The Design, Implementation, and Analysis of the Stony Brook Video Server

A DISSERTATION PRESENTED

BY

MICHAEL VERNICK

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

STATE UNIVERSITY OF NEW YORK

AT STONY BROOK

DECEMBER 1996

Abstract of the Dissertation

The Design, Implementation, and Analysis of the Stony Brook Video Server

by
Michael Vernick

Doctor of Philosophy
in
Computer Science
State University of New York at Stony Brook
1997

Advisor: Tzi-cker Chiueh

The combination of high-resolution graphics, high-speed computing, networking and storage devices, and excellent compression techniques has made it feasible for computer systems to deliver interactive multimedia to end-users via a high-speed network. This dissertation presents the design, implementation, and analysis of such a system, the *Stony Brook Video Server (SBVS)*. The *SBVS* employs only off-the-shelf PC components and is capable of guaranteeing the real-time delivery of digital video streams from the server's disk subsystem, through an Ethernet-based local-area network, to an end user's display. The *SBVS* integrates a software-based video storage subsystem *(VSS)*, with a real-time Ethernet-based subsystem, *RETHER*, which guarantees the smooth delivery of multimedia data on a shared Ethernet.

The main focus of the dissertation is on the design, implementation, and performance analysis of the *VSS*. The multimedia storage server design issues—including style of data delivery; the size, striping, and placement of media files; disk access algorithms; and buffer management schemes—are presented in the context of the methods used by the *VSS*. A detailed performance analysis shows that the *VSS*, with six disks and three SCSI controllers can support up to 89 simultaneous video streams at 1.5 megabits/second per stream.

In addition, the I/O-based admission control problem is stated and a framework for deciding whether or not to admit a new stream is provided. Five different admission control algorithms are presented and evaluated under a variety of hardware configurations and workloads. This is the first known work to present an empirical evaluation and comparison of video storage server admission control policies. It demonstrates that the statistical-based algorithm that uses measurements of the running system allows more streams than a deterministic algorithm which uses worst-case values.

This dissertation also presents a detailed analysis of methods for supporting fast forward and fast rewind of MPEG system streams. The analysis includes a comparison of implementation complexities, resource usage, and visual output for each of the methods. The comparison reveals that there is no clear "*best*" method. Each method has its advantages and disadvantages.

To Audrey, Jacob, Herb, Enid and Rich

Contents

List of Tables

List of Figures

Acknowledgements

I sincerely thank my advisor, Tzi-cker Chiueh, for his help, support, and guidance in creating the Stony Brook Video Server. His technical knowledge and problem-solving abilities are unsurpassed. I'd also like to thank my first advisor, Gary Schloss, who put me on the road and taught me to ask the right questions. Of course, this project could not have been completed without my colleague, Chitra Venkatramani, who built the networking portion of the system. I thank her for not getting too upset when I found system problems and suggested that they were bugs in her code (they were usually bugs in my code). I also want to thank my lab mates in the Experimental Computer Systems Lab, Manish Verma and T.N. Niranjan. Thanks to my office mate Karen Bernstein and my friends Daren Krebsbach and Michael Wynblatt who kept me up to date on all of the gossip. Thanks to Kathy and Betty who helped me get through all of the administrative hassles. Lastly, this work would never have been completed if it weren't for the unwavering love and support I recieved from dear wife, Audrey.

Chapter 1

Introduction

Although the punch card was invented in 1928, I was still using it as my programming tool as a college freshman in 1979. By the time I was a senior, I was using an ascii character terminal connected to an IBM mainframe. The desktop I now use has more power than that entire mainframe. Little did I know then how far computer systems would progress in such a short time.

The Alto, developed at Xerox Parc in 1974, was the first workstation. It performed at near mainframe speeds, allowed multiple windows to be opened on a monitor, used a mouse as an input device, and was connected to other machines by a network called an Ethernet. The PC revolution that followed in the 1980s gave us video cards that could show images with high resolution and sound cards for music and voice. The doubling of processor speeds every few years coupled with VLSI design technology has brought images, videos, and sound to our desktop. Although uncompressed videos and sound require massive amounts of digital storage, compressed versions can be stored using an order of magnitude less space. Hardware devices are able to decode compressed versions for real-time playback.

Back in the early 1970s, IBM developed the Winchester drive, a stacked platter of magnetically coated disks that could store 30 megabytes of data. Seagate produced the mini-Winchester drive in 1980 for the smaller computer; it could hold 5 megabytes of data. In 1996, a $200 hard disk can hold more than 2 gigabytes of data.

The combination of these technologies, high-resolution graphics, high-speed computing, networking and storage devices, and excellent compression techniques will make it feasible for computer systems to deliver interactive multimedia to end-users via a high-speed network.

This thesis is concerned with such a computer system, a *networked multimedia server*. It allows users to watch and listen to digital video and audio stored on a centralized server. In this type of system, the server stores digitized multimedia data and delivers it to client sites via high-speed networks in real-time.

Compared to the traditional networked file server, networked multimedia computing introduces two new problems. First, the large size of multimedia objects requires large aggregate I/O bandwidth (see table 1). The multimedia server must provide efficient methods for storing and retrieving large amounts of data at high speeds. Second, real-time performance must be guaranteed to ensure smooth playback at the client display. Multimedia objects convey meaning only when presented continuously in time. This is in contrast to textual objects for which spatial continuity is sufficient.

To support multimedia successfully, one must effectively address these two issues in almost every aspect of the computing system, including processors, memory, networking, I/O, and system software. Software must be written efficiently to minimize processor utilization. Main memory must be efficiently managed for staging data between the disks and the network. Multimedia files must be appropriately placed on the storage system. Disk accesses must be scheduled for optimal throughput and an admission control system must manage the number of streams that can be serviced concurrently.

This thesis presents the design, implementation, and analysis of a networked multimedia server called the Stony Brook Video Server. The server provides end-to-end smooth digital multimedia delivery from a server's I/O devices to each end user's display with real-time performance guarantees.

Specifications	Uncompressed	MPEG
Voice quality audio	64 Kbps	32 Kbps
CD quality audio (2 channels, 16 bit samples at 44.1 kHz)	1.4 Mbps	475 Kbps
MPEG-1 (352 x 240 x 30 frames/sec)	2.5 MBps	1.5 Mbps
MPEG-2, CCIR 601, Studio TV (720 x 480 x 30 frames/sec)	10.3 MBps	6 Mbps
HDTV quality video (1440 x 1152 x 30 frames/sec)	49.7 MBps	24 Mbps

Table 1: *Storage space requirements for digital multimedia data. Kbps = Kilobits per second, Mbps = Megabits per second, MBps = Megabytes per second.*

1.1 Types of Video Servers

Multimedia servers can be classified into three categories. First, the large-scale *video-on-demand server* is the digital replacement of the video rental store. It employs wide-area networks such as cable or telephone networks. In this type of system, video sequences are relatively long (on the order of hours), the number of simultaneous users is large (on

the order of thousands), and there is a high user tolerance of access latency (on the order of minutes). The storage servers for the interactive TV test trials conducted by several regional Bell operating companies belong to this category. These servers employ massive amounts of storage and processing power and are therefore very expensive. Second, the LAN-based *enterprise video server* is an NFS-style server that includes support for digital video files. The number of users to be serviced is fewer than a hundred and the access latency is expected to be on the order of seconds if there is enough system bandwidth to grant the access request. Commercial examples include Starlight's video server [Sta96] and Microsoft's Tiger server [B+96]. Lastly, the *specialized video server* serves niche markets, such as hotels and airplanes. In this case, the maximum access load is typically fixed and must be accommodated by the system. In addition, videos are played out at specific times so that many users can be serviced simultaneously.

1.2 Target Environment

The ultimate goal of the Stony Brook Video Server (*SBVS*) project is to build and analyze an *enterprise video server*. It is intended to support up to 100 users. The server uses off-the-shelf hardware, existing network infrastructures, and supports the playback of real-time multimedia streams. Off-the-shelf hardware is used to keep the costs of the server to a minimum. Commodity components are cheap and readily available. Existing network infrastructure, namely Ethernet, is also a high commodity item. This server will be able to be used in many present and future environments.

The first end-user application will allow students to view video-taped and digitized computer science lectures. Lectures will be videotaped using a standard VHS recording device. The lecture then will be compressed into MPEG format and stored on the video server for playback. MPEG was chosen because of its large acceptance as a video compression standard. Althouth the target application uses MPEG, the *SBVS* can support any multimedia compression standard. Once videos are stored, they are not edited. During the design process, decisions were made based on this kind of environment.

1.3 Dissertation Overview

Whereas a significant amount of research has been devoted to video servers, most published work is based on analytical or simulation methods. It is our view that more can be learned by the implementation and analysis of a working system. Some of the proposed algorithms that are shown to be analytcally superior to other algorithms cannot even be realized in a real system. Being experimental researchers, we build modular systems that work and then

evaluate them under realistic workloads. It is also extremely rewarding to see something that works and to have others use your system.

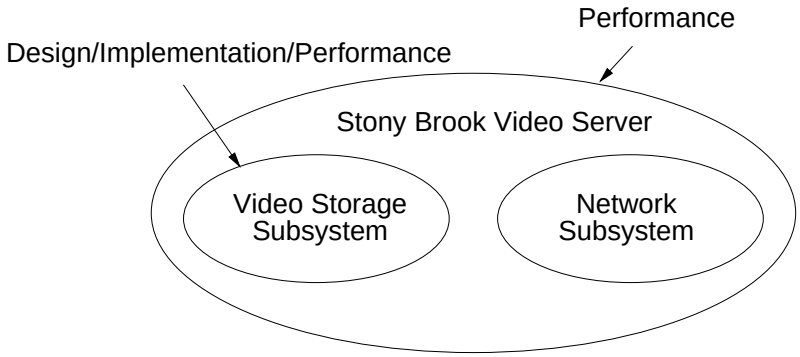

Figure 1: *Dissertation content overview.*

Figure 1 shows the major subsystems within the *SBVS*. It contains a video storage subsystem whose main responsibility is the storage and retrieval of video files on a software-driven disk array. During playback of video files, the video storage subsystem retrieves data and stores it in the main memory of the server. The network subsystem is responsible for transferring the video data from main memory to the clients over the network.

The thesis begins with a presentation of two of the underlying technologies that the *SBVS* uses: the SCSI protocol which allows peripheral components to communicate with the CPU, and the MPEG compression standard which defines how video and audio data is compressed and uncompressed.

Chapter 3 presents an overall introduction to the hardware and software components of the *SBVS*. This includes an overview of the network subsystem, *RETHER*, that is used by the *SBVS* to guarantee the delivery of multimedia data over standard Ethernet.

The following three chapters present the major body of this work: the design, implementation, measurement, and analysis of the video storage subsystem—the subsystem that is responsible for storing and retrieving multimedia data from an array of hard disks. The design and imlementation issues of buffer management, file placement, and disk scheduling are presented. In addition, the details of the storage system integration with *RETHER* is shown. The result of the integration is a working end-to-end video server. The guiding principle behind the design of the *SBVS* is the tight coupling of system components to improve the hardware utilization efficiency. The *SBVS* incorporates an integrated scheduling mechanism to guarantee both network and disk transport bandwidth, and a stream-by-stream disk scheduling algorithm that optimizes the effective disk bandwidth without scheduling disk requests on a cycle-by-cycle basis. It is believed that the *SBVS* is the first Ethernet-based video server to guarantee the real-time delivery of MPEG video data while allowing non-real-time transfers to exist on the same shared LAN. In addition to presenting performance

measurements of the video storage subsystem, performance measurements of the complete *SBVS* are also presented.

When a new user requests a new video to be delivered, the *admission control* system must determine if there are enough resources within the I/O system and network system to satisfy the request. Chapter 7 discusses and analyzes the video storage subsystem's admission control module. This module determines if there are enough resources within the storage server to accommodate a new request, while continuing to guarantee uninterrupted service to active requests. Most work pertaining to admission control has been analytical in nature. In this part of the work, a statistical admission control algorithm is implemented and compared to a proposed deterministic algorithm. A deterministic algorithm uses worst-case assumptions but guarantees that all clients are serviced in real-time. The statistical algorithm uses dynamic performance measurements and allows more streams to enter the system but does not provide 100% guarantees. The goal of the experiments is to determine the performance difference between the deterministic algorithm and various statitistical variants.

The last portion of this work, presented in Chapter 8, focuses on the VCR-functionality of the *SBVS*, in particular fast forward and rewind of MPEG streams. For the video server to be usable by the end-user, the system must provide VCR-like functions, i.e., pause, resume, fast forward/rewind, and slow motion. There have been several proposals for implementing these functions, yet there have been no published implementations. The goal of this portion of the project is to subjectively and objectively compare these proposals to the *SBVS* proposal and then implement the *SBVS* proposal. The main question to be explored in this part is: *How does the SBVS method compare to previously published proposals for implementing fast forward/rewind?*

Finally, a summary of the results and ideas for future work are presented in Chapter 9. Note that related work will be presented in each of the chapters to which it is appropriate.

Chapter 2

Preliminaries

2.1 SCSI

SCSI (Small Computer System Interface—pronounced *Scuzzy*) is the acronym for the best-known and most widely used ANSI (American National Standards Institute) interface. It's a standard for connecting peripherals to a computer. The second iteration of the standard, SCSI-2, is the most recent version of the SCSI command specification and allows for scanners, hard disk drives, CD-ROM players, tape drives, and many other devices to be connected to the I/O bus of a host computer.

One of the objectives of SCSI-2 is to move device-dependent intelligence out to the SCSI-2 devices. The command set definitions allow a sophisticated operating system to obtain all required initialization information from the attached SCSI-2 devices. The formalized sequence of requests identifies the type of attached SCSI-2 device, the characteristics of the device, and all the changeable parameters supported by the device. Further requests can determine the readiness of the device to operate, the types of media supported by the device, and all other pertinent system information. Those parameters not required by the operating system for operation, initialization, or system tuning are not exposed to the SCSI-2 interface, but are managed by the SCSI-2 device itself.

Figure 2 shows the architectural layout of a SCSI connection. There is a SCSI controller that communicates with both the CPU via the system I/O bus and with the SCSI devices via the SCSI bus. If the controller supports DMA, it can send and retrieve data to/from main memory with no interaction with the CPU. The SCSI-2 (Fast SCSI) can support synchronous transfers up to 10 megabytes per second (MBps). Fast-Wide SCSI can support transfers up to 20 MBps and Ultra SCSI can run at 40 MBps.

The SCSI interface uses logical rather than physical addressing for all data blocks. For direct-access devices such as hard disks, each logical unit may be interrogated to determine how many blocks it contains. A logical unit may coincide with all or part of a peripheral

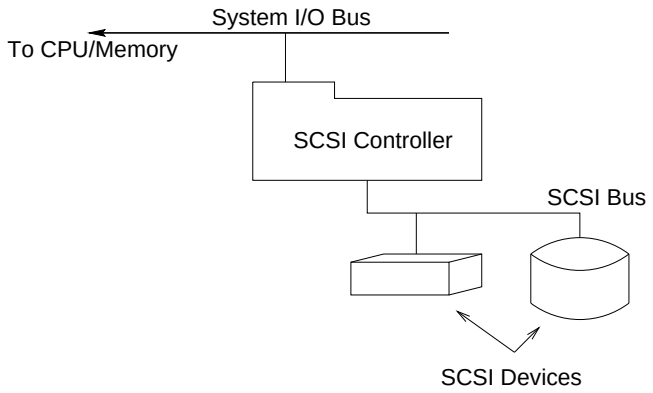

Figure 2: *Architectural layout of SCSI bus and devices.*

device. Thus, read and write requests sent to a disk include a logical block number rather than cylinder/track/sector. Translation from a logical block number to physical location is done by the device. Arbitration is needed to permit concurrent I/O operations. All SCSI devices are required to be capable of operating with the defined transfer protocol.

A SCSI command is executed by sending the command to the SCSI controller which in turn passes it to the SCSI device. For example, a disk read command contains the disk device number, the starting logical block to be accessed, the number of blocks to be read, and a memory location to place the data. The controller communicates with the appropriate device using the disk device number. It sends the request to the disk which executes the read. The retrieved data is sent back over the SCSI bus to the controller. The controller is then responsible for transferring the data from the SCSI bus to main memory via the I/O bus if DMA is supported. If the controller uses programmed I/O, the CPU is responsible for moving the data from the controller to main memory.

2.2 MPEG

MPEG (Moving Pictures Experimental Group—pronounced *Empeg*) is a standard for digital video and audio compression. The standard defines a compressed bitstream which implicitly defines a compressor and a decompresser.

There are several different types of MPEG bitstreams. MPEG video streams consist solely of video frames; MPEG audio contains audio only; and MPEG system streams contain both audio and video. There are currently two major MPEG standards. MPEG-1 defines bitstreams for low resolution video, 352 by 240 pixels by 30 frames per second, which is about the quality of VHS. The standard is optimized so that the average temporal bit rate of the compressed bitstream is about 1.5 megabits per second (Mbps). The MPEG-2 concept is similar to MPEG-1, but includes extensions to cover a wider range of applications. The

primary application target for MPEG-2 is the all-digital transmission of broadcast TV quality video at coded bitrates between 4 and 9 Mbps. The input to MPEG-2 is video recorded at 720 by 480 by 30 frames per second.

MPEG uses an inter-frame motion compensation method of video compression. The basic idea is to predict motion from frame to frame in the temporal direction, and then to use discrete cosine transforms to reduce the redundancy in the spatial domain.

An MPEG video stream consists of intra-coded frames (I), predictive frames (P), and bi-directional interpolated frames (B). A representative sequence of frames is shown in Figure 3. I frames are reference frames. They are self-contained and encoded independently of other frames, thus providing random access entry points into the video. P frames are encoded using motion estimation depending on a preceding I or P frame and receive a fairly high amount of compression. B frames are dependent on two "anchor" frames—the closest preceding I/P frame and a following I/P frame. These frames provide the highest amount of compression but require both a past and a future reference to be encoded. B frames are never used as references. Frames are grouped together (group of pictures or GOP) such that the first frame in the group is always an I frame. Thus, access to random parts of a video can begin at any GOP boundary.

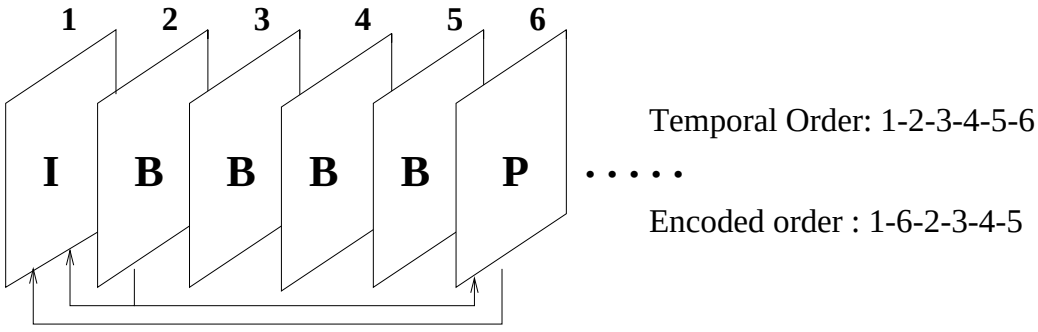

Figure 3: *Sample MPEG sequence showing inter-frame dependencies. B-frames depend on both the preceding I/P and following I/P frame while P frames depend on the preceding I/P frame. I frames have no inter-frame dependencies.*

Each of the frames is encoded on a macroblock basis, in which each macroblock consists of a 16-by-16 pixel block. In I frames, all of the macroblocks are intra-coded (I) while in P-frames, blocks could be either forward-dependent or intra-coded depending upon which scheme provides better compression. Similarly, in B-frames, blocks could be intra-coded, forward-dependent P, backward-dependent P, or interpolated B-macroblock (which is dependent upon both a forward and a backward block), depending on which results in the best compression. In blocks that are dependent on forward or backward reference blocks, the block is coded as the difference between the reference block and the block being compressed. Since P and B frames use inter-frame differences, they result in substantially smaller frames

than I frames and contribute significantly to the compression.

Both intra-coded blocks and difference blocks have high spatial redundancy. To reduce this redundancy, macroblocks are split into 8-by-8 pixel blocks and transformed from the spatial domain to the frequency domain with the Discrete Cosine Transform (DCT).

Next, the algorithm quantizes the frequency coefficients. Quantization is the process of approximating each frequency coefficient as one of a limited number of allowed values. The combination of DCT and quantization results in many of the frequency coefficients being zero, especially the coefficients for high spatial frequencies, which are not visually perceptible. By varying the quantization values when compressing, higher compression/lower quality or lower compression/higher quality can be achieved. Between the motion compensation and DCT processes, MPEG compression can yield compression ratios more than 200:1.

Lastly, to simplify buffering at the decoder, the MPEG standard requires that the frames be presented in a particular order. Specifically, a frame is presented to the decoder only after all the frames on which it is dependent are presented. Hence, the presentation order is different from the temporal order of video frames. For the example in Figure 3, the order of frames in the encoded stream would be 1-6-2-3-4-5.

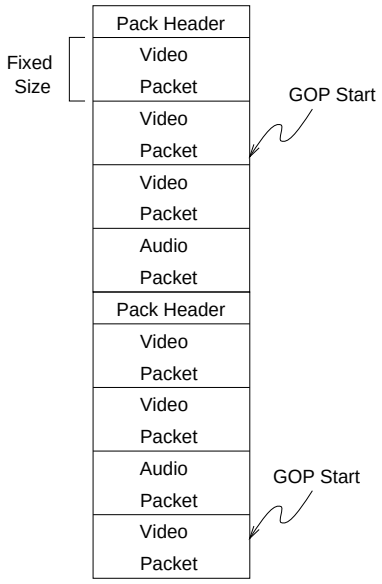

Figure 4: *Layout of an MPEG system stream.*

As previously mentioned, MPEG system streams consist of both audio and video. As shown in Figure 4, the system stream consists of packs and packets. A pack consists of a pack header and a fixed number of fixed size packets where each packet holds either video or audio data. The pack header contains information about each of the packets in the pack as well as information for synchronizing the video and audio. Since a packet has a fixed size, it is not associated with any particular piece of the video, thus a GOP and/or frame can begin

at any point within any packet.

Chapter 3

Stony Brook Video Server Overview

The Stony Brook Video Server (*SBVS*) is an *enterprise video server*. To minimize the costs of the system, it only uses off-the-shelf PC hardware components. A schematic of the supported system is shown in Figure 5. The server communicates with a set of user computers over standard (or Fast) Ethernet. The server and clients all share the network medium. This is in contrast to a switched type of network where each client has its own subnetwork, as in the Oracle Media Server [LOP94] or Microsoft Tiger [B+96] environments.

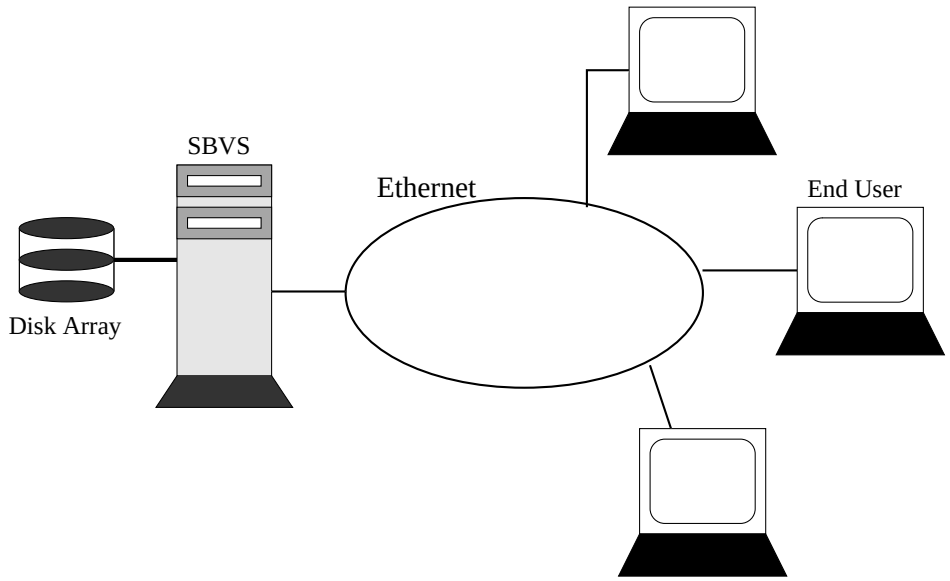

Figure 5: *Enterprise architecture*

3.1 Hardware Architecture

The first prototype, *SBVS-1*, was implemented on an Intel 486-66DX2 with 32 Megabytes of RAM and an EISA bus. It connects to five clients over a 10Mbps Ethernet. The server had a single Conner SCSI-2 disk, with 3-5 MBps transfer bandwidth, and an Adaptec 2742 EISA SCSI-2 controller. The SCSI bus transfers data at a rate of 10MBps.

The second prototype, *SBVS-2*, is based on an Intel Pentium 90MHz CPU with 32 Megabytes of RAM and a PCI bus. It uses a SCSI-2-based disk subsystem with three Adaptec PCI Fast SCSI-2 controllers and six Quantum Fireballer 1 Gigabyte disks. Table 2 shows the parameters of the Quantum disks. The Fast SCSI-2 bus transfers data at a rate of 10MBps. The network interface card is an SMC PCI Fast Ethernet 10/100, based on the DEC 21140 chip. The clients are based on the Intel Pentium 90MHz CPU with 16 Megabytes of RAM and PCI bus. They use either a SCSI-2/PCI- or IDE/ISA-based disk subsystem and SMC Fast Ethernet network card. In addition, each client uses an ISA-based Omni-Talisman card that performs hardware MPEG decompression with direct video output to the VGA adapter or NTSC compliant monitor. The network is a 100Mbps Fast Ethernet which connects the clients and the server and through the SMC Tiger Hub.

Formatted Capacity	1092 MB
Rotational Speed	5400 RPM
Zones per Surface	15
Sectors Inside Zone	88
Sectors Outside Zone	177
Avg Seek Time	12.0 ms
Track-to-Track Seek	3.1 ms
Maximum Seek	22 ms
Min Transfer Rate	5.4 MB/sec
Max Transfer Rate	10.5 MB/sec

Table 2: *Characteristics of Quantum Fireballer.*

3.2 Software Architecture

There are two main hardware/software subsystems in the *SBVS*. First, the network subsystem controls the underlying network hardware and is responsible for sending data from the main memory of the server, over the physical network, to the clients. The video storage subsystem (*VSS*) is responsible for video file placement and retrieval, stream admission control and VCR functionality. To maximize the efficiency of the network and video storage

subsystems, all controlling software was written from scratch inside the kernel. For example, rather than use expensive RAID hardware, a software driven disk-array was implemented.

3.2.1 Operating System

Both the servers and clients use the FreeBSD operating system, release 2.1. FreeBSD is based on BSD 4.4lite. Since the *SBVS* must have complete control over the system to provide real-time guarantees, the server code is implemented in the kernel. Thus, the *SBVS* only supports the retrieval and delivery of video data; it performs no other functions. A single system call places the *SBVS* into the video server mode and never returns. There are no other user processes which run that could affect the real-time nature of the system.[1]

3.2.2 Software Subsystems

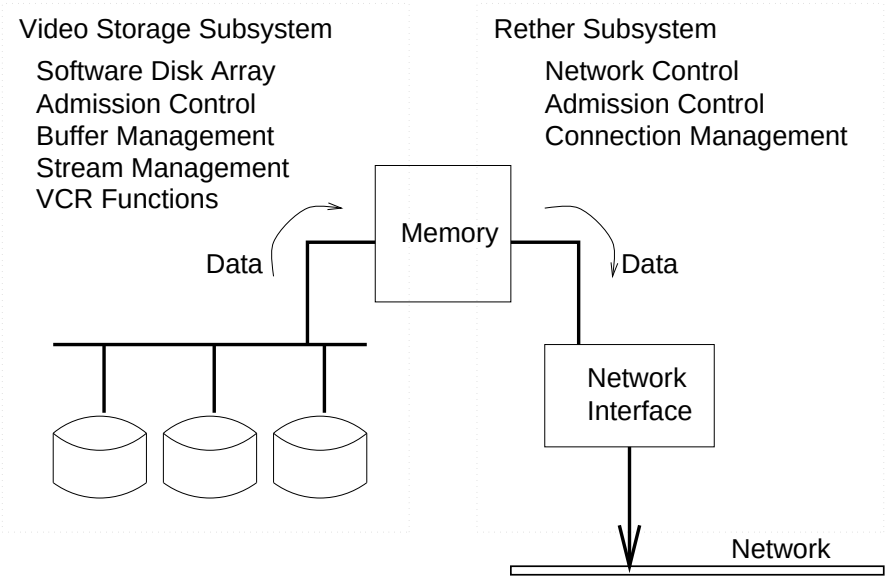

Figure 6: *System Overview of the SBVS.*

Figure 6 shows a schematic of the two major subsystems within the *SBVS*. The main task of the video storage subsystem is to schedule and move data from the disks in the disk-array to memory. The network subsystem's main task is to move data from main memory to the clients via the network interface. Both subsystems contain admission control software to guarantee that streams being entered into the system will not affect the real-time guarantees

[1] One trick we have learned to shorten development time is to write user-level code using copies of the kernel data structures. Kernel calls to devices are replaced by calls to a simulation library. Thus, hardware components are simulated. Our code, data structures, and algorithms can then be debugged first at user level and then once tested, moved into the kernel. In addition to shortening development time, we have a working simulation which directly models the kernel code.

of other active streams. Both subsystems also include stream management software that keeps track of the active streams in the system.

3.2.3 Client System

Clients are responsible for initiating the events that cause videos to be displayed, paused, and stopped. The application program interface (API) function calls for accessing the video server are

- *unsigned int videoBuffer(char *videoName):* This function call returns the size of the buffer needed by the client. This includes space for the client to pre-buffer data before initiating the display of the first frame. See Section 4.6.2 for a description of buffering at the client.

- *int videoPlay(char * videoName, int framesPerSec, int clientId, int portId):* This function call sends a message to the server to initiate the normal playback of the requested video. The *clientId* is a unique identifier returned by the function call *retherGetId*, and *portId* is the port to be used. If a stream is not already running on the given client and port, and can be admitted into the server system (returns 0), the client can display the frames of the video. If a stream is already playing on the client and port, the stream resumes normal playback at its recorded speed.

- *int videoDisplay(unsigned char *buffer):* This is a local function call which displays the data addressed by *buffer*. The buffer must first be allocated by the user using malloc. The size of the buffer is determined by using the function call *videoBuffer*.

- *videoPause(int portId):* This function call sends a message to the server to pause the playback of the stream at port *portId*. Any explicit or implicit stream reservations are held.

- *videoResume(int portId):* This function call sends a message to the server to resume the playback of the video of the stream at port *portId*. This is issued after a *videoPause*.

- *videoStop(int portId):* Causes the video to be stopped. All explicit or implicit reservations are canceled.

- *videoJump(int portId, int numSeconds):* This causes the video to jump forward or backward the specified amount of seconds. If *numSeconds* is less than 0, the video jumps backward, otherwise it jumps forward. The amount that is actually jumped is based on the playback rate of the video. For example, jumping forward 10 seconds while watching a video at a rate of 30fps causes the video to skip 300 frames. However,

if a user was watching the same video in slow motion at 20fps, a jump of 10 seconds only jumps forward 200 frames.

- *videoFwd(int portId):* This causes the video to be played in a fast-forward mode. Normal viewing is resumed by sending the videoPlay message.
- *videoRewind(int portId):* This causes the video to be played in a fast-reverse mode. Normal viewing is resumed by sending the videoPlay message.

3.3 Network System

For smooth playback of video over the network, the network subsystem must deliver data to the clients in real-time while allowing non-real-time traffic to coexist. The physical network used by the *SBVS* was chosen to be Ethernet. It is the most widely used network protocol and is extremely inexpensive. The Ethernet architecture, however, is inherently incapable of providing deterministic network access because of its contention-based medium access protocol. Although this protocol works well for non-real-time traffic under light load, it causes nondeterministic access delays to the network. This is usually unacceptable for transport of temporally constrained data such as video and audio. To overcome these deficiencies, the Real-Time Ethernet Protocol has been developed.

RETHER allows network applications to reserve bandwidth on an existing Ethernet without any hardware modifications—essentially turning Ethernet into a medium capable of supporting real-time traffic. The central features of *RETHER* are the following: First, *RETHER* is based on a software-controlled token-passing scheme to enforce collision-free operation on an Ethernet link. This software is part of the Ethernet driver and therefore is completely transparent to high-level networking software such as TCP/IP. Second, the performance of *RETHER* is identical to the original CSMA/CD protocol when there are no real-time connections. This is because the token-passing scheme of *RETHER* is turned on only when real-time connections exist, and is turned off when the last real-time connection terminates. Third, *RETHER* allows bandwidth reservations from user-level applications and guarantees them throughout the lifetime of the applications. The admission control algorithm used in *RETHER* is completely distributed across the network, i.e., there is no centralized admission controller.

Intuitively, time is divided into token rotation cycles, the length of which is a tunable parameter. In the *SBVS* prototype, 33 milliseconds (ms) was used, corresponding to a video frame rate of 30 frames per second (fps). After a node successfully makes a real-time *send* connection, the token arrives at that node every 33 ms until the connection is terminated. Bandwidth reservation is made in terms of the token holding time every time the token

arrives at a node. Thus, the bandwidth reservation is constant over the duration of the connection. For example, assuming that an MPEG-1 stream S has a bandwidth requirement of 1.5 Mbps, the server must send $\frac{1.5}{30} = .05$ Mbits every 33ms. The time to send .05 Mbits on a 100Mbps Ethernet is 500 microseconds (μs). Thus, for stream S, the server reserves $500\mu s$ of bandwidth for each 33 ms cycle.

The *VSS* has been integrated with *RETHER* to provide the real-time guaranteed delivery of multimedia data from the disk to the client. The full details of the *RETHER* protocol and its implementation can be found in [VC94].

