Technical Report

UCAM-CL-TR-438
ISSN 1476-2986

Number 438

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

An architecture for scalable and deterministic video servers

Feng Shi

November 1997

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted June 1997 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation is not substantially the same as any that I have submitted for a degree or diploma or any other qualification at any other University.

No part of this dissertation has already been or is being currently submitted for any such degree, diploma, or other qualification.

This dissertation, including tables, footnotes, and bibliography, but excluding appendices, photographs, and diagrams, does not exceed 60,000 words.

Acknowledgements

I would like to thank my supervisor, Andy Hopper, for the opportunity, advice, and encouragement to pursue this research.

I am also grateful to Martin Brown for his incessant chasing up, practical assistance, and fine proofreading of the whole dissertation.

Many thanks and much appreciation also go to various members of the Computer Laboratory, and of the Olivetti and Oracle Research Laboratory (ORL), notably, Mike Addlesee, Jean Bacon, Hiang-Swee Chiang, Damian Gilmurray, Gray Girling, Andy Harter, Alan Jones, Sai Lai Lo, Ken Moody, Brendan Murphy, Antony Rowstron, and Ian Wilson, for their help and discussion during the course of this research.

I am indebted to ORL and the Computer Laboratory for the use of equipment and computing facilities, with special thanks to Ian Luff for setting up the Cadmus project file space in ORL.

The first three years of this work was supported by a scholarship from the Cambridge Overseas Trust, and the final year was sponsored by ORL.

Summary

A video server is a storage system that can provide a repository for continuous media (CM) data and sustain CM stream delivery (playback or recording) through networks. The voluminous nature of CM data demands a video server to be scalable in order to serve a large number of concurrent client requests. In addition, deterministic services can be provided by a video server for playback because the characteristics of a variable bit rate (VBR) video can be analysed in advance and used in run-time admission control (AC) and data retrieval.

Recent research has made gigabit switches a reality, and the cost/performance ratio of microprocessors and standard PCs is dropping steadily. It would be more cost-effective and flexible to use off-the-shelf components inside a video server with a scalable switched network as the primary interconnect than to make a special purpose or massively parallel multiprocessor based video server. This work advocates and assumes such a scalable video server structure in which data is striped to multiple peripherals attached directly to a switched network.

However, most contemporary distributed file systems do not support data distribution across multiple networked nodes, let alone providing quality of service (QoS) to CM applications at the same time. It is the observation of this dissertation that the software system framework for network striped video servers is as important as the scalable hardware architecture itself. This leads to the development of a new system architecture, which is scalable, flexible, and QoS aware, for scalable and deterministic video servers. The resulting architecture is called Cadmus, named from sCAlable and Deterministic MUltimedia Servers.

Cadmus also provides integrated solutions to AC and actual QoS enforcement in storage nodes. This is achieved by considering resources, such as CPU, buffer, disk, and network, simultaneously but not independently and by including both real-time (RT) and non-real-time (NRT) activities. In addition, the potential to smooth the variability of VBR videos using read-ahead under client buffer constraints is identified. A new smoothing algorithm is presented, analysed, and incorporated into the Cadmus architecture.

A prototype implementation of Cadmus has been constructed based on distributed object computing and hardware modules directly connected to an Asynchronous Transfer Mode (ATM) network. Experiments were performed to evaluate the implementation and to demonstrate the utility and feasibility of the architecture and its AC criteria.

Contents

List of Figures

List of Tables

Glossary of Terms

The page number after each term represents the place where the term is first introduced in the body of this dissertation.

AC	Admission Control (p 2)
AFS	Andrew File System (p 13)
ATM	Asynchronous Transfer Mode (p 3)
BE	Best Effort (p 16)
CBR	Constant Bit Rate (p 20)
CM	Continuous Media (p 1)
CORBA	Common Object Request Broker Architecture (p 13)
CP	Client Part (p 37)
DCE	Distributed Computing Environment (p 13)
DS	Directory Server (p 36)
ECC	Error Correction Code (p 8)
EDF	Earliest Deadline First (p 16)
FC	Fibre Channel (p 9)
FC-AL	Fibre Channel Arbitrated Loop (p 12)
FCFS	First-Come-First-Served (p 72)
FFV	Fast Forward with Viewing (p 86)
FS	File Server (p 36)
FSID	File Server Identifier (p 42)
FS_ATTR	File Server Attributes (p 42)
GPS	Global Positioning System (p 89)
ID	Identifier (p 25)
IDL	Interface Definition Language (p 13)
LO	Logical Object (p 37)
LOID	Logical Object Identifier (p 37)
LO_ATTR	Logical Object Attributes (p 41)
MPEG	Motion Picture Expert Group (p 5)
NAP	Network Attached Peripheral (p 26)

NC	Network Computer (p 36)
NRT	Non-Real-Time (p 2)
NS	Name Server (p 36)
NTP	Network Time Protocol (p 89)
ORB	Object Request Broker (p 13)
OSF	Open Software Foundation (p 13)
PO	Physical Object (p 37)
POID	Physical Object Identifier (p 37)
PO_ATTR	Physical Object Attributes (p 42)
QoS	Quality of Service (p 2)
RAID	Redundant Array of Inexpensive Disks (p 9)
RM	Rate Monotonic (p 16)
RPC	Remote Procedure Call (p 13)
RT	Real-Time (p 2)
SCA	Stream Control Agent (p 37)
SCF	Stream Control Factory (p 37)
SCSI	Small Computer System Interface (p 9)
SS	Storage Server (p 36)
SUS	Striping Unit Size (p 10)
VBR	Variable Bit Rate (p 2)
VOD	Video On Demand (p 21)
ZBR	Zoned Bit Recording (p 8)

Chapter 1

Introduction

This chapter first describes the motivation for the research topics in this dissertation. The research contributions are identified and an overview of the other chapters is given.

1.1 Motivation

A video server is a storage system that provides a repository for continuous media (CM) data and sustains CM stream delivery (playback or recording) to or from clients through networks.

Unlike conventional data, a CM stream has a temporal requirement which places timeliness constraints on its data retrieval and processing, although it is tolerant to a certain degree of data loss or error during transmission. In addition, CM data is voluminous even after some form of compression, demanding high I/O bandwidth and large storage capacity.

For reasons of economy, large amounts of CM data will still be stored in magnetic disks in a video server in the foreseeable future. While the performance of CPUs and memory is increasing rapidly, the access time and transfer rates of disks have improved only modestly. The I/O bandwidth of many concurrent CM streams is well beyond the capability of one disk or a small scale disk array, so is the storage requirement of many different audio/video files. Consequently, for a large scale video server, multiple disk devices must be used to exploit their aggregate bandwidth and storage capacity.

From the perspective of the hardware architecture of a video server, the most important aspect is the interconnect structure between disk devices and clients. Generally, there are four types of interconnect: backplanes, channels, parallel computer interconnection, and networks. As recent research has made gigabit switches a reality, and the cost/performance ratio of microprocessors and standard PCs is dropping steadily, it would be more cost-effective and flexible to use off-the-shelf components inside a video server with a scalable switched network as the primary interconnect than to make a special purpose or massively parallel multiprocessor based video server.

This work assumes such a video server structure in which storage peripherals are attached directly to a switched network, and data is striped to multiple peripherals, each of which could be a single disk or a small scale disk array using a backplane or a non-switched channel internally. This server structure is scalable because the peripherals and the network

can be upgraded independently. It also has high performance because striping effectively utilises the aggregate I/O bandwidth and storage capacity of multiple devices.

However, most contemporary distributed file systems do not support data distribution across multiple networked nodes, let alone providing quality of service (QoS) to CM applications at the same time. It is the observation of this dissertation that the software system framework for network striped video servers is as important as the scalable hardware architecture itself. The framework should be flexible by providing basic mechanisms, but not dictating as many policies. It should also be scalable to accommodate the scalability of the hardware structure. Such a flexible and scalable software architecture is the main focus of this research.

It would be advantageous for a video server to provide deterministic services to its clients, something which is possible and is also assumed by this work. However, most current video server research addresses QoS related problems by making assumptions about its environment, for example, no network activities, no meta-data management, or no operating system support. Hence, an integrated and practical solution to admission control (AC) and QoS enforcement in peripheral nodes is very important and is another aim of this work.

