter to the smart_blkfilter_register_port() function is a pointer to this function. This filter function has as its parameters whether the operation is a read or write, pointers to the buffer_head and request structures in question, and the direction that the request is going at the time of the invocation




```
short smart_blkfilter_register_port(short port_num,
        void (*filter_func)(int rw, struct request *req, struct buffer_head *bh, int direction))
```

Fig. 9. Registering of filter applet via Linux stacked module mechanism

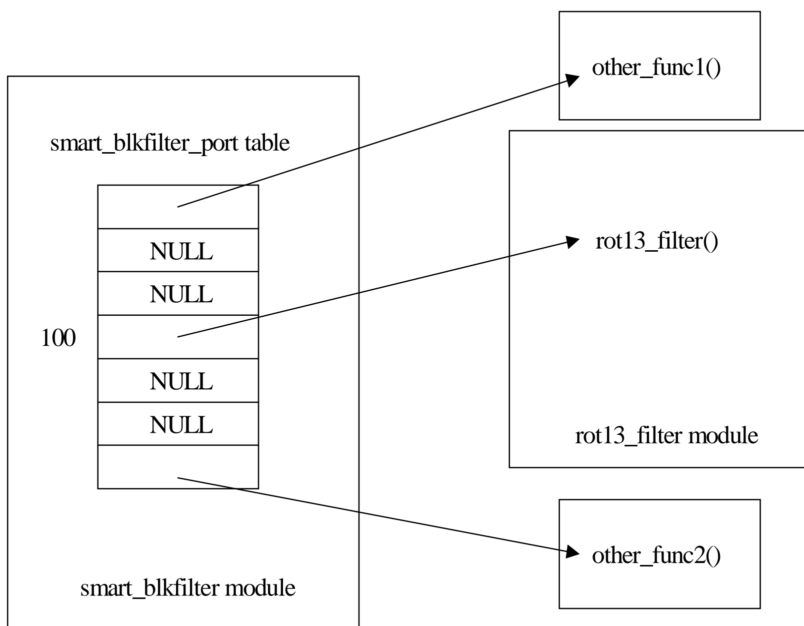


Fig. 10. Table of registered filter applets (functions)

(TOWARD_DISK or FROM_DISK). This is all of the information that the filter function needs to perform the necessary operations on the data.

The smart_blkfilter keeps a table of registered filter functions, with the index into the table being the port at which a function was registered. For example, a rot13 encryption filter applet could register a function called rot13_filter() at port 100 of the smart_blkfilter (see Fig. 10). Thus, the smart_blkfilter module merely provides a framework for registering and invoking filter functions. The rot13_filter() function would be the actual rot13 engine that would be invoked whenever a read or write goes through this registered port.

4. Preservation of original data

One problem with block filtering is preservation of the original data. The IDE driver deals with the buffer area passed to it from the buffer cache in the function call

ll_rw_block() in the original kernel implementation. However, by applying a filter to the data, we may compromise the integrity of the buffer cache. For example, let's assume we are writing a single 4K block of the character 'A' to the disk through a rot13 encryption port on the device. If we were to modify the data in place, the buffer in the cache block would now be filled with the letter 'N' instead. A subsequent read of the data block would result in a cache hit, but the returned block would contain the letter N. The buffer cache should not reflect the filter operation since it occurs below the buffer cache. Although we may want the data to be rot13 encrypted *on the disk*, the rot13 encryption may have no meaning to us at the application. The meaningful data was the block of A's that we wrote to the disk. We have essentially destroyed the original data in the buffer cache.

In order to remedy this problem, we have added two extra fields to the buffer_head structure – b_orig_data and b_orig_size. On a block write, we allocate a new buffer to hold the "filtered" data. In the context of the example, we save the 4K-block of N's in the newly allocated buffer. We then *switch the pointers* so that b_orig_data points to the original data (the block of A's), and b_data now points to the newly allocated buffer. We also save the size of the original data block in b_orig_size and store the size of the new block in b_size. For the current implementation, these two sizes should always be the same. Now the IDE device can deal with b_data and b_size as usual, completely unaware that we have changed the actual data area that they refer to. When our callback function is called upon returning from the disk write, we free the temporary buffer, reset the b_data pointer to point to the original buffer, and restore the

original size in b_size.

For a block read, we allocate the new buffer and adjust the pointers as we did for the write operation. However, since we are performing the filtering after the data is read from the disk, we do no more than allocate the new memory and adjust the pointer and size fields on the way down. When the callback is invoked, the filter operation is performed on the data read (pointed to by b_data, which is our new buffer) and the filtered data is then stored in b_orig_data (which is the buffer originally sent down by the buffer cache). We then reset the pointers so that the b_data pointer points to the filtered data (the original buffer). This makes sense on a file read, since the meaningful data on a file read is produced *after the filter operation*.

We have implemented two functions in our sample rot13_filter module, rot13_alloc_buf() and rot13_free_buf(), that take care of the dynamic memory allocation and pointer manipulation. They perform kmalloc() and kfree() kernel calls to perform memory allocation and deallocation, respectively. This keeps the main rot13 engine from having to worry about memory and pointer issues. Another design option would be to provide a smart_blkfilter_alloc_buf() and smart_blkfilter_free_buf() in the smart_blkfilter framework itself so that the applets would be free of having to deal with kernel memory and pointer management. These would be implemented exactly the same way as the rot13_alloc_buf() and rot13_free_buf() functions, except for the function name change and their location. These functions would be exported from smart_blkfilter the same way the registry functions are exported so that other modules could have access to them.

CHAPTER VI

EXPERIMENTATION AND MEASUREMENTS

A valid concern with the implementation described in the previous chapter, and with intelligent storage devices in general, is the impact that the extra layers of code at the device have on performance. Our implementation in particular adds extra layers of code at the device driver level, which obviously increases the I/O response time and reduces the throughput. The question is whether the performance hit taken by adding these extra layers of code outweighs the benefits. It is important to remember that we are dealing with disk I/O, and mechanical disks, although improving constantly, are still very slow compared to memory and the CPU. What would be significant delays in CPU-bound operations may be barely noticeable in I/O operations.

We devised several tests to measure the impact that the smart_blkfilter layer has on performance, both of filtered I/O operations as well as normal I/O operations that do not use the I/O filters at the device. The following sections outline the details of the experiments and the results.