3.4 Multimedia Systems Overviews

In [LV94], the technological considerations for designing a large-scale, distributed, interactive multimedia system are surveyed. The core problems are identified and proposed solutions are examined. The commercial market for video servers is discussed in [Tri95]. An overview of compression techniques and software for multimedia systems is presented in [Fur94]. [scfdv92] discusses the various challenges that are faced in delivering interactive digital video services to desktop computers in local area environments.

Chapter 4

Design of the Video Storage Subsystem

When building a multimedia storage server there are many design choices that must be made. These choices include the style of data delivery; the size, striping and placement of media files; disk access algorithms; and buffer management schemes. Several of these design choices, the style of data delivery, and the size of a media block, have implications in other components within the server. This chapter discusses these design choices and presents the methods adopted by the *SBVS*.

4.1 Push vs. Pull Data Delivery

The first design decision in building a multimedia storage server is the style of data delivery, i.e., client- or server-initiated. The server can use the *client-pull* method—the client constantly requests blocks of data that must be delivered by a given time and the server delivers them. Client software must be aware of the location of individual blocks of the media file and be able to determine exactly when a block is needed to guarantee continuity in the media stream. In this system, for example, the client would pause a stream by simply stopping the requests for more data. The main advantage of *client-pull* is that the client can easily speed up or slow down the rate of delivery by changing the time interval of its requests. In this type of environment, the server usually supports a general purpose real-time operating system that works with many types of real-time data. The main disadvantage is that the server does not have a long-term view of the requests' access patterns to optimize its retrieval. It simply responds to individual requests and tries its best to accommodate them. In the *server-push* system, the client makes an initial request to the server to start the delivery of data. The server continues to deliver the data until the end of the stream is

reached or the client requests some type of VCR function. The server has complete control of how and when accesses to the storage media are executed. If the client wants to pause, stop, use VCR-functions, etc., the client must send a message to the server and the server takes the appropriate action.

The *VSS* uses the *server-push* style of data delivery. Clients simply send high-level requests such as play, pause, etc., and the server is completely responsible for providing the data in real-time. By placing as much as possible of the intelligence of the system at the server side, the hardware/software requirements of the clients are simplified.

A generic technique to guarantee I/O bandwidth in the *server-push* system is called *cyclic scheduling*. Time is divided into *cycles* of length P_{io}. This is a tunable parameter which can be specified when the *VSS* starts operation. The consequences of changing the cycle time is discussed in Section 6.3.3. While users are consuming data over the network during cycle i, the *VSS* is inserting data into memory with the data needed for cycle $i + 1$. (Each active media stream is allocated a portion of the main memory buffer space.) During each cycle, all media streams are serviced. Whenever the i-th media stream is serviced, the disk subsystem transfers $P_{io} * C_i$ bytes of media data to the buffer, where C_i is the consumption rate of the i-th media stream in bytes/second. Depending on the type of retrieval algorithm, the consumption rate could be the average rate over the entire length of the media file or the rate of some smaller interval. This is examined more closely in the next section.

4.2 Media File Storage and Retrieval

The most important component of the *VSS* is the software that drives the hard disks where the media files are stored. The goal, when designing the software, is to optimize the retrieval of media files during normal playback so that the maximum number of users can be serviced. The technical issues in designing a software storage server are as follows: the size of each media block; placement of media blocks on a disk and within the disk array; the scheduling of disk accesses; and admission control to determine if streams can be admitted without violating any real-time guarantees. Once applications successfully reserve their required bandwidth, the system guarantees it throughout the entire playback session.

The following sections discuss these design issues, the alternatives of each issue, and the methods used in the *VSS*. Admission control will be discussed in Chapter 7. At the end of the chapter, a discussion of previous work on these design issues is presented.

4.2.1 Media Block Size and Retrieval

The storage server divides media files into blocks when storing them on disk. Each such data block may occupy several physical disk sectors or even disk tracks. Whereas data in a traditional file system is broken up into fixed size blocks, one can store media data blocks using a Constant Data Length (CDL) block or Constant Time Length (CTL) block. (It is assumed that media files are compressed into a variable bit rate (VBR) data stream.[1]) In CDL, each media block of a video file has the same size. In CTL, each media block corresponds to the same playback unit—for example, 1 second's worth of data. Since VBR streams are used, each CTL block may have a different size. One advantage of using CDL blocks is that the block size can be disk-sector-aligned. When retrieving data, requests are made specifying a number of disk sectors to retrieve. When using CTL blocks, extra code is needed to handle non-sector-aligned blocks.

Now, consider the amount of data that needs to be retrieved while servicing some stream. During each cycle, either a constant number of bytes can be retrieved, i.e., constant number of CDL blocks; or a variable number of bytes can be retrieved, i.e., variable number of CDL blocks, or constant/variable number of CTL blocks. If a constant rate is used, the sum of data retrieved for any N streams is also constant. (This does not imply that every stream must have the same rate. Different streams may have different, but constant rates. When a stream is added or deleted, however, the sum of the rates will change.) Again, the restriction is that the sum of the rates is constant for any N streams. Because of the characteristics of VBR streams, however, buffering is needed to smooth out the variations in the playback rate while reading data at a constant rate. (See Section 4.6.2 for a discussion of pre-buffering data.) If a variable number of bytes is retrieved from every stream during each cycle, then the sum of the rates is therefore also variable; thus it may be possible that this sum exceeds the disk read capacity, resulting in missed reads, or exceeds the network bandwidth, resulting in delayed frames.

The choice of a constant bandwidth per cycle or variable bandwidth per cycle has many implications within the storage server. It affects buffer management. If a constant rate is used, the amount of buffering is constant for each stream and easily manageable. On the other hand, pre-buffering of data must occur at the client so it never starves for data. Pre-buffering will increase the startup latency time. Because of the pre-buffering needed at the client, the CDL scheme has higher buffer requirements than the CTL scheme [CZ94]. The characteristics of MPEG-1 video streams, however, make the extra buffer needed very small. If variable rates are used, the streams may request a buffer equal to the maximum rate of the stream, resulting in wasted memory at the server when the maximum rate is not being

[1] For example, a video may be stored at 30 frames/second. Each frame, however, may be of a different size, resulting in a variable bit rate over the duration of the video.

serviced. To save space, a more complex buffer management scheme is needed so streams can share buffers. The choice of constant or variable rates also affects admission control. When using constant rates, if a new stream requests admission, the admission control module will know exactly how much the rate will increase if the new stream is added. This rate will be constant until another stream is added or deleted. If a variable rate is used, the rate will change every cycle. The admission control module must determine if, during any future cycle, the sum of the rates exceeds the disk or network capacity. Lastly, the network must be able to support whichever type of bandwidth rate is used.

When designing the *VSS*, the *RETHER* system had already been designed and implemented. As mentioned previously, *RETHER* supports a constant bandwidth rate. Each active stream on the network reserves a constant bandwidth and is held until the stream terminates. Because of its simplicity and its ease of integration with the *RETHER* system, the *VSS* uses a CDL block with constant delivery rate.

4.2.2 File Placement

Once the media block size is chosen, the allocation of media blocks to each individual disk and the disks in the disk array must be determined. The goal of the allocation scheme is to optimize normal playback since most users will be viewing videos in this mode. The *VSS* uses a contiguous allocation scheme on each disk. Each file is placed sequentially on the disk starting at the outer edge of the disk and working inward. Thus, all blocks of each file are contiguously placed on each disk. The main advantage of contiguous allocation is its simplicity. The only metadata needed for a media file is the size of the file, the size of each CDL block and its starting location. Random access into the file can be determined by the size and starting point of the first CDL block. If a random placement scheme is used, metadata is needed to point to each media block. In addition, assume streams A and B are being serviced. If all blocks of A occur before the blocks of B on the disk, then blocks of stream A are always accessed before blocks of stream B using the *CScan* algorithm. (See the next section.) The scheduling of disk access occurs only when streams are admitted. On the other hand, contiguous allocation may result in larger fragmentation of the disk compared to random placement. The *SBVS*, however, is primarily read-only. New media files are added to the disks infrequently, usually once a day, during non-active periods. When new files are added to the disks, the disks are reorganized to keep fragmentation at a minimum and to optimize the placement of the files to take advantage of user viewing patterns. More heavily viewed videos are placed toward the outside of the disk where the disk transfer rate is the highest.[2]

[2]All modern disks use a multi-zone organization where there are more sectors per track on the outside of the disk. The number of sectors per track decreases toward the inside of the disk.

Once the single disk allocation scheme is determined, the allocation scheme across the disks in the disk array must be determined. Media blocks from a file may reside on a single disk, may reside on a subset of the disks, or be striped across all of the disks in the array. In the *VSS*, media blocks are striped across all disks in the array. If the number of disks in the array is D, the stripe size S is chosen in such a way that $D * S = P_{io} * C_i$. (See Section 5.5 for retrieving files whose stripe size is not a multiple of the disk sector size.) During each retrieval cycle, one media block is accessed for each stream. By striping each media block across all disks, perfect load balancing is achieved with no software overhead. If the blocks are stored on a single disk only, heavily viewed files may cause disk bottlenecks. On the other hand, by completely striping each file across all of the disks in the array, if a single disk fails, all files will be missing blocks and the system will be unusable. (Although fault-tolerance is important, it is beyond the scope this work and left for future research.)

Figure 7 shows the placement of media blocks from a single file on two disks as well as how they would be placed in memory. The figure shows a single video file whose media block size is 8 disk sectors. Each media block has 4 sectors on disk 0 and 4 sectors on disk 1. When retrieving data during cycle i, sectors 1-4 are retrieved from disk 0 and sectors 5-8 are retrieved from disk 1. During cycle $i + 1$ sectors 9-12 are retrieved from disk 0 and sectors 13-16 are retrieved from disk 1. Note that the sector numbers in the figure are logical sectors. The logical sectors map to physical sectors on each disk.

4.2.2.1 Disk Scheduling

Again, time in the *VSS* is divided into fixed length cycles. During each cycle, one media block is retrieved for each active stream, where each media block is striped across all of the disks in the disk array. To maximize the number of streams that can be serviced in one cycle, disk requests must be scheduled so that seek overhead is minimized. Seek overhead is the only variable that a system can try to optimize when reading data. Rotational latency is a random variable which cannot be optimized and the transfer of data from the disk medium over interconnection buses is a constant based on the specific hardware used.

Traditionally, disk scheduling algorithms (e.g., first-come, first-served; elevator scan; shortest seek time first) have been employed by servers to reduce the seek time, to achieve a high throughput, and to provide fair access to each client. The addition of real-time constraints, however, makes direct application of traditional disk scheduling inappropriate for multimedia servers.

In traditional network file servers, disk read requests are not known in advance and arrive randomly. On the other hand, in a *server-push* multimedia server, the disk location for all read requests is known at the beginning of each cycle. Thus, to improve disk utilization the *CScan* algorithm is used. The set of block requests is first sorted according to its

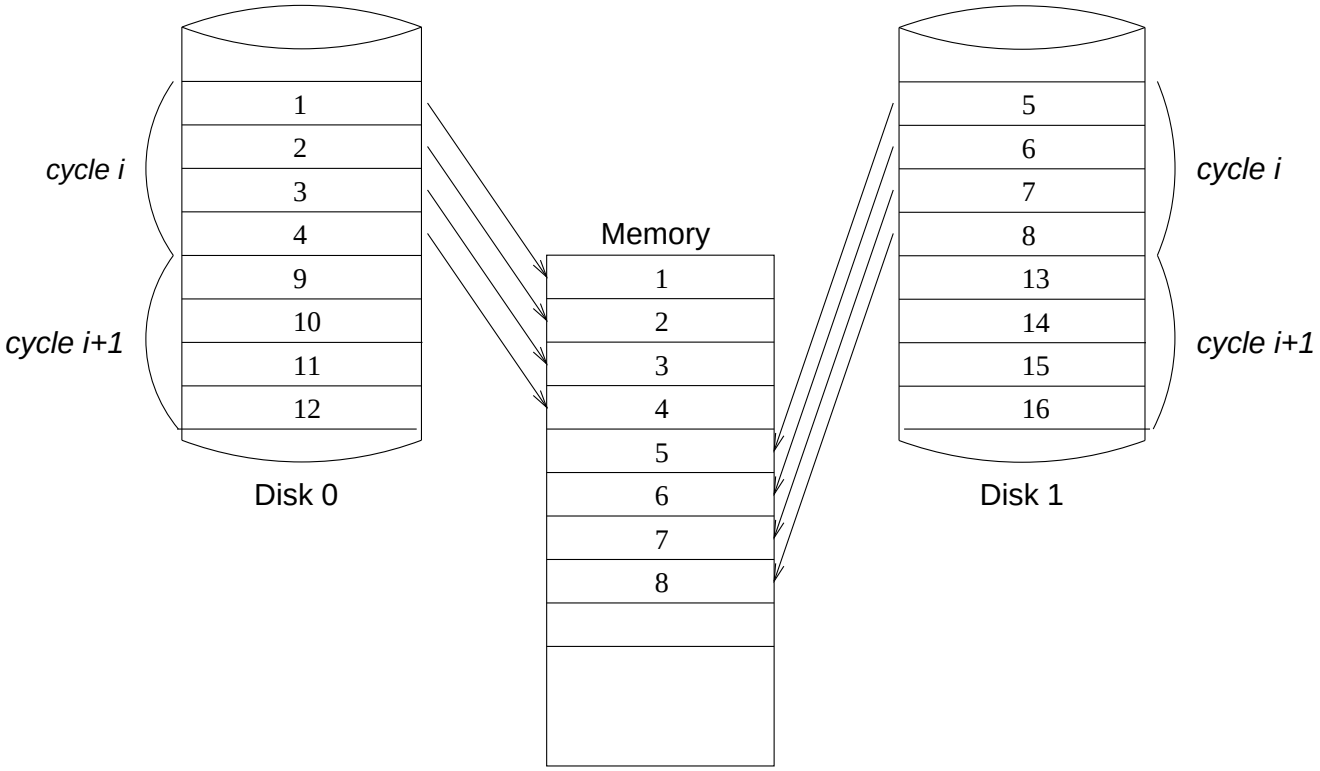

Figure 7: *Placement of video data on the disk vs. memory.*

cylinder coordinates, and then serviced within each cycle based on the sorted order. Because access requests are serviced while the disk arm is moving in one direction, unnecessary seek overheads are avoided.

Media streams consist of a sequence of access requests, so that disk scheduling can be done on a request-by-request or stream-by-stream basis. In stream-by-stream scheduling, access requests associated with a media stream are always serviced at a fixed slot within a cycle throughout the entire playback session, unless explicitly rescheduled. Rescheduling is performed only when streams enter or leave the system. In request-by-request scheduling, at the beginning of each cycle, the system determines the service order of the set of requests associated with the active video streams.

The *VSS* disk scheduling scheme gets the best of both the FIFO-based stream-by-stream scheduling and the request-by-request scheduling algorithms. Like the FIFO-based stream-by-stream scheduling, scheduling decisions only have to be made once per stream. Like request-by-request scheduling, the seek overhead is minimized because streams are serviced within a cycle according to their physical disk positions, not in the FIFO order. The main problem associated with stream-by-stream scheduling, however, is that non-video I/O requests must be handled separately. But since the *VSS* is a dedicated video server, this problem is not an issue.

4.3 Video Storage Systems Related Work

This section covers the previous works related to the digital video server design issues that have been discussed previously, including the placement of files on storage media, disk scheduling algorithms, buffer management and other storage issues. Although numerous proposals have been made on various aspects of these issues, there has been very little work that actually provides concrete implementation descriptions, let alone the results of a detailed performance evaluation of an operational video server. Most, if not all, of the previous work related to video servers has been analytical in nature. The following subsections discuss the related work in each of these areas and compares it to the implementation in the *SBVS*. [GVKR95, Mou96] are good survey papers where each of these design issues is discussed.

4.3.1 Disk Storage Systems

Because of the high bandwidth needed from the I/O system, multimedia servers employ some kind of disk array subsystem. The retrieval of data blocks for traditional file systems using disk arrays has been extensively reported in the literature [CP90, LK93, LKB87, GMS86, PGK88]. Whereas data blocks in traditional file systems are a Constant Data Length (CDL), multimedia data blocks can either be CDL or Constant Time Length (CTL). [CZ94, VRG95] discuss the advantages and disadvantages of storing multimedia data blocks using a CDL block or CTL block. It is shown that CTL blocks have lower system buffer requirements than CDL, but suffer from fragmentation during video editing. For the CDL block, [VRG95] presents an analytical model for determining an optimal range of sizes for a given set of configuration parameters. Simulation results that show the effects of changing the size of a CDL block are also presented. For CTL blocks, a method is described for distributing the workload amongst all the disks in the disk array. In [FJS95] a hybrid CDL/CTL scheme is proposed. Rather than use a CDL block that is based on the average bit rate of the entire video, the video is logically split into clips where the CDL block size is based on the average bit rate of each clip. At one extreme, if the clip is the entire video, the hybrid scheme is equivalent to the traditional CDL scheme. If the clip is small, on the order of a second, the hybrid scheme is equivalent to the traditional CTL scheme. Because of its simplicity, and its ease of integration with the RETHER system, the *SBVS* uses a CDL scheme based on the average bit rate of the entire video. Although the CDL scheme has higher buffer requirements than the CTL scheme, the characteristics of the MPEG video stream make the extra buffer needed very small. See Section 4.6.2 for a more detailed description.

The media blocks belonging to a file may be stored contiguously or scattered about the storage device. In [PV93], the *constrained placement* approach is a hybrid scheme where blocks may be scattered but bounded to some maximum distance. In [Bir95], the blocks

are scattered so that half of the file resides on the outside half of the disk and half of the file resides on the inside half of the disk. Since, the transfer times of disk vary depending on whether the data is on the outside partition or the inside partition, this scheme makes a disk's throughput become independent of the viewer's choices. In [BMC94], a constrained approach is used where the blocks are organized so the disk head traverses starting from the outer cylinder of the disk to the inner cylinder during one cycle and then traverses from the inner cylinder to the outer cylinder. Since the *SBVS* is a read-only system with additions and deletions of files occurring infrequently (once/twice a day), it uses the simpler, contiguous layout scheme of block placement.

Schemes for striping data using course-grain striping and fine-grain striping are examined in [ORS96]. For each scheme, the optimal amount of data to be retrieved for a video during each disk access is computed so that the number of concurrent streams supported by the server is maximized. [Mou94] analyzes the performance of using a RAID-3 array for delivery of video services.

In [PBC95], the authors compare two media block placement schemes where the size of a block is a group of frames (CTL). In the *balanced placement* approach, each media block is striped across all disks in the array and every disk is accessed for every stream every cycle. In the *periodic placement*, consecutive media blocks are placed on consecutive disks in a round-robin fashion. Each media block is retrieved from a single disk (compared to multiple disks in the balanced placement scheme). The balanced approach is shown to have lower startup latency since a stream can begin at any time. On the other hand, when using a large number of disks, the amount of data retrieved from each disk is small, lowering disk efficiency, i.e., the ratio of disk overhead to transfer time is high. The periodic approach is shown to have high startup latency, since accesses must be staggered so that the no disks become a bottleneck. In addition, to maximize the throughput, it is necessary to ensure that in each round the retrieval load is balanced across the disks. Because of the larger amount of data retrieved, disk utilization is high.

This same idea is discussed in more detail in [BGMJ94, CBR95, TPBG93]. Since striping across many disks lowers the disk utilization, in the *staggered striping* scheme, disks are grouped into clusters and each media block is striped across the disks in a single cluster. Performance evaluations by simulation discuss the impact of changing configuration parameters of staggered striping on the system performance.

Because of its simplicity and automatic load balancing, and since the *SBVS* uses a small number of disks, the *SBVS* uses the balanced approach where every disk is accessed for each stream during each round.

[CZ94] presents strategies for the placement of scalable video where the video is broken into several different resolutions. For example, a video bitstream may be comprised of low,

medium, and high resolution subset streams. For the user to view the low resolution stream, only the low resolution stream needs to be delivered. For the user to view the high resolution stream, the low, medium, and high resolution streams are delivered and merged.

In [CKLV95], a mechanism is described for the storage and retrieval of MPEG-encoded video from a single disk storage system. The scheme balances the need for the reliable delivery of MPEG frames with the desire to support the largest number of sessions. The approach reorganizes the MPEG-encoded video stream based on the relative importance of the frames and maps them to the storage device. The reorganization reduces the impact of frames lost due to missed deadlines.

Rather than stripe data across disks in the same system, [BP95, BB95] present methods where data is striped across multiple systems.

Traditional disk scheduling algorithms such as first-come, first-served (FCFS), and shortest seek time first (SSTF) are examined in [TP72]. There have been several works that have addressed this problem within the framework of a multimedia server. The simplist of the techniques is the *round-robin* algorithm, in which clients are serviced in the same order during each round. Thus, scheduling decisions can be made when streams are admitted or deleted, rather than on a cycle-by-cycle basis. However, the major drawback of round robin scheduling is that it does not exploit the relative positions of the media blocks being retrieved during a round [AOG92].

[NR93] compares three other popular scheduling algorithms. In the earliest deadline first (*EDF*) algorithm, deadlines are associated with each disk retrieval and serviced in time order. However, *EDF* does not minimize seek-time overheads and can result in poor disk utilization. (In [LL95], low-level algorithms are presented for an *EDF* scheduling policy.) The *CScan* algorithm is a scan type of disk-scheduling algorithm. The disk head scans for requests in one direction, from the outermost to the innermost track, serving requests in the order of its scan direction. In this policy, seek overheads are minimized. However, the relative order for servicing clients is based solely on the placement of blocks being retrieved. It is possible for a client to receive service at the beginning of one round and then at the end of the next round if data is randomly placed on the disk. In this case, *CScan* needs enough buffer space to satisfy consumption during two rounds. *Scan-EDF* is a hybrid scheduling algorithm that provides both seek optimization and earliest deadline first service. Requests are normally serviced in *EDF* order. If several requests have the same deadline, they are serviced according to their track locations on the disk. The performance evaluation showed that *CScan* could support the most real-time streams because of its goal of seek optimization. When non-real-time requests were interspersed with real-time requests, *Scan-EDF* supports the most streams.

Like *Scan-EDF*, the Grouped Sweeping Scheme (*GSS*) [Gem93, PY92] is a hybrid algorithm. This algorithm partitions each round into a number of groups. Each client is assigned to a certain group, and the groups are serviced in a fixed order in each round. For each group, the *CScan* algorithm is used. If all clients are assigned to the same group, the *GSS* is reduced to the *CScan* algorithm, while if only one client is assigned to a group, the *GSS* is reduced to the *round-robin* algorithm. By optimally deriving the number of groups, the server can balance the reduction of round length against the number of rounds between successive service.

In [SGM94], two different disk array environments were compared in a video server environment. In the *independent access array*, each disk has an independent rotational and head position. In the *parallel access array*, the rotation and head position are synchronized so that they move in unison. Results showed that the independent array performed better and had lower buffer requirements than the parallel array.

An analytical framework for disk scheduling algorithms is presented in [ORS95]. Although the algorithms are for single disk systems, they include both real-time and non real-time requests being serviced concurrently. Each of the algorithms is accompanied by an admission control scheme to restrict the number of concurrent requests being serviced at any given time.

The *SBVS* gets the best of both the *round-robin* and *CScan* algorithms. Since each video file is stored contiguously on the disk (see previous section), like the *round-robin* algorithm, scheduling decisions are made when the stream is admitted. In addition, the *CScan* algorithm is used during each round to minimize the seek overhead.

[HLL+15] presents an excellent performance evaluation of a mass storage system for a large-scale, video-on-demand server. The server has 31 SCSI channels, each connected to a RAID-3 disk array, 20 MIPS processors, and 768 MB of main memory. Experimental results show that the machine can support about 360 concurrent streams.

In all of the video-on-demand systems presented previously, users can start to display videos at random times. [Chi95, OBRS95] present a method for distributing video where the server supports a fixed number of streams where each stream is automatically started every n units of time, where n is based on the disk bandwidth. For example, assume a movie is an hour long and the disk subsystem can support a maximum of 30 concurrent streams. At any given time, there are 30 active streams where the streams differ by exactly 2 minutes (60/30). Users cannot start viewing a video until one of the streams starts retrieving data from the beginning of the movie, i.e., every 2 minutes. Moving around portions of the movie is accomplished by simply switching streams.

In all of the previous works, the video server is assumed to be of the *server-push* type. The client issues a request to the server to begin the delivery of a video stream. The server

continues to deliver data until the video completes or the user requests that the video be stopped. In [DS94, LS93], disk scheduling in a *client-pull* multimedia server is presented. In a *client-pull* server, clients are responsible for making individual requests for each media block.

4.3.2 Buffer Management

[TPBG93] discusses several methods for allocating memory to the active streams. When using the *CScan* algorithm and a scattered file placement, the order in which streams are accessed may change between rounds. A stream serviced at the beginning of one cycle may be serviced at the end of the next cycle. In this case, the *double buffering* scheme is used. Two buffers are allocated to each stream; while one buffer is being emptied, the other buffer is filled. The size of a buffer is the amount of data that is retrieved in a single cycle. This amount of memory can be decreased by sharing memory between streams. As data is emptied out of a buffer during the beginning of a cycle, data can be read into that same buffer later in the cycle. The paper shows that the memory requirement for this scheme is half of the amount needed for the double buffer scheme, but incurs increased complexity. [Mou96, LS93, Gem93, TPBG93, PY92] also discuss optimizations that reduce the amount of buffering by appropriately scheduling disk accesses.

Because of the way the RETHER system is integrated with the I/O subsystem, the *SBVS* adopts the simple double buffering algorithm. In addition, the maximum amount of physical memory required for 45 MPEG-1 video streams is 24MB based on the double buffering scheme. The cost of 32MB of DRAM is only about $200, quite low compared the cost of 45 client computers.

4.3.3 Miscelleneous Related Work

[Mou96] discusses the assignment of movies to the disk array to balance the load and maximize the throughput. Since the *SBVS* uses a balanced placement scheme with contiguous layout, it can only assign movies to different sections of the disk. Since the outer cylinders of the disks have higher transmission rates than the inner cylinders, frequently played videos are assigned to the outer cyclinders.

[Has93] presents a description of a continuous-media file server, but no performance results of an end-to-end system were available. [HK94] describes a performance analysis that compares two different I/O architectures. In the peer-to-peer mode, data flows directly from the I/O bus to the network adapter via the I/O bus, bypassing the main system bus and the system's main memory. This architecture was shown to perform better than the conventional I/O system where data flows from the I/O bus into main memory via the system

bus and back over the system bus to the network adapter.

A simulation study of a video-on-demand system is presented in [FD95]. The simulations determine the efect of varying a wide variety of parameters and design alternatives. The paper demonstrates the benefits of striping videos across many disks; the performance of different disk scheduling algorithms are compared; and buffer pool page replacement algorithms that minimize memory requirements are presented.

In [JSCB95], the architecture used for the video server is a parallel computer that has multiple server nodes connected by a high-speed interconnect. Interface nodes buffer data before transmitting it to the network. The paper investigates different scheduling algorithms to improve system throughput and response time.

To store large numbers of movies, the cost is very high to store the all of the movies on hard disks. The storage of some movies on slow, cheap, tertiary devices such as tape libraries, is investigated in [TKS94, DT94, FR94].

4.3.4 Summary of Related Work

Table 3 summarizes the various alternatives for multimedia server design issues and includes the method employed by the *SBVS*.

Design Issue	Alternatives	Used in *SBVS*
Delivery Style	Server-Push Client-Pull	Server-Push
Media Block Storage Size	Fixed (CDL) [CZ94] Variable (CTL)	Fixed (CDL)
Media Block Placement	Random Constrained [Bir95, PV93] Contiguous	Contiguous
Stripe Placement	All Disks [PBC95] Subset of Disks [BGMJ94, CBR95, TPBG93]	All Disks
Disk Scheduling	Round Robin [AH91] Earliest deadline first [NR93] CScan Scan/EDF Grouped Sweeping Scheme [Gem93, PY92]	CScan/Round Robin
Buffer Management	Single Buffering [TPBG93] Double Buffering	Double Buffering

Table 3: *Alternative methods for various design issues.*

4.4 Multi-Resolution Files

Typical video streams are encoded using compression parameters that affect the quality of the displayed video. These parameters include temporal resolution (frames per second) and spatial resolution (pixel height by pixel width). In the MPEG standard, other parameters include quantization levels which are used to increase or decrease the quality. By using large quantization steps, high frequency information is lost, thus lowering quality. The playback rate (in bytes per second), however, of the encoded video is proportionate to its quality. The better the quality, the higher the playback rate, and vice versa.

The basic observation behind multi-resolution video coding is that in a computing environment not all users will be viewing videos using their full resolution. Some users often open a portion of their display as a window for video playback; rarely will the window actually occupy the entire screen. Other users may be using software decoders that are unable to decompress a full resolution stream while users with high resolution displays and hardware decoders will want to retrieve and display the finest resolution of the video. During periods of high resource usage, the system may only be able to accommodate low resolution videos. If the video storage system only stores a single high-resolution file, users playing back at a lower resolution waste both storage and network resources.

Thus, in a general purpose multimedia system, it is advantageous if media files can be delivered using multiple resolutions/playback rates. The basic idea is to store multiple resolutions of a video sequence so that only the minimum amount of information needs to be delivered to the client side, depending on the requirements and capabilities of the client and the resource usage within the server and network.

Multi-resolution video coding does not necessarily imply additional storage overheads. For instance, the scalable modes in MPEG-2 allows multiple resolutions without extra storage requirements. Bitstreams can be partitioned into low resolution and high resolution components. Users wishing to view only the low resolution can be sent the low resolution component only. Users wishing to view the high resolution stream are sent both the low resolution and high resolution streams which are combined to produce the high quality output.

Although the scalable modes have been incorporated into the MPEG-2 standard, at the time of this writing, there are still no encoders nor decoders that actually support the scalable mode bitstreams. Thus, it was decided to support multiple resolution files in the *SBVS* by using separately encoded files. The user can request the appropriate video file that corresponds to its playback resolution. In addition, if the server cannot accommodate the request for a high resolution stream it may instead deliver a lower resolution stream.

4.5 VCR Functionality

In addition to supporting normal and multi-resolution playback, the multimedia server should also support VCR-like functions, i.e., pause, resume, fast forward and fast backward. Also, the random access capabilities of hard disk storage also allow for users to jump around an active stream. For example, a user can jump forward five minutes within the stream. In a VCR, the tape must be scanned. On a hard disk, however, the stream can be immediately updated so the next retrieval corresponds to the data that is five minutes ahead of the current data. By creating metadata that corresponds to various media blocks, streams can be randomly accessed. The implementation of the functions pause, stop, resume and jump forward/backward are straight forward and the exact details are covered in the next chapter. On the other hand, there has been very little published work dealing with actual implementations for supporting fast forward/rewind functionality for MPEG streams. Several papers propose methods for implementing fast forward but do not adequately deal with fast rewind. This subject will be examined in more detail in Chapter 8.

The *VSS* supports the fast forward and fast rewind of MPEG streams by storing a separate file that is specifically encoded to support both fast forward and rewind. In addition, this file can be used to support multi-resolution viewing as described in the previous section.

As an example, let the separately encoded file used for multi-resolution and fast forward/rewind be called the *ff-file*, and let the original playback file be called the *pb-file*. The *ff-file* is encoded at both a lower temporal resolution and lower quality than the *pb-file*. Therefore, the space requirements and average bit rate of the *ff-file* will be several times lower than the *pb-file*. Naturally, the main drawback to this scheme is the extra storage needed for storing the *ff-file*. For example, consider video V, whose *pb-file* was encoded at 30 frames per second with an average bit rate of 1.5Mbps. Let the *ff-file* be the separately encoded, lower resolution file, encoded at 10 frames per second and with an average bit rate of only .3Mbps. This file is encoded by skipping two of every three frames in the original video and using lower quality parameters. To support multi-resolution video, V can be shown using the *pb-file* at 1.5Mbps or can be show using the *ff-file* at .3Mbps. To support fast forward, the *ff-file* is shown at the same bit rate as its original *pb-file*, or 1.5Mbps, which is a speedup of five. Like the original playback file, the *ff-file* contains metadata so the file can be randomly accessed. Thus when the user requests the file be played back using fast forward, the *VSS* switches from the original playback file to the *ff-file*.

Normally, because of its motion estimation algorithms, MPEG streams can be viewed in the forward direction only. The *ff-file*, however, must be able to be displayed in both the forward and backward directions. We have developed an algorithm for allowing an MPEG stream to be viewed both forward and backward. This is also covered in Chapter 8.

4.6 Buffer Management

Figure 8 shows the data flow during playback of a media stream. Data is retrieved from the disk into the server's memory by the *VSS*, sent over the network and retrieved into memory by *RETHER*, and finally displayed on the user's monitor by the decoding device. As can be seen there needs to be mechanisms for managing buffer space in both the server and the client. The next two sections discuss the design of the buffer management schemes.

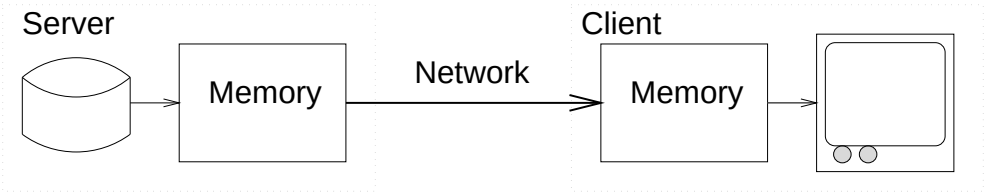

Figure 8: *Path of data from disk to client display.*

4.6.1 Server Buffer Management

There is a fundamental tradeoff between the buffer size and the effective disk transfer bandwidth. To improve the effective disk bandwidth, larger buffer space is needed to stage the data so that the seek/rotation overhead is amortized over larger transfer units. By increasing disk transfer bandwidth, more streams can be serviced. To a first approximation, the effective disk bandwidth can be represented as follows:

$$Bandwidth_{effective} = \frac{TransferSize}{Overhead + \frac{TransferSize}{RawTransferBandwidth}} \quad (1)$$

Overhead refers to seek and rotation delays and is determined by disk hardware. To increase $Bandwidth_{effective}$, one could increase either $TransferSize$ or $RawTransferBandwidth$. The former requires more buffer memory while the latter implies faster disks. Both incur extra cost, however. Since it is easier to incrementally add memory to a system than to replace disks, extending buffer space is a more popular technique to enhance the performance of the disk subsystem.