Most compression techniques for CM data produce variable bit rate (VBR) streams, which affect the efficiency of data retrieval from disks and complicate the resource allocation and the scheduling procedures in the whole system. Thus it is desirable to reduce this variability, which is possible because most of the time a video server deals with the playback of stored videos which can be subject to off-line or on-line analysis before their data is retrieved and sent to clients. A suitable smoothing method for VBR streams is the third target of this dissertation.

In summary, the research context is scalable network striped video servers providing deterministic services, and the research areas are software system architecture, VBR data smoothing, and integrated AC and QoS enforcement in storage nodes. The emphasis is also on implementing and evaluating a working prototype incorporating these three areas. The only assumption made in this research is that switched networks are not the bottleneck, and no QoS consideration is given to network communication.

1.2 Contributions

This work proposes a new system architecture, code named Cadmus[1], for network striped video servers. Cadmus separates the physical storage from the logical view of a data object, and stores system meta-data in the same places as CM data resides. Special components to control CM streams have been added as inherent parts of Cadmus, but staying outside the storage mechanisms. These characteristics distinguish Cadmus from other video server file systems and result in a scalable and flexible architecture.

AC and QoS enforcement in storage nodes are also described. The solution is integrated for two reasons. First, resources, such as CPU, buffer, disk, and network, are considered not independently but simultaneously. Second, non-real-time (NRT) activities coexist with their real-time (RT) counterparts but without disturbing the QoS to the latter. These are two important aspects that make a real system work, and they are both exhibited in the Cadmus prototype.

[1]Cadmus: sCAlable and Deterministic MUltimedia Servers, or: a Phoenician prince held in Greek Myth to have killed a dragon and sown its teeth from which sprang armed men who fought together.

A VBR data smoothing algorithm is presented. The algorithm takes into account client buffer limitations, performs or suppresses read-ahead, smoothes the data read in each fixed period, and eliminates small reads from disks. The algorithm is also formally and empirically analysed and the results are used as guidance for determining striping block sizes for VBR streams in Cadmus.

The above concepts are validated by describing the implementation and evaluation of a working prototype system. The Cadmus prototype is built on hardware modules that are directly connected to an Asynchronous Transfer Mode (ATM) network, and different system components interact through distributed object invocation. An NRT micro-kernel and its system level processes have been extended to provide QoS support in storage nodes together with one Cadmus component.

1.3 Outline

The chapters in this dissertation are arranged as follows:

Chapter 2 introduces the background material which is relevant to this work. The rationale for network striped video server structure is further identified.

Chapter 3 reviews the research work on video server design. Emphasis is put on those systems using network striping, and the limitations of these systems are pointed out.

Chapter 4 summaries the goals that a software system framework for network striped video servers should achieve. Then the Cadmus system architecture and its features are presented. This chapter also shows the functionalities of the Cadmus system components and describes the meta-data management method. Other architecture related topics such as QoS guarantee and the redundancy scheme are also discussed, as well as the evolution of the Cadmus development.

Chapter 5 first gives the motivation and the requirements for VBR data smoothing in a video server environment, then a smoothing algorithm is presented. The algorithm is analysed both theoretically and empirically, and the results are used to choose striping block sizes for VBR videos. Finally, the findings are compared to related work on VBR smoothing.

Chapter 6 describes the distributed AC procedures in Cadmus. After resources and activities in storage nodes are classified, AC criteria for both RT and NRT operations are presented. These are integrated solutions because many resources and activities are considered simultaneously.

Chapter 7 presents a prototype implementation of the Cadmus architecture. The system components, especially the one residing in storage nodes, are described. Chapter 8 evaluates the performance of the implementation and analyses the bottleneck of the hardware configuration used for the experiments.

Chapter 9 summarises the main results in this dissertation and makes some suggestions for future work.

In addition, Appendix A gives proofs for the lemmas and theorems about the smoothing algorithm presented in Chapter 5. Appendix B summarises the contracts between the AC criteria and QoS enforcement identified in Chapter 6. Finally, Appendix C describes the interfaces used in the Cadmus prototype implementation.

Chapter 2

Background

This chapter introduces some background information related to video server design, and the justification for network striping is further developed.

2.1 Continuous Media

CM is a general term used to refer to digital audio and video, which are the primary data types in a video server. As suggested in [**Hyden94**], CM have both *temporal* and *informational* properties. While the temporal property demands that CM data be retrieved, delivered, and processed in a timely fashion for continuous presentation or to reduce the possibility of data loss, the informational property says that CM is often tolerant of the loss of some of their information content.

Both properties have effects on video server design. For the timely retrieval and delivery of CM data, a video server should provide QoS to individual CM streams. To prevent overload situations and their adverse impact on overall QoS, AC functionality should also be incorporated. However, when overload does occur in a video server, data of some CM streams can be discarded to maintain service stability.

In addition, CM data generally has high bandwidth and storage requirements. Table 2.1 shows some typical parameters of several digital video formats. The non-compressed entries clearly demonstrate the *voluminous* nature of CM data. However, because of spatial, spectral, and temporal redundancies in video data, some form of lossy or lossless compression techniques can be applied before the data is stored or used for transmission. The most common compression standards are MPEG [**Gall91**], JPEG [**Wallace91**], and CCITT Recommendation H.261 [**Liou91**]. The results of compressing the videos using MPEG are also listed in Table 2.1.

Data compression also affects video server design. First, the I/O and storage requirements of CM data are still high even after compression, especially when considering multiple videos or streams. For example, 100 different videos compressed from the CCIR-601 PAL format in Table 2.1 will occupy 352 GB if each is 2 hours long, and 100 simultaneous streams of such videos will need 400 Mbps. These demands, coupled with the performance limits of current disk devices, call for scalable design for video servers.

Second, most compression techniques, such as the ones used in MPEG and JPEG, produce VBR streams. For example, Figure 2.1 shows a section of the compressed MPEG-

Table 2.1: Typical Parameters of Several Digital Video Formats

					no compression		compressed[1]		
format[2]	standard	X	Y	fps	rate (Mbps)	2 hour (GB)	rate (Mbps)	2 hour (GB)	compress ratio
Medusa		88	64	25	2.15	1.89	1.00	0.88	2.20 : 1
		256	192	25	18.75	16.48	1.00	0.88	18.8 : 1
SIF	NTSC	352	240	30	29.00	25.49	1.15	1.01	25.2 : 1
	PAL	352	288	25	29.00	25.49	1.15	1.01	25.2 : 1
CCIR-601	NTSC	720	480	30	118.65	104.28	4.00	3.52	29.6 : 1
	PAL	720	576	25	118.65	104.28	4.00	3.52	29.6 : 1
HDTV		1280	720	30	316.41	278.09	20.00	17.58	15.8 : 1
		1920	1080	30	711.91	625.71	20.00	17.58	35.6 : 1

1 frame size trace of the MTV sequence from [**Trace95**] [**Rose95**]. The rate variability will complicate the procedures of system resource management or waste resources if peak rate reservation is used. It would be desirable to reduce this variability by applying some smoothing method before CM data is read from a video server. This is possible because video servers are used to play back stored videos rather than record live videos most of the time. Consequently, smoothing algorithms and the coordination of activities associated with smoothing should be taken into account by a VBR video server design.