A. Kernel Overhead Measurements of Filter Driver

In this set of experiments we took time stamps in the smart_blkfilter driver to measure the added overhead due to having the layered driver in the I/O stack. We performed 5MB sequential file reads and file writes in blocks of 4KB. We measured the time spent executing our layered driver code for normal I/O requests, i.e., for processes using the normal open file function (open()). This causes the port value to be set to

zero, and our filter driver, upon discovering this, simply queues the request onto the next device without invoking a filter applet. We ran two file-read experiments–one with file read ahead enabled in the kernel and one with file read ahead disabled–to see the impact of the driver layer on these two cases. We also performed a 5MB file write with the O_SYNC flag turned on. By default, Linux does delayed writes. This means that when a write is performed, the data is written to buffers in the buffer cache, the buffers are marked as dirty, and the write call returns to the application. The actual flush to disk happens at a later time, perhaps several seconds after the write command returns. In order to get meaningful results, we turned on the O_SYNC flag, which causes the file to be written to disk immediately. The write operation does not return until the write occurs

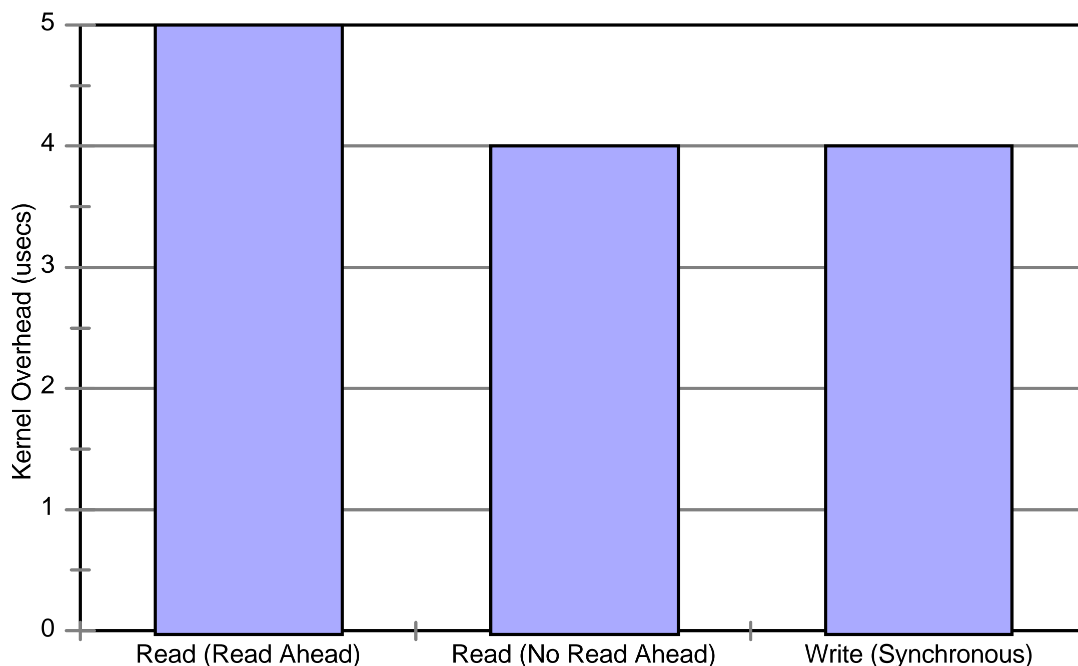


Fig. 11. Overhead due to presence of smart_blkfilter alone

on the media. We took the average of all of the 4KB requests for each case. Fig. 11 shows the results.

The average layered-driver overhead for file reads in Fig. 11 is 5 and 4 microseconds for file reads with read ahead enabled and read ahead disabled, respectively. For synchronous writes the overhead is 4 microseconds. Taking into consideration margins of error of the timers and averaging, we can say that the time is essentially the same for all three cases. File I/O is usually on the order of milliseconds, so we can safely say that the smart_blkfilter layer itself (without invoking filter applets) adds negligible overhead.

We implemented a sample filter "applet"–a rotation 13 (rot13) filter driver called rot13_filter. This filter module implements the rotation 13 encryption algorithm, which simply takes any alpha character and adds 13 to its ASCII value while preserving the case of the letter. Any addition that causes the character to overrun Z (or z) wraps back around to the beginning of the alphabet. For example, the letter *A* becomes an *N*, and the letter *V* becomes an *I*. The decryption algorithm simply subtracts 13 from the character, with additions that underrun the letter A (or a) wrapping back to Z (or z). Subtracting 13 from the letter *N* returns an *A*, and subtracting 13 from the letter *I* returns a *V*. Upon further observation, one will notice that adding and subtracting 13 give the same result. For example, subtracting 13 from the letter *A* also results in the letter *N*. Thus, encryption and decryption can be implemented with the same function.

We implemented a very efficient rot13 algorithm that uses bit-wise operations on

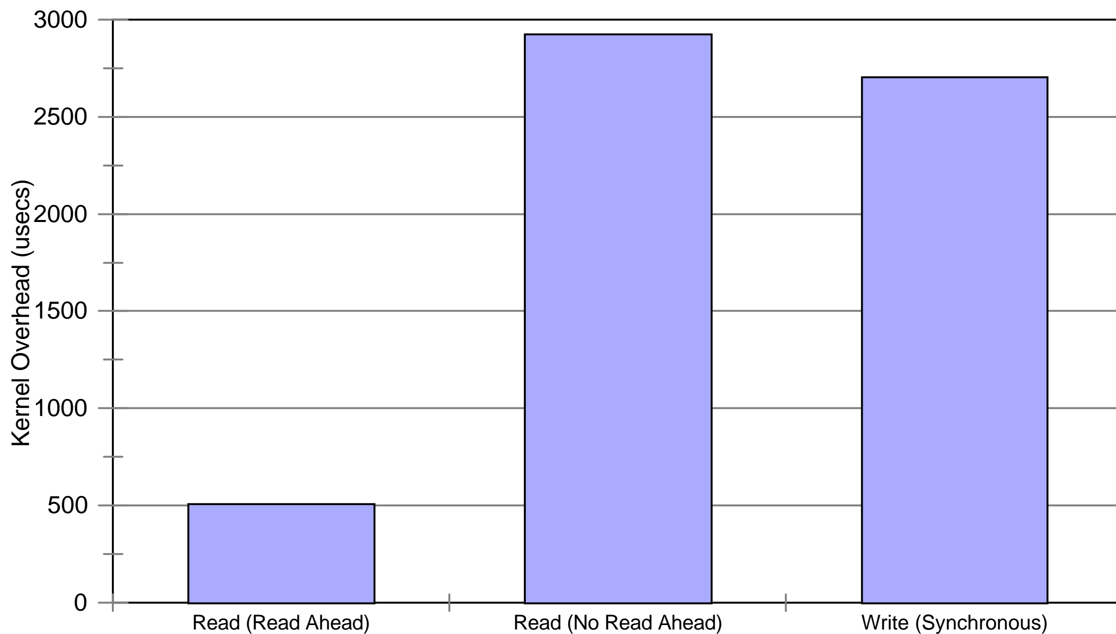


Fig. 12. Overhead of smart_blkfilter using rot13_filter port

the input character to return the rot13 result. We registered the rot13 module with smart_blkfilter using the exported registry function described previously. We then ran the same experiments described earlier in this section, except our test application used the open_smart_blkfilter() function call we implemented to pass the port value of the rot13 filter to the kernel. Fig. 12 shows the results.