Although there has been extensive research on buffering schemes for video servers [TABA93b], *double buffering* is used in *SBVS* because of its simplicity. In *double buffering* there are two buffers associated with each video stream. Within each cycle one buffer receives data from the disk while the other has its data sent out to the network. At the end of each cycle the roles of the two buffers switch. Thus, the total buffer space requirement for N_{max} streeams is

$$\sum_{i=1}^{N_{max}} 2 * Size_{sector} * \lceil \frac{C_i * P_{io}}{Size_{sector}} \rceil \quad (2)$$

When $C_i = C$, for all i, the total buffer memory requirement is approximately $2N_{max}CP_{io}$. It is possible to reduce the buffer memory by half if the space emptied out by network transfers is reused to store incoming I/O data. However, this scheme is much harder to implement when streams are added and deleted from the system. When $N_{max} = 45$ and $P_{io} = 1sec$, P_{net}, the amount of memory needed for double buffering is 24 Mbytes at MPEG-1 resolution (1.5 Mbps). Because this memory requirement is very modest—current DRAM prices are about \$5/MB — it was decided to trade memory for design and implementation simplicity. As shown in Equation (2), the buffer memory requirement is proportional to P_{io}, which in turn determines the disk efficiency.

In double buffering, the order in which the streams' buffers are filled does not have to be the same as the order in which they are emptied. The *VSS* simply guarantees that at the end of each cycle, the data that is needed by *RETHER* has been retrieved. Therefore, stream-by-stream scheduling will never cause underflows if double buffering is used. In video playback applications, the network data transfer unit is typically much smaller than the disk data transfer unit. The reason is that packet bursts should be reduced as much as possible to avoid network performance degradation in the presence of bursty traffic. On the other hand, the amount of data transferred in each I/O transaction should be as large as possible to amortize disk seek/rotation overhead. Consequently, there is a difference between the I/O cycle time, P_{io}, and the network cycle time, P_{net} (the time to send 1 network block to each active client), with P_{io} being an integral multiple of P_{net}. The size of each network block is thus

$$Network\ Block\ Size\ =\ \frac{C_i\ (in\ bytes\ per\ second)}{Network\ Cycles\ per\ second} \tag{3}$$

For example, when the I/O cycle time is 990 msec and the network cycle time is 33 msec, one network block is sent out every 33 msec from each video stream, but each I/O transaction brings in 30 network blocks that belong to a particular video stream. If the average consumption rate of the video is 180KB per I/O cycle, then the network block size is 6KB.

4.6.2 Client Buffer Management

Whereas server buffering is used to stage data between disks and networks, client buffering is used to accommodate bit rate fluctuations and network jitter. In the previous discussions, the system bases resource management and scheduling on the average bit rates of the video sequences, when in fact, the bit rates are variable. Therefore, it is essential to buffer data at the client side to eliminate potential underflow when the average bit rate is used to transfer the video stream. Again, an important advantage of the constant bit rate assumption is that it greatly simplifies the complexity of static disk data placement and run-time resource allocation. Consider a 6-frame video sequence, where the frame sizes are 8, 8, 8, 2, 6 and

4 kilobytes (KB). In this case, the average frame size is 6 KB. The average bit rate is computed as the product of the average frame size and the number of frames per second. In this example the average bit rate is 180KB (6 * 30 frames/sec). If the system only provides I/O and communication bandwidth equal to the average bit rate, the first three frames in the sequence will not be rendered smoothly since their frame size is greater than the average frame size.

To service a variable bit rate stream with a constant bit rate bandwidth, the system needs to accumulate sufficient "inventory" initially, to accommodate instantaneous bandwidth fluctuations in the future. Intuitively, the system must accumulate enough data in the buffer memory to ensure that, at any point in the video playback process, the amount of data that has been transferred from the disks is always larger than the accumulated data needed by the video stream up until that point. Formally, let L denote the number of frames in a video sequence, F_i the accumulated size from the first frame up to and including the ith frame, A the average frame size, and S_{inv} the amount of anticipatory inventory. Then

$$S_{inv} = \max_{i=1}^{i=L} (F_i - A * i) \quad (4)$$

In the previous example, the average frame size is 6KB and S_{inv} also works out to be 6KB. Table 4 shows the amount of data that is received displayed and accumulated during the first 7 cycles of playing the video file. During the first cycle, 6KB is received and pre-buffered. If the client had tried to display the first frame of 8KB it would not have had enough data. During the second cycle, the client can start displaying frames and can guarantee that it will never starve.

Received	Displayed	Accumulated
6	0	6
6	8	4
6	8	2
6	8	0
6	2	4
6	6	4
0	4	0

Table 4: *Amount received, displayed and accumulated during the first seven cycles of playing a video file whose first 6 frames are 8, 8, 8, 2, 6, and 4 kilobytes.*

Because S_{inv} can be pre-computed and stored with the data, the computation of Equation (4) is performed only once and will not affect run-time performance. Note that S_{inv} is dependent on the choice of synchronization granularity. In Equation (4), a video frame was chosen as the granularity because it is the natural visual display unit. In general, the size of

anticipatory buffer decreases with the granularity, at the expense of less room for mistakes in estimating the software/hardware overhead.

Number of Videos	Anticipatory Frames
9	< 10
4	10 20
1	$>= 20$

Table 5: *Results of sampling 14 videos for the size (in frames) of the anticipatory buffer.*

In [CW95], the anticipatory buffer size for various example video sequences are calculated according to Equation (4). Table 5 shows the results for 14 varying MPEG-1 streams. Although this sample is small, it shows that most videos will not need many frames to be accumulated. Assuming the normal 30 frames per second playback rate of MPEG, most videos will need to accumulate less than one second of data. In addition, the MPEG draft includes a specification for a Constrained Parameters Bitstreams (CPB). CPB are a limited set of sampling and bit-rate parameters designed to normalize computational complexity, buffer size, and memory bandwidth, thus keeping the anticipatory buffer size low.

Chapter 5

Implementation of the VSS

Figure 9 shows the software architecture of the two major components of the *SBVS*; *VSS*, the *Video Storage Subsystem*, and *RETHER*, the *Network Subsystem*. The main task of the *VSS* is to schedule and move data from the disks in the disk array to main memory. The network subsystem's main task is to move data from main memory of the server to the clients.

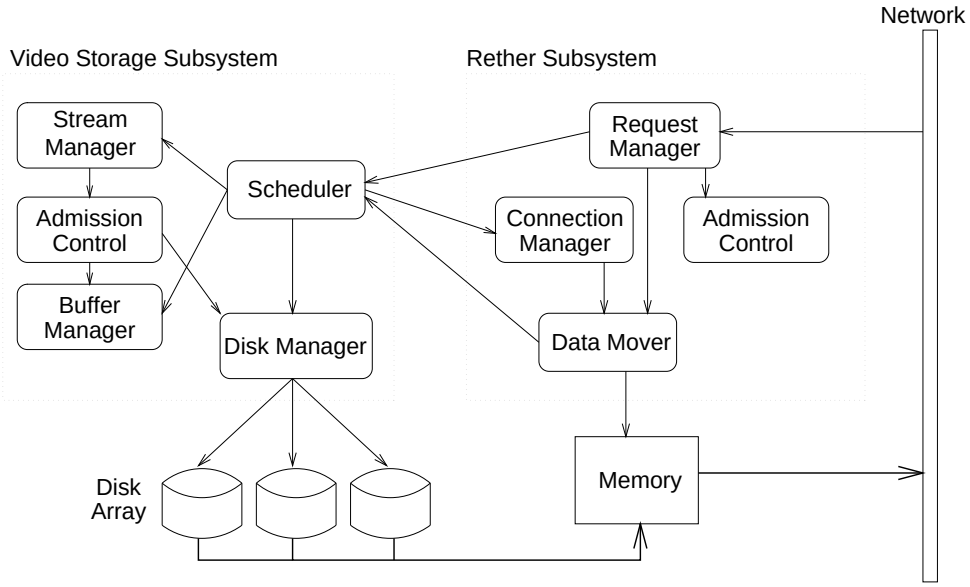

Figure 9: *System software modules in the SBVS.*

The responsibilities for each of the *VSS* modules are as follows. The *stream manager* manages stream-oriented functions such as stream initiation and tear-down, and the VCR functions, such as pause/resume, jump forward/backward, and fast forward/reverse. It creates a playback list that is used by the scheduler to schedule disk accesses. The *admission control* module determines if the *VSS* has enough resources to service a new stream. It uses the disk manager to retrieve statistical information about the systems I/O resources. The *scheduler* is the central coordinator of the *VSS*. It receives messages from *RETHER*, and

schedules the execution of the *stream manager* and the *disk array manager*. Using the playlist constructed by the stream manager, it sets up high-level access requests that are sent to the *disk array manager*. A retrieval at this level simply states that N sectors of data should be retrieved from logical block B into memory location M. Finally, the *disk array manager* breaks up a high-level request into several sub requests—one for each disk. This module creates the actual SCSI commands, sends the requests to the SCSI controllers, and processes the interrupts that are generated when each disk read finishes, thus, bypassing any normal file system processing.

The network subsystem of the *SBVS* is the *RETHER* subsystem. The responsibilities for each of the *RETHER* modules are as follows. The *request manager* processes messages from the network. These messages include video requests and token arrivals (see next section). The *connection manager* manages the real-time and non-real-time connections within the network. The *data mover* is responsible for setting up the network packets and network commands that transfer data from main memory to the network. The *admission control* module determines if the network subsystem has enough resources to service a new network stream.

The following sections discuss the implementation of the modules within the *VSS* . The *VSS* admission control system is described in Chapter 7 and details of the *RETHER* subsystem can be found in [VC94].

5.1 Scheduler

The *scheduler* is responsible for initiating the execution of the other modules in the *VSS* and executes at the beginning of each I/O cycle. Rather than use an internal timer to begin execution, the beginning of an I/O cycle is determined by the end of the last *RETHER* token cycle. (Each I/O cycle is comprised of N token cycles where $N = \frac{IOCycleTime}{TokenCycleTime}$. The cycle times are static parameters which are set when the video server is started.) This synchronizes the *VSS* with *RETHER*. See Section 5.9 for a detailed description. Algorithm 5.1 presents the algorithm used by the *scheduler*. When the *scheduler* is executed, it first invokes the *stream manager* to process any messages that have been sent by any users since its last invocation. The *stream manager* (see next section) constructs a playlist for all of the active streams. Using this playlist, the *scheduler* then sets up a queue of high-level access requests, one request for each active stream. This queue is then sent to the *disk array manager* which performs the I/O requests. Once the queue is constructed and the *disk array manager* invoked, the scheduler goes to sleep until the beginning of the next I/O cycle, when it is again woken up by *RETHER*.

Algorithm 5.1 Algorithm used by the VSS Scheduler.

```
While True
    Invoke stream manager to process pending messages
    Construct access list
    Invoke disk array manager to process access list
    Sleep until woken by RETHER
EndWhile
```

5.2 Stream Management

The *stream manager* is responsible for managing the information about the files available to clients and the information about the active streams in the system. The information about stored files is kept on the *SBVS* system disk and accessed using normal file system processing. Information about active streams is kept in the *playlist*. The *stream manager* maintains the playlist by inserting new streams, deleting old streams, and updating active streams, i.e., by changing the type of playback for such VCR-functions as fast forward, fast rewind, etc.

When the *stream manager* is started, it reads the media file directory into memory. The directory is updated whenever files are added to or deleted from the system, usually once a day. The information in the directory includes the starting block of each file, the size of the file, and the file's recorded rate (in bytes per I/O cycle). Once a stream becomes active (by a client invoking the *videoPlay* call), the *stream manager* keeps other information about each active stream in the playlist. This includes a stream identifier, the current location of the file being retrieved, etc. Table 6 shows the information that is kept by the *stream manager* about each stream.

Included in the active information is the next logical block number that must be retrieved for the stream on the next cycle. On hard disks logical block 0 is located on the outer cylinder of the disk and logical block N is located on the inner cylinder. The playlist is kept sorted by the next block that each stream must retrieved. This minimizes the seeks that each disk head must perform when retrieving data for multiple streams. When new streams become active, they are inserted into the playlist in sorted order.

Changes to the playlist are only done at the beginning of each I/O cycle. At this time, the *VSS* scheduler invokes the *stream manager*. The *stream manager* first checks to see if any streams have completed. If so, the stream is removed from the schedule and *RETHER* is notified that the stream has finished. The *stream manager* next checks a well-known input queue for messages from clients. Once all messages are processed, the *stream manager*

Variable	Description
firstBlock	Logical block number of first block in the file
nextBlock	Logical block number of next block to be retrieved
lastBlock	Logical block number of the last block in the file
dataBuffers[2]	Addresses of the two buffers for the stream
bufferPtr	Address in buffer where data should be placed
bufferLen	Length of a buffer
recordedRate	Average bit rate of the file in bytes per VSS cycle
playbackRate	The playback rate in bytes per cycle. It
	can never exceed the recorded rate
lowerBoundBytes	Number of bytes to be retrieved per cycle when using the lower bound block size. See Section 5.5
bufferedData	Amount of total data buffered in memory. Includes extra data due to sector mismatch.
frameLength	Size of each frame. Used by *RETHER*
streamId	Stream identifier for *RETHER*
destination	Destination node
port	Port on destination node
metadataSize	Size of the metadata in bytes
metadataBlocks	Number of sectors the metadata uses
metaData	Pointer to the metadata in memory loaded
extraBytesFwd	Number of extra bytes accumulated due to the mismatch between the playback rate and the sector size
extraBytesBack	Same as above, but used when going in reverse
fileIndex	Specifies which file is being shown, the main playback file or the fast forward/rewind file
status	Status of the stream
retherStatus	Status of the stream in relation to *RETHER*

Table 6: *Information kept about each stream.*

completes its processing and returns control to the *scheduler*. The following list describes the actions that are taken based on the type of message from the client.

- *videoPlay:* Since all disk requests for all active streams are sent to the disks at the beginning of each I/O cycle, streams can only be admitted into the system at the beginning of an I/O cycle.

  Figure 5.2 shows the algorithm that is performed by the *stream manager* when a new stream requests admission into the system.

  When a user requests to playback some video F, the *stream manager* first retrieves information about F from the media file directory. The user may play the video using

Algorithm 5.2 Algorithm for responding to a *videoPlay* message.

```
/* Assume user wants a new stream to playback file V */
Retrieve playback rate of V in directory of files
If file does notexist then return error
memNeeded = (2 * playbactRate) + extra
if memUsed + memNeeded > memCapacity then return error

/* Can RETHER admit the Stream? */
if retherAdmit(playbackRate) == FALSE then return error

/* Can the VSS admit the stream? */
ifV SSAdmit(playbackRate) == FALSE then return error

/* Add the stream */
Create stream information structure element

/* The metadata for each file is located in the logical blocks of the
file before the first actual media blocks */
Add request to playback schedule to retrieve the file header into memory
    The file header contains the mapping from frame number to
    disk block and is used for random access to the file
Add element to playback schedule sorting schedule by nextBlock
```

a default playback rate (the recorded rate) or a lower playback rate, i.e., slow motion. If F exists, the *stream manager* determines if there is enough memory capacity for buffering the data of the new stream for F. (Again, the amount of memory needed when using double buffering is $(2*playbackRate)+extraDueToSectoring$. See section 5.8 for details). If enough memory is available the *stream manager* then checks with the *RETHER* admission control system to determine if the network can handle a new request with the given playback rate. If there is enough network bandwidth the *stream manager* then checks with the *VSS* admission control system. If the *VSS* has enough bandwidth to accommodate the stream, it can be entered into the system. Again, the *stream manager* keeps a list of active streams, sorted by their physical locations on the disk. The location of the new stream is retrieved from the directory structure and the stream is entered into the playlist so that the list remains sorted. On the ensuing *VSS* cycle, data will be retrieved for the stream. Once the first cycle of a new stream finishes (data is ready to be sent out), the *RETHER* subsystem is notified that a new stream is ready to be serviced. The notification includes the playback rate of the stream and the addresses of the data buffers. *RETHER* can then begin to deliver the data.

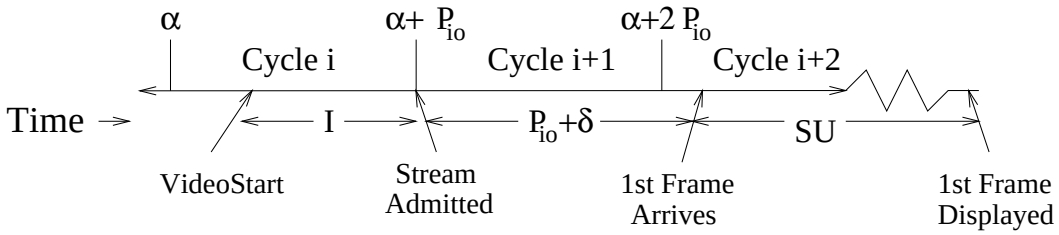

Figure 10: *Time-line showing events when a new stream is added.*

At the client, there is a startup delay between the initiation of the video stream request by the user and the display of the first frame. Figure 10 depicts the events that occur when a new stream is added. The time the user waits is defined to be $I + P_{io} + \delta + SU$. Since streams are only admitted at the beginning of an I/O cycle, I is the time between the user initiation until the beginning of the next I/O cycle. This occurs during cycle i in Figure 10. During cycle $i+1$ the first blocks of the stream are retrieved into memory which takes time P_{io}. During cycle $i+2$ *RETHER* can begin delivering the data that was retrieved during cycle $i+1$. It takes time δ from the beginning of the cycle to the time the first frame is delivered to the client. Finally, the client takes time SU to pre-buffer enough data so it does not starve (See section 4.6.2.)

- *videoPause:*

  The playback list is updated and the stream is simply marked as *paused*. Data blocks are not retrieved for the stream, but its network reservations are held. Data from the previous cycle that was retrieved is still sent to the client.

- *videoStop:*

  The streams buffers are freed, *RETHER* is notified, the stream is deleted from the playback list and the stream information structure is deleted from memory.

- *videoResume:*

  First, the stream is marked as active. During the next I/O cycle, say cycle i, the *VSS* will retrieve data from the disks. During cycle $i+1$, the data will be sent to the client. At the client, when the user initiated the *videoPause* the client software immediately stops sending data to the decoder, freezing the frame on the monitor. It may take, however, up to one I/O cycle before the server receives the message. During this time data is still being sent to the client from the server and buffered at the client. When the user executes *videoResume*, the client software can immediately start sending data to the decoder for processing. Even though it may take up to one I/O cycle for the message to be processed, the server had already sent enough data to satisfy the client

after the user had executed the pause action. Since the client stops consuming data as soon as the users initiates the pause action, any data that was pre-buffered at the client to ensure it does not starve, is not affected.

- *videoJump:*

  The user initiates this action and specifies how many seconds to jump forward or backward. The server translates the number of seconds into a number frames based on the playback rate. The current frame number is increased or decreased and the disk logical block is found using the stream's metadata which associates frame numbers with logical blocks. The stream is then treated as a new stream since its buffers have become invalid. Admission control does not have to be performed but data must be pre-buffered at the client.

- *videoFast,videoRewind:*

  Again, the *SBVS* uses a separately encoded, lower resolution version (*ff-file*) of the original video to support both fast forward and fast rewind of video files. The *ff-file* is encoded at both a lower temporal resolution and spatial resolution than the original file and is guaranteed to have a lower playback rate than the original file. (See section 4.5 for the design description.) The *stream manager* simply treats the *ff-file* as it does any other stream. It removes the original playback stream from the playlist, retrieves information about the *ff-file* from the media file directory and places the new stream into the playlist. Since the *ff-file* is guaranteed to have a lower playback rate than the original file, no admission control has to be performed by either *RETHER* or the *VSS* when switching from the normal playback file to the *ff-file*. (See section 5.4 for an example).

5.3 Disk Array Manager

The *disk array manager* is responsible for the retrieval of media blocks from the disk array. As shown in Figure 11, the input to the *disk array manager* is a playlist which specifies a queue of retrieval requests. For each request, a logical block number, number of blocks and a memory location is specified. The number of blocks, starting at the logical block number, are to be retrieved into the specified memory location. The design of the *VSS* specifies that each high-level request is spread across all of the disks in the disk array. The *disk array manager* translates each playlist request into a set of requests for each disk. Since media blocks are striped across the disk array such that they occupy the same logical block numbers on each disk, the *disk array manager* only needs to change the memory location and number of blocks

that must be retrieved. For example, as shown in Figure 11, the request S_1 specifies that 240 blocks should be retrieved from logical block 1000 and placed into location 0xA00000. Since there are two disks in this configuration, the *disk array manager* breaks the request into two requests, one for each disk. The first request specifies that 120 blocks from block 10000 should be placed into location 0xA00000 and the second request specifies that 120 blocks should be read from block 10000 and placed into location 0xA00F00.

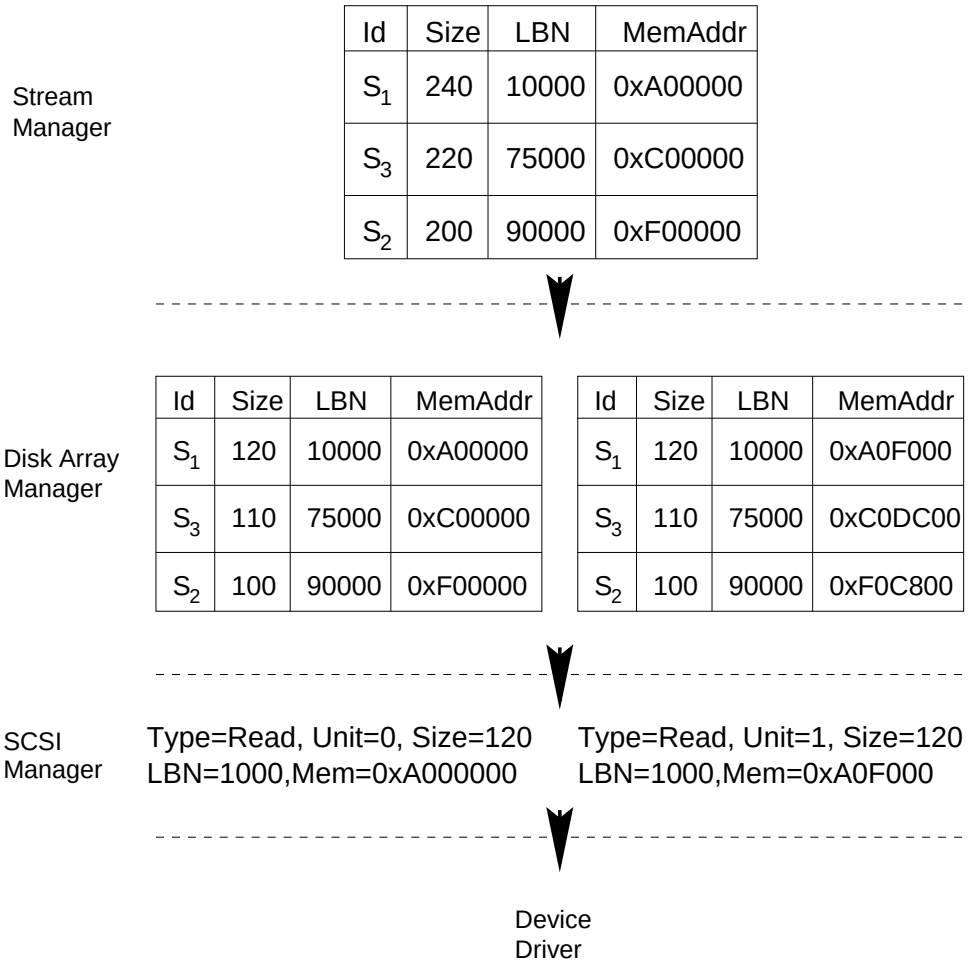

Figure 11: *Disk level modules.*

When the *disk array manager* is invoked, it executes the playlist as quickly as possible by streaming the requests to the disks. There is no explicit scheduling. Each of the requests is placed into a queue—one queue per disk. Once the queues are created, the actual retrievals can be executed. The *disk array manager* sends one or more requests from the head of each queue to the *SCSI manager*. The *SCSI manager* translates each request into the appropriate SCSI format and sends the SCSI command to the device driver of the SCSI controller. The device driver then outputs the command to the SCSI controller where the request is handled in hardware. The software device driver is notified by the SCSI controller when the request

has been satisfied and the data placed in memory. Since DMA controllers are used, the CPU does no processing while the data is being retrieved directly to memory. When the device driver is notified by the SCSI controller that a request is finished, it calls the *disk array manager* for processing. The disk that executed the request is contained in the device driver response. The *disk array manager* uses the disk number to interrogate the request queue for the associated disk. If there are outstanding requests in the queue, the *disk array manager* sends the next request to the *SCSI manager*. Once all of the requests are satisfied, the *disk array manager* notifies the scheduler that it is finished.

In addition to executing the playlist, the *disk array manager* records the total service time to execute the all of the requests in the playlist. The admission control system uses this information when deciding if the *VSS* has enough resources to accommodate a new stream.

5.4 Metadata

Each video in the system contains metadata that describes the properties of the file. The metadata for each file is placed in two locations. The first part, the directory metadata, is kept in the centralized *VSS directory.* The directory contains high level information about all of the files accessible by the *VSS*. This directory is kept as a normal Unix file and resides on the system disk. Table 7 describes the directory information that is kept about each video. This structure includes the name of the file which is used by clients for playing, the size of the file, the starting block of the file (again, each file occupies the same logical block numbers on every disk), the recorded rate of the file in bytes per second, and the size of the file header. The file header is the second part of the metadata and resides with each file on the video disks. The file header is used for random access into the file and is simply an array of records where each record contains a logical block number and a frame offset. Thus, the system can jump to random points in the file using the frame offset. For example, to jump 30 minutes into the file that is being played back at its recorded rate, the system would jump to frame $30 * 60 * framesPerSecond + currentFrame$.

Variable	**Description**
name	name of the video
size	total size in bytes of file
startBlock	Logical block number of first block
recordedRate	recorded rate in bytes per second
bytesInHeader	total number of bytes of the file header

Table 7: *Information kept about each video file.*

If a video has multiple, separately encoded versions for fast forward/rewind or multi-resolution, the file header for the multiple versions is included at the beginning of the main playback file. When a stream is started, the header information for every version of the video is read into memory. The size of the header metadata can be calculated as follows. Let the video be one hour long and recorded at 30 frames per second. If the headers contain an entry for every second of playback, there would be 3600 entries. Each entry contains a 4 byte block number and 2 byte frame offset for a total of 21KB.

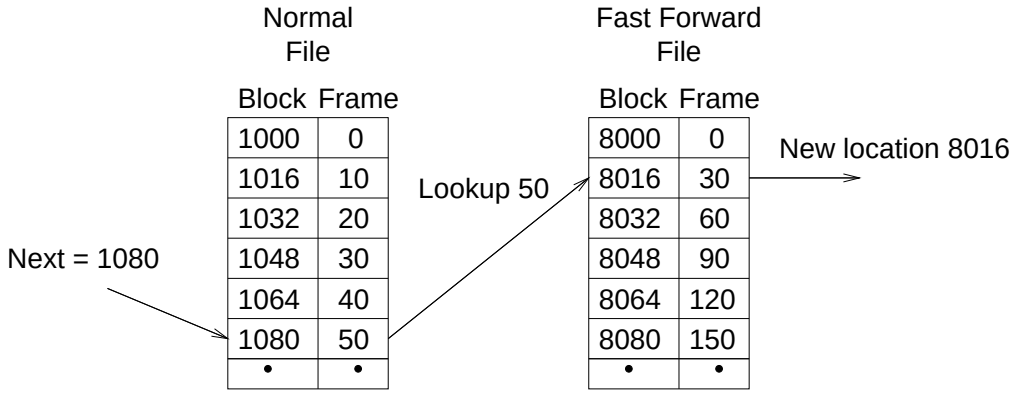

Figure 12: *How metadata is used to go from normal playback file to fast forward/rewind file.*

As an example, Figure 12 shows how the *stream manager* uses the file's header metadata to translate a logical block number in the *pb-file* into a logical block number of the *ff-file*. In the example, the stream is ready to retrieve block 1080 from the *pb-file* when the request for fast-forward is received. To go from normal viewing to fast forward, the *stream manager* uses the next logical block to be retrieved as an index into the metadata of the *pb-file*. The block number is translated into a frame offset into the *pb-file*, i.e., block 1080 is translated to frame number 50. Frame 50 is then used to find the closest media block in the *ff-file*, in this case, block 8016. If the exact frame number is not located, the closest frame number is used. The next logical block for the *ff-file* is then initialized to 8016. Because block boundaries do not necessarily align with frame boundaries, the *ff-file* will begin showing at frame 30 which causes a slight overlap, i.e, frames 30-50 are re-shown (less than one second). Thus, the granularity of the metadata determines how closely a frame in the *pb-file* corresponds to a frame in the *ff-file*.

5.5 Retrieval Algorithms

The *VSS* uses a constant data length (CDL) media block as its basis for disk retrieval, where a media block is the amount of data that is retrieved for a stream during one I/O cycle.

RETHER also uses a CDL media block as its basis for transmission during a network cycle, where the length of a *RETHER* media block is equal to the average bit rate of the video stream [VVC96].[1] If it were possible, the *VSS* would read into memory in one I/O cycle the same amount of data that is sent out in the next cycle. However, there is a problem in that the sector size on a disk is a fixed value. If the average bit rate is not a multiple of the sector size, the amount of data read into memory will be different than the amount sent out for transmission.

The *VSS* needs to prevent overflows caused by buffer accumulation due to disk sectoring. For example, assuming a disk array with four disks, let the transmission size (average bit rate) of some video be 64000 bytes per cycle. However, due to disk sectoring and our constraint that the same number of sectors is accessed from every disk every cycle, either 63488 bytes (124 sectors) or 65536 bytes (128 sectors) is retrieved. This section presents the algorithms that are used to read and write I/O media blocks due to the mismatch between the retrieval size and transmission size.

Let:

- $M_i = playbackrate$: bytes to be transmitted in an I/O cycle for stream i. This is the average bit rate of the video.
- $S = sector\ size$: 512 bytes on most disks.
- $N = number\ of\ disks$: In disk array.

First, calculate the upper bound (UB) and lower bound (LB) of the total number of sectors that need to be retrieved such that $LB <= M/S <= UB$. The lower bound for stream i is:

$$LB_i = \lfloor \frac{M_i}{N * S} \rfloor * N \tag{5}$$

The upper bound for stream i is:

$$UB_i = \lceil \frac{M_i}{N * S} \rceil * N \tag{6}$$

5.5.1 Forward Retrieval Algorithm

The writing of media blocks to the disk array should optimize the performance of reading media blocks during normal playback. Reading is optimized if the same number of sectors is read from each disk (load balancing), the number of disk read commands is minimized, and the amount of memory copies is minimized. The algorithm in Figure 5.3 is used when

[1] Pre-buffering of data must be performed at the client to smooth out the fluctuations caused by the variable bit-rate MPEG file.

Algorithm 5.3 *Algorithm for writing data to disk and reading data from disk during normal playback for each stream*

```
extra = 0
lowerBoundBytes = LB * S
upperBoundBytes = UB * S
nextSector = firstSector
while moreCycles do
    if lowerBoundBytes + extra >= M then
        Read/Write LB sectors    /* LB/N Per Disk */
        nextSector += LB/N
        extra = (lowerBoundBytes + extra) - M
    else
        Read/Write UB sectors    /* UB/N Per Disk */
        nextSector += UB/N
        extra = extra + (upperBoundBytes - M)
    endIf
    Copy extra bytes from bottom of
    current buffer to top of next buffer
endWhile
```

writing the video to the disk array, and when reading the video during normal playback. The variable $nextSector$ is the first logical sector number of a media block. Since data from each video file occupies the same logical sectors on every disk, $nextSector$ is valid for every disk. Each time the disk manager services a stream, it uses the algorithm to determine the exact physical I/O size. The value $extra$ specifies the amount of data that has accumulated in the buffer due to the differences between the retrieval size, which is either an LB or UB media block, and the transmittal size. After the data is read into a buffer, the accumulated extra data, i.e., the data retrieved but not sent out, is copied to the next buffer that will be filled. This algorithm is used for each active video stream.

In this algorithm, the value $extra$ is calculated based on the previous value of $extra$, the amount of data to be retrieved, and the trasmittal rate of the video. It is possible, however, to calculate $extra$ without using its previous value. It can be calculated as follows:

$$extra = ((nextSector - firstSector) * S * N) \ mod \ M \tag{7}$$

As an example in using the algorithm, let $M = 64000$ bytes, $N = 4$ disks, $S = 512$ bytes, and $firstSector = 100$. The lower bound, LB, is 124 sectors or 63488 bytes, and the upper bound, UB, is 128 sectors or 65536 bytes. Table 8 shows, for the first five cycles, the amount of data that is retrieved from the disk, the logical sector number of the first sector in the block, the total amount of data for the stream that is buffered in memory, and the amount

of extra data that has accumulated once the 64000 bytes has been shipped to the client.