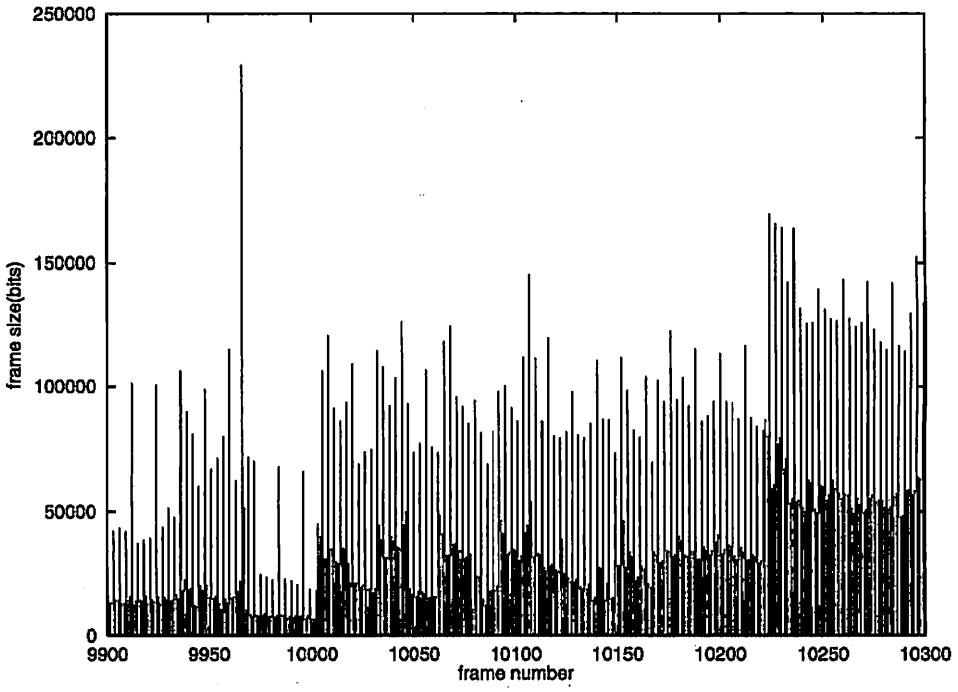

Figure 2.1: Frame Size Trace for MTV: Segment

[1]The compression method used for the Medusa and the SIF videos is MPEG-1 [**ISO/IEC93a**], while MPEG-2 [**ISO/IEC93b**] is used for the other two formats. Medusa videos are compressed using the MPEG-2 video encoder/decoder ported to Medusa by the author. The Medusa MPEG package can produce both MPEG-1 and MPEG-2 video sequences.

[2]The Medusa [**Wray94**] videos in the table use RGB555 encoding with 16 bits per pixel. The other three assume a chroma format of 4:2:0, where the two chrominance channels have half the sample grid density in both horizontal and vertical directions with respect to luminance. That results in an average 12 bits per pixel when the resolution for each sample is 8 bits. The HDTV examples are two of the several formats supported by the Grand Alliance System [**Hopkins94**] [**Petajan95**].

Finally, different compression schemes or standards use different video file formats. There are multiple compression standards, such as MPEG, JPEG, and H.261. There are also other compression techniques that will potentially be used in the future, such as wavelet [**Hilton94**] and fractal [**Barnsley88**] [**Fisher95**]. Besides, proprietary formats, such as QuickTime, AVI, and DVI, are widely used as well. If the data retrieval component of a video server attempted to understand all these formats, the design would be greatly complicated and the server would be a very inflexible and inefficient one. However, if video format analysis and data retrieval functionalities are separated, the resulting system will be more modular and extensible. This latter approach is taken by Cadmus.

2.2 Storage Devices

2.2.1 Disks

Characteristics

Disks are important video server components. A disk consists of a stack of platters coated with magnetic media, with the platters rotating on a common spindle at constant angular velocity past the read/write heads (one per surface). Data is organised into *cylinders* of different radii from the spindle axis. The intersection of a cylinder on a surface is called a *track*, which is divided into a number of *sectors*, each storing a fixed size of user data (typically 512 B).

Before reading or writing a sector, the disk heads have to *seek* to the specific cylinder, then wait until that sector is *rotated* and positioned under the heads. Contiguous sectors on the same track can be read or written continuously, with the transfer rate subject to the rotation speed of the spindle and the number of sectors per track. However, reading or writing logically consecutive sectors that are stored on different tracks or cylinders requires *head switches*. Normally these consecutive sectors are skewed so that the next sector is rotated just under the heads after the switch, making the switch time depend on the rotation speed and the number of sectors per track as well. Table 2.2 shows the general characteristics of an example drive ST12550N [**Seagate93**]. Some of the parameters are useful for estimating disk service time in order to do AC in a video server. The Cadmus prototype used a disk of this type.

Table 2.2: General Characteristics of the ST12550N Disk Drive

drive size	3.5 *in*	rotation speed	7200 rpm
interface	fast SCSI-2 [3]	full rotational latency	8.3 ms
formatted drive capacity	2139 MB [4]	full stroke seek (read)	17 ms
area density	152 Mbits/in^2	full stroke seek (write)	19 ms
on-drive cache	960 KB	internal data rate	3.56–5.96 MB/s
read/write heads	19	SCSI interface rate	5.0 MB/s
servo head	1	seek error rate	< 1 in 10^7 seeks
total cylinders	2707	recoverable error rate	< 1 in 10^{10} bits
total zones	24	MTBF	500,000 hours

[3] Can also be operated according to SCSI-1 protocol.

[4] Formatted as follows: 512 data bytes per sector; 9 spare sectors per cylinder; and 1 cylinder at the inner track reserved for spares.

As tracks near the outside of each surface have greater circumferences than those near the spindle, zoned bit recording (ZBR) is used in most modern disks. This approach groups sets of adjacent cylinders into *zones*, with the number of sectors per track being constant within each zone, but successively larger in the outer zones than that in the inner. Table 2.3 shows the zone information for the ST12550N. Cyl stands for cylinder and the unit of skew is a sector. Because different zones may have large transfer rate differences, zone information should be used by a video server in AC to better utilise the disk resource.

Table 2.3: ST12550N Zone Information

zone	start cyl	end cyl	track sectors	track skew	cyl skew	zone	start cyl	end cyl	track sectors	track skew	cyl skew
1	0	125	97	9	19	13	1654	1725	78	7	16
2	126	341	97	9	19	14	1726	1801	76	7	15
3	342	589	95	9	19	15	1802	1889	74	7	15
4	590	742	93	8	19	16	1890	1975	72	7	15
5	743	902	91	8	18	17	1976	2069	71	7	14
6	903	1034	90	8	18	18	2070	2182	69	6	14
7	1035	1163	88	8	18	19	2183	2295	67	6	14
8	1164	1283	86	8	17	20	2296	2389	65	6	13
9	1284	1395	84	8	17	21	2390	2495	64	6	13
10	1396	1486	83	8	17	22	2496	2587	62	6	13
11	1487	1572	81	7	16	23	2588	2679	60	6	12
12	1573	1653	79	7	16	24	2680	2706	58	5	12

Anomalies

The mechanical nature of disks will lead to unpredictable down-time such as: *thermal calibration (T-cal)*, *seek error recovery*, *read error recovery*, *write error recovery*, and *defect management*. During these periods, a disk will not do any useful work for user requests. The following paragraphs will briefly discuss these anomalies using the ST12550N as an example.

In the ST12550N, T-cal happens approximately once every 10 minutes, and a full T-cal will compensate 19 heads with each head taking about 56.6 ms, resulting in a full T-cal of 1.0754 seconds. Automatic T-cal also occurs at other times such as during read error recovery, and during reassign block functions when a read or write detects defective disk blocks and automatic reallocation is enabled. However, in most future disks, T-cal will be eliminated using embedded servo technology.

A seek error is defined as a failure of the drive to position the heads to the addressed track. After detecting an initial seek error, the drive automatically re-seeks to the addressed track up to 3 times. Write errors can occur as a result of media defects, environmental interference, or equipment malfunctions. The drive will also attempt its recovery algorithm up to 3 times after a write error. If a defect block is detected and write reallocation is enabled, then the reassign block function is performed to automatically relocate the bad block to a spare one.

The default retry count for the application of the read recovery algorithm during a read error is 27. Even if read retry is disabled and early recovery is enabled to allow the disk drive to apply ECC (Error Correction Code) correction as soon as possible, there are still 3 hidden retries. If a defect block is detected and read reallocation is enabled, the disk will also perform the reassign block function to relocate the bad block.

The defect management is three-fold: slip sparing at format time; 9 spare sectors per cylinder; and 1 cylinder at the inner track reserved for spares. There is no perfect disk surface. For example, the author tested 3 new ST12550N disks formatted with 4178873 sectors of size 512 B, and found 131, 136, and 466 bad sectors which are skipped on each disk. If at run-time, more defect sectors are formed and read/write reallocation is enabled, then a range of logically contiguous sectors may later be allocated a sector far away in the innermost cylinder. This will incur extra seek and rotational latency which is out of user control.