File reads with read ahead enabled incur only a 500 *microsecond* overhead cost per 4KB block read, while reads with read ahead disabled and synchronous writes incur about a 2.9 and 2.7 *millisecond* cost, respectively. With read ahead enabled, traces showed that each request structure has a substantial list of buffer_head structures on its buffer_head queue. With read ahead disabled each request structure has at most four 1KB buffers on its buffer list (the buffer size matches the block size of the underlying

disk). Request coalescing increases the efficiency of the filter driver. It reduces the number of requests on the request queue, instead putting multiple buffer_head structures *on a single request*. This is advantageous because smart_blkfilter incurs extra work when it is finished processing a request. It must remove the request from its internal queue and then queue it onto the next device (ide) *in scan order*. This means traversing a linked list. By coalescing requests, buffers are already in scan order on the request structure, so the number of times the smart_blkfilter has to manipulate the ide request queue are fewer. This means less time traversing the linked list and less overhead incurred.

B. Application-Level Measurements

To get a better understanding of the impact that the overhead measurements examined in the previous section have on overall performance, we ran several experiments to measure application-level response time/throughput. We again analyzed file reads with and without read ahead enabled as well as synchronous file writes. We first took application-level read/write times for the normal case (no smart_blkfilter layer). We then installed the smart_blkfilter only and took measurements. Next, we registered our rot13_filter applet with the smart_blkfilter and took read/write time measurements. Finally, for comparison purposes, we coded the rot13 operations into our test application and performed the rot13 on each block of data at the application layer. During this last experiment we did have the smart_blkfilter layer installed (no registered applets) to achieve a fair comparison. We expressed all results in terms of average time to read/write a single 4K block of data for easy comparison to kernel overhead figures of the previous section. Throughput values can be easily obtained from these figures through simple

arithmetic. Fig. 13 shows the results for file reads.

The first thing we notice is that for the baseline case of normal file reads (no smart_blkfilter layer), both read-ahead and no-read-ahead results are almost the same, around 530 microseconds. The reason for this is that even though read ahead does not occur in the kernel for the no-read-ahead case, read ahead does occur at the disk itself. All hard disks today come with on-board cache memory to speed up disk access. When reads are performed, the disk itself performs read ahead so that requests for subsequent blocks on the disk are satisfied from the disk cache instead of having to go to the media. This eliminates the mechanical delays for requests of subsequent blocks. For normal sequential reads, the read ahead performed by the disk itself is good enough that the gains in carrying out additional read ahead in the file system are minor. The "Layer Only" bars

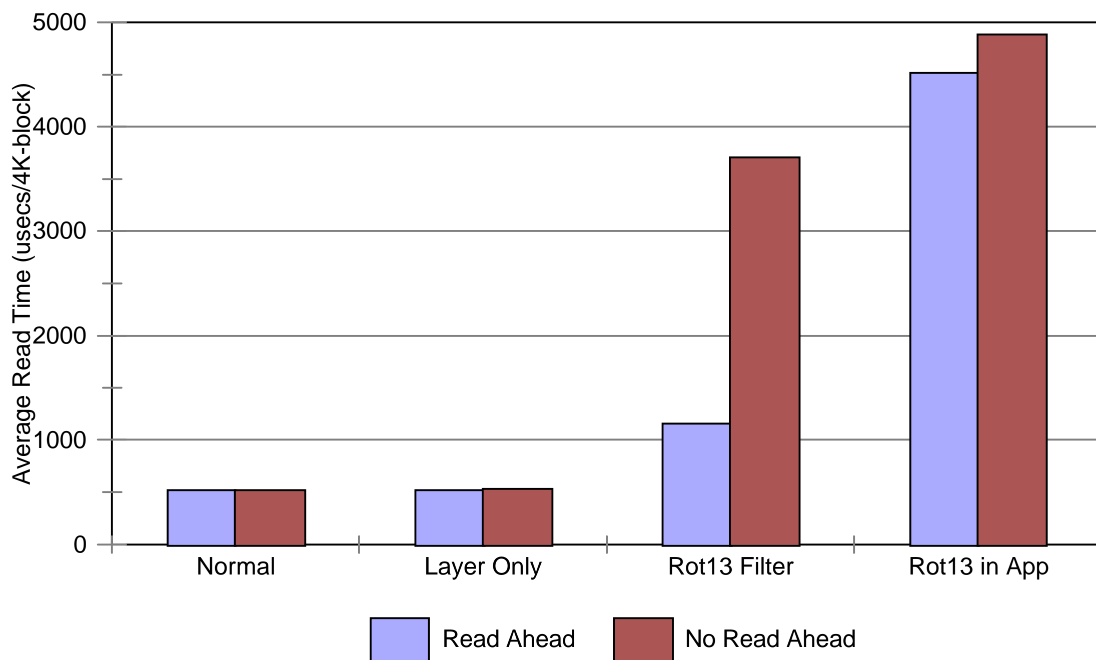


Fig. 13. Application-level read response times

in Fig. 13. show negligible increase in response time from the baseline (normal) case. We saw in Fig. 11 that the smart_blkfilter layer alone adds very little overhead to the I/O call stack, which confirms these results.