Cycle	Start Sector	Sectors Per Disk	Total Bytes Retrieved	Bytes Buffered	Bytes Extra
0	100	32	65536	65536	1536
1	132	31	63488	65024	1024
2	163	31	63488	64512	512
3	194	31	63488	64000	0
4	225	32	65536	65536	1536

Table 8: *Amount of retrieved, buffered, and extra data accumulated at the end of each cycle.*

Lastly, the buffer space for each read for stream i needs to be calculated. The maximum that can be read is $UB_i + extra$ where $extra < M_i - LB_i$, otherwise LB would have been read. Thus, the buffer space (in bytes) is:

$$(UB * S) + M - (LB * S) = M + (N * S) \tag{8}$$

5.6 Rewind Retrieval Algorithm

Although there have been numerous proposals in the literature to support fast forward and rewind, the easiest way to support the rewind of MPEG-1 system files using current technology is to use a separate, specially encoded file. See Chapter 8 for a more detailed description of this problem.

To implement rewind, media blocks must be retrieved from the disks in reverse order. In addition, *RETHER* must send packets of each media block in reverse order. Like retrieving blocks for forward motion, the amount of data retrieved for each stream during each cycle will either be an LB or UB media block. For example, assume the server is proceeding in reverse, starting at sector 257, and using the media blocks as show in Table 8. First, 32 blocks are retrieved from each disk at sector 225, then 31 blocks are retrieved starting at sector 194, then 31 blocks are retrieved from sector 163, etc. The algorithm in Figure 5.4 handles the reverse retrieval of the media blocks.

First, the value $extraBack$ is set to be $N * S$. Like the forward motion algorithm, this value contains the difference between the amount of buffered data and the transmission size. The amount buffered at any time during reverse retrieval is $extraBack + retrievalSize$. It must be guaranteed that this value is always greater than the transmission size M, otherwise there would not be enough data to send. In addition, the user must be able to change from forward motion to rewind at any time. Since rewind is implemented using a separate file, the server must be able to jump between random points in the normal file and rewind file. The

server uses metadata to calculate the appropriate entry point into the rewind file. However, the calculated entry point in the rewind file may be an LB media block. For example, let $extraBack$ be initialized to 0, assume that the file in Table 8 is the rewind file, and the server has to start retrieving data backwards from sector 194. The size of the media block is $LB * S$ or 63488 bytes, $extraBack$ is 0, but the network is expecting to send out 64000 (M) bytes during the cycle after the data is retrieved. In this case, there is not enough buffered data to satisfy the network subsystem. One solution would be to read the extra data needed to ensure that the amount of buffered data is greater than or equal to the transmission size. This would mean retrieving data from the beginning of the next media block and would incur an extra read at the disk. Instead, $extraBack$ is initialized to be $N * S$ (the maximum amount of extra data that can accumulate during any retrieval) and the $extraBack$ bytes in memory are all initialized to be 0. This guarantees that the calculated value $extraBack + retrievalSize$ is always greater than the transmission size, M. When the network sends the first cycle of data for rewind, the first $extraBytes$ of data will be 0. The client program simply discards this data. The data will be correct for all other cycles.

Next, the algorithm uses $extraFwd$ to determine how much data is to be retrieved during the cycle. At any given time, the system knows the current $nextSector$ and can calculate $extraFwd$. The algorithm must then determine the size of the previous media block which is based on the previous value of $extraFwd$—let's call it $extraLast$. From the forward retrieval algorithm, only one of the following equations can be true.

$$extraFwd = extraLast + (UB * S) - M \tag{9}$$

$$extraFwd = extraLast + (LB * S) - M \tag{10}$$

For example, assume the media block at sector 225 has been retrieved using reverse mode (Table 8). The value $extraFwd$ for sector 225 is calculated to be 0 using Equation (7). The algorithm must determine the size of the media block that comes before sector 225 so that $0 = extraLast + ReadSize - M$ where $ReadSize = UB * S$ or $ReadSize = LB * S$. By definition, $0 \leq extraFwd \leq (N * S)$ and $0 \leq extraLast \leq (N * S)$. If $extraLast$ is less than 0, Equation (10) must be true, otherwise Equation (9) is true. For $extraLast$ to be less than 0 in Equation (9), $extraFwd + M - UB < 0$. If this is the case, then the previous media block is an LB block, else it is a UB block. Using this equation with $extraFwd = 0$, $extraLast$ is less than 0 in Equation (9). The media block before sector 225 is an LB block (31 sectors/disk) so $nextSector$ is set to 194.

At the end of each cycle, $extraBack$ bytes must be copied from the top of the buffer that was filled to the bottom of the the buffer that is next to be filled. See section 5.8 for a description of how the buffering works during rewind.

Algorithm 5.4 *Algorithm for reading data from disk during rewind*

$extraBack = numDisks * S$
while $moreCycles$ **do**
  $extraFwd = ((nextSector - firstSector) * S * numDisks) \ mod \ M$
  if $(extraFwd + M - upperBoundBytes < 0)$
    $nextSector- = LB/numDisks$
    Retrieve LB sectors /* $LB/numDisks$ Per Disk */
    $extraBack = lowerBoundBytes + extraBack - M$
  else
    $nextSector- = UB/numDisks$
    Retrieve UB sectors /* $UB/numDisks$ Per Disk */
    $extraBack = upperBoundBytes + extraBack - M$
  endif
  Copy $extraBack$ bytes from top of current buffer to bottom of next buffer
endWhile

5.7 Support for Variable Playback Rates

There are three different rates at which a video can flow from the disk to the client display. First, the video data has a retrieval rate going from the server's disk array to main memory. Second, the data has a network transmission rate going from the server's memory to the client's memory. Last, the client decoder has a consumption rate at which it displays data from the client's memory. This section discusses the support within the *SBVS* for allowing the client's consumption rate to be changed. Since fast forward is handled by encoding a separate file, the support is for client consumption rates that are less than or equal to the normal playback rate, i.e., slow-motion.[2]

Since a video server proceeds in cycles, there are two methods to allow the client to change its consumption rate. In the *steady rate* method, the server changes its rate to match the new consumption rate. For example, assuming the client decoder is consuming data at 30fps and then requests the data to be sent at half speed, the server slows its retrieval down to retrieve and send 15 frames every second. In the *variable rate* method, the server has active and non-active rounds. During active cycles, the server will send data to the client at a higher rate than being consumed but buffer the data at the client. During non-active cycles, data is still consumed by the client from its buffer, however, no data is transmitted from the server to the client. The long term average production and consumption rates are still equal. For example, assuming a cycle is one second and the client is consuming data at 15fps, the server would send 30 frames during the first cycle while the client shows 15

[2]Slow-motion can only be performed at the client end by a software decoder. Current MPEG hardware decoders do not allow variable playback rates.

frames and buffers 15 frames. During the next cycle, no data is sent but the client still can show 15 frames. The main advantage of using the *steady rate* over the *variable rate* is that less resources need to be reserved. When using a variable rate, the reservation of resources is usually made using the maximum bursty rate. In the above example, when using a *steady rate*, 15fps would be reserved, but when using a *variable rate* 30fps would be reserved, even though the long term average is still 15fps.

The next sections discusses how these two methods are implemented in the *SBVS*. Assume that the normal playback rate (recorded rate of the video) is R and that the new requested rate is P.

5.7.1 Steady Rate Network Transmission

In the *steady rate* method, the network slows its transmission rate to match that of the client. Even though the network sends data at a constant rate, the *VSS* can produce data for the network using either a *steady rate* or a *variable rate*. When using a *steady rate* to retrieve data, the *VSS* decreases the amount of data that is retrieved during each cycle to match the network transmission rate. When using a *variable rate*, the *VSS* continues to retrieve the same amount of data as it does during normal playback. To keep the long-term production rate from the *VSS* equal to the slower network transmission rate, data is retrieved only during certain cycles. Because of its easier implementation, the *VSS* uses the *variable rate* scheme with the *steady rate RETHER* network transmission. Although the *VSS* doesn't use a *steady rate* scheme, the implications of implementing it are presented next. Then, the *VSS variable rate* is discussed.

5.7.1.1 Steady Rate I/O Transmission

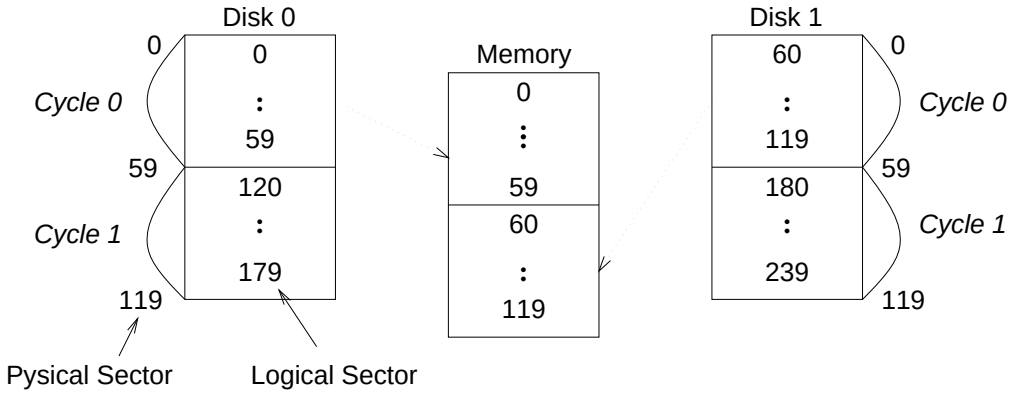

Figure 13: *Transfer of data from disk to main memory during normal playback with 2 disks and a playback rate of 120 sectors per second.*

If the *VSS* were to match its retrieval rate to that of the constant network transmission rate, one potential problem can occur. Figure 13 shows a possible block placement of a video file that was written using two disks and an I/O cycle time of 1 second. Assuming the normal playback rate is 120 sectors per second (sps) for the video, the stripe size is 60 sectors. 60 sectors are retrieved from each disk during each cycle. This corresponds to logical sectors 0-59 being retrieved from disk 0, and logical sectors 60-119 being retrieved from disk 1. (Each logical sector maps to a physical sector on one of the disks in the array. In the figure logical sectors 60-119 map to physical sectors 0-59 on disk 1.) During cycle 1, logical sectors 120-179 are retrieved from disk 0, and sectors 180-239 are retrieved from disk 1. Now, assume that some client wants to view the same video at 4/5 speed or 100 sectors per second. (The decoder at this client may only support a bandwidth of 4/5 the normal rate. In the implemented system the user would specify the lower rate in frames per second. The quantity sectors per second is used for ease of discussion.) Rather than retrieving 120 sectors per cycle, only 100 sectors are needed per cycle. In this example, logical sectors 0-59 would be read from disk 0, and logical sectors 60-99 would be read from disk 1 during cycle 0. During cycle 1, sectors 100-119 would be read from disk 1, sectors 120-179 would be read from disk 0, and sectors 180-199 would also be read from disk 1. Although, sectors 100-119 and 180-199 on disk 1 are physically contiguous, they do not correspond to a consecutive portion of the MPEG file. Within main memory, the media block's logical sectors must be stored consecutively so that network packets are sent out to clients in the correct order.

The problem is that two non-contiguous logical regions of the video are accessed from the same disk during the same cycle. However, this only occurs when using certain playback rates. Without loss of generality, assume that disk 1 in a disk array of N disks has two non-contiguous logical regions that need to be accessed during one cycle. For this problem to occur, the start of the logical region starts on disk 1. Since data is placed on the disks in a round-robin fashion, the logical region traverses every other disk and then returns to disk 1. Thus, during slow motion the total retrieval size per round must be at least $(N-1) * stripeSize$. During normal playback, $N * stripeSize$ blocks are retrieved. So, if there are N disks in the disk array, the problem occurs if $P > \frac{N-1}{N} * R$. For example, if there are three disks in the array, it occurs if P is greater than two thirds of the normal playback. If the normal playback rate is 120 sectors/second, this would occur for rates greater than 80 sectors/second.

To handle the problem of retrieving logically non-contiguous data, there are three options—these are described in the context of Figure 13: 1) Three read requests can be performed, one for sectors 100-119, one for sectors 120-179 and one for sectors 180-199; 2) Two read requests can be performed, one for sectors 120-179 and one for 100-119 and 180-199. In this case, data must be rearranged in memory; and 3) Two reads can be performed

using scatter-gather. Scatter-gather allows a single read to be executed at the disk. The SCSI controller, however, breaks up the retrieved data and can send subsets of the data to different places in memory. For example, a single request is sent to disk 1 to retrieve the sectors 100-119 and 180-199. The SCSI controller is able to send sectors 100-119 to one location in memory and sectors 180-199 to another location. In summary, the first option results in extra disk overhead, the second option results in large memory copies, and the third option must be supported by the SCSI controller.

5.7.1.2 Variable Rate I/O Transmission

Because of its simple implementation, the *VSS* uses a *variable rate* method of retrieval together with the *steady rate* network transmission rate. The media block placement is optimized so that during normal playback the same number of blocks are retrieved from each disk during each round. Rather than retrieve less data if the network transmission rate decreases, the *VSS* continues to retrieve data as if the client is viewing in normal playback mode. During each cycle, the amount of accumulated extra data is recorded. This extra data is due to the difference between the retrieval size and the slower network transmission rate. Eventually the amount of accumulated data in memory will be enough to satisfy the data requirements of the network during an entire cycle. During this cycle no data needs to be retrieved for the stream, and the amount of accumulated data is decremented appropriately. In essence, very few changes need to be made to the server to support variable rates.

When using this *variable rate* retrieval there is one restriction. The *SBVS* will allow the user to specify only certain playback rates. The reason for these restrictions has to do with the implementation of the buffer manager and the integration with *RETHER*. The *VSS* uses a double buffer mechanism and shares its buffers with *RETHER*. During normal playback data is filled into one buffer by the *VSS* while *RETHER* empties the other buffer. Problems can occur, however, if the *VSS* retrieval rate differs from the network transmission rate.

For example, let the recorded rate of the video be 120sps and the cycle time be one second. The size of each of the two *VSS* buffers is 120 sectors. Assume that the user wants the transmission rate to be 4/5 of the normal rate, i.e., 100sps. Table 9 shows how the buffers are filled and emptied during the first 3 cycles of the video. During the first cycle, buffer 0 gets filled with 120 sectors but no data gets sent out (One buffer must be filled before the data starts to get emptied). During cycle 2, buffer 1 gets filled with 120 sectors while 100 are emptied from buffer 0. At this point there are 20 sectors in buffer 0 and 120 sectors in buffer 1. During cycle 3, 20 sectors are transmitted from buffer 0 and 80 from buffer 1. Since both buffers are being emptied during the same cycle, the *VSS* is unable to retrieve any data. If it were to retrieve data, depending on the timing, it may overwrite data that had not yet been transmitted. There is now only 40 sectors left for the next cycle, which is

not enough to satisfy the client's consumption rate of 100sps.

Cycle	Fill	Empty	Buffer 0 Left	Buffer 1 Left
1	$B_0(120)$		120	0
2	$B_1(120)$	$B_0(100)$	20	120
3		$B_0(20)$+$B_1(80)$	0	40
4		$B_0(60)$+$B_1(40)$	-60	0

Table 9: *Table shows how buffers are filled and emptied during the first 3 cycles of accessing a stream whose recorded rate is 120sps and whose requested playback rate is 100sps. $B_i(N)$ means that buffer i is filled/emptied of N sectors. At the end of cycle 3, RETHER needs 100 sectors to transmit but only 40 are left.*

The problem occurred because the playback rate caused both buffers to be accessed on two consecutive cycles. For example, during cycle 3 in Table 9, data is emptied from both buffers and neither can be filled. During the next cycle, data is again to be transmitted from both buffers but there is not enough data to satisfy the request since no data was retrieved on the previous cycle. Again, this will occur when both buffers are accessed on two consecutive rounds. Thus, to be able to use a different *VSS* retrieval rate from the *RETHER* transmission rate, it must be guaranteed that the network transmission does not cross the buffer boundaries during two consecutive cycles.

The restrictions of the playback rates if double buffering is used are as follows. Any rates less than one half of the recorded rate are acceptable. Any rates over two-thirds of the recorded rate are not acceptable, and rates between one half and two thirds must meet the criteria that the playback rate must be $\frac{n+1}{2n+1}$ of the recorded rate for any positive integer n. These restrictions will now be enumerated. Let R be the number of sectors that are retrieved during a single I/O cycle and let P be the number of sectors that are transmitted during one I/O cycle. R is also the buffer size of each of the two buffers.

First, any rate where $P < \frac{1}{2}R$ is valid. Assume that during some cycle i, a piece of the network transmission of P sectors comes from buffer 0 and a piece comes from buffer 1. Let δ sectors come from buffer 0 and $P - \delta$ come from buffer 1, leaving $R - (P - \delta)$. Since $R > 2P$ there must be at least P sectors left in buffer 1. During cycle $i + 1$ all P sectors are transmitted from buffer 1 while buffer 0 is filled. Thus, it is guaranteed that the network transmission cannot access both buffers on two consecutive cycles.

Next, any rate where $P > \frac{2}{3}R$ is not valid. Let $P = \frac{2}{3}R + \delta$ where $\delta > 0$ and $\delta < \frac{1}{3}R$. During the first cycle, P sectors are transferred from buffer 0 while buffer 1 is filled. During cycle 2, $R-P$ sectors are transferred from buffer 0 and $P-(R-P) = 2P-R$ are transferred from buffer 1. No data can be retrieved since both buffers are accessed during transmission. In buffer 1 there are $R-(2P-R) = 2R-2P$ sectors left. And, $2R-2P = 2R-(\frac{4}{3}R+2\delta) =$

$\frac{2}{3}R - 2\delta$. But, since $P = \frac{2}{3}R + \delta$ buffer 1 does not have enough data to completely satisfy the transmission of data during cycle 3. Since no data could be retrieved during cycle 2, there is not enough valid data for cycle 3 and the network is starved.

Lastly, any rate where $P = \frac{(n+1)}{2n+1}R$ and $n > 0$, is valid. Figure 14 shows how this formula is derived. In Figure 14a, $P = \frac{2}{3}R$. During cycle $C1$, P sectors are transmitted from buffer 0. During cycle $C2$, $R - P$ sectors are transmitted from buffer 0 and $P - (R - P) = 2P - R$ from buffer 1, leaving $R - (2P - R) = 2R - 2P$ left in buffer 1. Since $P = \frac{2}{3}R$, $2R - 2P = 2 * \frac{3}{2}P - 2P = P$. During cycle $C3$ buffer 1 is emptied while buffer 0 is filled and the same pattern repeats. In this case, $2R - 2P = P$. Using other rates, however, $2R - 2P$ could be less than P or greater than P. If $2R - 2P < P$, then $P > \frac{2}{3}R$, which was shown to be an invalid rate. If $2R - 2P > P$ then buffer 1 has enough bytes to satisfy the transmission during cycle $C3$. This is shown in Figure 14b. While P is being transmitted from buffer 1 during $C3$, buffer 0 can be refilled. After cycle $C4$ there are $3R - 4P$ sectors left in buffer 0. In this case $3R - 4P = P$ and the pattern repeats. Thus another valid rate would be $P = \frac{3}{5}R$. If $3R - 4P < P$ there would not be enough data in buffer 0 so buffer 1 would have to be accessed. This violates the restriction that *RETHER* cannot transmit from both buffers on two consecutive cycles. Figure 14c, shows what would happen if $3R - 4P > P$. During cycle $C5$, P sectors are transmitted from buffer 0 while buffer 1 is filled. During $C6$, $3R - 5P$ are transmitted from buffer 0 and $6P - 3R$ are transmitted from buffer 1, leaving $4R - 5P$. If $4R - 5P = P$ the rate $P = \frac{4}{7}R$ is valid. If $4R - 5P < P$, the rate is invalid and if $4R - 5P > P$ it may be valid or invalid.

As can be seen, a pattern has emerged. Valid rates are $P = \frac{2}{3}R$, $P = \frac{3}{5}R$, $P = \frac{4}{7}R$, etc. In general, rates where $\frac{1}{2}R <= P <= \frac{2}{3}R$, $P = \frac{(n+1)}{2n+1}R$, and $n > 0$ for positive integer n, is a valid rate. Essentially, $R - P$ should be $n * (2P - R)$, where $R - P$ is the excess data for the first cycle and $2P - R$ is the excess every two cycles. If $R - P$ is not an integral multiple of $n * (2P - R)$, consecutive cycles of accesses to both buffers will occur, thus causing starvation.

5.7.2 Variable Rate Network Transmission

The main advantage of using a lower *steady rate* network transmission is to decrease the resources used by the network transmission. Unlike the *VSS*, using the lower *steady rate* network transmission does not increase the complexity of *RETHER*. (The *VSS* places the media blocks of a video on the disk array based on its recorded rate). However, only certain playback rates are allowed and this may not be acceptable to the end user. On the other hand, if a *variable rate* method is used, bursty traffic occurs and the network must reserve the maximum amount of resources needed during the bursty transfers. In addition, a slight problem surfaced while testing the *SBVS*. As mentioned in the last chapter, the *SBVS* pre-buffers and sends data to the client based on the average statistical bit rate of the video.

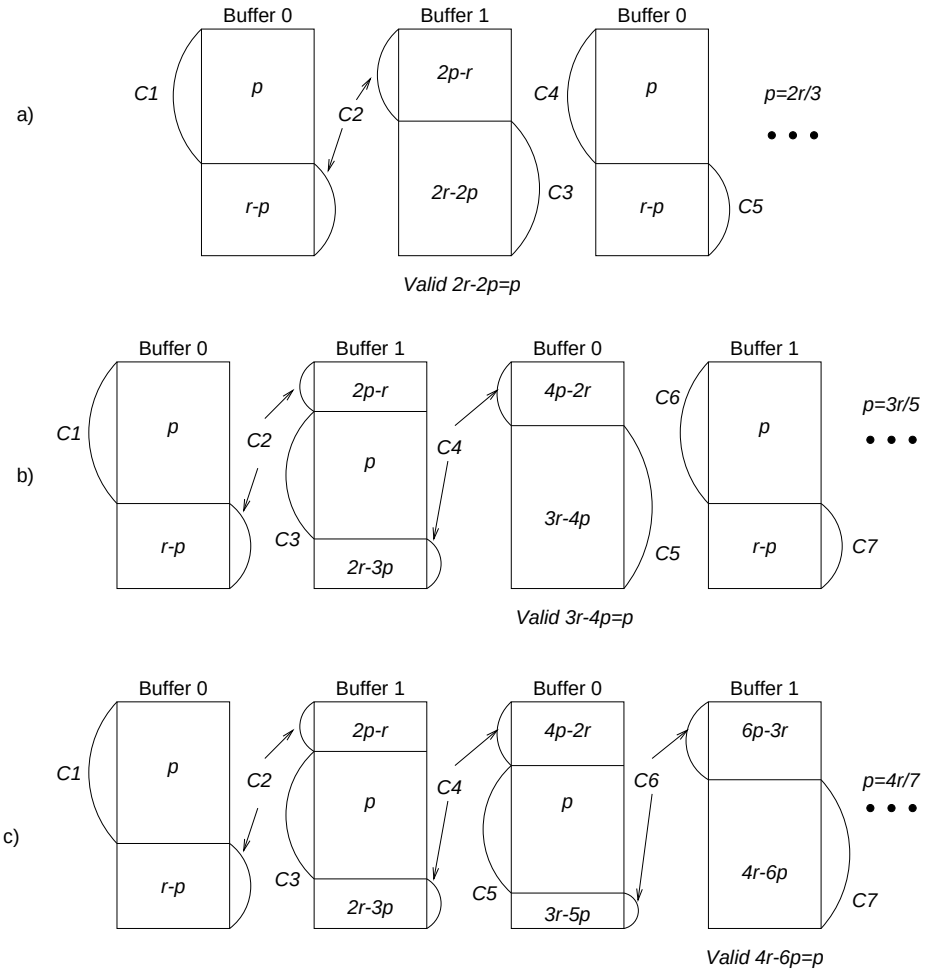

Figure 14: *Figure a shows how data is transmitted from each buffer for a playback rate P and a record rate R where $P = \frac{2}{3}R$. $C1$ is the cycle in which the data is transmitted. Figure b shows a rate of $P = \frac{3}{5}R$ and Figure c shows a rate of $P = \frac{4}{7}R$.*

It was discovered that, although the system was delivering the correct amount of data, the hardware MPEG decoder was consuming data at a faster rate than it should. Even though data was prebuffered at the client so that it would not starve, because the decoder consumed its data too fast, it eventually consumed its prebuffered amount and became starved. This resulted in slight hiccups in the display of the video. In fact, this problem would occur if the opposite case were true. If the decoder decoded its video too slow, the buffer would eventually overflow.

To allow the user unlimited playback rates and to allow for differences between the client consumption rate and the *RETHER* transmission rate, the *SBVS* also allows for a *variable rate* network transmission scheme. The normal playback rate is delivered to the client where data is buffered and consumed. Since the transmission rate from the server is greater than the consumption rate by the client, the client's buffer would eventually overflow. So that

this doesn't happen, the client periodically sends a message to the server to stop sending data for c cycles. During this time, no data is sent to the client, but the client continues to consume data and decreases the amount of buffered data. Thus, the client is allowed unlimited playback rates by controlling the server through a feedback mechanism. So that the decoder never starves during normal playback at the recorded rate, the video is stored and transmitted with a slightly higher rate than the average bit rate. If the client's buffer reaches some pre-defined high-water mark, a message is sent to the server and a cycle of retrieving and transmitting of data is skipped.

5.7.3 Summary

In summary, to allow the user to specify a consumption rate that is less than or equal to the recorded rate of the video, there are two methods. In the *steady rate* method, the network transmission rate matches that of the client's consumption rate. The advantage is that the network only reserves enough bandwidth to support the network transmission rate. On the other hand, this method only allows the user to specify certain playback rates. For unlimited playback rates, the network transmission uses a *variable rate* method where, during some cycles, the recorded rate of data is sent to the client, and during other cycles no data is sent. The client uses a feedback mechanism when it wants the server to stop sending data. The network, however, must reserve resources based on the recorded rate rather than the playback rate to accommodate the fluctuations.

5.8 Server Buffering

The *SBVS* employs a *double buffering* mechanism to stage data retrieved from disk and destined for the network. In the double buffering scheme, each stream has two buffers. During each cycle one buffer is filled while the the other buffer is empied. The roles are then reversed during the next cycle.

Each buffer must be able to hold the largest media block of the file plus any extra data accumulated due to the mismatch between the *RETHER* transmission size, which is based on the average bit rate of the video, and the *VSS* retrieval size, which must be disk sector aligned. At the end of each cycle, any extra data that was retrieved (*retrieve size + accumulated data - network transmission size*) needs to copied to the beginning of the other buffer.

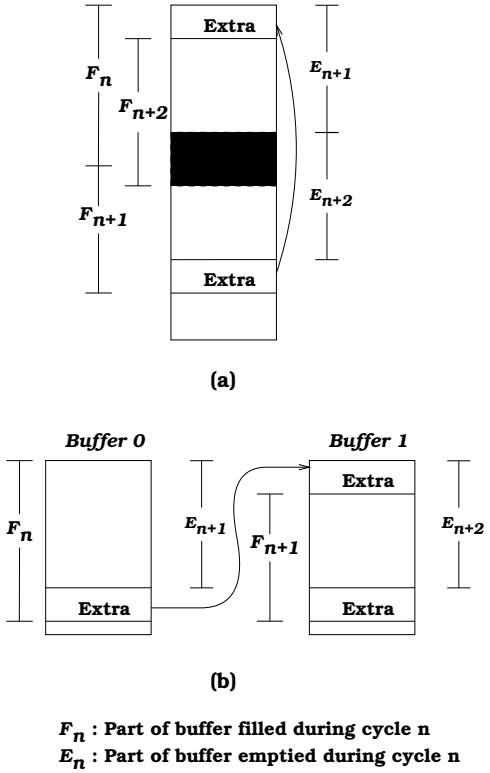

Figure 15: *Double Buffer Management. a) Using a single large buffer for both buffers; b) Two distinct buffers.*

5.8.1 Buffer Management for Normal Playback

There are two options for the implementation of the double buffering scheme. These are shown in Figure 15. The first method is to use a single large buffer as shown in Figure 15a. In this scheme the accumulated data needs only to be copied at the end of every other cycle, rather than every cycle.

Figure 15b shows the separate buffering scheme. Like its logical equivalent, data is retrieved into two separate buffers. At the end of each cycle, any accumulated data is copied to the top of the other buffer. This guarantees that the *VSS* and *RETHER* do not access the same buffer on the same cycle. If the data were not copied, *RETHER* would be transmitting data from both buffers at the same time that the *VSS* was retrieving data into one of the buffers. This could cause a problem if the *VSS* retrieved the data before *RETHER* was be able to transmit it. This is the scenario that occured during the first implementation of the buffer manager.

During the initial implementation of the buffer manager, the single large buffer scheme was used. It supported the methods for slow-motion, and only one memory copy had to be executed for every two cycles. Once this scheme was implemented, however, a major flaw was found and can be seen in Figure 15a. During cycle n, data is read into the top part of

the buffer. During cycle $n+1$, data is read into the next part of the buffer while data is emptied that was retrieved during cycle n. At the end of cycle $n+1$, any accumulated data is copied back to the top of the buffer. During cycle $n+2$, data is again retrieved into the top part of the buffer while data is emptied from the bottom part of the buffer. As can be seen in the figure, this causes problems. Since more data is retrieved than transmitted, the data that is retrieved spills over into the bottom buffer, overlapping with the data that is being sent out, as shown by the gray area in Figure 15a. If, by chance, the data gets retrieved during cycle $n+2$ before the data is sent out that was read in from cycle $n+1$, good data is overwritten and incorrect data gets transmitted, causing problems at the client decoder. The implementation was thus changed to the separate buffering scheme which guarantees that the *VSS* and *RETHER* never access the same buffer on the same cycle.

5.8.2 Buffer Management for Rewind

Figure 16 shows how the buffer manager implements rewind. During cycle F_n, data is read into the buffer so the last byte of the retrieved data occupies the last byte of the buffer. At the end of the cycle, the extra data that is not transmitted is copied to the bottom of buffer 1. During cycle F_{n+1}, the data read during cycle F_n is sent out from buffer 0, and the new data is read into buffer 1. The data is read so that the last byte retrieved occupies the byte just before the first byte of the accumulated data that was copied from the previous cycle. The *RETHER* subsystem sends data out in packets starting from the bottom of the buffer, working upward.

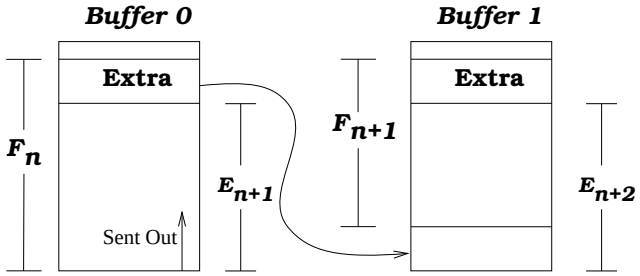

Figure 16: *Buffer management for supporting rewind*

5.9 Integration of the VSS with RETHER

The main task of the video server software is to copy data from the disk to main memory, while copying previously retrieved data from main memory to the network interface card for transmission. The network copying operation is triggered by token arrival interrupts to ensure that the server observes the token access rules of *RETHER*. Only when the token arrives will the server actually send out data in network-data units to the clients. If tokens are delayed, network-data units will be delayed at the client, resulting in poor video service. Figure 17 presents an overview of the way data is moved in to and out of the server. Within *RETHER*, a network cycle is defined by the token interrupt time and during each network cycle one network-data unit is transferred to each active client. The size of a network-data unit is determined by the token time of *RETHER* and the recorded bit rate of the video stream. As show in Figure 17 there are ten network cycles per I/O cycle. The number of network cycles per I/O cycle is a tunable parameter. If the I/O cycle is one second, each network cycle is 100ms. If the average bit rate of the stream is 180KBps, data is sent to the client at 10 network-units per second and the network-data unit is 18KB. Logically, during each network cycle, the I/O system transfers one network-data unit for each stream from the disk into the associated buffer. In practice, however, the small size of network-data units makes it infeasible to retrieve only one network-data unit of data per disk read. Thus, the server retrieves 10 network-data units for each stream during 10 network cycles. So the long-term average is still one network-data unit for each stream per network cycle. This approach offers the advantage of better utilization of disk transfer bandwidth, since the overhead is one, rather than N, seeks and rotations per stream per I/O cycle.

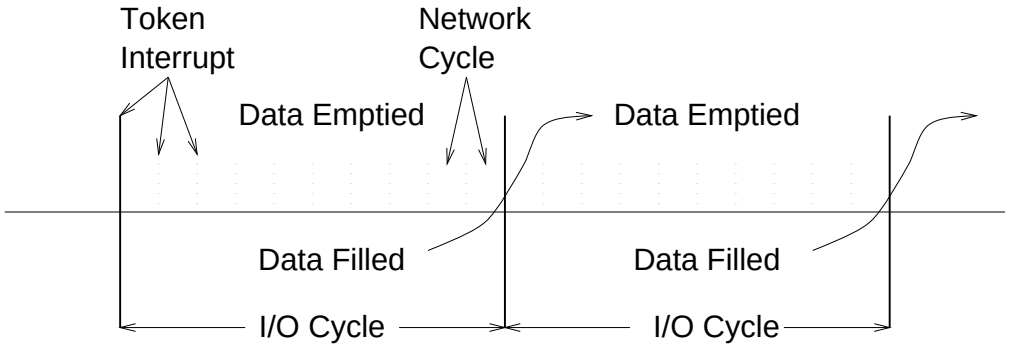

Figure 17: *Coordination of RETHER and VSS.*

5.9.1 Integration in the SBVS-1

The first prototype, *SBVS-1*, was an EISA-based machine connected to a 10Mbps Ethernet with a DMA-based SCSI controller and programmed I/O network interface card. The disk,

the Ethernet link, and the EISA bus all have sufficiently high rated transfer bandwidth for our performance target—5 MPEG-I video streams or 7.5Mbps. So, in the very first implementation, we assumed that the high bandwidth of the EISA bus would be sufficient to allow the I/ O and networking transactions to compete for the bus without any performance penalty. However, the results were far from satisfactory. If an I/O transfer over the EISA bus occurs while a token interrupt is generated, the interrupt is not serviced until the transfer is complete, thus delaying the token and affecting the network service. In addition, the CPU uses programmed-I/O to copy data from memory to the network card. This also tends to be delayed because of bus contention. It was thus necessary to schedule the use of EISA bus more carefully.