There are two characteristics of the above disk anomalies. First, most of them are unpredictable and it is very difficult to track them or to keep a record to anticipate when the anomalies will happen. Second, they do occur in a real system and thus cannot be ignored. Cadmus takes a defensive approach rather than a preventive one. It does not reserve resources for the anomalies in order to smooth them out, but does provide mechanisms to maintain system stability during overload situations, whose cause could be the disk anomalies.

Technology Trends

Traditionally, the area density (measured in megabits per square inch (Mbits/in^2)) of disk drives increased at a rate of roughly 27% per year [**Chen94b**]. Recently, new technology, such as MR (Magneto-Resistive) heads, PRML (Partial Response Maximum Likelihood) signal detection, and no-ID sector formatting, has lifted the figure to 60% per year with today's density in the 600–700 Mbits/in^2 range [**Seagate96a**]. By the year 2000, the area density is expected to reach 10 Gbits/in^2. In addition, embedded servo technology, which distributes operational information evenly in relation to user data on each platter, eliminates the need for T-cal.

The spinning speed of the disk spindle has increased from 3600 rpm to 5400 rpm, 7200 rpm, and 10,000 rpm in the last decade. Disk interfaces have been evolved from SCSI-1 (4 MB/s) to SCSI-2 (10–40 MB/s) and Fibre Channel (FC) (100 MB/s). The most advanced commodity disk example today can be found in the Cheetah family from Seagate [**Seagate96b**]. The Cheetah disks spin at 10,000 rpm; use MR heads, PRML signalling, and embedded servo; and provide FC interfaces. Such a 3.5 *in* disk stores 9 GB, and a 5.2 *in* one 23 GB. It should be noted that this new technology does not eliminate the disk anomalies mentioned above except T-cal.

2.2.2 Disk Arrays

Concepts

A disk array [**Lawlor81**] organises multiple, independent disks into a large, high-performance logical disk. The motivation for disk arrays is the performance gap between CPUs and secondary storage, as well as the need for high performance secondary storage systems. Disk arrays stripe data across multiple disks and access them in parallel to achieve both high data transfer rates on large data access and high I/O rates on small data access. Data striping also results in uniform load balancing across all of the disks, eliminating hot spots that otherwise saturate a small number of disks while the majority of disks sit idle. Redundancy information is usually provided to tolerate disk failures because of the decreased reliability of disk arrays. Such a disk array is normally called a RAID (Redundant Array of Inexpensive Disks) [**Patterson88**].

Data striping is accomplished by distributing consecutive logical data units (called *striping units*) among the disks in a certain order such as round-robin. *Fine-grained striping* has small striping unit size (SUS) and distributes data so that all of the disks cooperate in servicing every request, while *coarse-grained striping* has large SUS and allows the disks to cooperate on large requests and to service small requests independently. SUS governs the trade-off between transfer parallelism and access concurrency.

Redundancy information is normally in the form of replication or parity. Redundant data is either concentrated on a small number of disks or distributed uniformly across all of the disks. Different data and redundancy distribution leads to different RAID levels [**Patterson88**] [**Chen94b**].

RAID Levels

RAID level 0 (non-redundant) does not employ any redundancy at all, while RAID level 1 (mirrored) uses replication for redundancy. Both levels use coarse-grained striping. RAID level 2 (memory-style ECC) divides the array into data disks and check disks. User data is bit or byte striped over the data disks, and the check disks hold a Hamming error correcting code computed over the data in the corresponding bits or bytes on the data disks.

In RAID level 3 (bit-interleaved parity), user data is bit or byte striped across the data disks, and a simple parity code is used to protect against data loss. A single check disk (called the *parity disk*) stores the parity. RAID level 4 (block-interleaved parity) is identical to level 3 except that striping units are relatively coarse grained. RAID level 5 (block-interleaved distributed-parity) uses coarse-grained striping but distributes parity blocks on all the data disks. There is no fixed parity layout scheme for RAID5.

A RAID level specifies not a specific implementation of a disk array but rather its configuration and use [**Chen94b**]. For example, RAID levels 1 and 3 can be viewed as subclasses of level 5. Since RAID levels 2 and 4 are practically inferior to level 5, the problem of selecting among RAID levels 1 through 5 is a subset of the more general problem of choosing an appropriate *parity group* size and SUS for RAID level 5 disk arrays. A parity group size close to two may indicate the use of RAID level 1 disk arrays; a striping unit much smaller than the size of an average request may indicate the use of a level 3 array.

2.3 Video Server Structure

2.3.1 The Storage Hierarchy

The storage hierarchy of a video server could consist of tapes, disks, and DRAMs. Although tapes are the cheapest per MB and have high capacity per unit, they also have the slowest transfer rates and the longest average seek time. These characteristics make them unsuitable to serve multiple concurrent CM streams in a video server, but they are good candidates for backup repositories.

DRAMs have the highest bandwidth among the three, but they are also the most expensive per MB. [**Hennessy90**] identifies the two orders of magnitude gap in costs between DRAMs and disks, which still holds today. At the time of writing this dissertation[5], the

[5] January 1997

prices per MB for DRAMs[6], disks[7], and tapes[8] are around \$4, \$0.18, and \$0.045. Storing 352 GB CM data will thus cost \$1.44 million, \$0.065 million, and \$0.016 million respectively. The reduced cost of magnetic disks, along with their reasonable random access time and transfer rates, makes them the desirable choices to store and serve CM data in a video server.

Another aspect to consider is unit capacity. Although the density of DRAMs increases 60% per year, which is faster than the average 27% density increase per year for disks in the past, current DRAM chips are of 64 Mbits. 1 Gbits DRAMs will not be available until 2002, while 4 Gbits ones are likely in 10 to 12 years time [**SMART96**]. In contrast, current 3.5 *in* disks have a capacity of 9 GB, and their area density now increases at 60% per year. This unit capacity gap and the persistence characteristic of data stored on disks, further advocate that in the near future, magnetic disks will play an important role in video servers for storing and streaming large amounts of data.

On the other hand, there are performance gaps between DRAMs and disks. The bandwidth of 32-bit 60 ns DRAMs is 67 MB/s, increasing at 7% per year; while the transfer rate of a disk today could reach 11.3–16.8 MB/s, increasing at 22% in the past and about 60% [**Grochowski96**] now. Hence, DRAM bandwidth and disk transfer rates will be comparable in a few years. However, the average random access time of disks is at around 10 ms, which is of five orders of magnitude slower than that of DRAMs. For performance reasons, it would be attractive to store the most frequently accessed videos in DRAMs. There are trade-offs between costs and performance in arranging storage hierarchies for video servers, but these topics are outside the scope of this dissertation. There is research comparing different storage hierarchies and their costs, such as [**Chervenak95**] [**Stoller95**], and all of them conclude that most video servers will store videos on disks in the foreseeable future. Disk based video servers are assumed by this dissertation.

2.3.2 Basic Structure

Figure 2.2 shows the basic structure of a disk based video server system (backup storage and CPUs are omitted for simplicity). Disks and DRAMs storing CM data are linked by some interconnect and are regulated by some management system, which is contacted by clients for playing back or recording CM streams. CM data is transferred between clients and the video server through networks, which are often switched ones in large scale systems because of their scalability.

The most important aspect of the hardware part of a video server to support a large number of streams is the interconnect structure. Generally, there are four types of interconnect: *backplanes*, *channels* [**Katz92**], *parallel computer interconnection* [**Ibbett89**], and *networks*. Most current high-end video servers use a centralised server architecture exploiting the combination of the first three interconnect methods, and networks are only used to connect servers to clients. Several problems exist for this kind of structure. First, if disks are attached to a single machine, then that machine's memory and I/O subsystem are likely to become a performance bottleneck. Second, backplanes and non-switched channel style interconnect do not scale well in both bandwidth and distance [**Boxer95**]. Third, as video server applications are I/O intensive but not computationally intensive, exploiting massively parallel multiprocessor architecture is not a cost-effective approach.