For the case of the rot13 driver module in Fig. 13, we see that the read ahead test shows around a 600-microsecond increase (from 0.5 to about 1.1 milliseconds) in read time per 4K block over the layer-only case. This is consistent with the overhead results in Fig. 12 that showed the rot13 filter adding an extra 500 microseconds to the I/O call path. The no-read-ahead case shows a much larger increase, from about a 500 microsecond to a 3.7 millisecond average read time. Again, this is consistent with the added overhead figures shown in Fig. 12. The application-level read times exhibit slightly higher overheads than the kernel times of the previous section due to the fact that we are measuring the *entire time spent* to satisfy the request in the user program. Also, normal-priority user processes can be frequently interrupted, put in wait queues, etc., during their execution. The overhead measurements in the previous section were taken in the kernel, so they were not subject to the high frequency of interrupts that the user processes were subject to.

We finally see that performing the rot13 encryption in the application itself caused a nearly fourfold increase in response time over the case when rot13 was performed in the filter driver in the read ahead experiment. Again, this is due to the fact that the rot13 execution in the application itself was subject to frequent interrupts, context switches, etc., while the rot13 filter module was operating in kernel space and was not subject to frequent interrupts. The no-read-ahead case also showed an increase of over a

millisecond of average read time for rot13 performed in the application over rot13 performed in the filter driver.

Fig. 14 shows the results for synchronous writes. The first observation is that synchronous writes take much longer than reads in general. This is due to the fact that writes obviously do not benefit from disk read ahead as disk reads do. The application may have to wait for an actual disk write for every 4KB block, which can take a significant amount of time. Most likely the kernel programmers were aware of this problem, which is why delayed writes are performed by default. Indeed, synchronous writing of files is often not necessary. Delayed writes decouple the application from the disk, so most of the time file write latency will not be an issue. We see from Fig. 14 that the rot13 filter driver adds a few milliseconds of latency to the block write on

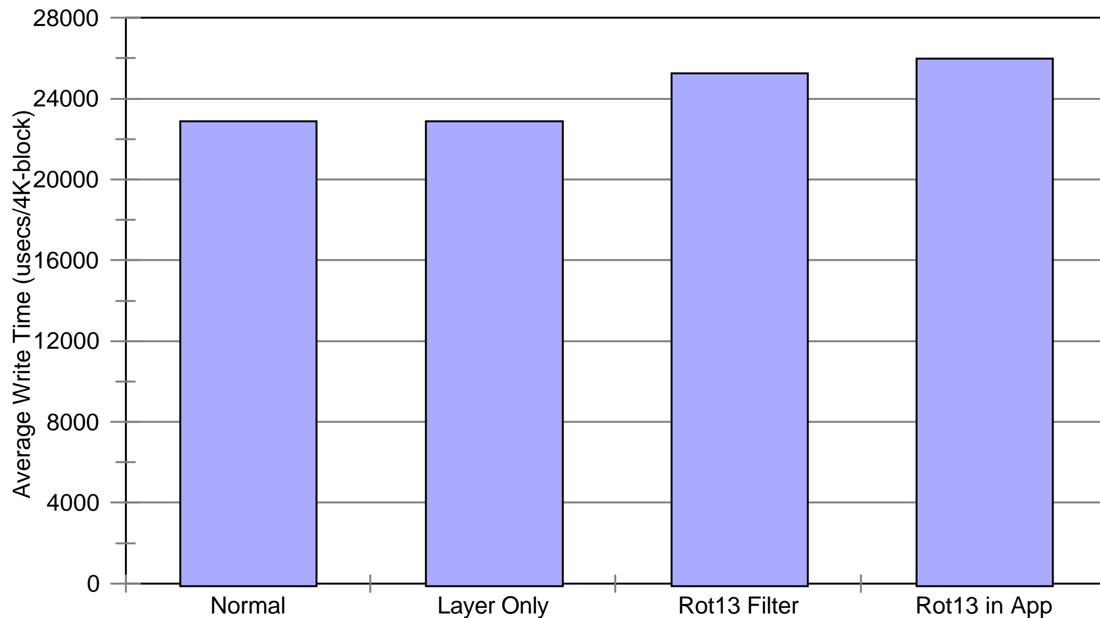


Fig. 14. Application-level write response times

average. The average write times for rot13 in the driver versus rot13 in the application are closer to each other than in the read case. Again, most of the time write operations are delayed. In fact, delayed writes decouple our filter driver from the user application altogether, so that no penalty is seen in the user application. We performed synchronous writes purely for measurement purposes. Synchronous writes have large latencies in general, so the performance impact of our filter driver is not as significant as it might have been otherwise.

The results discussed here are promising, especially for file reads. They show that it is possible to have an increase in performance using multiported storage, where application-level code is migrated to the disk, versus conventional storage, where the "filtering" of the data occurs in the application. We comment more on this later.

C. Overhead Measurements for Periodic Requests and Multiple Processes/Ports

Our final set of experiments involved taking kernel-level measurements in the filter driver to see the effect of having multiple processes reading from multiple ports simultaneously. We also had our user-level test applications issue periodic 4KB requests as opposed to continuous requests. This could be a likely scenario for a multimedia server, where several clients could be making periodic requests to the server through several ports. We analyze only file reads and not file writes in this section, as we are assuming than in many applications, such as multimedia servers, the writing of files to the disk is likely to happen "off-line." Once files are stored on disk and placed on-line, only file reads would come into the server from clients. Read response time and throughput are many times of greatest concern in these situations. Moreover, delayed writes can be

used to amortize the latency in file writes as previously discussed.

In all of the following experiments we vary the number of processes running our test program from one to four. Each simultaneously-running process reads a different 2MB file from a different port of the smart_blkfilter. We avail ourselves of the no-op port (port 1) of the smart_blkfilter to measure the overhead due to the smart_blkfilter layer alone. We vary the number of processes that are reading through the no-op port from one to four. We run these same experiments again, this time varying the number of instances of the rot13_filter driver module registered with smart_blkfilter from one to four, one for each process. All simultaneously active processes communicate with the disk at the same rate. We vary the rate between experiments from 10 I/O requests of 4KB per second to 40 I/Os per second by increments of 10. Finally, we run all of these