Figure 18 illustrates the timing details of the integration between the *VSS* and *RETHER* in the *SBVS-1*. The key idea in the I/O bus scheduling is *overlapping disk seek, rotation and transfer of data from the disk medium to the disk cache, with network data copying.* This requires the SCSI-2 *prefetch* command[3]. At the beginning of each network cycle the *VSS* issues a prefetch command for one active stream. When a disk receives this command it moves the disk arm to the appropriate track and transfers the data from the disk medium to the disk buffer without transferring the data over the SCSI or I/O bus. While the disk performs its prefetch operation, the network transaction has complete control of the EISA bus to copy data from memory to the network card. At the end of the network copying operation, the target data, which has been transferred to the disk drive's internal buffer, can be read over the EISA bus to main memory without incurring contention delays.

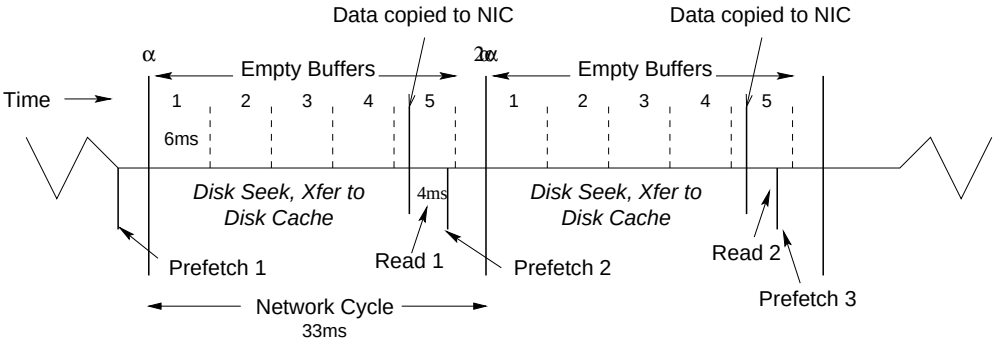

Figure 18: *Coordination of RETHER and VSS in the SBVS-1.*

Even if the data to be read from the disk incurs the worst-case seek and rotation it can be guaranteed that the data is ready to be read from the disk bus when *RETHER* finishes

[3]The particular disk drive being used does not support the prefetch command, but performs prefetching on its own. When a disk read is complete, the disk continues to read data into its buffer until an entire track is transferred or a new request is queued. To simulate prefetching, a read request of 512 bytes is sent to the disk. The 512 bytes are transferred over the EISA bus while the disk continues to prefetch data automatically. This small transfer over the EISA bus did not cause any bus contention.

emptying the data buffers for each network cycle. To make this guarantee, data from only one active stream is retrieved during each network cycle. Data is then read in a round-robin manner during each successive network cycle. For example, if there are five active streams, data for stream 1 is read during network cycle i, data for stream 2 is read during cycle $i+1$, etc. The size of an I/O cycle is five network cycles or 165ms. As shown in Figure 18, each network cycle is 33ms. Considering an I/O cycle of 165ms and an average playout rate of 180KBps, each media block to be retrieved is 30KB. There are 5 network cycles per I/O cycle so 6K is transmitted for each stream every network cycle. During each network cycle it was measured to take 24ms to transmit the first 4 streams over the network. Sending data to a client from the server incurs two operations, the copying of data from memory to the network interface card (NIC), and the copying of data over the physical wire. It was measured to take 2ms to send 6KB of data from memory to the NIC. Once the data for the 5th and last stream has been copied over the EISA bus (26ms from the beginning of the network cycle), the *VSS* can start to retrieve its data from the disk buffer. It was measured to take 4 ms to retrieve 30KB from the disk buffer. Once the data has been transferred from the disk buffer to main memory, the prefetch command is issued for the next I/O cycle. This occurs 30ms after the initiation of the last network cycle. Thus, the disk must be able to guarantee that it can seek, rotate and transfer its data to the disk buffer within 29 ms (3 ms from previous cycle plus 26ms from current cycle). This is certainly within acceptable ranges of current hard-disk technology.

5.9.2 Integration in the SBVS-2

The *SBVS-2* uses a Pentium-90 processor and a PCI bus. The rated transfer bandwidth of the disk, the Ethernet link, and the PCI bus are all sufficiently high for our performance target.

Although it was necessary to explicitly schedule the use of the EISA bus in the implementation of the *SBVS-1*, it was not necessary to perform such scheduling in the *SBVS-2*. By using a PCI bus which is rated at 132MBps, a DMA-based network interface card and a DMA-based SCSI controller, token times were not affected, even while data is being copied from the disk system to memory.

Figure 19 depicts a time line of the interaction between *RETHER* and the *VSS*. Below the time line, streams are read into one of two buffers. Above the time line, network-data units are delivered out of the other buffer. For example, in the figure, five streams are being serviced. At times $t0, t1, t2$, and $t4$, a *RETHER* token arrives and one network-data unit for each stream is delivered. At time $t3$, all streams from buffer 1 have been delivered and

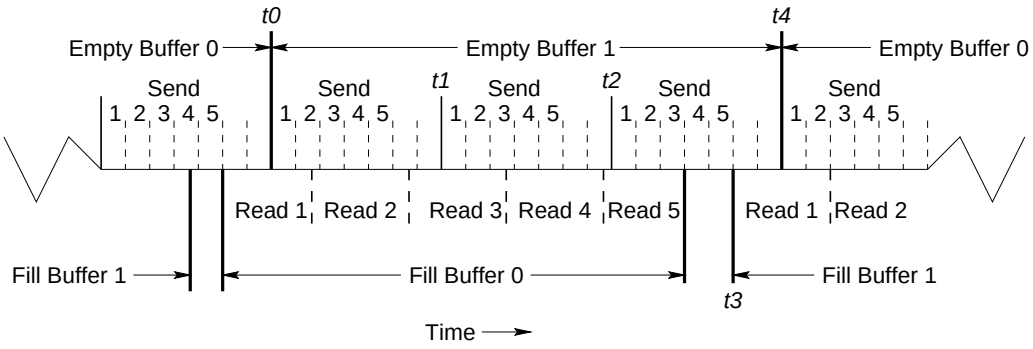

Figure 19: *Coordination of RETHER and VSS in the SBVS-2.*

RETHER wakes up the *VSS* to begin filling buffer 1 which has just been emptied.[4]

5.9.3 Serving Streams on Multiple Ethernet Segments

In addition to delivering data to clients on a single shared Ethernet, the *RETHER* subsystem is able to deliver videos to clients on different Ethernet segments. But, as far as the *VSS* is concerned, it is unaware of the network over which each of the admitted streams has to be served. The information about the network to which streams are delivered is maintained by *RETHER*. There is, however, one issue that must dealt with when integrating the *VSS* and *RETHER* when *RETHER* is serving multiple segments. This section discusses this issue.

As described previously, the network subsystem drives the *VSS* by supplying a periodic wakeup signal at the beginning of every I/O cycle for the *VSS* to retrieve data from the disk for all the streams. As shown in Figure20a, the *VSS* has a cycle time of P_{io} and each I/O cycle corresponds to four network cycles of duration $\alpha = \frac{P_{io}}{4}$. At the end of the last network cycle *RETHER* issues the wakeup signal. This occurs at times $t0, t1$, and $t2$ in the figure. Between times $t0$ and $t1$, data is retrieved from storage to memory. Between times $t1$ and $t2$, the data is delivered to the clients. As long as the *VSS* guarantees that data is available at time $t1$, the timing is correct.

If the *SBVS* begins to service streams on a second network, this timing may become slightly skewed. *RETHER* does not guarantee that the network cycles amongst multiple networks are synchronized. The skew between networks can be seen in Figure 20b. A second network, Net2, starts delivering data at time $t0$. Net1 finishes its last network cycle at time $t1$ and Net2 finishes its last network cycle at time $t2$. The difference between $t1$ and $t2$ is the network *skew*. *RETHER* guarantees that the maximum skew between N networks in one network cycle or α.

As can be seen, the *VSS* can start a new I/O cycle only after the last stream on the

[4]In reality, each stream has two buffers which are not synchronized. For example, during some cycle, stream v_1 can be filling its buffer 0 while stream v_2 can be filling its buffer 1.

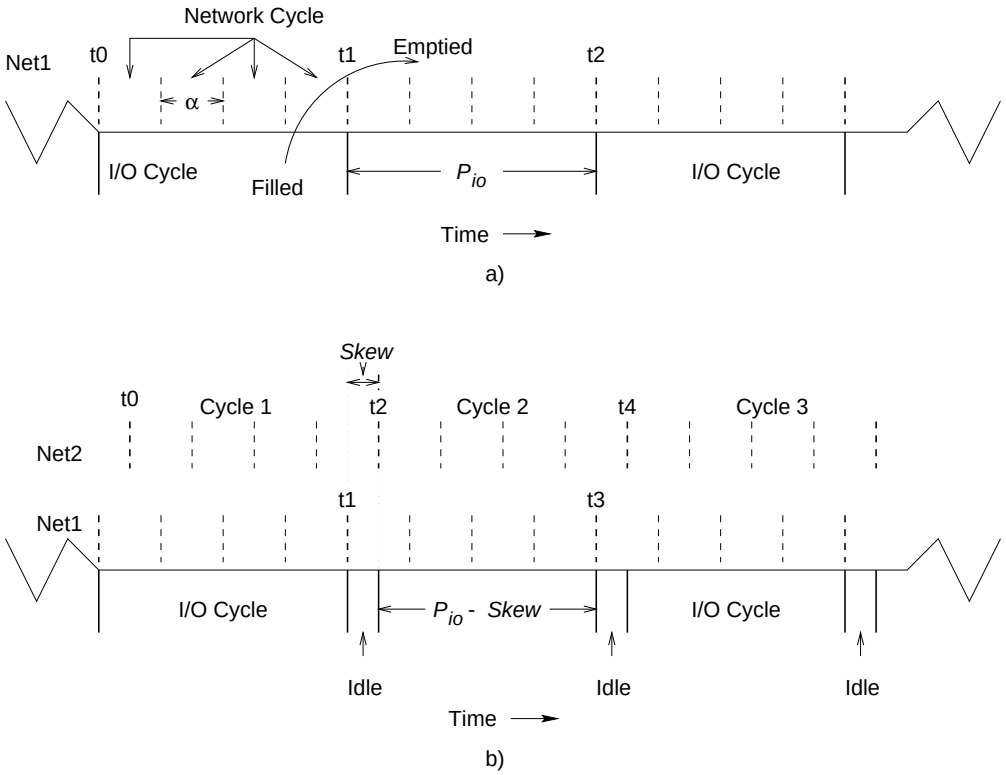

Figure 20: *a) SBVS serving streams on a single network segment. b) SBVS serving streams on two network segments. This introduces a skew between the token arriveal times on the multiple segments.*

last network has been serviced. The last network is the network with the latest ending last network cycle. This guarantees that *RETHER* has enough time to finish emptying its buffers before being overwritten by new data. In the figure this occurs at times $t2$ and $t4$ since Net2 is the last network. In addition, the *VSS* must retrieve the data for the next I/O cycle before the first stream on the first network requires data for that cycle. This guarantees that the first network has valid data when it starts to deliver it. This occurs at time $t1$ and time $t3$. Thus, during cycle 2, the *VSS* can only retrieve data between times $t2$ and $t3$. Since the time between $t1$ and $t3$ is P_{io} or the original cycle time, and the time between $t1$ and $t2$ is the *skew*, the retrieval time, or effective I/O cycle time, becomes $P_{io} - skew$. Since the maximum skew for N networks is the network cycle time α, in the worst case, the *VSS* has one network cycle less time to fetch data than it had when there was only one network segment being served. Thus, when servicing multiple networks the *VSS* must guarantee that it takes at most $P_{io} - \alpha$ to service all of the streams during one I/O cycle. This is the time that the *VSS* admission control uses when determining if a new stream can be added. If there is only one network being serviced, the admission control system uses P_{io} as its upper bound. Admission control will be discussed in Chapter 7.

5.10 Other Implementations

Most of the work regarding distributed multimedia servers has been analytical in nature or based on simulations. However, there have been several papers that discuss the implementation of prototype video servers. The IBM LAN Server is presented in [Bau93]. Built to work over a token ring network, the system uses a *client-pull* style of delivery and can distribute up to 28 1Mbps video streams over three 16 Mbps networks. Microsoft's Tiger Video File server is described in [B+96]. Like the *SBVS*, Tiger uses uses commodity computing hardware. The goal of Tiger is to distribute a large number constant rate bit streams over a switched network, such as ATM or switched Ethernet. Tiger provides fault-tolerance by mirroring disks. To display videos that have different bit rates, i.e., MPEG-1 vs. MPEG-2, one would need a separate server for each bit rate to be delivered. In [NT94], the implementation of a video server that was built on top of a real-time file system is presented. This system also uses a *client-pull* style of delivery where I/O requests to the operating system include a deadline. Lastly, the Starlight StarWorks product [Sta96] uses a RAID based system to provide networked video. But, because it is a proprietary software system, there has been little published about its implementation.

To provide real-time delivery of data, each of these systems assumes that the network is completely controlled by the server. Unlike the *SBVS*, these systems cannot allow other real-time or non-real-time connections to coexist. An example would be an Ethernet switch where the server has a dedicated fast interconnect (100Mbps) to the switch, and each user is also connected to the switch. Thus, each client has its own dedicated 10Mbps Ethernet LAN. Unfortunately, even in this set-up, real-time performance is not necessarily guaranteed.

Chapter 6

Performance Analysis of the SBVS

6.1 Performance Goals

The main goal of the *SBVS* is to maximize the available bandwidth of an off-the-shelf PC-based system and network. There are two main objectives for measuring the performance of the *SBVS*. The first objective is to determine the maximum available bandwidth of each of the hardware components. Then measurements of a running *SBVS* video server can be compared to the maximum measurements of the individual components to determine the percentage of maximum throughput that can be achieved. The second objective is to determine the maximum number of streams that can be supported under different hardware configurations with varying system parameters.

The results of three sets of experiments are reported in this chapter. First, the maximum throughput of individual components (*VSS* and *RETHER*) of the *SBVS* will be measured in isolation. Second, these components will be measured using the *SBVS* software running actual video server code to determine the maximum number of streams that each component can accommodate. Lastly, the performance of the entire *SBVS* will be measured. This includes both the storage and network components working together. The goal is to identify potential performance overhead when system components are integrated into a complete system.

6.2 Performance Evaluation of the SBVS-1

The first prototype, *SBVS-1*, was implemented on an Intel 486-66DX2 with 32MB of RAM and an EISA bus. It connects to five clients over a 10Mbps Ethernet. The server has a single, Conner SCSI-II disk.

The objective of the first experiment is to determine the maximum attainable bandwidth

on the Ethernet given the hardware and device driver overheads. Packets, equal in size to an Ethernet MTU (Maximum Transmission Unit = 1500 bytes), are transferred from one node to another and the time to transmit 2000 packets is measured. The transmission times were measured by reading the time at the beginning of processing the first packet, until after receiving the interrupt indicating the completion of transmission of the last packet of the message. The results of 7.5Mbps showed that we are able to obtain only 75% of the Ethernet bandwidth due to software and hardware overheads.

In the next experiment, the single SCSI disk was measured to have a transfer rate of 1.5MBps on the inner portion of the disk and about 3MBps on the outside portion of the disk. Thus, the bandwidth of the disk is enough to saturate a 10Mbps network.

Lastly, the *VSS* and *RETHER* were assembled into a working system. When running the end-to-end *SBVS-1*, we found that a maximum of 5 MPEG-1 streams at 1.5 Mbps could be supported with a network utilization still at 75%. Thus, the integration of the *VSS* subsystem with the *RETHER* system had no effect on the maximum number of streams supportable by the network. See section 5.9.1 for the implementation details of integrating the *VSS* with *RETHER*.

6.3 Performance Evaluation of the SBVS-2

The second prototype, *SBVS-2*, is based on an Intel Pentium 90MHz CPU with 32MB of RAM and a PCI bus. It uses a SCSI-II-based disk subsystem with three Adaptec PCI Fast SCSI-II bus controllers and six Quantum Fireballer 1 Gigabyte disks. The SCSI bus transfers data at a rate of 10MBps. The network is a 100Mbps Fast Ethernet.

6.3.1 Video Storage Subsystem Performance in Isolation

The objective of the following experiments is to determine the maximum attainable performance of the *VSS* without the network subsystem. See [LK93] for a discussion of maximizing performance in a disk array. In each of the following experiments, a benchmark program within the kernel is responsible for sending SCSI read commands directly to the SCSI adapter device driver. All normal file system code is bypassed. When disk requests finish, the adapter device driver calls an interrupt routine within the benchmark program, again bypassing any file system code. As soon as the service of a disk request finishes, the next request for the disk is immediately sent. At the start of each experiment, two commands are sent to each disk. Since the disks cannot queue commands, one command is sent to the disk, while the other command waits at the SCSI adapter, which does allow commands to be queued. When a disk is finished processing a command, it can retrieve its next command from the adapter,

without waiting for the OS to process the next request. While the disk processes its next command, the OS queues another command to the adapter. Also, in each of the experiments, the disks are not spindle-synchronized.

6.3.2 Maximum Performance of Disk Array

The goal of the first experiment is to determine the maximum available bandwidth of a SCSI bus. By repeatedly requesting the same data from each disk, no seek or rotation overheads are incurred by the disk since each request can be satisfied by the cache on the disk. Although the bus transfers data at a rate of 10MBps, the actual transfer rate will be lower because of command overhead, arbitration for the SCSI bus, etc.

	# of Controllers/Total # Disks										
Size	**1/1**	**1/2**	**1/3**	**1/4**	**1/5**	**2/2**	**2/4**	**2/5**	**2/6**	**3/6**	**4/8**
80KB	8.2	8.5	8.5	8.5	8.5	16.5	17.0	16.2	16.9	25.5	35

Table 10: *Maximum throughput of disk array in MBps for varying numbers of bus controllers and disks. Data is read directly from disk cache.*

Table 10 shows that the maximum bandwidth that can be achieved on a single bus is 8.5MBps with a retrieval size of 80KB. Since this experiment reads data directly from the disk cache, the size of the cache determines the largest retrieval size that can be retrieved.[1] Theoretically, as the retrieval size increases, the overhead would decrease proportionally until a maximum rate of 10MBps would be achieved with an infinite retrieval size. As bus controllers are added to the system there is a linear speedup. For example, retrieving 80KB from each of 2 disks with 1 bus controller yields 8.5MBps while retrieving 80KB from each of 8 disks with 4 bus controllers yields 35MBps.

	# of Controllers/Total # Disks								
Size	**1/1**	**1/2**	**1/3**	**1/4**	**1/5**	**2/2**	**2/4**	**2/6**	**3/6**
30KB	1.5	3.0	3.7	4.5	5.1	3.0	5.9	6.7	9.1
60KB	2.3	4.3	5.4	6.0	6.3	4.5	8.9	9.7	13.4
90KB	2.9	4.7	5.5	6.2	6.4	5.4	9.0	10.6	13.7
120KB	3.0	5.2	5.9	6.3	6.6	6.0	9.9	10.9	15.3

Table 11: *Total disk array throughput in MBps for varying numbers of bus controllers, disks and retrieval sizes. Requests are continuously streamed to disks and evenly distributed across all tracks.*

Table 11 presents the results of experiments where the data is read from the disk medium, rather than the disk cache, using varying numbers of disks and bus controllers. Reads are

[1] The Quantum Fireballer has an available cache of 80KB.

performed to varying places on each disk beginning at the outer edge of the disk and working inward. Forty reads are performed during each outward to inward sweep and each read is equidistant from each other, thus maximizing the seek time between each read. The idea is to model the random access of videos. Once a sweep is finished, the disk starts again at the outer edge.[2]

Requests are streamed to each disk so that as soon as a disk finishes a read, another command is sent immediately back to the disk. Using five disks on a single SCSI bus, and a retrieval size of 120KB the maximum bandwidth is only 6.6MBps. Using five disks we had hoped to achieve the same 8.5MBps as in the previous disk cache experiment, since the seeks and rotations of each disk would be masked while the other disks are transferring data. For example, without loss of generality, assume that Disk 1 in the disk array receives a read command. Before Disk 1 can access the SCSI bus to transfer its data, Disks 2 thru 5 also must transfer their own data since it is assumed that each disk is always working on a read command. While Disk 1 seeks, rotates, and transfers its data into its cache, Disks 2 thru 5 will be transferring data over the SCSI bus. From the previous experiment it was determined that it takes about 7ms to transfer 64K from the disk cache into system memory. Thus, it takes about 28ms for Disks 2 thru 5 to transfer all of their data. On average it takes Disk 1 about 12ms to seek to its correct track and then at most 11.1ms (1 complete rotation) to transfer its data into the disk cache. The Quantum Fireballer begins reading data as soon as it gets to the correct track so it can take at most one complete revolution to transfer its data from the medium into the cache. It only takes Disk 1 about 23ms to have its data ready to transfer but it takes the other four disks 28ms to hold transfer their data. In essence, by using enough disks, seeks and rotations should be masked while other disks in the array spend time transferring.

However, this was not the case. Once a disk finishes transferring its data over the bus it must then retrieve its next command (assuming there is another command to be executed). The command may already be in the disk controller, the command may be waiting in the SCSI bus controller, or the command may be waiting in an OS data structure queue. If the command is in the SCSI bus controller or the OS, the disk must acquire the SCSI bus before it can get its next command. Since the disk transfers are large, the disk must wait in an idle state. The Quantum Fireball cannot queue commands, so in addition to a disk seeking, rotating, and transferring data, it may also have to wait until it can actually begin executing its command, thus increasing the total time to complete a read command. This hypothesis was validated by a disk bus simulation. Using measurements of the Quantum Fireball disk, our simulation showed 8.7MBps maximum utilization with no seeks and rotations compared to 8.5MBps of empirically measured utilization. However,

[2]Increasing the reads per sweep above forty had negligible effect on throughput.

the simulation yielded only 6.8MBps with seeks and rotations, compared to 6.6MBps of measured utilization. We then changed the simulation to model a disk controller that can queue commands. At the beginning of the simulation, two commands are sent to each disk and as each read completes another command is immediately sent to the disk. Thus, the disk always has at least one command in its disk queue. Using this model, the simulation showed a maximum bandwidth of 8.7MBps. Thus, we recommend that to achieve the best utilization of the disks, one should purchase disks that can queue commands. Using the video server software, at the beginning of an I/O cycle all reads should be sent to all disks, allowing the commands to queue up while the disks begin to seek to the first stream. Thus, as soon as a disk finishes one command, the next command is already in its queue.

	# of Controllers/Total # Disks								
Size	**1/1**	**1/2**	**1/3**	**1/4**	**1/5**	**2/2**	**2/4**	**2/6**	**3/6**
30KB	1.3	2.7	3.4	4.0	4.5	2.6	4.9	6.7	7.4
60KB	2.1	4.2	4.9	5.4	5.8	4.2	7.4	9.5	11.1
90KB	2.7	4.6	5.3	5.6	5.9	5.3	8.4	10.4	12.5
120KB	3.0	5.1	5.5	5.9	6.1	6.0	9.3	10.6	13.9

Table 12: *Total disk array throughput in MBps using the VSS algorithm for varying numbers of bus controllers, disks and retrieval sizes. Data is read in cycles where each cycle consists of 40 reads per disk, evenly distributed starting from the outside track to the inside track. A new cycle does not begin until all reads from the previous cycle complete.*

Table 12 shows the throughput of the system when the *VSS* algorithm is used. In the previous experiment, read commands are sent to the disk as soon as the previous read finishes—for the entire experiment duration. However, the video server reads data in I/O cycles. It must wait at the end of each cycle for all retrieves to finish before it can start the next cycle. Otherwise, the network may not have transmitted its data before being overwritten with new data. During each cycle, however, reads are sent to the disk as soon as the previous read finishes. Thus the total time for each cycle is the sum of the times of the slowest disk for that cycle. The results from Tables 11 and 12 show that as the number of disks increase, the loss in disk throughput increases. Using 6 disks with 3 bus controllers, the video server algorithm attains about 90% of the maximum bandwidth.

6.3.3 Maximum Number of Streams that the I/O System can Support

The objective of these experiments is to determine the maximum number of streams that the disk subsystem can accommodate under two different hardware configurations and under varying I/O cycle times from .5 seconds to 10 seconds. It is assumed that each stream has

an average bit rate of 180KBps. During each run 6 movies are evenly distributed across the disks. The active streams access the videos in sequential order such that disk accesses are also evenly distributed across the disk. Again, the I/O cycle time determines the amount of data that must be retrieved for each stream during each cycle. For example, assuming an I/O cycle time of one second, 180KB must be retrieved for each stream during each cycle. Each experiment is run for 1 hour and the time is measured for each cycle. If any cycle takes longer than one second then that number of streams cannot be accommodated. Thus, the number of streams is decreased until all measured cycle times during the 1 hour are less than one second. This is conservative in that the system allows no missed deadlines.

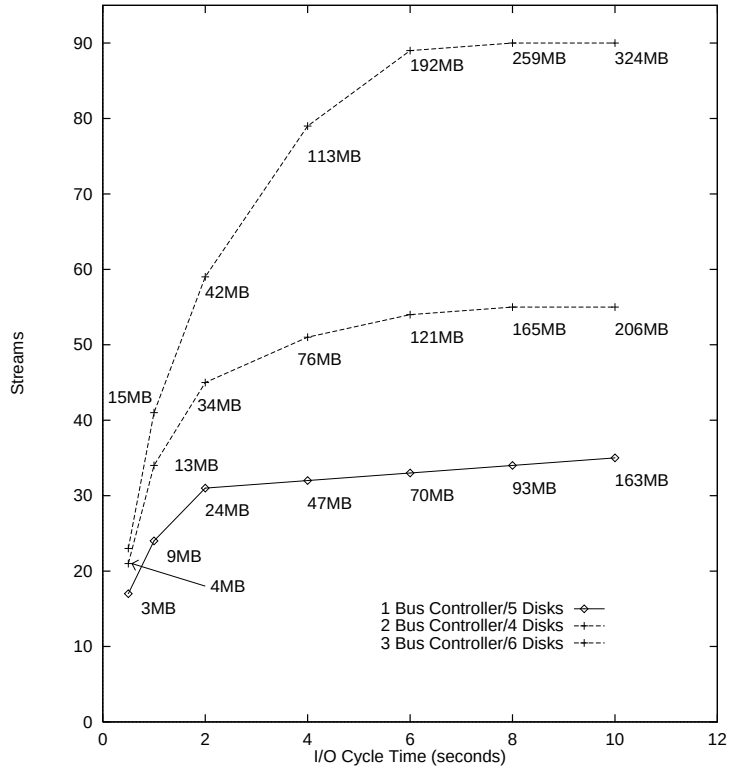

Figure 21: *Maximum number of streams that can be accommodated by disk subsystem. Numbers adjacent to data points specify the amount of physical memory needed to support the associated number of streams.*

Figure 21 shows that a maximum of 35 streams can be accommodated on a single SCSI bus controller with five disks. For a cycle time of 2 seconds, 31 streams at 1.5Mbps require 24MB of physical memory. 55 streams can be accommodated with two bus controllers and four disks using an I/O cycle time of 8 seconds. However, increasing the I/O cycle time increases the amount of data that must be retrieved and stored in memory, increasing the physical memory requirements of the server. In addition, increasing the I/O cycle increases the startup latency at every client. This figure also shows the minimum amount of physical

memory needed to support the maximum number of streams using a double buffering scheme.

6.3.4 Network Performance in Isolation

The goal of the next set of experiments is to measure the performance of the Fast Ethernet network subsystem in isolation. Like the previous I/O experiments, the network experiments were performed entirely in the kernel, bypassing any high-level network protocols.

The objective of the first experiment is to determine the maximum attainable bandwidth on the Fast Ethernet given the hardware and device driver overheads. This experiment was set up similar to the experiment on the 10Mbps Ethernet described in section 6.2. The results of 70Mbps showed that we are able to obtain only 70% of the Ethernet bandwidth due to software and hardware overheads.[3].

The objective of the next experiment is to measure the maximum number of *RETHER* streams that can be supported. The number of streams is determined by the maximum throughput of the network and the *RETHER* interrupt overhead per stream. In the worst case the *RETHER* token may interrupt the server once for every stream. This occurs when there are other real-time applications, such as video-conferencing, also reserving bandwidth. In this case, the token interrupt overhead reduces the maximum number of possible streams. *RETHER* can support 41 MPEG-1 streams at 1.5Mbps with a token cycle time of 33ms, with one interrupt per stream. With only a single interrupt for all the streams, 45 streams can be supported. To measure the worst case, the token is bounced back and forth between two node. Each node processes real-time streams alternately, thereby causing one interrupt per stream. These experiments yielded a network bandwidth of 67-68Mbps, which corresponds well with the network throughput of 70Mbps that was measured in the previous experiment. The loss in bandwidth of around 2Mbps is due to sending data in network-data units which incurs a per-stream protocol processing overhead.

Bit-rate (Mbps)	Num Streams
1.5	45
3.0	22
4.5	15
6.0	11

Table 13: Maximum Number of streams supported by RETHER with one network interrupt for all streams.

The objective of the next set of experiments was to measure the number of streams that can be supported with only one interrupt per stream with varying bit rates and varying

[3]This number was similar to the measurements found in [Con95]

token cycle times. It turns out that the token cycle time had no effect on the number of streams for the following reason: Although one interrupt occurs in the 132ms token cycle case compared to four interrupts in the 33ms token cycle case, the savings in token overhead is not sufficient to support any additional streams at these bit-rates. However, more streams may be supported at smaller bit-rates. The maximum number of streams is indicated in Table 13. The full details of *RETHER* and its performance evaluation can be found in [Ven96].

6.4 SBVS-2 Integrated System Performance

Whereas the two previous sections presented the results in terms of the maximum number of streams that the individual network and storage components can support, this section presents performance results of an integrated *SBVS* system.

In the setup for the following experiments, the network hub that is used by the Fast Ethernet can only support a maximum of 16 nodes. The experiments, however, may run many more clients. This is done by having several nodes read multiple streams using multiple processes. [4]

6.4.1 Maximum Number of Streams that SBVS-2 can Support

The goal of the following experiments is to determine the maximum number of streams that can be serviced under varying hardware configurations and video bit rates. The maximum number of streams is determined by two numbers, the token cycle time of the *RETHER* subsystem and the I/O cycle time of the disk array subsystem. See Section 5.9 for a description of these two values. During each network cycle, each client is sent one network-data unit. The network cycle time is a system parameter. To determine the maximum number of streams, the number of streams is increased until either all of the streams cannot be processed within a network cycle or I/O cycle.

When running the end-to-end *SBVS*, we found that a maximum of 45 streams could be supported when using two bus controllers and five disks assuming an average bit rate of 1.5Mbps per stream. 31 streams can be supported when using 1 bus controller and five disks. The maximum of 45 streams with two bus controllers in the integrated system corresponds directly to the maximum number of streams that can be supported by the network running in isolation. The I/O bandwidth of 80Mbps in this case (Table 12) is sufficient to saturate the network and the network becomes the bottleneck. Thus, the integration of the I/O subsystem with the *RETHER* system had no effect on the maximum number of streams supportable by

[4] The nodes, however, do not have enough CPU to also display the data.

the network. Conversely, the maximum of 31 streams with one bus controller corresponds directly to the maximum number of streams that can be supported by the I/O system in isolation. The maximum I/O bandwidth for this configuration is 48Mbps, well below the saturation point of 70Mbps for the network. Again, integration of the two subsystems did not have any impact on the network throughput. The CPU, memory system, and I/O buses are able to process the flow of data.

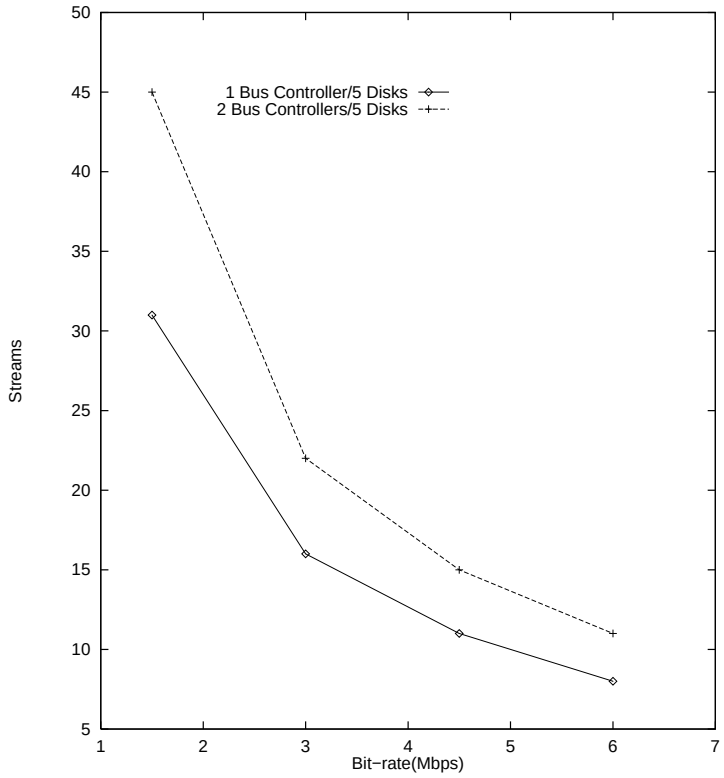

Figure 22: *Maximum number of streams that can be accommodated by* SBVS *with varying bit rates.*

The objective of the next set of experiments is to to measure the number of streams that can be supported with varying bit-rates and to determine if there was any per-stream overhead. Figure 22 indicates that the maximum number of streams supportable decreased linearly with increasing bit rates, indicating that the network utilization was unaffected by the different bit-rates. For example, using two bus controllers and 5 disks, enough to saturate the network, 45 1.5Mbps streams can be supported. Doubling the bit-rate to 3Mbps decreases the number of streams by half, to 22.