[6] 16 MB 60 ns EDO 72-pin non-parity
[7] 4.3 GB 7200 rpm 3.5 *in* SCSI
[8] 8.0 GB internal SCSI

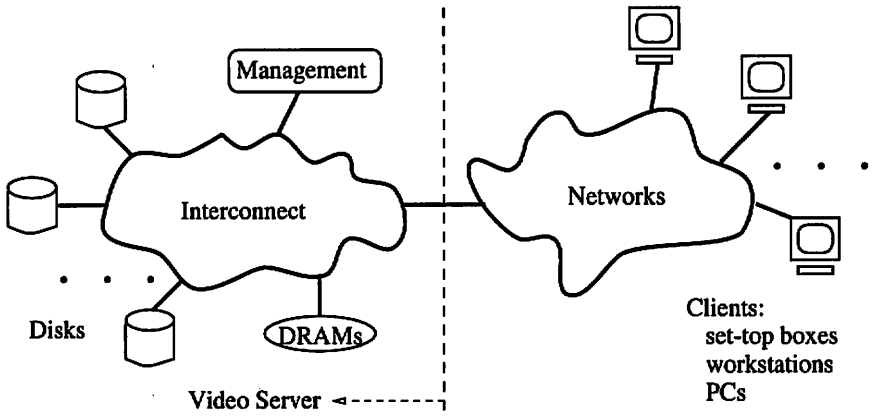

Figure 2.2: Basic Video Server System Structure

It is likely that the main interconnect in future video servers will be based on channels and networks. Two prominent candidates are FC [**Bryan94**] [**Sachs96**] and ATM [**Prycker93**]. FC is a high-speed computer or storage devices to computer interconnect over a bit-serial connection. It is a hardware intensive, switched, looped, or point-to-point technology purposefully blurring the distinctions between channels and networks. As surveyed in [**Sachs94**], LAN and I/O architecture are converging, especially in the datalink and physical layers. And their traditional difference in the communication length and the master-slave vs. peer-to-peer computational models is disappearing. Also FC can support high level protocols such as AAL5 and SCSI. For storage devices such as magnetic disks, an FC Arbitrated Loop (FC-AL) [**ANSI94**] architecture has been proposed as an economic solution for disk arrays to avoid the more costly switching fabric.

2.3.3 Network Striping

Recent research has made gigabit switches a reality [**Kung92**] [**McKeown96**], while the cost/performance ratio of microprocessors is dropping steadily. These make it attractive to use off-the-shelf components inside a video server with scalable switched networks as the primary interconnect. Storage peripherals can be attached directly to a switched network such as ATM or switched FC, and CM data can be striped to multiple peripherals, each of which could be a single disk or a cost-effective small scale RAID using a backplane or a non-switched channel such as SCSI or FC-AL internally. The rationales of this network striped video server architecture are *cost-effectiveness*, *flexibility*, *scalability*, and *high-performance*.

The server is cost-effective because it can be constructed from off-the-shelf commercial components and switched network interconnect. Flexibility comes from the fact that networks are generic and any storage devices or processing units can be attached and detached without worrying about modification of the interconnect. Because peripherals and networks can be independently upgraded, and each peripheral only needs to manage its own storage objects, the server structure is highly modular, extensible, and scalable. High performance comes from striping and the network attached approach. Striping allows the usage of the aggregate resources of multiple storage nodes to serve multiple clients, while the network attached approach enables data to be moved between clients and storage devices directly and eliminates the potential data path bottleneck present in single node video servers.

Though network striped video servers rely on networks as the interconnect, this work does not dictate an actual network technology to be used. However, as the working environment has ATM installed, further discussion of the structure will assume an ATM network.

Because ATM and FC are converging in certain aspects, the results of this research can equally be applied to FC based interconnect. The current industrial trend is that ATM will be the future network technology, while FC-AL will be used in future disk arrays. There has not been much interest in SCSI/IPI over ATM or SCSI over switched FC.

However, network striped video servers have the disadvantage of reduced reliability because multiple components can fail independently. There is a point beyond which it would be more desirable to replicate videos among different and unrelated network striped servers. Such hierarchical design is studied elsewhere [**Lougher96**] [**Pegler97**], while this work concentrates only on network striped video servers. Nevertheless, this research does consider reliability issues and provides a redundancy scheme to achieve fault tolerance.

2.4 Distributed Computing

A network striped video server is a typical distributed system, which is defined as several computers doing something together [**Schroeder93**]. The canonical example of a general purpose distributed system is a networked system — a set of workstations/PCs and servers interconnected with a network. Networked systems are gaining popularity because of their advantages in the areas of sharing, cost, growth, and autonomy over traditional centralised systems. However, centralised systems are often easier to use because they are more accessible, coherent, and manageable. To reduce this usage complexity for networked systems, a *distributed environment* is introduced to give users and applications transparent access to data, computation, and other resources across collections of multi-vendor, heterogeneous systems.

An example distributed environment is Open Software Foundation (OSF)'s Distributed Computing Environment (DCE) [**OSF91**]. DCE is based on the client/server model and the remote procedure call (RPC) [**Birrell84**] paradigm. It consists of an integrated set of tools and operating system and network independent services that support the development, use, and maintenance of distributed applications. The services are divided into the *fundamental distributed services* and the *data sharing services.* The fundamental distributed services include threads, RPCs, a directory service, a time service, and a security service, while the data sharing services build on top of the fundamental services and include a distributed file system and diskless support. The DCE distributed file system is based on the Andrew File System (AFS) [**Howard88**].

Another distributed environment is the Object Management Group (OMG)'s Object Management Architecture (OMA) [**OMA90**], which is based on the object model rather than the procedural call paradigm. OMA attempts to define, at a high level of abstraction, the various facilities necessary for distributed object-oriented computing. It consists of four components: the *object request broker (ORB)*, *object services*, *common facilities*, and *application objects.* The core of OMA is the ORB, a mechanism by which objects transparently make requests and receive responses. The ORB provides interoperability between applications on different machines in heterogeneous distributed environment and interconnects multiple object systems. OMG's Common Object Request Broker Architecture (CORBA) specification [**CORBA91**] [**CORBA94**] concretely describes the interfaces and services that must be provided by compliant ORBs. CORBA is composed of five major components: the *ORB core*, the *interface definition language (IDL)*, the *dynamic invocation interface*, the *interface repository*, and *object adapters.* CORBA based distributed object computing is the paradigm used for component interaction in the Cadmus prototype implementation.

2.5 File Systems

2.5.1 File Implementation

A file system determines how files are structured, named, accessed, used, protected, and implemented [**Tanenbaum92**]. In a dedicated video server environment, a proper implementation that has good performance and can effectively support QoS enforcement mechanisms is more important than other issues. Implementation of files on physical devices is generally based on *sequential*, *indexed*, or *log-structured* file organisation. A typical file system always uses the same file implementation for consistency reasons.

A sequential file is stored as a contiguous block of data on a disk. This has the advantages of simple implementation and good performance, but the disadvantages of fragmentation and wasted space, which are more relevant for large CM files. A variation of this organisation is linked sequential files where fixed-size disk blocks of the same file are linked together by pointers embedded in each block and no space is lost to disk fragmentation. However, random access is very slow in this case.

An index file has a separately stored disk position index table apart from the file data itself. According to how file data is further organised, indexed implementation can be categorised as either *block-based* or *extent-based*. Block-based file systems store file data in fixed-size blocks (typically 512 B–8 KB) on disks, and each block has an entry in an index table. Data is read from or written to disk in block units by the file system, thus incurring a seek and rotational overhead for each block access. Also if a file is large, its index table could be very large as well. UNIX [**Bach86**] uses such a block-based file system, but it also provides a raw disk interface which bypasses the file system and buffer cache and can be used to read or write large amounts of data.

Extent-based file systems do not force block sizes to be fixed. A file is composed of a number of chunks, or *extents*, each of which is stored contiguously on a disk. The file system records the starts and lengths of the extents into an index table, also called an *extent list*, for each file. When extent sizes are fixed for all files, an extent-based file system reduces to a block-based one. When a file has only one extent, it becomes a sequential file. Hence, extent-based organisation can achieve the effects of both sequential and block-based systems, yet it is far more flexible than the other two, which makes it the desirable choice for CM applications. For example, large extents can be allocated to CM files for better performance, while the fragmentation problem is not severe because of variable extent sizes.

The fundamental idea of a log-structured file system [**Ousterhout89**] [**Rosenblum92**] is to improve write performance by buffering a sequence of file system changes in file caches and then writing all the changes to disks sequentially in a log-like structure using a single disk write operation. The file-system's only representation on disks is in the form of the append-only log. However, the motivation for log-structured file systems does not hold in a video server environment. This is because most activities in a video server are reads, and recording will normally generate large writes. Furthermore, caching is not very useful because of the lack of temporal locality for CM data. In addition, although log-structured file systems have very good write performance, their read performance is at the same level as that of the UNIX style block-based file systems.