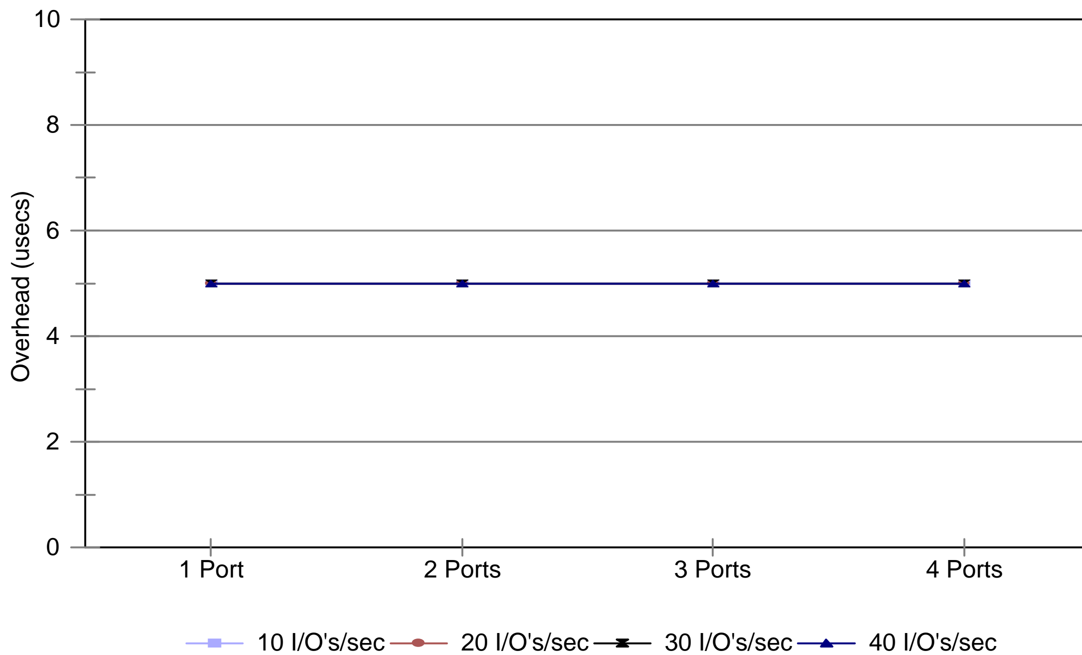


Fig. 15. Processes reading through no-op port - read ahead enabled

experiments twice, once with read ahead enabled and again with read ahead disabled.

Fig. 15 shows the smart_blkfilter overhead incurred in the kernel for all processes reading through the no-op port. As we vary the number of ports from one to four, we see that for all periodic rates, the overhead stays constant. Again, the overhead due to the layer itself with read ahead enabled is negligible.

Fig. 16 shows the results for the same set of experiments but with read ahead disabled. We see that, in general, the overhead is a little higher than for the read-ahead case. However, we are still only experiencing average overhead values of 10 microseconds and lower, which are negligible compared to the overall I/O response time for a given request. Here we observe that as we increase the number of active processes

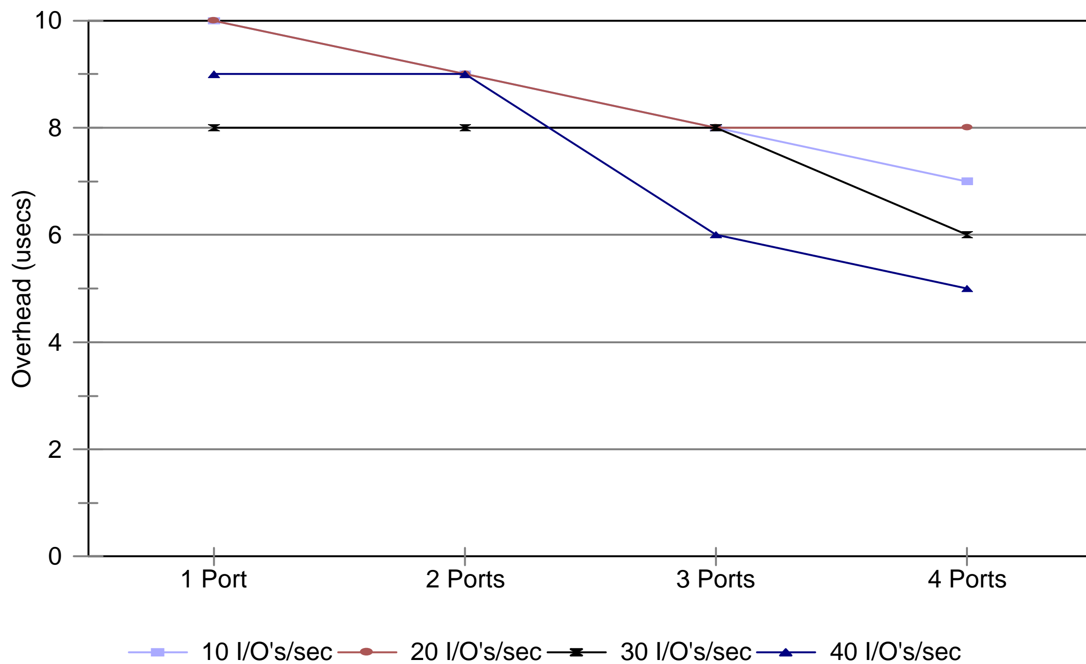


Fig. 16. Processes reading through no-op port - read ahead disabled

from one to four, the overhead actually decreases slightly. At these resolutions (several microseconds), measurement margins of error can be a factor. The overall observation is that for both cases–with read enabled and with read ahead disabled–the overhead of the smart_blkfilter layer alone is negligible.

Fig. 17 shows the kernel overhead for the driver layer when the rot13 filters are registered with smart_blkfilter and read ahead is enabled. We vary the number of ports registered (and, thus, the number of simultaneous processes) from one to four. The results are mixed. For the 10 I/Os-per-second experiment, we see that the overhead oscillates from around 400 to 355 to near 380 to around 360 microseconds as we increase the number of rot13 ports being read. The plots for 20, 30, and 40 I/Os-per-second also show mixed results. It is difficult to make any definitive conclusions from