A number of experiments were run with different token cycle times of 66ms, 99ms and 132ms. It was found that the number of streams that could be supported for the different bit-rates remained unaffected.

When utilizing a disk array that can saturate the network, measurements showed that the

network throughput was a steady 67-68Mbps compared to a maximum attainable throughput of 70Mbps. Varying cycle times and bit-rates had no effect. The low overhead of the I/O and *RETHER* subsystems allow us to efficiently utilize the available hardware bandwidth.

6.5 Minimum Configuration Requirements For One Network

The minimum hardware configuration needed the *SBVS* system includes a disk array that can saturate the maximum allowable network bandwidth of 70Mbps, a sufficient amount of physical memory that can support the maximum number of streams, and a processor fast enough to efficiently run the software algorithms. Table 12 shows that a minimum of two bus controllers, each with two disks, has a throughput greater than 70Mbps running the video server algorithms. However, by adding disks, higher throughput can be achieved, thus decreasing the minimum I/O cycle time required to process all of the streams. When the minimum I/O cycle time is decreased, the physical memory requirements are also decreased. There is a trade-off between adding disks for throughput or adding memory and allowing an increase in cycle times to increase the maximum supportable number of streams. Table 14 shows the results of three experiments to determine the minimum cycle time required for four, five or six disks using two bus controllers. The required physical memory to support this cycle time is also presented. Each experiment was run assuming an average bit rate of 1.5Mbps. Although adding disks does lower the cycle time and physical memory requirements, it is not significant. In this case, adding 2 more disks at a cost of $200 each, would save only 3MB of required memory, approximately $20 (assuming 4MB chips are purchased). Note that although 24MB of physical memory is the minimum required to support 45 streams using a double buffering scheme, more memory will be needed depending on the operating system being used.

	Disks		
	4	**5**	**6**
Cycle Time in Sec	1.66	1.60	1.50
Memory Required in MB	27	25.9	24

Table 14: *Minimum cycle time and associated required physical memory for two bus controllers and four to six disks.*

As for processor speeds, we have only experimented with a Pentium 90MHz CPU. This processor was quite sufficient to handle our workloads on a single Ethernet segment. This, however, was not the case when supporting nodes on two Ethernet segments and is discussed

in the next section.

6.6 Scalability

The previous experiments show that the current bottleneck in the *SBVS* is the single Fast Ethernet network that it serves. This section examines the issues in scaling the *SBVS* to support more streams. Since the bandwidth on the Ethernet cannot be increased, the next logical alternative is to support multiple Ethernet segments. Like the previous experiments using a single network, the objective of the next set of experiments is to first measure the *RETHER* component in isolation and then measure a fully assembled system. A second Fast Ethernet interface card was added to the server such that the server had two network cards and three SCSI busses, each bus having two SCSI disks.

Like the experiments detailed in section 6.3.4 the objective of the first experiment is to determine the maximum attainable bandwidth on two Ethernet segments with no *RETHER* overhead. Data is transferred from one node to two other nodes on two separate networks and the time to transmit all of the data from the sender end is measured. When transferring data over two separate networks the maximum attainable transfer bandwidth was measured to be 125.5 Mbps. Compared to the single network maximum bandwith of 70Mbps, the improvement was not linear. There are two possible reasons for the non-linear performance, CPU utilization or memory/bus bandwidth. In section 6.3.2, however, the I/O system was reported to be able to transfer 35MBps of data into memory. Thus, the bottleneck is the CPU.

The objective of the next experiment is to measure the maximum number of *RETHER* streams that can be supported on the two networks. Measurements showed that 81 streams (1.5Mbps) could be supported—about 121.5Mbps. This is slightly lower than the 125.5Mbps measured in the previous experiment. This is due to the additional *RETHER* overhead introduced into a system whose CPU is already saturated.

In the next set of experiments the objective is to measure the maximum number of streams that can be supported in an integrated system with two networks. To saturate two networks running at about 125Mbps, the *SBVS* must have a disk subsystem that can deliver 125Mbps into memory and a memory subsystem that can handle $2 * 125$ or 250Mbps (125Mbps written into memory by the *VSS* and 125Mbps read out of memory by *RETHER*). Again, memory can support at least 35MBps or 280Mbps. Using three bus controllers and six disks, as shown in Table 12, the I/O system can support only 13.9MBps or 111Mbps. Thus, the current configuration does not have enough bandwidth to support 125Mbps. To increase the bandwidth of the I/O, there are several alternatives. The first alternative is to use disks that can queue commands. The results in section 6.3.2 show that by using disks

that can queue commands, a total of 25.5MBps or 204Mbps can be supported using three bus controllers and six disks. Another alternative would be to simply use faster SCSI buses and disks. The current *SBVS* configuration supports 10MBps buses and disks. Fast-Wide SCSI, however, can support speeds of 20MBps and Ultra SCSI can support speeds up to 40MBps. Thus, it is easy to increase the speed of the I/O system with faster disks and/or SCSI bus controllers. Due to our budget, however, we were constrained to 10MBps buses and disks. To increase the bandwidth of the SCSI bus for our experiments without incurring any extra costs, the experiments always reads data from the disk cache rather than the disk medium. This simulates disks that have command queueing. In this case, the I/O subsystem is able to deliver about 25MBps, easily able to saturate two networks.[5] With such a setup, the integrated *SBVS* with two networks can support 65 streams.

Configuration	**VSS**	**RETHER**	**SBVS**
1 Ethernet/2 SCSI Buses 4 Disks	55	45	45
2 Ethernets/3 SCSI Buses 6 Disks	74	81	65

Table 15: *Maximum number of 1.5Mbps streams that can be supported under multiple configurations by the VSS in isolation, RETHER in isolation, and the integrated SBVS.*

Table 15 shows the maximum number of streams that can be supported by the *VSS* in isolation, *RETHER* in isolation, and the integrated *SBVS*. In a single network environment, the number of streams is limited by the network bandwidth since the maximum number of streams, 45, can be supported by both *RETHER* in isolation and the *SBVS*. In a two network environment, doubling the network bandwidth resulted in only a 44% increase over the single network case. Although, *RETHER* in isolation, can support 81 streams, the CPU is saturated. When additional CPU processing is required due to the integration of the *VSS*, the total number of video streams that can be supported is reduced to 65 streams. Therefore, unlike the single network case, integration of the *VSS* and *RETHER* does affect system performance.

[5] In this case, the data transfered to the clients is still random bytes and cannot be displayed.

Chapter 7

Video Storage System Admission Control

7.1 Overview

When a user requests a new stream to be delivered, the admission control module within the *SBVS* determines if the addition of the new stream will affect the real-time guarantees of other streams. If there is not enough additional bandwidth in either the network or the I/O system to service the new stream, the stream is not allowed to enter the system, and the user's request is denied. Depending on the protocol, the system may queue requests until resources become available or users may simply wait some amount of time before re-sending the request. The design goal of the admission control module is to optimize the number of video streams being serviced simultaneously without breaking any real-time bandwidth guarantees.

This chapter presents the design and analysis of the *VSS* admission control module. Network admission control in the *SBVS* is handled by the *RETHER* system and not discussed here. The chapter starts with a framework that describes the video server's admission control algorithms in terms of how various resource parameters, such as disk seek, rotation, transfer time, etc., are estimated. Then, five admission control algorithms are presented within the framework where each of the algorithms predicts how much bandwidth would be used by the *VSS* if a new stream were to be added. Each algorithm uses either worst-case assumptions about resources or dynamically measured values when creating its prediction. Finally, the analysis section compares the empirical performance of the different algorithms under various workloads.

7.2 Description of the Admission Control Problem

The admission control problem can simply be stated as: *How does the system guarantee that all or most (depending on the level of quality of service) active streams are guaranteed uninterrupted service while admitting as many streams as possible?*

Much of the work regarding quality of service has been intended for network congestion control. However, such work is equally applicable in networked multimedia servers. [FV90] has proposed three levels of *quality of service* which we use in our experiments. They are as follows:

1. *Deterministic:* All deadlines are guaranteed to be met. For this level of service the admission control algorithm considers worst-case scenarios in admitting new clients.

2. *Statistical:* Deadlines are not 100% guaranteed to be met but are usually reliable. To provide such guarantees, admission control algorithms must consider dynamic statistical behavior of the system while admitting new clients.

3. *Best Effort:* No guarantees are given for meeting deadlines. The server just "tries its best," i.e., it schedules such accesses only when there is time left over after servicing all guaranteed and statistical clients.

As discussed in Section 4.1, the *SBVS* uses *cyclic scheduling* where time is divided into *cycles* of length P_{io}. While *RETHER* is sending data to users over the network during cycle i, the *VSS* is inserting data into memory with the data needed for cycle $i+1$. Let SV_i be the actual time the *VSS* takes to read into memory the data needed for every stream during cycle i. For all active streams to have uninterrupted service, for every cycle, SV_i must be less than P_{io}. This guarantees that *RETHER* has valid data at the beginning of every cycle.

The goal of the *VSS* admission control policy is to admit as many streams as possible, while maintaining that $SV_i < P_{io}$ for every cycle i. Thus, when a request for a new stream is made, the admission control algorithm attempts to predict the service time, PSV_i, for every future cycle that includes the new request. If, for every cycle i, $PSV_i < P_{io}$, then the stream is admitted. If the components of the I/O service time were all deterministic, then the admission control would be straightforward since it would be relatively simple to calculate PSV_i. The existence of several non-deterministic components, namely the rotational latency, head switches and track-to-track seeks, complicates the design of admission control algorithms since it is difficult to compute PSV_i.

Note that the consumption rate of the jth video, C_j, is constant throughout the duration of its playback. This is because the *VSS* uses a constant data length (CDL) media block (See Section 4.2.1). For any given time duration, from time t to time $t+\delta$, if no streams are added, deleted, or paused, the amount of data retrieved during any cycle within the duration

is constant. Thus, the admission control algorithm is simplified since it only needs to make a single prediction for all future cycles. For example, let a new request for stream r enter the system between cycle i and $i+1$. Assuming there are ns streams running during cycle i, the total amount of data to be read during cycle i is

$$ReadAmount \;=\; \sum_{j=1}^{j=ns} \;(\; C_j \;*\; P_{io} \;) \tag{11}$$

If r is entered into the system at cycle $i+1$, the amount of data read during cycle $i+1$ and any future cycle (until either a stream is admitted, deleted or paused) is

$$ReadAmount \;=\; \left(\sum_{j=1}^{j=ns} \;(\; C_j \;*\; P_{io} \;) \right) + \;(\; C_r \;*\; P_{io} \;) \tag{12}$$

If a constant time length (CTL) block is used, then C_j and, hence, $ReadAmount$, would vary from cycle to cycle. This complicates the admission control algorithm since the amount of data to be retrieved in future cycles will vary.

The admission control test for a single disk is now presented and then will be generalized to a multi-disk scheme. The output of the admission control algorithm is a prediction of the service time if a new stream were to be inserted into the system. Assuming ns streams are running and stream r requests admittance, a simple prediction (total elapsed time for servicing $ns+1$ video streams within a cycle for a single disk) would be as follows.

$$PSV_{ns+1} \;=\; PSV_{ns} \;+\; PSV_r \;=\; \left(\sum_{j=1}^{j=ns} \;PSV_j \right) \;+\; PSV_r \tag{13}$$

This formula creates a prediction based on the sum of the predictions for each active stream and the prediction for the new stream. Thus, a new video stream r can be admitted if an only if $PSV_{ns+1} \;<\; P_{io}$ If the stream is admitted, it is retrofitted into the current schedule to minimize the seek overhead.

The prediction for a single stream j, PSV_j, has several terms; the time the disk takes to seek from stream $j-1$ to stream j ($SEEK_{j-1,j}$); the disk rotational latency ($ROTLAT_j$), which is the time to rotate the disk to the target sector; and the time to transfer the data from the disk medium to the disk cache ($XFER_j$).[1] In modern disks, while the data is being transfered from the disk medium to the disk cache, the data can also be transferred over the disk bus to main memory. Thus, the prediction for a single stream to transfer $C_j * P_{io}$ bytes in a single cycle is

$$PSV_j \;=\; SEEK_{j-1,j} \;+\; ROTLAT_j \;+\; XFER_j(C_j * P_{io}) \tag{14}$$

[1] This is the hardware cache inside the disk drive.

The disk seek time is a function of the location of the data to be retrieved. A worst-case seek time can be used in the equation or a measured value can be used that is acquired by a benchmark program that seeks to various locations on the disk. The rotational latency depends on the location of the data to be retrieved and the speed of the disk. Since disks are always turning, rotational latency cannot be accurately predicted on a request by request basis. The disk transfer time depends on the location the data, the speed of the disk, and the amount of data to be retrieved. As a first approximation, the transfer time for stream j can be calculated as follows.

$$XFER_j(C_j * P_{io}) = \frac{C_j * P_{io}}{DISK_{total}} \quad (15)$$

$DISK_{total}$ is the raw transfer bandwidth of the disk in bytes/second. In modern disks $DISK_{total}$ will vary depending on the zone of the retrieved block. Because of the larger physical area, tracks on the outside zones of a disk have more sectors than the inside tracks. Measurements show that the bandwidth on the outside of the disk to be about twice the bandwidth of the inside of the disk. In addition, when retrieving consecutive sectors it is also possible that a disk read will cause a disk head switch between cylinders and/or a track to track seek if data crosses track boundaries. In essence, $DISK_{total}$ will vary depending on the location of the data to be read, and the number of track-to-track seeks or head switches. Thus a better approximation of the transfer time would be the following. Let $Z_{sectors}(j)$ be the number of sectors on a track within the zone in which stream j resides, $Size_{sector}$ the size of a sector, $O_{disk}(j)$ the overhead due to any head switches or track-to-track seeks for transferring stream j, and ROT be the time for the disk to rotate once. A more exact time to transfer data from the disk medium to the disk cache is:

$$XFER_j(C_j * P_{io}) = \left(\frac{C_i * P_{io}}{Z_{sectors}(j) * Size_{sector}} * ROT \right) + O_{disk}(j) \quad (16)$$

The first term, $\frac{C_i * P_{io}}{Z_{sectors}(j) * Size_{sector}}$ determines the number of disk rotations required to retrieve the data. Multiplying it by ROT gives the total amount of time to rotate the disk to retrieve the data. Although the times to perform a head switch or track-to-track seek are fixed, during retrieval of data for a stream, some cycles may incur this overhead, while some may not. Since the SCSI protocol provides an access interface where the logical block number is specified rather than cylinder/track/sector, it may not be possible to predict whether a disk read will incur a head switch or track-to-track seek.

Equation 14 is the prediction for servicing a stream on a single disk. This assumes that there is no contention when sending data from the disk cache to main memory. It also assumes that the data can be transferred over the disk bus while it is being transferred from the disk medium to the disk cache. However, if there are several disks on the disk bus, only

Algorithm 7.1 *Deterministic admission control algorithm. Worst-case values are used for all varying prediction terms.*

```
/* Do a worst case scenario */
total = 0;
for(i = 0; i < numStreams; i++)
    total += MAXIMUM_SEEK+
        PredictWorstCaseTransfer(Stream[I]->sectorsToGet);
if(total > cycleTime) do not admit stream
else admit stream
```

one disk at a time can send its data to main memory over the bus. On the other hand, the disks work in parallel and can be transferring data from the disk medium into the disk cache while other disks transfer data over the disk bus. For example, in a system with N disks that supports disk striping, and for an I/O request size of K bytes, each disk transfers $\frac{K}{N}$ bytes from the disk medium to the disk cache, and the disk bus takes $BUS_j(K)$ to transfer K bytes serially. If $C_j * P_{io}$ is the number of bytes to be retrieved by stream j in one I/O cycle, Equation 14 is changed to be the following.

$$PSV_j = SEEK_{j-1,j} + ROTLAT_j + XFER_j(\frac{C_j * P_{io}}{N}) + BUS(C_j * P_{io}) \qquad (17)$$

In summary, the prediction formula for a cycle is made up of several varying terms and several fixed terms. The fixed terms are C_j, the consumption rate for stream j, N, the number of disks, P_{io}, the cycle time, $Size_{sector}$, the size of a disk sector, ROT, the time for a single revolution of the disk, and $BUS(K)$ the time to transfer K bytes over the disk bus. The varying terms are $SEEK$, the time to perform a disk seek, $ROTLAT$, the rotational latency, and O_{disk}, the overhead due to disk head switches and/or track-to-track seeks.

Because the delays contributed by the varying terms cannot be accurately modeled statically, they have to be dynamically estimated for admission control. Deterministic algorithms assume worst-case values for the varying terms, whereas statistical algorithms estimate varying terms according to run-time measurements from past history. The statistical algorithms themselves can be further classified according to whether the worst-case or average values of the run-time measurements are used in the estimation.

7.3 Admission Control Algorithms

Five different admission control algorithms are described in this section. Each of these algorithms differs in how it assigns values to each of the terms in the prediction, Equation

17.

7.3.1 Deterministic/Worst-Case

Algorithm 7.1 shows the deterministic algorithm that guarantees the service time never exceeds the cycle time. Worst-case values are used for each of the varying terms in the prediction equation. Note that the value of $MAXIMUM\_SEEK$ in Algorithm 7.1 is not the maximum seek from the outside cylinder to the inside cylinder. The *VSS* uses a SCAN algorithm to retrieve data where streams are serviced starting at the outside cylinder and progressing inward on the disk. In the worst-case, if there are ns active streams, the streams would be uniformly distributed across the entire disk, maximizing the seek time between each stream. A benchmark program was run to determine the seek times for various numbers of streams. For more than 10 streams, the maximum seek was measured to be 6ms and is used in the deterministic algorithm.

This deterministic scheme is not based on any dynamically measured values. The procedure *PredictWorstCaseTransfer* is shown in Algorithm 7.2. It is the method for estimating the worst-case time for the *VSS* to service a single stream for one cycle. This algorithm is absolutely conservative in that it uses worst-case assumptions about rotational delays, head switches and transfer times, but does not include any disk seek overhead.

7.3.2 Deterministic/Average-Case

In the previous deterministic algorithm, worst-case values are assigned to all of the terms in the prediction formula. However, this method is extremely conservative. Over any period of time, there is an extremely low probability that every request assumes its worst-case values. For example, the *PredictWorstCaseTransfer* algorithm assumes worst-case rotational delay and disk overhead for every stream in every cycle. This is unlikely to occur in a running system. Algorithm 7.3 presents the Deterministic/Average algorithm, an algorithm that uses measured statistics of the running system but will still guarantee that all streams are serviced. First, a prediction is made on the service time for the active streams. Service times over some window of time are measured and the average service time is calculated. Note that $CalculateStatistics()$ takes $numStreams$, the number of streams currently in the system, as an argument because the *VSS* maintains a separate history of service time measurements for each distinct number of active streams. It is assumed that the rotatational latency for every request to every disk takes the average, or one half of one rotation. The prediction for the active streams is updated so that a worst-case rotation delay is included. This results in a service time prediction that assumes a worst-case rotation delay with other terms derived from measurements. This is not done on a per disk basis, but a stream by stream basis.

Algorithm 7.2 *Worst case time to transfer sectorsToGet blocks from the disk array to main memory.*

PredictWorstCaseTransfer(totalSectors)

$total = 0$; /* This is the total prediction */

/* Each request is striped across all of the disks in the array */
$sectorsPerDisk = totalSectors/numDisks$

/* The first variable is the time to rotate the disk to the
* appropriate sector. Assumes a worst-case rotational delay. */
$total = ROTATION\_TIME$;

/* The next term is the time to transfer the data from the
disk medium into the disk cache. This assumes that the
video is located on the inner portion of the disk. */
$mediumXfer = (sectorsPerDisk/INNER\_SECTORS) * ROTATION\_TIME$;

/* The retrieval may have to cross cylinder or track boundaries. */
$headSwitch = ((secotrsPerDisk/INNER\_SECTORS) + 1) * HEAD\_SWITCH$;
$mediumXfer += headSwitch$;

/* Add in the time to transfer the data over the SCSI bus. */
$busXfer = ((sectorsPerDisk * SECTOR\_SIZE)/SCSI\_BUS\_TRANSFER) +$
$SCSI\_OVERHEAD$;

/* In modern disks, data can be transferred from the disk medium to the
* disk buffer while other data is being transferred from the disk buffer
* over the SCSI bus. The time to get the data from the disk medium into
* main memory is the maximum of the disk transfer time and bus transfer. */
$firstDisk = MAX(mediumXfer, busXfer)$;

/* For the other disks, the transfer off of the medium is done concurrently
* with the first disk. So just add the time to transfer over the disk bus. */
$otherDisks = (disksPerScsiBus - 1) * busXfer$;

$total = total + firstDisk + otherDisks$;

Again it is assumed that disks work in parallel. The service time is bounded by the time it takes to retrieve the data from the slowest disk. Thus if ASV_{ns} is the average service time for ns active streams, PSV_{ns} is calculated as follows.

$$PSV_{ns} = ASV_{ns} + (ns * .5 * ROTATION\_TIME)$$

Algorithm 7.3 *Deterministic/Average-case admission control algorithm.*

```
/* The procedure CalculateStatistics returns the total service time
* for numCycles for numStreams. It is assumed that each request
* took one half of a rotation. Then add in another half rotation
* time for each stream so that the rotation delay is now the worst-case value.
* This assumes that only one disk in the array takes a full rotation.
* The other disks work in parallel and can start transferring while
* the slow disk is rotating.
*/
CalculateStatistics(numStreams, numCycles, totalTime)
average = totalTime/numCycles;
total = average + (numStreams * 0.5 * ROTATION_TIME);

/* Finally, add a worst case calculation for the next stream.
*/
total+ = PredictWorstCaseTransfer(Stream[r]- > sectorsToGet);

/* If the predicted total is less than the cycle time, admit the stream
*/
if(total < cycleTime) admit stream
else do not admit stream
```

Finally, a worst-case service time estimate of adding the new stream is added to the measured service time for the active streams, PSV_{ns}. If the total time is less than the cycle time, the stream is admitted.

7.3.3 Statistical/Worst-Case

Although a deterministic approach guarantees 100% reliability, we believe that the users of the *SBVS* are willing to tolerate a few missed deadlines. A few lost frames, or breaks in the audio can be tolerated especially if users are warned beforehand. Therefore, the *VSS* uses a statistical approach for admission control.

To evaluate PSV_{ns} in Equation 13, the time to service the active streams, the actual service times for ns streams is measured over some time interval. Depending on the type of statistical admission control, PSV_{ns} can be a worst case, average, or some other calculated value derived from the measured service times. The time to service the new stream, PSV_r, is estimated based on the amount of data to be retrieved and the location on the disk. This can be pre-calculated by running benchmark programs that retrieve varying amounts of data from different parts of the disk, can be based on dynamic statistics, or a worst-case

value can be used.[2] In addition to pre-calculating transfer times, seek times can also be pre-calculated. Seek times between various zones of a disk can be measured and saved. Using these calculated values, Equation 13 is evaluated to determine if the stream can be added.

In the Statistical/Worst-Case method, as shown in Algorithm 7.4, the actual service times are measured over some number of cycles. This measurement includes the average service time, the standard deviation, and the maximum service time. In this algorithm, PSV_{ns} is assigned to be the maximum service time over the interval. Next, the additional time to service the new stream is estimated using the worst-case time to service a single stream. This is simply the maximum service time divided by the number of active streams. If the sum of the maximum service time, standard deviation, and estimated transfer time of the new stream is less than the cycle time, the new stream is admitted into the system.

One problem occurred during the testing of this algorithm. The interval that the service times are used may have been very short. For example, assume at time 0 there are 40 active streams. Five cycles later, another stream requests admittance. The algorithm makes its prediction based on the service times that were measured while running 40 streams. This is the most accurate measurement. Realistically, five cycles does not give a long term view of the service times while running 40 streams. If the five measured times were low, the new stream will be admitted. What happened is that 40 streams was about the maximum number of streams that could be serviced. When running 41 streams, overloads occurred. At some point a stream would finish and assuming there is a waiting stream, the waiting stream would immediately request admittance. Although there were many measured service times while running 41 streams, when the request arrived only 40 streams were running and the algorithm only used the five measured times while running the 40 streams. These five measured values were low and always allowed a new stream to be admitted. This scenario kept repeating and when running 41 streams the system kept overloading. The problem is that there were only five measured service times while running 40 streams, although there were many measured service times while running 41 streams. Thus, the algorithm was changed to estimate the service time for running $N + 1$ streams to be the larger of the predicted value from running N streams and the actual measured service times while running $N + 1$ streams, if they are available.

7.3.4 Statistical/Average-Case

This algorithm is exactly the same as the previous Statistical/Worst-Case algorithm. Rather than using the maximum service time over some interval, it uses the average service time from the calculated statistics. In addition, when predicting the service time for the new

[2]Benchmark programs are run when the system is installed or its configuration is changed.

stream, the algorithm also uses per-stream average service time, $\frac{average}{numStreams}$. This estimate is less conservative than the worst-case algorithm.

7.3.5 No Prediction

In this algorithm, no prediction is made. As shown in Algorithm 7.5, the system will allow a stream to enter if, during the measured service times, no service time has exceeded the cycle time.

7.4 Performance Analysis of Admission Control Algorithms

In this section, the results of experiments that compare the performance of the above admission control algorithms under various workloads are reported. The first objective of the experiments is to measure the maximum number of streams admitted by each algorithm without causing a overload. The second objective is to compared the predicted service time to the actual service time.

7.4.1 Experiment Methodology

A generator program is run that randomly generates video numbers and starting times. The starting times are exponentially distributed with a mean time of 10 cycles between each start. The output of the generator program is input to the admission control experiments such that each run of the five experiments uses the same input. The program that runs each experiment simply proceeds in cycles. At the end of each cycle, the service time of the cycle and the number of running streams is saved. The program then checks the generated input to determine if another stream needs to be added, and if so, uses the appropriate admission control algorithm to determine if the stream can be admitted. Each experiment is run so that it takes about one hour. Each algorithm is measured on three hardware configurations, two disks/one SCSI controller, four disks/two SCSI controllers, and six disk/three SCSI controllers. In addition to measuring the maximum number of allowable streams, the number of cycles that cannot service all of the streams is measured. A cycle does not service all of the streams if the total time to service the streams is greater than the cycle time. For example, as shown in Table 16, when running the *No Prediction* algorithm on six disks, a maximum of 42 streams were admitted into the system. But when running 42 streams, 23% of the cycles caused an overload. When running 41 streams, no cycles caused an overload.

7.4.2 Uniformly Distributed Workload/Long Videos

In this experiment there are six videos evenly striped across each disk such that each video takes up one sixth of every disk. Each video is accessed in uniformly distributed manner. The playback rate for each video is the average MPEG-1 bit-rate or 180KBps. When using 2 disks each video is about 30 minutes long. When using 4 disks, each video is an hour long and when using 6 disks, each video is an hour and a half long. With this setup, the physical disk array parameters such as stripe unit size and degree of striping per SCSI bus remain fixed across all three configurations.

Disks	Determ. Worst-Case	Determ. Average	Statistical Worst-Case	Statistical Average	No Prediction
2	16	21	24	25(3%)	26(32%)
4	26	29	32	33(4%)	35(66%)
6	31	33	38(.05%)	41(11%)	42(23%)

Table 16: *Admission control performance for a cycle time of 1 second, using a uniformly distributed workload with long videos.*

Disks	Determ. Worst-Case	Determ. Average	Statistical Worst-Case	Statistical Average	No Prediction
2	21	29	31(2%)	31(8%)	33(99%)
4	35	42	45	46(7%)	46(11%)
6	46	53	58	61(3%)	62(16%)

Table 17: *Admission control performance for a cycle time of 2 second, using a uniformly distributed workload with long videos.*

Tables 16 and 17 show the results of the admission control experiments for cycle times of one and two seconds, respectively. Each value represents the maximum number of streams that the algorithm allowed to enter the system. (Each experiment was run for one hour.) The number in parenthesis is the percentage of cycles where the service time exceeded the cycle time. The *No Prediction* algorithm determines the maximum number of streams that could have been allowed to enter with no overflows. For example, with a one second I/O cycle time and using two disks, 25 streams could be admitted with no overflows. When 26 streams were running, 32% of the cycles overflowed. The results show that both statistical algorithms perform very closely to what a perfect prediction algorithm can achieve, and the difference between *Statistical/Worst-Case* and *Statistical/Average* is rather minor. On the other hand, the *Statistical/Worst-Case* is more reliable because it shows much smaller percentages of overloaded cycles.

Compared to the *Deterministic/Average* algorithms, the *Statistical/Worst-Case* is about 10% to 20% better, across different I/O cycle times and disk array configurations. This result is less optimistic than are reported in earlier papers [Mou94] [VGG94b, VGG94a]. This is because the I/O service time model presented in the previous section is more accurate and realistic. In addition, the measurements from real implementations include various overheads, such as SCSI bus arbitration overheads, that are simply overlooked in earlier studies.

The relative advantage of the *Statistical/Worst-Case* algorithm over the *Deterministic/Worst-Case* algorithm shrinks as more disks are added. When increasing the number of disks, more streams can be serviced in the same amount of time because of the increased parallelism. As more streams are serviced, there is more randomness because of the increase in the varying resources. Thus, the standard deviation of the service time increases. As a result, since the statistical predictions use the standard deviation, the predictions are relatively more conservative when there are more disks. For example, measurements showed that the standard deviations are 45%, 75%, and 105% of the average when there are 2, 4, and 6 disks in the I/O system.

In several cases, the service time exceeded the prediction when using the *Statistical/Worst-Case* algorithm. We conjecture this may be due to the short history used in the prediction. In each of the above experiments, the average time between requests for admission is only 10 cycles. This keeps the total time duration of each experiment run to about an hour. As a result, the admission control system only has a small time window from which to collect the measured service times used for prediction. In a real production environment, it would be unlikely that users request to display a new video every 10 cycles (10 or 20 seconds). Thus, on average, the admission control system has a small interval to which it can use the measured times. For example, when asking to admit stream number $r + 1$, the statistical algorithms use service times that were measured while running r streams. In the next experiment the *Statistical/Worst-Case* algorithm was run to determine the effect of changing the mean waiting time between requests, and thus the time window for measurement collection, on the accuracy of service time predictions. Tables 18 and 19 show the results. As expected, the percentage of overloaded cycles decreases because the prediction tends to be more conservative when a larger amount of history is used in the prediction.

7.4.3 Uniformly Distributed Workload/Low Bit-Rate Videos

The objective of the next experiment is to test the effects of lowering the playback rate of the videos. In this case, the non-deterministic variables in the admission control algorithm constitute a slightly larger proportion of the total service time. Like the previous experiment

Mean Wait	10		20		30	
Cycle Time	1	2	1	2	1	2
2 Disks	26(36%)	31(9%)	25(8%)	30	24	30
4 Disks	34(9%)	45	31	45	31	46
6 Disks	39	60	38	61	40	60

Table 18: *Effect of mean waiting time for requesting a new video on percentages of overload for Statistical/Worst-Case algorithm. Uses distributed workload and long videos. Results show that longer wait times and/or more disks, give better overall performance. Numbers in parenthesis specify percentage of overflow cycles.*

Mean Wait	40		50		60	
Cycle Time	1	2	1	2	1	2
2 Disks	24	30	24	29	24(.6%)	29
4 Disks	33(2%)	45	31	45	32	45
6 Disks	39	60	38	61	38	59

Table 19: *Effect of mean waiting time for longer waiting times.*

six videos are evenly striped across each disk such that each video takes up one sixth of every disk. However, the playback rate of each video is lowered to 60KBps or one third of a normal MPEG video.

Disks	Determ. Worst-Case	Determ. Average	Statistical Worst-Case	Statistical Average	No Prediction
2	32	40	46(1%) 47(7%)	49(4%) 50(54%)	49(41%)
4	39	45	55(.3%) 56(.4%)	58(14%) 59(24%)	58(17%)
6	42	48	61(.8%)	63(6%) 64(27%)	66(54%)

Table 20: *Admission control performance for a cycle time of 1 second, using a uniformly distributed workload with low bit rate videos.*

Tables 20 and 21 shows the results of the experiments. Since the playback rates of the videos are lowered, more streams can be admitted than in the previous experiments. Comparing Table 20 with Table 16, the ratio of streams admitted by the deterministic algorithm to the streams admitted by the *No Prediction* algorithm is lower when using a lower bit rate. For example, when using six disks and a playback rate of 180KBps, the *Deterministic/Worst-case* algorithm allows 31 streams or 76% of the maximum number of streams allowed by the *No Prediction* algorithm. When lowering the playback rate to 60KBps, the algorithm allows 42 videos but only 64% of the maximum. The change between

Disks	Determ. Worst-Case	Determ. Average	Statistical Worst-Case	Statistical Average	No Prediction
2	46	59	66	70(7%)	69(2%)
4	65	70	83	85(5%) 86(10%)	86(14%)
6	74	82	98	102(2%) 103(24%)	102(2%)

Table 21: *Admission control performance for a cycle time of 2 seconds, using a uniformly distributed workload with short videos.*

the 180KBps case and the 60KBps, however, is not dramatic. The only variable that is changed in the admission control prediction is the estimated time to transfer the data from the disk medium to the disk cache. Other variables, i.e., seek, rotation, and overhead remain constant. As a result, the negative performance effects due to worst-case assumptions used in the deterministic algorithms are slightly more pronounced when the bit rates are low than when they are high. In the deterministic algorithms, the transfer time is a small percentage of the total service time when assuming other worst-case values. For the 60KBps experiment the disk transfer time is 2.5ms compared to the predicted total service time (seek, rotation, transfer, overhead) of 23ms.