2.5.2 Distributed File Systems

A distributed file system is used to provide persistent storage and data sharing in a distributed system. Some important issues in the design of a distributed file system are naming

structure, programming interface, file implementation, system integrity, existence control, concurrency control, security, file location, availability, and scalability [**Satyanarayanan93**]. A distributed file system typically has two reasonably distinct components [**Bacon93**]: a *file service* and a *directory service*. The former specifies the file system interface for operations on files, while the latter is concerned with directory management. These services are normally implemented by a *file server*, typically a process running on some networked machine. A more flexible approach described in [**Lo94**] separates the file service into two layers, resulting in a modular and extensible architecture that can support multiple file abstractions.

Many of the design issues and their solutions for a distributed file system can be applied to a network striped video server as well because of the distributed nature of the latter. However, most contemporary distributed file systems do not support file implementation on multiple storage nodes across networks, let alone providing QoS for CM applications. In addition, most of them are optimised according to file usage patterns on traditional UNIX systems where most files are small (less than 10 KB) and have short lifetime [**Satyanarayanan81**], neither of which is true in a video server environment. To better utilise system resources, a flexible architecture which is optimised for network striped video servers and can support QoS for CM data retrieval and delivery would be desirable.

2.6 Real-Time Systems

2.6.1 Concepts

RT systems are defined as those systems in which the correctness of a system depends not only on the logical results of computation, but also on the time at which the results are produced [**Ramamritham94**]. An RT system is *hard*, if the consequences of a timing failure can be catastrophic; otherwise, it is classified as a *soft* RT system. Video servers are typical soft RT systems because they have to satisfy the temporal requirement of CM data, but occasional delay or data loss is generally tolerable.

Activities in an RT system have timing constraints and need computational, communication, data, and I/O resources to proceed. Some activities may also depend on others. Therefore, *scheduling algorithms* are used to determine, for a given set of activities, whether a schedule (the sequence and the time periods) for executing the activities exists such that the timing, precedence, and resource constraints of the activities are satisfied, and to calculate such a schedule if one exists.

RT operating systems (RTOSs) are integrated parts of RT systems. They stress predictability and include features to support RT constraints. They should perform integrated CPU scheduling and resource allocation so that collections of activities can obtain the resources they need, at the right time, in order to meet timing constraints. Scheduling and operating system (OS) support are two important aspects in video server design in order to do AC and guarantee QoS.

2.6.2 Real-Time Scheduling

Static Scheduling

RT systems can be classified as either *static* or *dynamic*, as can RT scheduling. A static RT system has three characteristics: tasks to be scheduled are known a priori; schedulability

is analysed off-line; and no new tasks can be invoked at run-time. Scheduling in such an environment can be static; and a schedule can be computed off-line and stored in a table, or it can be constructed at run-time according to some pre-specified heuristics.

One such static scheduling approach is the cyclic executive model [**Baker88**]. A cyclic executive is a control structure or program for explicitly interleaving the execution of several periodic processes on a single CPU. A pre-computed *cyclic schedule* specifies an interleaving of activities that will enable processes to execute within their periods and deadlines. Such a schedule is called a *major schedule*, whose duration is called a *major cycle*. The major schedule is further divided into *minor schedules* of equal duration, i.e., the *minor cycle*, also known as a *frame*. The disadvantage of the cyclic executive is that static analysis must be applied to find the cycles and the schedules. The advantage of the approach is that it is efficient, predictable, and simple to implement.

Dynamic Scheduling

A non-static RT system is called dynamic and is characterised by dynamic scheduling which computes schedules directly or indirectly at run-time according to on-line information. Dynamic scheduling almost always relies on some *heuristics* to select which task to run first. Such heuristics may be in the forms of priorities, periods, deadlines, laxities, or functions which synthesise various characteristics and requirements of the tasks to be scheduled.

Priorities can be either pre-assigned or derived from other task characteristics such as periods or deadlines. Highest priority first scheduling suffers from the unbounded priority inversion problem [**Cornhill87**] when there is contention on multiple resources. In such a situation, a higher priority task can be blocked unpredictably long by a lower priority task. Priority inheritance protocol (PIP) [**Sha90**], priority ceiling protocol (PCP) [**Sha90**], and stack resource policy (SRP) [**Baker90**] have been proposed to bound the duration of priority inversion, although priority inversion itself cannot be eliminated.

Rate monotonic (RM) scheduling is a static priority scheduling method and assigns priorities to tasks based on their periods, with higher priorities to tasks with shorter periods. It is shown that RM scheduling is optimal among static priority schemes for periodic and independent tasks with constant computation time [**Liu73**]. Earliest deadline first (EDF) is a dynamic priority assignment algorithm: the closer a task's deadline, the higher its priority. EDF is shown to be globally optimal under the same scheduling environment. Processor utilisation bounds exist for both schemes [**Liu73**] [**Lehoczky89**], and full processor utilisation can be achieved using EDF. In addition to deadlines, a task's laxity (given by the amount of time one can wait and still meet its deadline) can be used to deduce its dynamic priority as well.

However, [**Locke86**] shows that, during overload situations, deadline or laxity based scheduling has counter-productive effects. A time-value function is used in [**Locke86**] to characterise a task, with an importance value assigned to each point of time when the task is finished, and a Best Effort (BE) scheduling algorithm is proposed to maximise the sum of values of all tasks during overload. The ideas have evolved into the Benefit Accrual Model (BAM) in [**Jensen94**], where a benefit function is used to generalise the deadline of an RT computation. Similar BE based approaches are also used to establish formal performance bounds for on-line algorithms in [**Baruah91**] [**Koren92**], where a task's nature is unknown when it is issued; and the focus is the *competitive factor*, which measures the value an algorithm guarantees it will achieve compared to a clairvoyant scheduler.

RT scheduling with precedence and resource constraints are generally NP-complete problems [**Cheng88**] [**Audsley90**]. [**Zhao87b**] and [**Zhao87a**] propose suboptimal algorithms

that employ computationally simple heuristics based on resource and timing requirements of the tasks being scheduled. [**Spuri94**] extends the results from PCP and SRP to accommodate precedence constraints and shared resources. For BE scheduling, [**Clark90**] develops an algorithm to deal with dependency (precedence and resource conflicts) which is not known in advance.

2.6.3 Real-Time Operating Systems

Four main functional areas that an RTOS supports are: process management and synchronisation, memory management, interprocess communication, and I/O. Three general categories of RTOSs exist: small and proprietary kernels, RT extensions to commercial time-sharing OSs such as UNIX, and research RT kernels.

The small and proprietary kernels are often used for embedded systems when very fast and highly predictable execution must be guaranteed. They are normally based on the micro-kernel architecture where a minimal kernel and a set of system server processes cooperate to provide OS functionalities. These kernels usually have fast context switch time, short interrupt delay, and simple primitives. They also normally maintain an RT clock, use single-address space memory management, and provide simple RT scheduling according to priorities or deadlines. These features can be used as a basis upon which simple RT systems can be built. Video servers belong to such RT systems because the types of the activities and their resource requirements are known in advance, although the timing constraints of the tasks cannot be anticipated. Cadmus uses a modified micro-kernel to provide QoS guarantees in storage nodes, which will be further described in later chapters.

Recently, there have been efforts to build even smaller kernels and to put many of the OS functionalities into applications through shared libraries instead of system server processes. Nemesis [**Leslie96**] is such a vertically structured kernel built to support multimedia applications in a workstation environment. Nemesis schedules periodic processes using the EDF method [**Roscoe95**], and it also supports NRT activities. There is no concept of priority in Nemesis, which eliminates the problem of priority inversion. More recent work presents solutions to network communication [**Black95**] and multimedia I/O [**Barham96**] under the Nemesis environment. Other similar kernels are SPIN [**Bershad94**] and Exokernel [**Engler94**] [**Engler95**], which are not driven by multimedia applications.