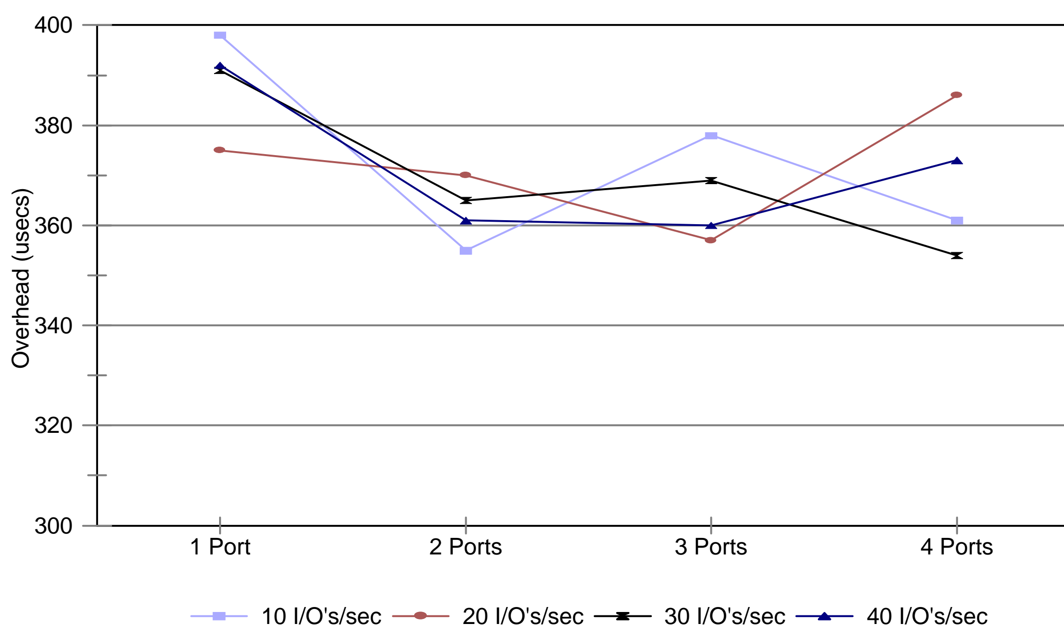


Fig. 17. Processes reading through rot13 port - read ahead enabled

this chart alone. One observation we make is that all overhead times lie within the 350-400 microsecond range, which is comparable to the 500-microsecond overhead value for sequential I/O with read ahead shown in Fig. 12. Another observation is that the general trend seems to be a slight decrease in overhead as we increase in the number of ports being simultaneously read for all rates except the 20I/Os-per-second plot. Margins of error in measurement may be a factor again here. We comment more on this chart in the context of the rest of the data in the next section.

The results for file reads with no read ahead is a bit more definitive than the read-ahead case, as shown in Fig. 18. Here we see that for all periodic rates we have an average filter driver overhead of about 3.2 milliseconds at 1 I/O port/process, which is comparable to the kernel overhead measurement for the no-read-ahead case of Fig. 12.

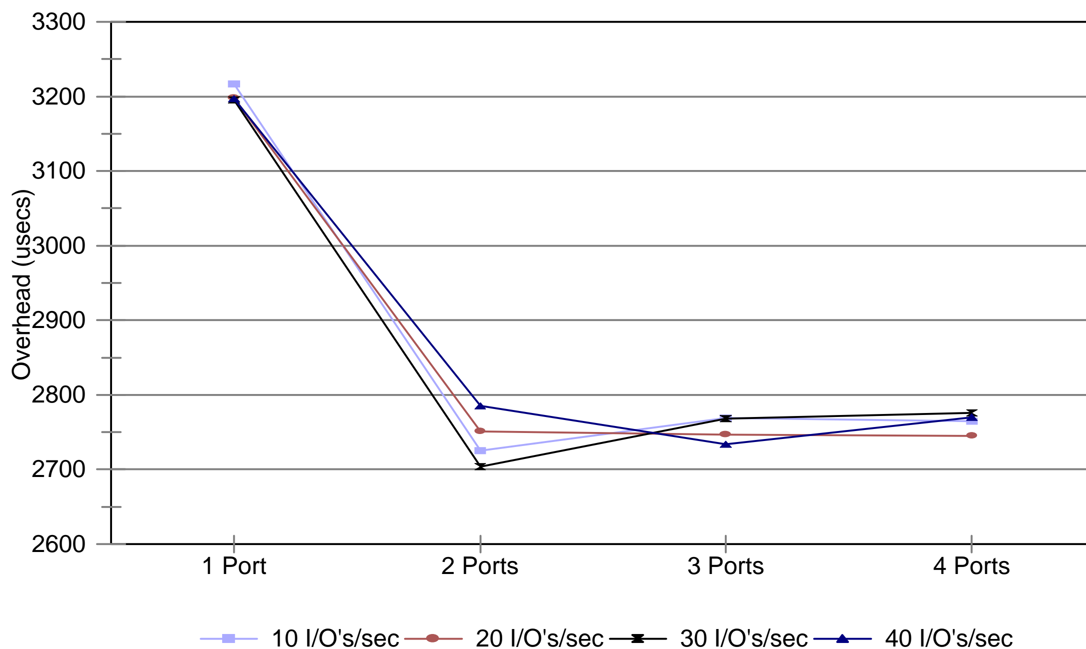


Fig. 18. Processes reading through rot13 port - read ahead disabled

We see here that as we increase the number of ports being simultaneously read, the overhead incurred at the filter driver decreases in a definitive manner, much more so than the read-ahead case of Fig. 17. The average overhead drops down nearly half a millisecond to 2.7 milliseconds. The filter driver performs better when multiple processes are reading the disk than when a single process is reading the disk. We further explore these results and the others in this chapter in the next section.

D. General Observations and Analysis of Experimental Results

We observed some interesting results from the previously-outlined experiments. One general observation is that the impact on performance of the smart_blkfilter alone is inconsequential, which is good news for those applications not wanting to use the services of smart_blkfilter. The performance of these applications is not affected by the presence of the layer.

A second observation is that performing rot13 operations in the filter applet gave us better performance than performing the operations in the user application. This is due to the fact that rot13 in the filter driver is executing at a higher priority than rot13 in the application. The rot13 kernel driver is not subject to frequent context switches and interrupts that the user application is subject to.

Another interesting observation is that there is less driver overhead per 4KB request when we have multiple outstanding requests than when we issue single 4KB requests. The following is the explanation. The overhead for read ahead was significantly less than for no read ahead as observed in Fig. 12. As explained previously, this is because read ahead causes a large amount of request coalescing.

For the periodic request experiments, we saw that, in general, for higher numbers of simultaneous processes, the overhead measured in the kernel went down. The read-ahead case of Fig. 17 showed only a small decrease in average overhead. Since read ahead is occurring, the periodic requests at the application are actually being turned into bursts of sequential requests at the driver due to read ahead. This means that there is already much request coalescing occurring, therefore increasing the I/O rate does not have as much of an effect on overhead values.