Again, as Tables 20 and 21 show, increasing the cycle time causes the *Statistical/Worst-case* algorithm to become more conservative and more reliable. As the cycle times are increased and more streams are serviced, the standard deviations increase, causing the statistical algorithms to be more conservative.

7.4.4 Non-Uniform Workload

In this experiment there are 10 movies evenly distributed across each disk. However, they are ordered such that the videos on the outside of the disk are considered 'favorite' videos and are accessed more frequently than the videos on the inside of the disk. The distribution of accesses is show in Figure 22. Since the streams are read from the outer portion of the disk, the disk transfer rate will be higher than a normally distributed workload that accesses data across all parts of the disk. In addition, seek times will be lower since the reads are closer together. The expectation is that the worst-case deterministic algorithms will perform less well while the statistical algorithms continue to perform close to the optimum.

As shown in Table 16 and Table 23, the performance gap between the *Statistical/Worst-Case* and *Deterministic/Worst-Case* algorithms indeed widens, although not as significantly

Video	0	1	2	3	4	5	6	7	8	9
No. of Requests	140	86	45	31	26	6	7	4	2	1

Table 22: *Distribution of videos used in the non-uniform admission control experiments. For each video, the associated number represents the number of times during the experiment that the video is requested to be displayed.*

Disks	Determ. Worst-Case	Determ. Average	Statistical Worst-Case	Statistical Average	No Prediction
2	17	23	26(.9%)	26(2%) 27(39%)	27(37%)
4	26	30	35	35(2%) 36(7%)	37(46%)
6	32	35	42(3%)	42(3%) 43(13%)	43(12%)

Table 23: *Admission control performance for a cycle time of 1 second, using a "favorite movie" workload with medium length videos.*

as expected. This seems to indicate that non-disk-related overheads that exist independently of the access patterns of the workload play a non-trivial role in the overall resource consumption. Similar conclusions can be drawn when comparing Table 17 and 24.

7.4.5 Accuracy of Service Time Prediction

In the previous experiments, the comparison of algorithms is based on the maximum number of streams that can be admitted. In the next experiment, the objective is to compare the prediction estimates for each of the algorithms with the measured service times for the duration of the entire experiment. The experiment is run by randomly (exponential distribution with a mean waiting time of 30 cycles) adding streams until 100 streams is reached. Each stream reads 360KB per cycle, equivalent to an MPEG-1 stream using a two second I/O cycle time. Whenever a new stream requests to be admitted, the four prediction

Disks	Determ. Worst-Case	Determ. Average	Statistical Worst-Case	Statistical Average	No Prediction
2	21	29	31(.7%)	32(15%)	32(10%)
4	35	43	47	48(5%) 49(40%)	49(33%)
6	46	57	66(1%)	65(.6%)	67(39%)
6	46	57	66(1%)	66(10%)	

Table 24: *Admission control performance for a cycle time of 2 seconds, using a "favorite movie" workload with medium length videos.*

algorithms are executed to calculate the prediction that would be used if the corresponding admission control were running. However, only the prediction is made and recorded, streams are never denied admittance until 100 streams is reached. (The *No Prediction* algorithm is not used since it does not calculate a prediction.)

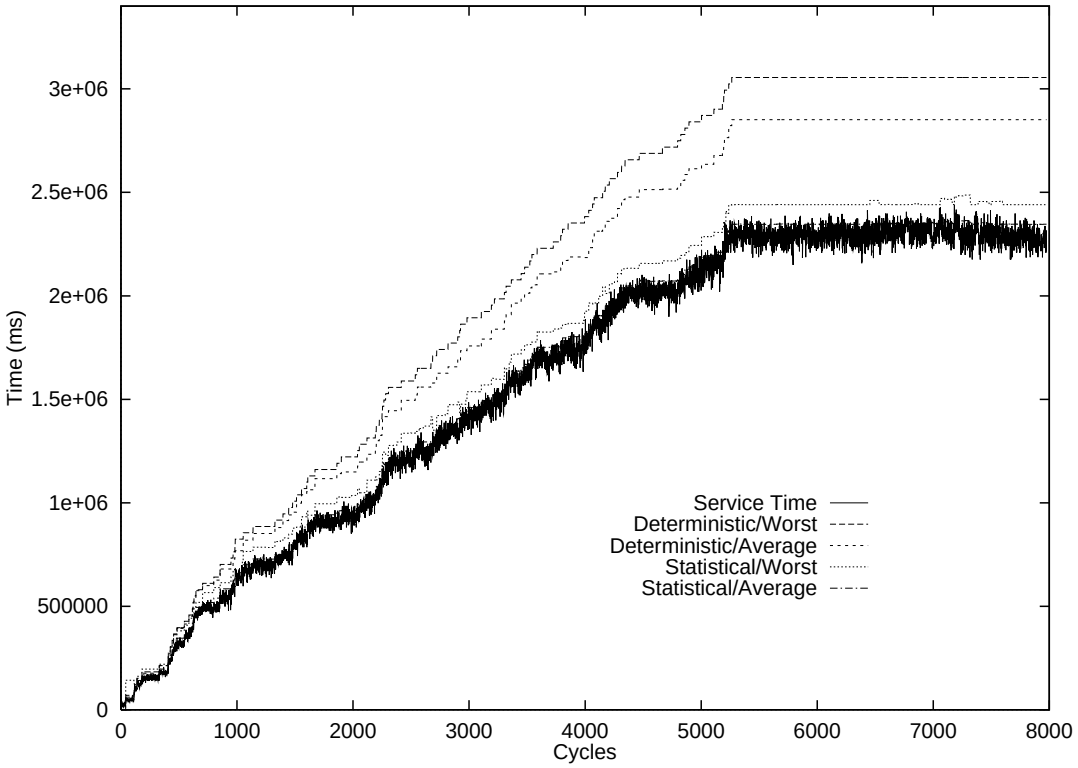

Figure 23: *Distributed workload admission control. Service times are actual service times. The four other plots are the prediction using the appropriate algorithm.*

Figure 23 shows the result of an experiment to gauge how well the prediction mechanism works under the uniformly distributed workload. The X-axis is the time line in I/O cycles and the Y-axis is the accumulated service time as more and more streams are added. Because no streams are allowed to exit, the accumulated service time is monotonically increasing until 100 streams are being serviced. The *Statistical/Worst-Case* algorithm works best in that it almost matches the measured service time without any overloaded cycles. The two *Deterministic* algorithms as expected are too conservative, whereas the *Statistical/Average* algorithm is too aggressive. Figure 24 provides a zoomed-in version of Figure 23.

7.4.6 Summary

This chapter presents an empirical comparison of several I/O-based admission control algorithms that are used to guarantee the smooth playback of existing streams when new

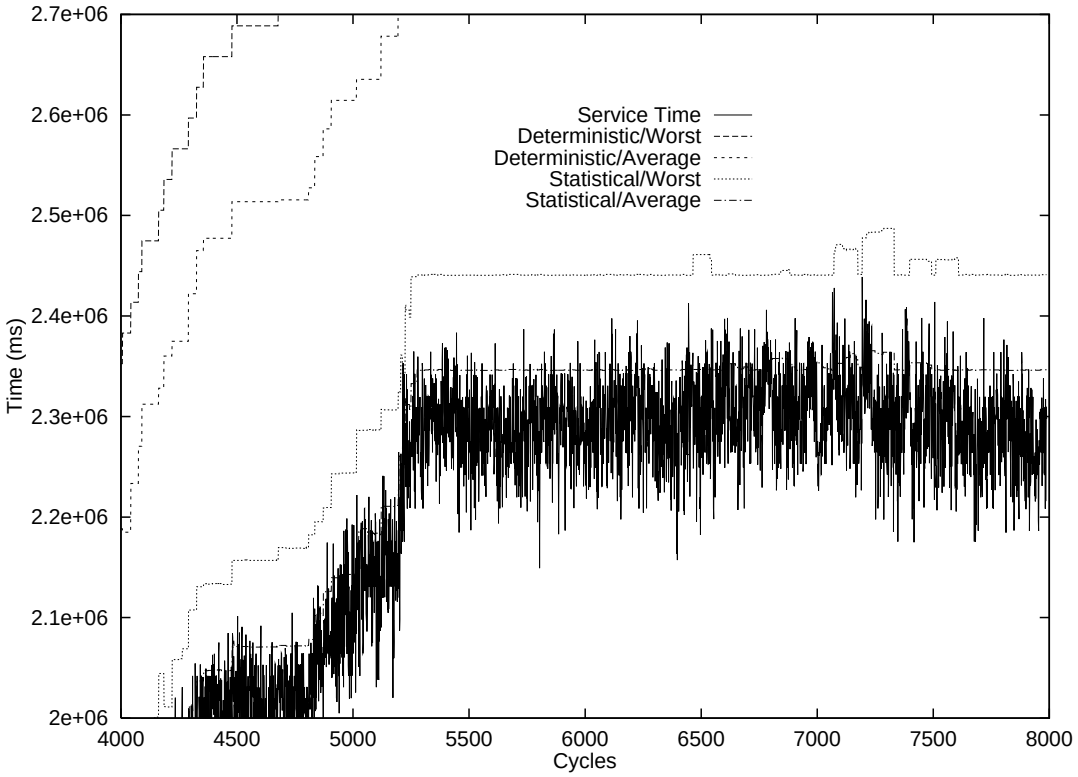

Figure 24: *This is the same graph as in Figure 23 but zoomed in more closely so the relationship between the two statistical algorithms and the actual service time can be better viewed.*

streams are to be added. A general framework is described for the admission control algorithms in terms of how varying parameters in the I/O service time prediction equation are estimated. Based on empirical measurements, the performance of these algorithms under various workloads and architectural parameters are compared. It was found that the *Statistical/Worst-Case* algorithm achieves a performance level that is rather close to the optimum while keeping the percentage of overloaded cycles to a minimum. Therefore, the *Statistical/Worst-Case* algorithm is the admission control algorithm used in *SBVS*. Also, contrary to the results reported by earlier work, the performance gap between the *Statistical* and *Deterministic* algorithms is between 10% to 40%, which is much smaller than were reported in earlier works[VGG94b, VGG94a]. This discrepancy is due to the over-simplified assumptions by the simulators as well as other fixed software/hardware overheads that are completely overlooked. The experimental approach used in these experiments completely eliminates all of these problems.

7.4.7 Related Work

Most of the existing work on admission control algorithms for multimedia servers has focused on developing techniques for providing deterministic service guarantees to clients. Worst-case seek and latency times are used to decide if new clients can be admitted [AOG92, RVR92, RV91, VR93]. In each of these papers, the algorithms for single disk admission control are presented but no performance results are given.

In [Mou94], a deterministic algorithm is compared to a statistical algorithm. Simulations show that the statistical algorithm can service up to 70% more streams than the deterministic algorithm while minimizing the probability that an overflow occurs to be on the order of 10^{-7}. This work, however, does not describe the statistical algorithm.

[CZ94, CZ96] compares the effects of using two different deterministic algorithms when using variable bit rate data streams for CTL blocks. The admission control algorithm is more complex when using CTL blocks, than when using CDL blocks. The amount of data retrieved varies from cycle to cycle if CTL blocks are used since the size of CTL blocks vary. The amount of data retrieved when using CDL blocks is constant until a stream is added, deleted or paused. In the first algorithm, actual bit traces of each video stream are used to determine how much data is retrieved during each future cycle. In the second algorithm, statistics about the bit traces are used. Simulations show that the deterministic approach has better reliability. In the simulations, however, the disk seek and rotation times are kept at worst-case, fixed values. The disk transfer times used are average values and do not take into account the different transfer times of data from different portions of the disk.

[VGG94b, VGG94a] discuss statistical admission control algorithms. In these type of algorithms, like the statistical algorithms presented for the *SBVS*, clients are admitted for service only if the extrapolation from the past measurements of the storage server performance characteristics indicate that the service requirements of all the clients can be met satisfactorily. The service requirement is the percentage of deadlines that can be missed. For example, a client can specify that it can tolerate up to 5% of missed deadlines. The experiments, however, are simulated, assume large numbers of users (> 1000) and disks (> 100), and do not take into account variables such as head switch and bus transfer overhead. The algorithm also assumes that a CTL media block size is used rather that a CDL media block. The results presented determine the maximum number of streams that can be admitted by using the statistical algorithm vs. a deterministic algorithm. When using the statistical algorithm and allowing up to 3% of the cycles to overload, three times the number of streams can be admitted compared to the deterministic approach. The results presented in this chapter extend this work by comparing several other algorithms to the measurement based algorithm, and by performing the experiments on a real disk-array under a variety of workloads.

Compared to previous work, the results reported in this chapter are new in that they are based on empirical measurements from actual implementations of the admission control algorithms in an operational disk-array video server. Moreover, because *SBVS* implements a CDL data retrieval strategy, this chapter focuses on admission control algorithms for CDL-based rather than CTL-based video servers.

Algorithm 7.4 *Statistical admission control algorithm based on maximum service time and standard deviation over recorded interval.*

/* Calculate the statistics of the service times for the interval. It
* is based on the current number of active streams. */
$CalculateStatistics(numStreams, numCycles, totalTime, totalSq, max)$
$average = totalTime/numCycles$
$stdev = sqrt((totalSq/valid)/(average * average))$

/* Get the position of the new stream */
$r = GetPosition(newStream)$

/* The SeekTime(f, t) function in the following code returns the measured
* seek time from stream 'f' to stream 't'. These measurements
* are made when the system is first initialized and stored on
* the system disk for use by the admission control algorithm. */
$if\ r == 0$ /* New stream is first stream on disk */
$\quad additional = SeekTime(r, 0) - SeekTime(m, 0) + SeekTime(m, r)$
$else\ if (r == m)$ /* New stream is the last stream */
$\quad additional = SeekTime(m, r) - SeekTime(0, m) + SeekTime(0, r)$
$else$ /* New stream goes between stream k-1 and k */
$\quad additional = SeekTime(k-1, r) + SeekTime(r, k) - SeekTime(k-1, k)$
$endif$

/* Use worst-case measured value to estimate additional
* service time for new stream */
$additional\ +\!= max/numStreams;$
$prediction = max + stdev + additional$

/* Calculate the statistics based on numStreams+1 streams. */
$CalculateStatistics(numStreams + 1, numCycles, totalTime, totalSq, max)$
$average = totalTime/numCycles$
$stdev = sqrt((totalSq/valid)/(average * average))$

/* If maximum time for numStreams+1 and the stdev is greater
* than the prediction for numStreams, then use the greater value. */
$if\ (max + stdev > prediction)\ prediction = max + stdev$

$if\ (prediction > cycleTime)\ do\ not\ admit\ stream$
$else\ admit\ stream$

Algorithm 7.5 *No Prediction admission control algorithm. If the maximum service time over some duration is less than the I/O cycle time, the stream is allowed to enter.*

$CalculateStatistics(numStreams, numCycles, totalTime, max)$;
$if(max < cycleTime)$ *admit stream*
else do not admit stream

Chapter 8

Fast Forward/Rewind of MPEG Files

In a distributed video server environment like the *SBVS*, users should have the ability to quickly scan videos both forward and backward to find specific scenes. Although this is a natural requirement, there has been little published work dealing with the methods for supporting this functionality. Several works propose methods for implementing fast forward but do not adequately deal with fast rewind.

This chapter discusses and analyzes several different methods for supporting fast forward/rewind (FF/FR), specifically in MPEG bit-streams. The goal of this work is to compare the proposed alternatives by analyzing; 1) the implementation complexity of each method; 2) the increases in resources needed by the server and the client when users are using FF/FR; 3) the flexibility in the playback speeds; and 4) the visual quality of the output generated by each method. The chapter starts by discussing the implications of showing an MPEG video in both the forward and backward directions. Next, various proposed methods for implementing FF/FR are discussed, including implementation complexities and resource overhead needed to support the method. When switching from normal playback to FF/FR, extra resources may be needed by the server. When designing the methods for FF/FR, an important design goal is to ensure that when a client switches from normal to fast playback, the bit-rate being sent to the client does not increase. Finally, the visual output that is generated by using each of these methods is subjectively compared and discussed.

8.1 Differences between Forward and Backward

As discussed in Section 2.2, an MPEG bitstream consists of I-frames, which are independently decodeable, P-frames, which are backward dependent on the previous P- or I-frame, and B-frames which are both forward and backward dependent on the future/previous P- or I-frame. In addition, frames cannot be displayed until all the frames on which it depends have been presented to the decoder. Since B-frames rely on at most two frames, the decoder needs to

be able to store a maximum of two frames in its memory.

Frames in a bitstream are grouped into segments called a group of pictures (GOP). So that these bitstreams are randomly accessible, GOPs begin with I-frames. There are two types of GOPs, *closed* GOPs which are independently decodeable and end with a P-frame or I-frame, and *open* GOPs which end with a B-frame. For example, let the display order of a closed GOP consist of the frames $IBBPBBP$. Since all of the dependent frames exist within this GOP, it can be decoded independently of any other GOPs. Now consider the open GOP, $IBBPBB$. The first two B-frames are dependent on the first I and P-frames. However, the second set of B-frames depend on the first P-frame and the I-frame from the next GOP. Thus, frames in this GOP are dependent on frames in another GOP.

To implement rewind, let a sequence consist of the frames I_1 B_2 B_3 P_4 B_5 B_6 P_7 B_8 B_9 P_{10} I_{11} B_{12}... where I_1 and I_{11} are the first frames in successive GOPs. To display frame P_{10}, three frames must first be decoded, P_7, which P_{10} depends upon, P_4, which P_7 depends upon, and I_1, which P_4 depends upon. In this case four frames must be decoded and held in memory before P_{10} can be displayed. If the GOP were longer, the decoder would need to store more uncompressed frames. Using current technology, however, hardware decoders are unable to do this. They expect the MPEG stream to be moving in the forward direction.[1] For example, as described previously, both P- and B-frames have directional dependencies. P-frames depend on the previous I- or P-frame, and B-frames depend on the previous I/P-frame and next I/P-frame. In this example when playing frames in reverse order, frame I_{11} would be presented to the decoder then frame P_{10}. Since a P-frame depends on the previous I-frame, P_{10} would be decoded based on the contents of I_{11} rather than its true parent frame, I_1, and the picture would be garbled. The same argument also holds for B-frames.

Another possibility for supporting rewind is presented in [CK95]. Rather than the server support rewind, it is completely handled by the client. As frames of video are processed at the client, rather than discard the displayed frames, the client saves the frames to a temporary buffer in main memory, or to a buffer on the hard disk. When the user requests the mode to change to rewind, the client plays back frames from the local buffer. Depending on the size of the buffer, anywhere from several seconds to many minutes can be saved. Since P-frames cannot be decoded in reverse, the method in [CK95] proposes to save all processed I-frames, discard B-frames, and convert P-frames to I-frames. The resulting file is comprised of only I-frames and can be play both forward and backward. The authors showed that the conversion from P-frame to I-frame can be done in real-time. To calculate the storage needed to save this file, statistics from several MPEG-1 streams are used. Each of the files had a GOP constisting of the frames $IBBPBBPBBPBBPBB$ and an average I-frame size of 13KB with a resolution of 320x240 at 30fps. For each second of display, 10 frames, or

[1] It may be possible to do this in software, but high-end, expensive machines are needed.

about 130KB would have to be stored. Assuming a client with 16MB of main memory, about two minutes of rewind data could be stored. And since the saved file contains one of every three frames from the original, this corresponds to 6 minutes of the original file. Of course, if the hard disk were used to store the frames, more information could be kept. The main advantage of this method is that the server needs no extra processing to support rewind. The main disadvantage is that the amount of rewind data is constrained by the size of the temporary buffer. In addition, the user must view the video in the forward direction before switching to rewind.

In the following five sections, different methods for implementing FF/FR using MPEG bitstreams are presented. For each of the methods, the related work is cited.

8.2 Increased Playback Rate

Consider a video being sent to a client at 30 frames per second (fps) during normal playback. In the *increased playback rate* method proposed in [DSSKT94], for a speedup of three, the server would deliver 90fps.

The main advantage of this method is that the quality of the playback will be the same as the normal playback quality. In addition, multiple playback rates can be achieved. For a speedup of 4, 120fps are delivered.

There are two major disadvantages to this scheme. First the hardware decoder must be able to display the video using the new playback rate. Currently, there are no hardware or software MPEG decoders that can run faster than 30fps. In addition, if a television set (NTSC input) is being used as the output device, it cannot show more than 30fps. Thus, this method cannot be implemented using current technology.

The second problem with this method, assuming that it were possible to play back MPEG bitstreams faster than 30fps, is that this method increases the resource load on both the server and the network. For a speedup of three, the server must retrieve three times as much data from the storage system and send three times as much data over the network. It may be possible that the server, or the network, does not have enough bandwidth to accommodate this additional load. This may be unacceptable. In [DSSKT94], the authors propose that the server reserve bandwidth to accommodate users going into FF/FR. This, however, decreases the total number of users that can be accommodated for normal playback, thus increasing the dollar per stream for the system. Lastly, MPEG bitstreams have P-frames that are interpolated based on previous I or P-frames. As shown above, this makes it very complex to show these bitstreams in reverse order.

8.3 Skipping Segments

In the *skipping segments* method proposed in [CKY95], entire GOPs are either skipped or sent to the client. For example, to achieve a speedup of N, every Nth GOP is sent to and consumed by the client. The client continues to decode at the normal decoding rate.

By skipping various numbers of GOPs, multiple playback rates can be achieved. In addition, this method is also easy to implement if the server delivers data based on a constant time length (CTL) data block, especially if the size of the data block is equivalent to the number of frames in each GOP. This method is harder to implement if constant data length (CDL) blocks are used since GOPs will cross data block boundaries. When MPEG bitstreams are broken up into fixed size blocks, and the underlying system is optimized for fixed length blocks, it would be hard to skip and send GOPs of varying length.

In addition, if closed GOPs are used as the encoded bitstream, each GOP is independently decodeable. If open GOPs are used, some processing must occur at the client. Consider the display sequence of an open GOP consisting of the frames $I_1B_2B_3P_4B_5B_6$. If this GOP was sent to the decoder and the next GOP was skipped, frames B_5 and B_6 would have to be discarded by the client since the forward I-frame that they depend on (I_7), would be skipped. Lastly, there are no increases in resources at either the server or client when users switch to FF/FR.

For rewind, since current hardware MPEG-1 hardware decoders cannot play the frames of an MPEG GOP in reverse order, the GOPs are sent in reverse order but the individual frames are played in forward order. The visual output does not look faster since the playback rate of each GOP does not change. This, however, may be acceptable to the end user who is searching backward for a particular part of a video. The visual quality aspect will be discussed in the last section of this chapter.

Another problem arises that the authors of [CKY95] did not address. They only consider MPEG-1 Video streams. The problem occurs when one tries to implement the skipping segment method using MPEG-1 system streams. (MPEG system streams are discussed in Section 2.2.) The authors proposed to divide the MPEG bitstream into GOPs and place each GOP on different disks in a disk array. During normal playback, one GOP and thus, one disk is accessed during each cycle. It is possible, however, for a system stream packet to be divided between two disks. Also, the pack header which describes the packets may reside on a different disk. This is shown in Figure 25. If GOP_i is stored on disk 1 and GOP_{i+1} is stored on disk 2, the pack header that describes the packets that are part of the GOP_{i+1} may reside on disk 1 while the actual data resides on disk 2. If the skipping segment method is used this may cause problems at the decoder.

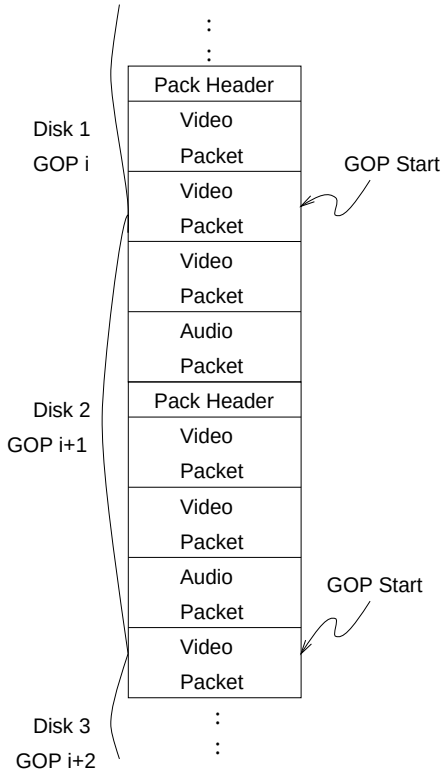

Figure 25: *Example of how an MPEG system stream is organized. If GOPs are placed on different disks, the pack header for some video packet may reside on a different disk.*

8.4 Skipping Subsegments

One problem with skipping full GOPs is the number of frames that are skipped at any one time. Let a GOP consist of 15 frames, the normal playback rate be 30fps and the speedup rate be three. Thus from the original stream, 15 frames are displayed, 30 are skipped, 15 are displayed, etc. One second of video is skipped between the display of each GOP. During periods of high motion, this may not be optimal. Another alternative, proposed in [OBRS95], is to skip independent subsequences rather than full GOPs.

Consider a closed GOP consisting of frames $IBBPBBPBBP$. The first four frames are considered an independent sequence because they can be decoded independently. The first seven frames are also an independent sequence. For fast playback, independent sequences are retrieved and sent while the rest of the frames in each GOP are skipped. When using this method there are only a fixed number of speedup rates that can be achieved. If four frames are sent from each GOP, six frames are skipped for a speedup of 2.5.

Like the skipping segment method, the same playback rate is maintained at the client. The bitrate that needs to be sent over the network, however, will increase as will the resources at the server. As an example, statistics from a Three Stooges MPEG-1 stream were gathered.

Each GOP consists of the frames $IBBPBBPBBPBBPBB$. The average frame size is 4206 bytes and with a frame rate of 30fps, the average playback rate over the duration of the video is 126180Bps. The average size of an I-frame is 13291 bytes, the average size of a P-frame is 4577, bytes and the average size of a B-frame is 3146 bytes. Thus, the average size of the first 4 frames of each GOP would be 24160 bytes (13291+4577+(2*3146)). If the same frame rate, 30fps, were consumed by the client, then the average playback rate would rise to 181200Bps ($\frac{30}{4} * 26140$) an increase of 43% over the normal playback rate. This increases the resources used by both the network and the I/O subsystems.

In addition to the higher playback rate required, extra overhead may be incurred by the I/O subsystem. If a GOP is used as the retrieval block, normally the GOP is stored contiguously on disk. Assuming that during each I/O cycle 1 GOP (15 frames) is retrieved, the disk executes one rotation, seek, and transfer. If four different frames from three GOPS (12 frames) and three frames from the next GOP are retrieved during each cycle, the disk must perform four rotations, seeks, and retrievals for every 15 frames retrieved, again increasing the overhead to process the fast playback request.

8.5 Skip Frames

In [CKLV95], the proposed method for supporting FF/FR is to send only I-, or I- and P-frames. By using this method a limited number of speedup rates is achieved. For example, in a 15 frame GOP $IBBPBBPBBPBBPBB$, sending only I-frames results in a speedup of 15 and sending only the I- and P-frames results in a speedup of 3. Since the data being displayed is not contiguous, the authors recommend first reordering the MPEG file. If the data were kept contiguous on the disk, a disk rotation, seek and transfer must be performed for each frame, increasing the disk overhead to support faster playback. Rather, the data is reordered as follows. GOPs are grouped together in packs. The frames in each pack are reordered such that all of the I-frames are grouped together then the P-frames and then the B-frames.

For example, 2 GOPs of the above sequence are reordered into the sequence $IIPPPPPPPPBBBBB$.... During normal playback the entire sequence is retrieved from the disk and sent to the client where the frames are re-ordered and presented to the decoder. Assuming a playback rate of 30 frames per second, 2 GOPs are sent to the client every second. During Fast Forward the first 2 I-frames and 8 P-frames are sent to the client. This is done for 3 consecutive sequences. So instead of sending 2 GOPs (30 frames) consisting of I-, P-, and B frames during each cycle as in normal playback, the I- and P-frames from 3 sequences (6 GOPs) are sent. This is also 30fps but skips 2 out of 3 frames.

Like the method where independent sequences are skipped, the main drawback with this

method is that it increases the load on the server and the network. If the Three Stooges video were reordered, during fast playback, 6 I-frames and 24 P-frames are sent each second. Based on the average frame sizes, the total amount of data for 30 frames is $(6 * 13291) + (24 * 4577)$ or 189595Bps, a 50% increase in the playback rate. And since 3 sequences must be accessed every second, extra disk overhead is incurred.

This problem of extra resource consumption is addressed in [SV95]. The bitstream is partitioned into three substreams. The B-frames constitute one substream. The I and P-frames are partitioned into a low-resolution and a residual component stream. During normal playback the low-resolution, residual and B-frame streams are merged to create the original bit-stream. During fast forward, only the low-resolution stream is sent to the client. This reduces the amount of data that needs to be retrieved and sent to the client, although it still introduces extra seek and latency overhead on the disk storage system.

Lastly, fast rewind can only be implemented by sending the I-frames. P-frames are based on forward prediction only and cannot be shown backwards. Just sending I-frames results in (30*13291)=398730Bps or an increased load of 310% on the server and network.

8.6 Special File

In this method, a special file is created specifically for the support of FF/FR. The main advantages of using this method are that it does not increase the load on the server since it can be created with the same playback rate as the original file. Also, the file can be created such that it can be shown forward and backward (See next section). [2] The main disadvantage of using this scheme is that a separate file must be created for each speedup rate, increasing the amount of storage used in the system, and hence, the cost. To keep the amount of storage used by the extra file low, however, it can be created using a lower quality than the original. As outlined in Section 4.4, this file can be used to support both FF/FR and multi-resolution viewing. The following section discusses the implementation of the FF/FR in the *SBVS*.

8.6.1 SBVS Support for Rewind

In the *SBVS*, the fast forward/rewind file (*ff-file*) is generated by removing frames from the original stream. The default speedup rate for the *ff-file* is three, so two of every three frames are removed from the original. To support multiple playback speeds, a different file needs to be encoded for each playback speed. The encoding of the *ff-file* can be done in two ways. If

[2] The file could also be created based on the motion within the original file. During periods of little motion, large numbers of frames are skipped from the original. During periods of high motion, lower numbers of frames are skipped. This, however, is beyond the scope of this thesis.

the original uncompressed stream is available, then every nth frame is encoded for a speedup of n. If the original does not exist, the compressed stream is first uncompressed and every nth frame is then re-compressed. Rather than simply cutting frames from the compressed file, this allows the quality parameters of the *ff-file* to be varied. Parameters are chosen so that during the creation of the *ff-file*, the size of the file is approximately 20-25% of the original file and the average bit rate of the resulting file is less than or equal to the average bit rate of the original. This guarantees that when switching from normal playback to fast playback, there is no increase in load on the server nor the network.

In addition, when creating the *ff-file*, the bitstream is updated so that it can support both fast forward and fast rewind. As discussed in Section 2.2, MPEG frames are encoded on a macroblock basis, where each macroblock consists of a 16x16 pixel block. In I-frames, all of the macroblocks are intra-coded (I) while in P-frames, blocks could be either forward dependent or intra-coded depending upon which scheme provides better compression. Similarly, in B-frames, blocks could be intra-coded, forward dependent P, backward dependent P or interpolated B-macroblock (which is dependent upon both, a forward and a backward block), depending on which results in the best compression. In blocks that are dependent on forward or backward reference blocks, the block is coded as a distance vector which is the difference between the reference block and the block being compressed. For example, as shown in Figure 26a, the solid lines show dependencies between blocks during forward playback. The macroblock at coordinate (3,3) in frame B is decoded based on block (2,3) in I1 and block (3,5) in I2. The dotted lines show how the decoder would decode the block if the frames were presented in the reverse order. The macroblock in B at (3,3) would be decoded based on block (2,3) in I2 and block (3,5) in I1 since I2 is presented before I1. Since both macroblocks are incorrect, the frame is decoded improperly and the quality is poor.

To solve these problems, macroblocks are forced to be either intra-coded (I) or purely interpolated (B). P-macroblocks are not acceptable because they have a unidirectional dependency which would make it impossible to play frames back in the reverse direction. This would not entirely solve the problem since the interpolated B-macroblocks have different distance vectors with respect to the two reference frames. To solve this problem, the dependency is forced to be equal in both directions, and set equal to the smaller of the two vectors. This makes the interpolated frames symmetrical with respect to both the forward and the backward reference frames and enables them to be decoded correctly even if the reference frames are presented in the reverse order. For example, as shown in Figure 26b, the block at (3,3) in frame B now becomes dependent on block (2,3) in both I1 and I2.