The approach to extend commercial OSs such as UNIX to RTOSs has limited applicability because the extensions are generally slower and less predictable than the proprietary kernels. It is questionable whether this is the correct approach, because many basic and inappropriate underlying assumptions in time-sharing systems still exist in the extensions, such as optimising for the average case, assigning resources on demand, and independent CPU scheduling and resource allocation. All these will bring unpredictability, which is at odds with RT systems.

While micro-kernels are effective for simple RT applications, they generally provide no direct support for solving more complex or larger problems. For example, it is very difficult to craft a solution based on priority-driven scheduling where all timing, computation time, resource, precedence, and value requirements are mapped to a single priority for each task. It would be desirable for an RTOS to understand the timing and resource requirements of the activities and to provide predictability in both the kernel and the application levels, instead of treating each task as a random process and allocating resources independently. Some research kernels, such as ARTS [**Tokuda89**], MARS [**Kopetz89**], CHAOS [**Gopinath89**], Spring [**Stankovic91**], and Alpha [**Clark92**], are aiming at these directions.

2.7 Summary

This chapter has examined some background concepts that have effects on video server design. Particularly, the characteristics of CM data and storage devices demand scalable and cost-effective video server structure, where disk based network striping is a good candidate. In addition, these characteristics also require the following support from a video server: QoS, AC, overload processing, and VBR smoothing. On the other hand, video server systems are closely related to distributed systems, file systems, and RT systems. Some basic concepts of the latter three have been described and their influences on video server design were identified.

Chapter 3

Related Work

This chapter surveys the research work on various aspects of video server design. An emphasis is put on scalable storage systems using network striping, particularly those designed for CM applications.

3.1 Disk Scheduling

A common approach for disk scheduling in a video server is to retrieve data on a *cycle* basis, with the amount of data read or written for each stream in a cycle being proportional to the stream's playback rate. In [**Rangan92**], this is known as the *quality proportional multi-subscriber servicing algorithm*, in which data retrieved for each subscriber in a cycle may not be stored contiguously on a disk, but the cycle length may change dynamically. In [**Lougher92**], this method is known as *data block normalisation* or *stream re-mapping*, where data to be read for one stream in a cycle is stored as contiguous disk blocks, but the cycle length is fixed.

In each cycle, the retrieval order for the streams can use either the *round-robin* method or the *SCAN* algorithm, in which disk heads scan the disk platter from end to end back and forth while servicing the requests along the way. Their main difference lies in their effects on the latency between retrieving the same stream, and the latency has implications on playback initiation delay and buffer requirements. [**Yu92**] combines these two methods by partitioning each cycle into groups and assigning each stream to one group. The groups are serviced in a fixed order in each cycle, but SCAN scheduling is used inside each group. This *grouped sweeping scheme (GSS)* can easily reduce to either the round-robin or the SCAN algorithm by adjusting the group size and the group number.

[**Barham96**] proposes an *RSCAN* algorithm to provide per-client rate guarantees at the possible expense of disk utilisation. A disk I/O scheduler maintains a list of pending transactions for all streams and services the transactions using SCAN. However, the scheduler accounts for the actual cost of each transaction in arrears and uses a credit scheme based on leaky-buckets to rate control each stream. NRT disk requests are also supported by estimating the slack-time in the system. The scheduler may split a long contiguous transfer at a disk block boundary in order to meet the QoS guarantee of another client.

Deadline based disk scheduling is normally used in RT database systems (RTDBSs). [**Abbott90**] studies the *earliest deadline SCAN (ED-SCAN)* method where disk heads seek

in the direction of the read request with the earliest deadline while servicing all the requests along the way. It also examines the *feasible deadline SCAN (FD-SCAN)* algorithm where only read requests with feasible deadlines are chosen as targets that determine scanning direction. A deadline is feasible if it can be met by estimation. [**Haritsa91**] proposes an *adaptive earliest deadline (AED)* method which uses a feedback control mechanism which detects overload conditions and modifies transaction priority assignment accordingly. The rationale is to schedule the largest set of transactions that can be completed by their deadlines in all situations, which is similar to the one of BE scheduling [**Locke86**].

For multimedia I/O, [**Reddy94**] proposes a *SCAN-EDF* algorithm where requests are normally served in EDF order, but SCAN is used for requests with the same deadline. The efficiency of SCAN-EDF depends on how often seek optimisation can be applied. Therefore, deadline engineering methods, e.g., batching or delaying deadlines, are proposed to improve efficiency. When requests from all streams are batched, SCAN-EDF in fact becomes the cycle based SCAN approach.

Another deadline based disk scheduling method is the *RT-WINDOW* algorithm presented in [**Chen96**], which has two adjustable parameters: a window size and a threshold. Requests are sorted in ascending deadline order and the first several requests constitute a window. The first request is served immediately if its deadline is less than the threshold. Otherwise, disk heads are moved toward the first request while servicing those requests which are within the window and are also along the way. By adjusting the parameters, RT-WINDOW can be degenerated to either EDF or SCAN scheduling.

3.2 Data Placement

When multiple streams are served from a video server, disk seek and rotational overhead can be classified into *inter-stream* and *intra-stream seeks*. Assume at any time an access request to a disk bears the data of only one stream, then inter-stream seeks are those incurred between different requests, while intra-stream ones are those needed to serve one request when its data is not stored contiguously. Because seek and rotational latency is significant compared to the time spent on disk data transfer, data placement strategies are used to reduce its occurrence or shorten its duration.

A stream's data can be stored contiguously or scattered on a storage device. *Contiguous placement* has no intra-stream seeks, although inter-stream seeks may still persist. However, sequential file implementation, as described in Section 2.5.1, is subject to fragmentation problems and may necessitate enormous copying overhead during insertions and deletions to maintain contiguity. *Scattered placement* is more flexible but may incur intra-stream seeks. Nonetheless, intra-stream seeks can be eliminated by storing the data of a request contiguously on a disk, although data of different requests for a stream can be scattered. This strategy is used in [**Lougher92**] for constant bit rate (CBR) videos.

Instead of avoiding intra-stream seeks, another approach is to reduce them to a reasonable bound. This rationale is behind *constrained placement* [**Rangan91**] [**Anderson92**], where the separation between successive blocks of a file is bounded. The bound on separation is generally not enforced for each pair of successive blocks of a file but only on average over a finite sequence of blocks. Methods for merging multiple CM files on disks and still preserving the placement constraints of individual files are studied in [**Yu89**] [**Rangan93**]. Constraint placement is useful when block sizes must be small, but it needs elaborate algorithms in the implementation to guarantee the constraints. Furthermore, its advantage may

disappear in the cycle based SCAN approach, where intra-stream and inter-stream seeks can be mixed and optimised together.

To reduce inter-stream seeks, [**Ghandeharizadeh95**] proposes a *region based block allocation mechanism.* This approach partitions a disk into a number of regions and stores successive blocks of a file in successive regions. In each cycle, a server only retrieves data for all active streams from a single region using SCAN, thus limiting the inter-stream seek overhead. However, this method needs to synchronise all streams to the same region, and therefore the latency observed by a stream's first request may be large.

[**Ozden96b**] proposes a *phase-constrained allocation scheme* in order to eliminate both inter-stream and intra-stream seeks. A phase refers to the video transmission starting at a given time. This scheme concatenates all videos into a single super-video and supports maximum concurrent phases allowed by the disk transfer rate. It segments the super-video into rows and columns and stores data contiguously on a disk using the column major order. In each cycle, a column of data corresponding to the maximum number of phases are read sequentially, eliminating both seek and rotational latency. However, the latency for a new phase to start may be even longer, and the resulting system is highly inflexible. For example, even in pure video-on-demand (VOD) applications, it may be necessary for videos of variable lengths to be added or deleted dynamically, thus breaking the column major order.

To maintain temporal relationships among objects such as video, audio, text, and image, [**Chen93**] *interleaves* related media objects within a block stored contiguously on a disk, thus removing the inter-stream seeks incurred when these objects were stored separately. Similarly, by observing that files recorded at the same time are likely to be replayed at the same time, [**Lougher93**] uses *log-structured placement* to group together related files on disks. Then when these related streams are played back together, their inter-stream seeks can be minimised or even eliminated. However, the advantages of both schemes disappear for independent streams.