For the no-read-ahead case of Fig. 18, however, there are fewer opportunities for request coalescing since we are reading from different files. Nevertheless, we see a larger overhead for the 1 port/process case than for the 2-4 ports/processes case of Fig. 18. The reason for the reduction of overhead in this case is the following. There is some overhead incurred when the request queue is empty and the first request is being queued onto the device. The next device in the I/O stack (ide.c) is not actually started immediately, but a task is loaded onto a *task queue* to be run at a later time [16]. This task, in our case, is the one that actually performs the disk read. When the request queue is kept busy, however, requests are queued immediately into the already running request queue. This saves some time due to the elimination of the overhead in the kernel of setting up the task queue and waiting for it to be triggered. The extra work of setting up the task queue for a request is not performed as often as we increase the number of ports simultaneously read since the request queue is already initialized and loaded.

CHAPTER VII

SUMMARY AND CONCLUSIONS

We have introduced an intelligent I/O device called a multiported storage device. Multiported storage devices, inspired by Active Disks, exploit general-purpose processing power at the device itself. However, unlike Active Disks, a multiported storage device allows application-specific code to be downloaded to the device while still maintaining the simple block-level interface. In addition, a multiported device can have several software "ports" through which applications can communicate with the device. Application-specific code, or "filter applets," can be registered at these ports. This allows the flexibility of having several different, unrelated pieces of application code at the device.

We implemented a prototype multiported storage device on a PC running the Linux operating system. We implemented the multiported layer of code, called *smart_blkfilter*, as a kernel module. This module intercepts requests as they enter/leave the disk, performing filtering functions based on the port value sent to the device. The smart_blkfilter also provides a mechanism for registering filter applets implemented as kernel modules using the Linux stacked module mechanism.

The Linux kernel module implementation outlined in this thesis has the advantage of being transparent to user applications not needing its services. For applications needing its services, the only change necessary at the application level was the implementation of the open_smart_bklfilter() system call, which allows a file to be opened

with an extra parameter, the port parameter. Unfortunately, this requires applications to be changed to use this special open function call. While sufficient for test purposes, another method of passing the port value to the driver that does not require changing the application is desirable. One possibility is using the UNIX directory namespace to indicate the port number. For example, we could open a file with the full path of /port1/home/mgrande/myfile. While parsing the directory entries, the file system could be modified to recognize the "port1" at the beginning of the path as indicating the port value to be used for all file reads/writes. The rest of the path points to the actual directory path of the file in the file system. This pseudo-file namespace is a concept similar to the /proc file system in Linux, where many entries in the /proc file system are not files at all, but interfaces to things such as system information and statistics [16]. This would require further modifications of the kernel, but applications would not have to be changed.

Another potential problem with the implementation presented in this thesis is the fact that filter applets, which are kernel modules, are operating in kernel space and have access to critical kernel data structures, such as the request queue and buffer_head lists. Pointers to these structures are passed to the filter functions even though the functions do not need to manipulate them–they need only to manipulate the data areas that they point to. A poorly programmed or malicious filter module, however, can compromise file and even file system integrity by manipulating these data structures incorrectly. [1] discusses the problem of protecting against potential damage at the device from faulty code. Several possibilities are explored, including compilation, pre-compilation, compile-time, load-time, and run-time safety checks. Another solution explored is proof-carrying code,

where the runtime system publishes a safety policy that all programs must adhere to. The code producer is responsible for providing a proof that the program meets the limitations of the policy. While this method has been demonstrated for small functions and simple policies, generating proofs for arbitrary programs is known to be intractable [1]. The feasibility of all of these methods would have to be analyzed in the context of kernel modules for our particular implementation.

A related problem to access to critical kernel data structures is memory allocation in the kernel. A renegade filter module can leave memory leaks, exhaust kernel memory, or both. One partial solution to the memory allocation problem was outlined in V.C.4. We could define a smart_blkfilter_alloc() and smart_blkfilter_free() interface in our smart_blkfilter module for memory allocation/deallocation. These functions would take care of all buffer and pointer manipulation. They could be implemented exactly as the rot13_alloc() and rot13_free() functions in the sample rot13 filter, except that they would be moved into the smart_blkfilter driver for use by all filter modules. An alternate implementation, and one less prone to consuming large amounts of kernel memory, would be to allocate a large buffer in smart_blkfilter upon initiation of the driver. Memory allocation for filter modules would actually be taken from this pre-allocated buffer. This would require memory management code for the internal buffer, which is more complex than the current kmalloc/kfree method of dynamic memory allocation that we implemented.

Another area that may require further investigation is the buffer cache semantics. In our implementation, we took measures to insure the integrity of the buffer cache

buffers by making copies of intermediate data to dynamically allocated buffers. Several questions concerning the buffer cache remain to be answered. If process A reads a disk block through port 5, and process B tries to read the block through port 6, the second process' request may not even go to disk since the buffer is already in memory. Maybe process A read a multimedia file at 1Mbps through port 5, and process B is trying to read it at 56Kbps through port 6. Would it be easier just to not cache any blocks that have gone through filters? Perhaps the kernel could be modified to check for non-zero port values in the buffer_head structure of cached blocks. If the port value is non-zero and does not match the port value of the current request, then the kernel might mark that buffer invalid and retrieve the buffer again from the disk through the new port. The question of defining buffer cache semantics that are suitable for all possible combinations of user applications is an issue that may require further investigation.

Finally, we stated in V.C.4 that we save the original size of the buffer cache block in b_orig_size in case the size changes. The size of all buffers was kept the same in our example. However, a filter applet that performs compression, for example, could cause a change in block size. This can actually be quite problematic, as the buffer cache block size and disk block sizes may no longer match. In fact, block sizes may not even be multiples of sector sizes after compression. Padding with zeroes to compensate could negate any compression. This is a complex issue that would have to be investigated further.

REFERENCES

[1]Erik Riedel and Garth Gibson, "Active Disks - Remote Execution for Network-Attached Storage," Technical Report CMU-CS-97-198, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1997.

[2]Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel and David Rochberg, "A Case for Network-Attached Secure Disks," Technical Report CMU-CS-96-142, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1996.

[3]Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg and Jim Zelenka, "File Server Scaling with Network-Attached Secure Disks," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, Seattle, WA, June 15-18, 1997. Available via http://www.pdl.cs.cmu.edu/Publications/publications.html.

[4]Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Erik Riedel, David Rochberg and Jim Zelenka, "Filesystems for Network-Attached Secure Disks," Technical Report CMU-CS-97-118, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1997.

[5]Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg and Jim Zelenka, "A Cost-Effective, High-Bandwidth Storage Architecture," in *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, 1998. Available via http://www.pdl.cs.cmu.edu/Publications/publications.html.

[6]Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst and Jim Zelenka, "NASD Scalable Storage Systems," in *USENIX 99, Extreme Linux Workshop*, Monterey, CA, June 1999. Available via http://www.pdl.cs.cmu.edu/Publications/publications.html.