The size of the updated macroblock bitstream, however, is slightly larger than would be if the bitstream were not updated. Again, the dependency between blocks is coded as the difference between the referenced macroblock and the estimated macroblock. By forcing

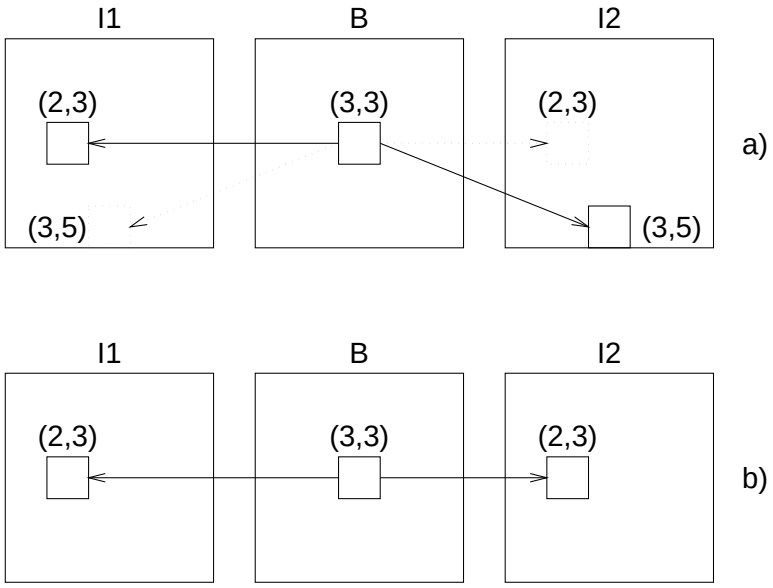

Figure 26: *a) Solid lines show dependencies between blocks going in the forward direction. The macroblock at coordinate (3,3) in frame B is decoded based on block (2,3) in I1 and block (3,5) in I2. The dotted lines show how the decoder would decode the block if the frames were presented in the reverse order. The macroblock in B at (3,3) would be decoded based on block (2,3) in I2 and block (3,5) in I1, both of which are wrong. b) Solid lines show how the updated macroblock dependencies are updated. The decoding is performed on blocks at (2,3) in both the forward and backward direction. The frame can be decoded correctly in both directions.*

Movie	# Frames	Original	New	Loss %
Sukhoi	760	611172	778268	27
Mjackson	564	379168	468100	23
Alien	263	216304	270218	24
Olympics	4008	3383771	4138911	22
3 Stooges	1797	8577387	10859203	26

Table 25: *Loss in compression efficiency when using the SBVS encoding scheme.*

macroblocks to be dependent on specific macroblocks, this difference is higher since the optimal dependent block may not be used.

Table 25 shows that the loss in efficiency from using the above scheme is around 25%. All of the movies listed in the table were compressed with the frame format IBBBBI. The quantization scale used was 8 for I-frames and 16 for P- and B-frames.

During rewind of the file, the frames must be presented to the decoder in reverse order. For implementation purposes, the MPEG syntax is modified and a header is added to each GOP that specifies how the frames should be rearranged to result in a reverse playback. This header is included so that the frames can be rearranged efficiently by a software layer at the

client before being presented to the decoder. The header lists the offset of each frame within the GOP and its size. It also stores an offset to the previous and next header for efficiency in processing. The software at the client preprocesses the arriving data and rearranges the frames in the input buffer of the decoder based on the header of the GOP. To play the stream in fast-forward mode, the client software strips the stream of the header before supplying it to the decoder [Ven96].[3]

8.7 Subjective Analysis

In this section the visual output that is generated by each of these methods is analyzed. Since an objective analysis of the output is beyond the scope of this thesis, a subjective analysis is done. Several subjects were asked to comment on three main aspects of each video, quality, searching ability, i.e., the ability to find a particular scene, and how motion is displayed. Note that the increased playback rate method is not presented since there are no software or hardware devices that can display videos at a speed greater than 30fps.

Version	# Frames	Size (Bytes)	BitRate (bps)	Network (ms)	I/O (ms)
Original	4008	20399475	1008492	28	37
Skip Frames	1336	8552763	1516539	43	58
Skip 2 Segments	1338	5698427	1008933	34	37
Skip Subsegments	1068	6543847	1452251	41	208
Alternate File	1337	4138911	733953	21	35

Table 26: *Clip information for the different versions of the Olympics video. The network time is the time to send each media block during one cycle. The I/O time is the average time to retrieve one media block.*

For the experiment, two MPEG-1 videos were used for the analysis, a one minute clip from a Three Stooges movie where most of the action was people moving at a walking pace. The second video was an Olympic clip from CNN, that combined both head-shots with little motion and fast motion gymnastics. Both clips were encoded at 30fps with a 15 frame open GOP consisting of the frames *IBBPBBPBBPBBPBB*.

For each of the two videos, clips were created that simulated the different fast forward/rewind algorithms. Since the *SBVS* does not implement each of these methods, the clips were created off-line with a program to parse the input MPEG stream and create the appropriate output stream. Tables 26 and 27 show the number of frames, the total size, the average bit rate and the network and I/O delay for each video. The I/O delay is the average

[3]The overhead due to the header is 60 bytes per GOP.

Version	# Frames	Size (Bytes)	BitRate (bps)	Network (ms)	I/O (ms)
Original	1797	11228563	1180574	34	39
Skip Frames	591	7221007	2315977	66	82
Skip 2 Segments	584	3644371	1177279	34	39
Skip Subsegments	591	5731255	1837634	52	209
Alt. File	585	3272963	886204	25	37

Table 27: *Clip information for the different versions of the Three stooges video. The network time is the time to send each media block during one cycle. The I/O time is the average time to retrieve one media block.*

time (in ms) to retrieve data for the stream for one I/O cycle of two seconds. In the skipping frames method it is assumed that the file is reorganized as described previously. Every four GOPs are reorganized into one segment that begins with 4 I-frames, then 16 P-frames and finally 40 B-frames. To retrieve 60 frames (2 seconds), three segments must be accessed, thus, the I/O delay for the skipping frames method includes three extra seeks per cycle. In the skipping subsegment method, four frames are retrieved from each of 15 different GOPs, thus incurring 15 extra seeks. The network delay is simply the time to transfer the data over the network assuming a network transfer speed of 70Mbps. The table shows that the skipping frames and skipping subsegments methods both increase the time needed to retrieve the data from the disk and send it over the network.

The clips will now be enumerated. For each clip, the method used to create the clip is specified, and the subjective analysis done by the three subjects is summarized.

Original

Method: The Three Stooges clip was taken from a commercially produced CD-ROM. The Olympics clip was download from a WWW site that encodes CNN news clips.

Analysis: The quality of both clips is close to VCR quality, which is not as good as broadcast TV.

Skipping Segments/Skip 2 Open GOPS

Method: In this version, the resulting clip contains 1 of every three GOPs where each GOP is an open GOP. Since an open GOP is shown, the first 2 B-frames in each GOP should be decoded using a P-frame in the previous GOP.

Analysis: The resulting clip is jerky and slightly uncomfortable because of the long skips of frames where 30 out of every 45 frames are completely skipped. It did not flow smoothly since there was a break in the action every half second. The actors were moving at normal speed but motion was skipped. Thus, it did not look like fast forward. Also, when showing

an open GOP, the first two B-frames depend on the P-frame in the previous GOP. But since the previous GOP is skipped, the decoder uses the wrong reference frames when decoding the first two B-frames. The resulting clip is very garbled. Although this was not a preferred method, the subjects felt that it could be used as a searching mechanism if one did not have to use this display for a long period of time.

Skipping Segments/Skip 2 Closed GOPS

Method: To create this version, the first two B-frames in each GOP are removed since they depend on a frame in another GOP. This creates a closed GOP. Then two of every three GOPs are removed from the file.

Analysis: This is a more acceptable version than the previous version that uses open GOPs. The quality is better since all of the frames are decoded properly, but again, the clips are jerky.

Skipping Segments/Skip 2 Closed GOPS in reverse

Method: The GOPs in the above video are reversed, but the display of each GOP, however, is performed in the forward direction.

Analysis: Like the two previous clips, this was also jerky. Although, this version seemed to be acceptable for searching, it was sometimes confusing because the actors were still moving forward. In some instances a scene change occurred in the middle of the GOP which also created confusion. For example, assume the 15 frames in GOP 1 are from scene A, of the 15 frames in GOP 2, 10 are from scene A and 5 frames are from scene B, and 15 frames from GOP 3 are from scene B. When showing the individual frames of the GOPs in the forward direction but the independent GOPs in reverse order, the order would be 15 frames from scene B, 10 from scene A, 5 from scene B, and then 15 from scene A. Rather than a smooth transition from scene B to scene A, it went back and forth.

Skipping Subsegments

Method: In this version, all but the first four frames of each GOP are stripped from the original. The first four frames are an independent sequence and can be decoded independently. Since 4 of the original 15 frames are shown a speedup up 3.75 is the result.

Analysis: This version seemed to be more jerky than the skipping GOP method since frames were skipped every $\frac{2}{15}$ of a second. It was harder to tell if the characters were moving normally or in fast motion. The subjects complained that this was the most "*disturbing*" clip because of the jerkiness and felt that it would be hard to use for long periods of time. On the other hand, again, they felt that it could be used to find a specific scene.

Skipping Subsegments in reverse

Method: The GOPs in the above video are reversed, but each independent sequence was shown in the forward direction.

Analysis: This was acceptable for searching, but again, was uncomfortable to watch compared to the other methods.

Skipping Frames/I and P- frames only

Method: All of the B-frames are removed from each clip. Since there are two B-frames to every I or P-frame, this results in a speedup of 3.

Analysis: This version looks like true fast forward. The actors in the clips seem to be moving in fast motion. This was the preferred method for fast forward. It was not uncomfortable to watch and was easy to use for searching.

Skipping Frames/I-frames only

Method: Both P- and B-frames are removed. Since there is a single I-frame for every 15 frames, a speedup of 15 is achieved. The actual speedup rate, however, is lower than 15 since the hardware decoder cannot decode 15 I-frames every second. The measured speedup at the decoder was about five.

Analysis: This was comfortable to watch. It seemed that the actors were moving in very fast motion. As for searching, it was acceptable using the Stooges clip since most of the movement was at normal walking speed. The version was not acceptable for the fast gymnastics motion in the Olympics clip. Skipping 14 of every 15 frames during fast motion resulted in missed information. The gymnastics action occurred too fast to follow.

Skipping Frames/I-frames in reverse

Method: Both P and B-frames are removed, then the resulting I-frames are reversed.

Analysis: The characters seem to be moving backwards in fast motion. Like the previous version of I-frames going forward, this version was acceptable for the Stooges clip but unacceptable for the Olympics clip.

Skipping Frames/Duplicate I-frames in reverse

Method: Since the previous version was unacceptable using the Olympics clip, this version is created by removing all of the P and B-frames, duplicating each I-frame and then reversing the frames.

Analysis: This was a more acceptable method for the Olympic clip since the fast motion was slowed down to an acceptable rate.

Re-encoded Non-Updated File

Method: In this version the compressed, original file is first uncompressed. Every third frame from the uncompressed version is re-encoded into a new file with a GOP of IBBBBBBI. This is equivalent to skipping frames. The preferred method of encoding this file would be to use the original uncompressed movie version before encoding but it was unavailable. Since the MPEG compression standard is lossy, quality is degraded since the clip is compressed, uncompressed and then compressed again.

Analysis: This version is very acceptable. The action is smooth and it looks as though the actors are moving in fast motion. The quality is slightly lower than the skipping frames method, but only because the video used for encoding was not the original uncompressed version, but the result of uncompressing the compressed version.

Re-encoded Non-Updated File in Reverse

Method: This version is created by reversing all of the frames created in the previous version.

Analysis: The resulting clip is unacceptable. As metioned previously, B-frames have macroblocks that are dependent on macroblocks in reference frames in both the forward and backward directions. By reversing the direction of the frames, the referenced macroblocks are wrong and the quality is quite bad.

Re-encoded Updated File

Method: In this version the original file is first uncompressed. Every third frame from the uncompressed version is re-encoded into a new file with a GOP of IBBBBBBI. In addition, the B frames are updated so that the referenced macroblocks are equal for both the forward and backward references. This assures that the file can be shown in both the forward and backward directions.

Analysis: Since the encoder may not use the optimal reference block, the quality is not as good as in the previous version where the B-frames are not updated.

Re-encoded Updated File, 4-16 in Reverse

Method: This version is created by reversing all of the frames created in the previous version. Again, the B-frames are updated.

Analysis: The quality in this version is acceptable, compared to the version where the B-frames were not updated. So, although the fast forward version of the non-updated clip is better than the updated clip, the rewind version of the updated clip is much better than the non-updated clip. Thus, it is a tradeoff. For a slightly lower quality fast forward version, the rewind version is acceptable.

8.8 Summary

Version	Bit Rate	SpeedUp	Implementation	Visual Output
Increased playback rate	Proportional to speedup rate	Any	Cannot be done with current hardware	Unknown
Skip GOPs	No Increase if 1 GOP/disk	Any	Possible problems with MPEG system streams	Jerky but acceptable
Skip Segments	$> 50\%$ increase	Limited	Possible problems with MPEG system streams	Very jerky. Least favorite
Skip Frames	$> 100\%$ increase	Limited	Special processing at client	Looks like fast forward/ fast rewind
Alternate File	No Increase	1 per file	Extra Storage	Looks like fast forward/ fast rewind. Slight loss in quality.

Table 28: *Summary of the various methods for implementing fast forward rewind in a distributed video server.*

Table 28 briefly summarizes the various aspects of each FF/FR method. Normally, when comparing various alternative algorithms, one would like to fix all but one parameter and make a comparison based on individual parameters. For example, to compare FF/FR methods, the bit rate and amount of storage should be fixed and then the quality can be compared. In this case, however, this is not possible. The resulting bit rate and quality are fixed for each method.

It is clear that there is no "best" solution. Each of the alternatives has its advantages and disadvantages and represents a tradeoff. For example, the alternate file method could be shown forward and backward but each speedup rate that is supported increases the storage capacity needed. The other methods don't increase the storage space needed but may increase the bit rate for FF/FR.

The underlying system may also dictate which method is used. In the *SBVS*, the server knows nothing about frames or GOPs. It simply stores a file consisting of fixed blocks that must be delievered at a certain rate. Thus, for the *SBVS* to use the skipping GOP method, additional complexity must be added to the system for the server to recognize GOP boundaries.

It was also clear though that scene content had an effect on the display. When displaying only I-frames in the Stooges video which had normal walking motion, the display was good

and smooth. But when displaying only I-frames in the Olympics video, during periods of fast motion, the clip looked jumpy.

More work needs to be done in the area of visual display. Our assumption has been that a video server should support the same functions as our VCR, but should it? Is fast forward used only for scanning for a particular scene? If that is the case, there may be other methods for more efficient searching that are not discussed here.

Chapter 9

Conclusion

The high speed and low cost of commodity computer hardware has made it feasible to use off-the-shelf computers for the delivery of digital audio and video. Over the last several years, both researchers and systems builders have given the *Video Server* much attention. Some have focused on the *Large-Scale Video Server* that is intended to support thousands of users over a wide-area network. This work, however, focuses on the *LAN-Based Video Server*, a small-scale system that is intended to support up to 100 users.

The *LAN-Based Video Server* must be able to provide the real-time delivery of video streams from the server's I/O subsystem, through the interconnecting local-area network, to the client's display. The *Stony Brook Video Server (SBVS)* described in this thesis is a system that provides exactly such an end-to-end performance guarantee. The *SBVS* prototype is based on off-the-shelf PC commodity hardware components. The *SBVS* software consists of a real-time disk array-based video storage subsystem integrated with a real-time Ethernet protocol.

Whereas most related work on video servers has been analytical in nature and based on simulations, this thesis presents a detailed design and implementation description, and a performance study of a working system. At the start of this project, there were no descriptions of concrete implementations, nor any performace studies of working systems. Thus, being experimental researchers, we chose to base our research on a working prototype. Like analytical and simulation methods that provide general insight into certain problems, the building of experimental prototypes provides lessons and insight that can only be acquired by implementing and analyzing an end-to-end complete system. The lessons learned from building the *SBVS* are included throughout this thesis and it is hoped that other researchers can use these experiences as building blocks for future development.

This chapter outlines the research contributions of this thesis and presents ideas for future work.

9.1 Research Contributions

There are two major software subsystems in the *SBVS—RETHER*, a real-time Ethernet-based protocol that allows applications to reserve network bandwidth on an existing Ethernet, and the *Video Storage Subsystem (VSS)*, a software-driven disk-array subsystem that provides real-time access to striped video files.

At the beginning of the *SBVS* project, *RETHER* had already been designed and a prototype system was in operation[Ven96]. This thesis focuses on the design and implementation of the *VSS*, the performance of the *VSS* in isolation, and the performance of the *VSS* integrated with *RETHER*. In addition to presenting the design of the *VSS* module, Chapter 4 discusses the central issues involved in building a video storage server. One of the first design decisions to be made concerns the size of a media block. Video files can be broken into *Constant Data Length (CDL)* blocks, e.g., 64KB, or *Constant Time Length (CTL)* blocks, e.g., 1 second's worth of data. This design decision is crucial, since the size of the media block has implications in many other design issues, including buffer management, disk load balancing, admission control, and network communication. Since *RETHER* had already been designed and used a *CDL* block for network transmission, the *VSS* was also designed to use a *CDL* block. It would have been extremely complicated to incorporate a *VSS CTL*-block-based system with a *RETHER CDL*-block-based system. In general, whenever a design choice had to be made, the choice that incurred the least implementation complexity was used. This allowed us to quickly build a working prototype. It also shows that one can build a low-cost video server without using any complicated storage policies or buffer mechanisms.

Details of the *VSS* implementation is presented in Chapter 5. The implementation description includes the specific algorithms that are used by the *VSS* to retrieve media blocks from the disks for both forward and backward retrieval. Implementation details of the buffer management scheme, *double buffering*, are also presented. Lastly, the chapter includes the specific integration mechanisms between *RETHER* and the *VSS*. In the first prototype (*SBVS-1*), an EISA-based system, bus contention and slow programmed-I/O peripherals created the need for explicit scheduling of the EISA bus to meet the performance target. In the *SBVS-2*, a PCI-based system, DMA peripherals were used and no explicit scheduling of the PCI bus had to be performed.

Chapter 6 presents a detailed performance evaluation of the *VSS*. The raw performance of the disk array is compared with the cycle-based video server retrieval algorithms. Due to the synchronization requirements at the end of each I/O cycle, about 10% utilization is lost. Using six SCSI-2 disks and three SCSI bus controllers, the *VSS* can support up to 89 MPEG-1 streams at 1.5Mbps. Also in Chapter 6, a performance evaluation of the running end-to-end *SBVS* video server is presented. When supporting a single Ethernet segment, results show that the integration of the *VSS* subsystem with *RETHER* had no effect on the

maximum number of streams that can be supported by either subsystem. When using a disk array that has a higher bandwidth than the network, the maximum number of streams is only limited by the network bandwidth. In addition, the low software overhead of the *VSS* and *RETHER* subsystems allows for efficient utilization of the available hardware bandwidth. It was shown that the maximum bandwidth of a single network, when running in isolation with no video server overhead, was 70Mbps, while the complete integrated system has a maximum network bandwidth of 68Mbps. Measurements also show that, using a Pentium-90 CPU, a four-drive disk array, and 100Mbps Fast Ethernet, the *SBVS* is capable of delivering 45 MPEG-1 streams, each at 1.5 Mbps. This work is unique in that it provides the first detailed implementation and performance evaluation of a *CDL*-based, working end-to-end video server.

The performance implications of various system parameters, such as the I/O cycle time, network token cycle time, and disk array organizations, are examined in detail. Finally, the scalability issues and performance of running the *SBVS* over two networks is presented. In the case of two networks, the CPU overhead for processing the network packets and I/O load is the bottleneck and should be alleviated by a faster processor.

In Chapter 7, the I/O-based admission control problem is stated and a framework for deciding whether or not to admit a new stream is given. Five different algorithms are presented in which each algorithm uses various values to evaluate an admission control formula. These algorithms are then compared on the working *VSS* under a variety of hardware configurations and workloads. This is the first known work to present an empirical evaluation and comparison of video storage server admission control policies. It is shown that the statistical-based algorithm that uses measurements of the running system allows more streams than a deterministic algorithm which uses worst-case values. In addition, contrary to the results reported by earlier work based on simulations, the performance gap between the *Statistical* and *Deterministic* algorithms is between 10% to 40%, which is much smaller than was previously believed. This discrepancy is due to the over-simplified assumptions by the simulators as well as other fixed software/hardware overheads that are completely overlooked. The experimental approach used in these experiments completely eliminates all of these problems.

In addition to supporting normal playback, the *SBVS* supports VCR-functions, such as pause, resume, jump forward/backward, and fast forward/rewind. The design and implementation for supporting the VCR-like functions are included in Chapters 4 and 5. In addition, Chapter 8 presents a detailed analysis of methods for supporting fast forward and fast rewind. The analysis includes a comparison of implementation complexities, resource usage, and visual output for each of the methods. The comparison reveals that there is no clear "*best*" method. Each method has its advantages and disadvantages. The analysis

can be used by designers to choose the appropriate method, given the constraints of each individual environment.

9.2 Future Work

When scaling the *SBVS* to two networks, the Pentium 90 CPU was shown to be the bottleneck. Our first task will be to upgrade the CPU of the video server to a Pentium Pro 200 to alleviate this bottleneck.

Currently, users make requests to watch a video simply based on the name of the video. We are planning to provide, however, a user-friendly front-end to access the video sequences such as speech-based video indexing.

As more disks are added to the *SBVS*, the stripe size is reduced since media blocks are striped across all disks in the array. When the stripe size is lowered, the ratio of disk overhead to disk transfer time increases, thus reducing the effective utilization of the disks. An alternative is to stripe the media blocks across a subset of the disks, as in the staggered-striping approach. In the *SBVS*, we plan to stripe data across disks on the same SCSI bus, rather than across buses. Issues of load-balancing must then be addressed.

In any server, when data flows from the disk to the network, it first travels from the disk, over the I/O bus, into main memory, and then back over the I/O bus to network controller. We are now investigating the implemention of an integrated disk/network controller so that data can flow from the disk to the network directly, bypassing the system bus [SV96].

In the current prototype, if a disk fails, all of the movies must be re-written to the remaining disks in the array. As more disks are added to the system, the probability that a disk failure will occur increases. While the data is rewritten, the system is not usable. A mechanism is needed to handle single disk failures without shutting down the system.

In most video servers, the server is responsible for handling fast rewind. Another possibility is for the client to handle rewind. In this method, the client is responsible for saving processed frames in a temporary buffer. The user can rewind through the saved frames with no interaction from the server.

Lastly, our group is designing a broadcast-based video server architecture that takes the *SBVS* into the realm of large-scale video services [Chi95]. In a video-on-demand server like the *SBVS*, users can request to start a video at any time. In this environment, one stream on the server serves one user. In a broadcast-based system, videos are shown at periodic intervals, say every five minutes. In this case, a single stream can serve many users, thus decreasing the cost per user.

Bibliography

[AH91] D. Andersond and G. Homsy. A continuous media i/o server and its synchronization. *IEEE Computer*, page 51, October 1991.

[AOG92] D. Anderson, Y. Osawa, and R. Govindan. A file system for continuous media. In *ACM Transactions on Computer Systems*, volume 10, page 331, Nov 1992.

[B+96] W. Bolosky et al. The tiger video fileserver. In *Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1996.

[Bau93] M. Baugher. A multimedia client to the ibm lan server. In *Proceedings of the ACM Multimedia*, Anaheim, CA, Aug 1993.

[BB95] C. Bernhardt and E. Biersack. A scalable video server: Architecture, design, and implementation. In *Realtime Systems Conference*, page 63, Paris, France, Jan 1995.

[BGMJ94] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered striping in multimedia information systems. In *Proceedings of the Fifth International Conference on Management of Data*, May 94.

[Bir95] Y. Birk. Track-pairing: a novel data layout for vod servers with multi-zone-recording disks. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 248, Washington, DC, 1995.

[BMC94] P. Bocheck, H. Meadows, and S. Chang. Disk partitioning technique for reducing multimedia access delay. In *Proceeding of the IASTED/ISMM International Conference. Distributed Multimedia Systems and Applications*, page 27, Honolulu, HI, Aug 1994.

[BP95] M. Buddhikot and G. Parulkar. Efficient data layout, scheduling and playout control in mars. In *Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1995.

[CBR95] A. Cohen, W. Burkhard, and P. Rangan. Pipelined disk arrays for digital movie retrieval. In *Proceedings ACM Multimedia*, page 312, San Fransisco, CA, 1995.

[Chi95] T. Chiueh. A periodic broadcasting approach to video-on-demand service. In *SPIE First International Symposium on Technoloigies and Systems for Voice, Video and Data Communications*, volume 2615, page 162, Philadelphia, PA, Oct 1995.

[CK95] M. Chen and D. Kandlur. Downloading and stream conversion: Supporting interactive playout of videos in a client station. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 73, Washington, DC, 1995.

[CKLV95] H. Chen, A. Krishnamurthy, T. Little, and D. Venkatesh. A scalable video-on-demand service for the provision of vcr-like functions. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 65, Washington, DC, 1995.

[CKY95] M. Chen, D. Kandlur, and P. Yu. Storage and retrieval methods to support fully interactive playout in a disk-array-based video server. *Multimedia Systems*, 3:126, 1995.

[Con95] J Conover. Fast, fast, fast. In *Network Computing*, page 46. CMP Media Inc., Nov 1 1995.

[CP90] P. Chen and D. Patterson. Maximizing performance in a striped disk array. In *Proceedings of ACM SIGARCH Conference on Computer Architecture*, page 322, Seattle, WA, 1990.

[CW95] T. Chiueh and L. Wang. Burst handling of digital video traffic. In *State University of New York at Stony Brook, 1995*, 1995.

[CZ94] E. Chang and A. Zakhor. Scalable video data placement on parallel disk arrays. In *SPIE International Symposium on Imaging Technology*, 1994.

[CZ96] E. Chang and A. Zakhor. Cost analyses for vbr video servers. In *SPIE International Symposium on Electronic Imaging, Science and Technology; Multimedia Computing and Networks*, volume 2667, page 1, San Jose, CA, July 1996.

[CZ94] E. Chang and A. Zakhor. Variable bit-rate mpeg video storage on parallel disk arrays. In *Proceedings IEEE First International Workshop on Community Networking*, page 127, San Francisco, CA, July 94.

[DS94] S. Daigle and J. Strosnider. Disk scheduling for multimedia data streams. In *Proceedings SPIE—International Society of Optical Engineers: High-Speed Networking and Multimedia Computing*, volume 2188, page 212, San Jose, CA, Feb 1994. SPIE.

[DSSKT94] J. Dey-Sinclair, J. Salehi, J. Kurose, and D. Towsley. Providing vcr capabilities in large-scale video servers. In *Proceedings ACM Multimedia*, page 25, San Fransisco, CA, Oct 1994.

[DT94] Y. Doganata and T. Tantawi. A cost/performance study of video servers with hierarchical storage. In *IEEE International Conference on Multimedia Computing and Systems*, Boston, MA, 1994.

[FD95] C. Freedman and D. DeWitt. The spiffi scalable video-on-demand system. In *Proceedings of ACM Multimedia*, page 353, San Jose, CA, Nov 1995.

[FJS95] W. Feng, F. Jahanian, and S. Sechrest. Providing vcr functionality in a constant quality video-on-demand transporation service. Technical Report 271-95, University of Michigan, Ann Arbor, MI, Dec 1995.

[FR94] C. Federighi and L. Rowe. A distributed hierarchical storage manager for a video-on-demand system. In *IS&T/SPIE Symposium on Electronic Imaging, Science, and Technology: Storage and Retrieval for Image and Video Databases II*, page 1, San Jose, CA, Feb 1994.

[Fur94] B. Furht. Multimedia systems: An overview. *IEEE Multimedia*, page 47, Spring 1994.

[FV90] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. In *IEEE Journal on Selected Areas in Communications*, page 386, Apr 1990.

[Gem93] D. Gemmell. Multimedia network file servers: Multi-channel delay sensitive data retrieval. In *Proceedings ACM Multimedia*, page 243, Anaheim, CA, 1993.

[GMS86] H. Garcia-Molina and K. Salem. Disk striping. In *Proceedings of the IEEE International Conference on Data Engineering*, page 332, Feb 1986.

[GVKR95] D. Gemmell, H. Vin, D. Kandlur, and P. Rangan. Multimedia storage servers: A tutorial and survey. In *IEEE Multimedia*, May 1995.

[Has93] R. Haskin. The shark continuous-media file server. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 12, Washington, DC, 1993.

[HK94] K. Huynh and T. Khoshgoftaar. Performance analysis of advanced i/o architectures for pc-based video servers. *Multimedia Systems*, 2:36, 1994.

[HLL+15] J. Hsieh, M. Lin, J. Liu, D. Du, and T. Ruwart. Performance of a mass storage system for video-on-demand. *Parallel and Distributed Computing*, 30:147, 19915.

[JSCB95] D. Jadav, C. Srinilta, A. Choudhary, and P Berra. Design and evaluation of data access strategies in a high performance multimedia-on-demand server. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 286, Washington DC, 1995.

[LK93] E. Lee and R Katz. An analytic performance model of disk arrays. In *Proceedings of ACM SIGMETRICS*, page 98, May 1993.

[LKB87] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, page 69, 1987.

[LL95] S. Lau and J Lui. A novel video-on-demand storage architecture for supporting constant frame rate with variable bit rate retrieval. In *Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, page 294, Durham, NH, Apr 1995.

[LOP94] A. Laursen, J. Olkin, and M. Porter. Oracle media server: providing consumer based interactive access to multimedia data. In *ACM SIGMOD*, volume 2, page 470, 1994.

[LS93] P. Lougher and D. Shepherd. The design of a sotrage server for continuous media. *The Computer Journal*, 36(1):32, 1993.

[LV94] T. Little and D Venkatesh. Prospects for interactive video-on-demand. *IEEE Multimedia*, page 14, Fall 1994.

[Mou94] A. Mourad. I/o scheduling in a storage server for video-on-demand applications. In *Proceeding of the IASTED/ISMM International Conference. Distributed Multimedia Systems and Applications*, page 31, Honolulu, HI, Aug 1994.

[Mou96] A. Mourad. Issues in the design of a storage server for video-on-demand. *Multimedia Systems*, 4:70, 1996. General paper.

[NR93] J. Wyllie N. Reddy. Disk scheduling in a multimedia i/o system. In *Proceedings ACM Multimedia*, page 225, Anaheim, CA, Aug 1993.

[NT94] T. Nakajima and H. Tezuka. A continuous media application supporting dynamic qos control on real-time mach. In *Proceedings of ACM Multimedia*, page 289, San Fransisco, CA, Oct 1994.

[OBRS95] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A low-cost storage server for movie on demand databases. *Multimedia Systems*, 1995.

[ORS95] B. Ozden, R. Rastogi, and A. Silberschatz. A framework for the storage and retrieval of continous media data. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 2, 1995.

[ORS96] B. Ozden, R. Rastogi, and A. Silberschatz. 'disk striping in video server environments. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. IEEE, Jun 1996.

[PBC95] S. Paek, P. Bocheck, and S. Chang. Scalable mpeg2 video servers with heterogeneous qos on parallel disk arrays. In *Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, page 342, Apr 1995.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD Confere nce on Management of Data*, Chicago, IL, June 1988.

[PV93] Rangan P. and H. Vin. Efficient storage techniques for digital continuous multimedia. In *IEEE Transactions on Knowledge and Data Engineering*, page 564, Aug 1993.

[PY92] D. Kandlur P. Yu, M. Chen. Design and analysis of a grouped sweeping scheme for multimedia storage management. In *Proceedings Third International Workshop on Network and Operating Systems Support for Digital Audio and Video*, page 44, La Jolla, CA, Nov 1992.

[RV91] P. Rangan and H. Vin. Designing file systems for digital video and audio. In *Proceedings of the 13th Symposium on Operating Systems Principles, Operating Systems Review*, volume 25, page 81, October 1991.

[RVR92] P. Rangan, H. Vin, and S. Ramanathan. Designing an on-demand multimedia service. *IEEE Communications Maganzine*, page 56, July 1992.

[scfdv92] Client server challenges for digital video. Tobagi, f. and long, j. In *Digest of Papers IEEE Spring CompCon*, page 88, Spring 1992.

[SGM94] M. Saito and H. Garcia-Molina. Independent access array for continuous media storage management. In *Proceeding of the IASTED/ISMM International Conference. Distributed Multimedia Systems and Applications*, Honolulu, HI, Aug 1994.

[Sta96] Starlight Networks. *http://www.starlight.com*, 1996.

[SV95] P. Shenoy and H. Vin. Efficient support for scan operations in video servers. In *Proceedings ACM Multimedia*, page 131, San Jose, CA, Nov 1995.

[SV96] G. Schloss and M. Vernick. *Input/Output in Parallel and Distributed Systems*, chapter HCSA: A Hybrid Client-Server Architecture. Kluwer Publishers, Aug 1996.

[TKS94] W. Tetzlaff, M. Kienzle, and D. Sitaram. A methodolgy for evaluating storage systems in distributed and hierarchical video servers. In *Digest of Papers of IEEE CompCon*, page 430, San Fransisco, CA, 1994 1994.

[TP72] T. Teorey and T.B. Pinkerton. A comparative analysis of disk scheduling policies. In *Communications of the ACM*, page 177, Mar 1972.

[TPBG93] F. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming raid - a disk array management system for video files. In *Proceedings ACM Multimedia*, page 393, Anaheim, CA, 1993.

[Tri95] C. Tristram. Stream on: Video servers in the real world. *New Media*, page 46, April 1995.

[VC94] C. Venkatramani and T. Chiueh. Supporting real-time traffic on ethernet. In *Real-Time Systems Symposium*, page 282, San Juan, Puerto Rico, Dec 1994.

[Ven96] C. Venkatramani. *The Design, Implementation and Evaluation of RETHER: A Real-Time Ethernet Protocol.* PhD thesis, State University of New York at Stony Brook, Stony Brook, NY, Dec 1996.

[VGG94a] H. Vin, A. Goyal, and P. Goyal. An observation-based admission control algorithm for multimedia servers. In *Proceedings of the First IEEE International Conference on Multimedia Computing and Systems*, page 234, Boston, MA, 1994.

[VGG94b] H. Vin, P. Goyal, and A Goyal. A statistical admission control algorithm for multimedia servers. In *Proceedings ACM Multimedia*, page 33, San Francisco, CA, Oct 1994.

[VR93] H. Vin and P Rangan. Designing a multi-user hdtv storage server. *IEEE Journal on Selected Areas in Communications*, 11:153, Jan 1993.

[VRG95] H. Vin, S. Rao, and P. Goyal. Optimizing the placement of multimedia objects on disk arrays. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 158, Washington, D.C., 1995.

[VVC96] M. Vernick, C. Venkatramani, and T. Chiueh. Adventures in building the stony brook video server. In *Proceedings ACM Multimedia*, Nov 1996.