[7]Erik Reidel, Garth Gibson, Christos Faloustos, "Active Storage for Large-Scale Data Mining and Multimedia," in *Proceedings of the 24th Conference on Very Large Databases (VLDB '98)*, New York, NY, August 24-27, 1998. Available via http://www.pdl.cs.cmu.edu/Publications/publications.html.

[8]Anurag Acharya, Mustafa Uysal and Joel Saltz, "Active Disks," Technical Report TRCS98-06, Department of Computer Science, University of California Santa Barbara, CA, March 17, 1998.

[9] Mustafa Uysal, Anurag Acharya and Joel Saltz, “Evaluation of Active Disks for Large Decision Support Databases,” Technical Report TRCS99-25, Department of Computer Science, University of California Santa Barbara, CA, July 1999.

[10] *The Free On-Line Dictionary of Computing*. Available via http://foldoc.doc.ic.ac.uk/. [Accessed February 2000]

[11] Steven McCanne, Martin Vetterli, and Van Jacobson, “Low-complexity Video Coding for Receiver-driven Layered Multicast,” *IEEE Journal on Selected Areas in Communications*, Vol. 16, No. 6, pp. 983-1001, August 1997.

[12] Rémy Card, Éric Dumas and Franck Mével, *The Linux Kernel Book*, John Wiley & Sons Ltd, Chichester, West Sussex, England, 1998.

[13] David A. Rusling, *The Linux Kernel*, January 1998. Part of the Linux Documentation Project. Available via http://www.linuxdoc.org/docs.html.

[14] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice Hall PTR, Englewood Cliffs, NJ, 1986.

[15] Helen Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.

[16] Alessandro Rubini, *Linux Device Drivers*, O’Reilly & Associates, Inc., Sebastopol, CA, 1998.

[17] Art Baker, *The Windows NT Device Driver Book: A Guide for Programmers*, Prentice Hall PTR, Upper Saddle River, NJ, 1997.

APPENDIX A

SELECTED KERNEL DATA STRUCTURES

The following are the definitions for the buffer_head, request, and blk_dev_struct structures. The buffer_head structure is located in /usr/src/linux/include/linux/fs.h, and the request and blk_dev_struct structures are in /usr/src/linux/include/linux/blkdev.h. These definitions show the changes made to support the multiported storage device implementation described in this thesis.

```
struct buffer_head {
	/* First cache line: */
	unsigned long b_blocknr;		/* block number */
	kdev_t b_dev;			/* device (B_FREE = free) */
	kdev_t b_rdev;			/* Real device */
	unsigned long b_rsector;		/* Real buffer location on disk */
	struct buffer_head * b_next;	/* Hash queue list */
	struct buffer_head * b_this_page;	/* circular list of buffers in one page */

	/* Second cache line: */
	unsigned long b_state;		/* buffer state bitmap (see above) */
	struct buffer_head * b_next_free;
	unsigned int b_count;		/* users using this block */
	unsigned long b_size;		/* block size */

	/* Non-performance-critical data follows. */
	char * b_data;			/* pointer to data block (1024 bytes) */
	unsigned int b_list;		/* List that this buffer appears */
	unsigned long b_flushtime;	/* Time when this (dirty) buffer should be written */
	unsigned long b_lru_time;	/* Time when this buffer was  last used. */
	struct wait_queue * b_wait;
	struct buffer_head * b_prev;		/* doubly linked list of hash-queue */
	struct buffer_head * b_prev_free;	/* doubly linked list of buffers */
	struct buffer_head * b_reqnext;	/* request queue */

	/*
```

```
 * Marcus Grande - layered driver support
 * 10/29/99
 */
char * b_orig_data;/*
                 * used for keeping track of pointer to original
                 * data after the filter operation.  b_data
                 * (and b_size) must be changed to point to
                 * the modified (filtered) data on writes, and
                 * b_orig_data points to the actual buffer
                 * of the buffer cache.
                 * On reads, data comes from the disk, so we
                 * have to keep track of the original buffer
                 * cache buffer using this pointer and copy
                 * the filtered data into this buffer, then
                 * readjust the pointers, and finally free
                 * the temp buffer.  See rot13_filter.c for
                 * more details
                 */
unsigned long b_orig_size; /* size of b_orig_data buffer */
short int port;    /* what port, or "filter", should be applied on
                 * data. port 0 will be the no-op port, i.e.,
                 * don't do anything to the data, just pass it
                 * to/from ide driver as is
                 */
/*
 * Some MD stuff like RAID5 needs special event handlers and
 * special private buffer_head fields:
 */
        void * personality;
        void * private_bh;
};



struct request {
        volatile int rq_status; /* should split this into a few status bits */
#define RQ_INACTIVE             (-1)
#define RQ_ACTIVE               1
#define RQ_SCSI_BUSY            0xffff
#define RQ_SCSI_DONE            0xfffe
#define RQ_SCSI_DISCONNECTING   0xffe0

        kdev_t rq_dev;
        int cmd;                /* READ or WRITE */
```

```
        int errors;
        unsigned long sector;
        unsigned long nr_sectors;
        unsigned long current_nr_sectors;
        char * buffer;
        struct semaphore * sem;
        struct buffer_head * bh;
        struct buffer_head * bhtail;
        struct request * next;
        /*
         * smart_blkfilter support
         * Marcus Grande
         * 11/19/99
         */
        short port;
};
```

VITA

Marcus Bryan Grande received a B.S. degree in Electrical Engineering in December of 1997 and an M.S. degree in Computer Engineering in May of 2000, both from Texas A&M University in College Station, TX. He spent August 1993 to December 1993 as a research engineer co-op at the National Institutes of Health in Bethesda, MD. Here he worked in a blood research lab on the development of a prototype high-speed calorimeter. From May 1994 to May 1995 he worked as a research engineer co-op at the Walter Reed Army Institute of Research's Blood Research Detachment in Rockville, MD. Here he continued his work on the calorimeter. He also served a mission for the Church of Jesus Christ of Latter-day Saints from June 1990 to May 1992 in the San José, CA area, working primarily with the Latin community. Marcus is currently employed as a software engineer in the AIX I/O Subsystems (Base Device Drivers) department of the UNIX division at IBM in Austin, TX. He is married and has one daughter.

His contact address is P.O. Box 271376, Corpus Christi, TX 78427-1376.

The typist for this thesis was Marcus Bryan Grande.