Most of the above work assumes CBR streams and disks with constant transfer rates. For VBR videos, [**Chang96**] compares three placement and retrieval techniques: *constant time length (CTL)* places and retrieves data in blocks corresponding to equal playback durations; *constant data length (CDL)* places and retrieves constant size data blocks; and a *hybrid* solution uses CDL placement but retrieves a variable number of blocks in each cycle. The hybrid approach is shown to have moderate buffer requirements and low fragmentation. For ZBR disks, [**Tewari96b**] considers the popularity and access rates of data blocks and proposes an optimal *skewed organ pipe* data placement strategy. However, the approach is questionable because the placement optimisation criterion is to minimise the mean response time for block access, instead of guaranteeing QoS for individual streams.

3.3 Quality of Service Enforcement

While a video server supporting CBR streams can retrieve and send a fixed amount of data periodically for each client, VBR videos are more difficult to handle in the sense that their rates vary, thus imposing variable load to both the server and networks. On the other hand, there is server *service time variability* as well. This is because given a fixed amount of data, the time needed to read it from a disk would vary depending on the disk scheduling method used and the place the data is stored. The QoS enforcement scheme in a VBR video server should take into account of both aspects of variability.

One advantage for stored VBR videos is that their characteristics can be analysed off-line. For video servers, *statistical* and *deterministic* services have been proposed. Statistical schemes will utilise the disk service time characterisation [**Vin94b**] [**Vin94a**] and the bit rate statistics of stored videos [**Chang94b**] [**Chang96**] to perform statistical analysis during AC. However, although statistical strategies are applicable to various situations, for commercial applications such as VOD, it would be better to provide deterministic services to clients.

Deterministic schemes for VBR videos are either *cycle based* or *deadline based.* Cycle scenarios [**Dey94**] [**Chang94a**] [**Asai95**] [**Neufeld96a**] extend the cycle model for CBR videos in [**Rangan91**] [**Lougher92**] [**Vin93**] and perform AC for the cycle lifetime of a new stream based on the amount of data needed in each cycle, which is known a priori. Deadline based methods [**Lau95**] recompute a schedule for each disk at AC time based on deadlines of the requests that would be issued in the lifetime of a new stream, while the deadlines are obtained by analysing the corresponding stored VBR video in advance. In deadline based approaches, each request would retrieve a fixed amount of data for a video; while in cycle based schemes, a variable amount of data would be read in fixed-length cycles. Similar deterministic service ideas were also proposed as "dynamically scripted pre-fetching" in [**Staehli93**], but without further elaboration.

Another category of deadline based design that could support VBR videos [**Reddy94**] [**Chen96**] [**Shi96**] can be classified as *dynamic deadline based schemes*, as their service schedules are computed dynamically at run-time instead of at AC time. Their drawback is that one cannot reason about predictability and QoS guarantees, and they can easily reduce to best-effort RT services. Both deterministic and dynamic deadline based mechanisms have the disadvantage that QoS will be degraded unpredictably during temporary overload situations. While effective service (e.g., when a disk is always servicing client requests) overload for different streams will not happen in deterministic deadline based schemes, practical service overload may occur due to disk anomalies. Consequently, a deterministic cycle based scheme would be a better approach if stringent QoS guarantees are required, such as in the case of VOD.

However, most of the above work only considers service guarantees from storage devices and is based on modelling or simulation. [**Ramakrishnan95**] is different in that it describes the implementation of a prototype video server and the scheduling and AC for accessing the server's processor in addition to storage resources. The CPU scheduler uses a combination of RM and weighted round-robin scheduling algorithms and supports three classes of tasks including NRT activities. The disk scheduling is a cycle based round-robin one using worst case seek time estimation. However, the server is oriented to CBR videos because peak rate reservation is used for playing back VBR streams. There is also no consideration on shared resources, such as buses, during AC.

[**Jardetzky92**] and [**Jardetzky95**] also implement a prototype CM file server and consider CPU scheduling. The disk scheduling is based on C-SCAN (Circular-SCAN), but there is no concept of a cycle in the system. Dedicated disk threads are used for read-ahead and write-behind. In addition, each stream has a processing thread created for it, and different classes of threads are scheduled by priorities. The periodic processing threads calculate their own deadlines and request to be scheduled accordingly, resulting in an RM based EDF scheduling for these threads. However, there is no consideration of AC, and the dynamic deadline based scheme used may be unstable during abnormal situations. Furthermore, the one thread per stream approach may incur much stream jitter due to increasing contention between the threads.

3.4 Redundancy Schemes

Because a scalable video server may employ components that could fail independently, system reliability will become an issue and fault tolerance problems should be considered. Two different but closely related issues are *failure recovery* and *video data availability*. When a component fails due to server, node, or media failures, that component is unavailable until it is recovered. Traditional transaction recovery techniques [**Spector89**], such as intention lists, shadow paging, and write-ahead logging, can be used in all components to maintain system state and meta-data integrity. Video data lost due to unrecoverable media failures in disks should be restored from backup storage or reconstructed from redundant information stored on other disks.

However, during failure recovery, video data should be still available to maintain existing streaming services. This can only be achieved by using some forms of data redundancy schemes. The same idea as behind RAID applies, and either *parity* or *replication* information can be stored in the system. Parity based schemes similar to RAID3 or RAID5 but for multimedia applications are studied in [**Tobagi93**] [**Berson95**] [**Tewari96a**] and [**Ozden96a**]. The general approach is to reserve resources for fault situations, and to evenly distribute load when a disk fails by properly placing data and parity information on multiple disks. The drawback for parity based schemes compared to replication is that data needs to be reconstructed before it can be sent to clients, thus requiring extra resources and more complex scheduling methods to maintain RT services.

Replication schemes will store a *backup* copy of a *primary* data block in a disk other than the one in which the primary block resides. Then when only the primary data is lost, its backup is immediately available. *Declustering* is a term used to describe how to distribute backups in replication schemes. Several alternative declustering techniques are: *mirrored* (or duplexing, mirroring, shadowing, or mirrored striping) [**Katzman78**] [**Tandem87**], *chained* [**Hsiao90**], *multi-chained* [**Lee95**], *rotational mirrored* [**Chen95**], and *interleaved* [**Teradata85**] [**Copeland89**] declustering. In all these cases, a file may be striped across multiple disks. However, most of them are studied under time-sharing environment where performance metrics are average response time to multiple requests and average load on each disk.

In mirrored declustering, a disk will store either the primary or backup blocks but not both, and primary blocks striped on one disk have their backup counterparts stored on another single disk. When a primary disk fails, all its load is assumed by the single corresponding backup disk. In contrast, chained declustering allows primary and backup blocks to coexist on one disk, places the backup of a primary block on the successive disk relative to the one on which the primary block resides, and stripes primary blocks onto all disks in a round-robin manner. Then when one disk fails, its load can be shed to its adjacent two disks. A disadvantage of chained declustering is that it is less reliable than mirrored declustering.

Multi-chained declustering further relaxes placement restrictions on backup blocks. A primary block of a file can have its backup stored on one of the subsequent d disks relative to the one where the primary block is stored, and d is termed as the *declustering degree* of the file. The *stride*, s, of a primary block is defined to be the distance between the disks where the backup copy and the primary block reside. In multi-chained declustering, the strides of the primary blocks of a file on one disk are evenly distributed between 1 and d, so that when one disk fails, requests for the file on the failed disk can be evenly offloaded to the subsequent d disks. Chained declustering is thus a special case of multi-chained declustering where $d = s = 1$.

Rotational mirrored declustering is also a special case of multi-chained declustering if disks in a disk array are permuted. It is designed for replicating a video onto another disk array to increase the number of streams which can be served for that single video, and therefore does not apply to network striped video servers which use striping for the same purpose. However, when a set of disks where a single video is striped has reached a certain reliability limit and it is inappropriate to stripe a video to more disks in order to serve more requests, replicating videos to another set of disks would then be more desirable.

While multi-chained declustering tries to store the backup copies of different primary blocks in different disks, interleaved declustering stores the backup copy of each primary block in different disks. With a declustering degree of d, interleaved declustering will segment a backup block into d fragments of equal size and store them in the subsequent d disks relative to the one in which the primary block is stored. Then when the disk where a primary block is stored fails, the load for accessing the primary block is always equally distributed to the subsequent d disks.

3.5 Scalable Storage Servers

This section performs a survey and evaluation of research work on scalable storage systems, especially network striped video servers. Some industrial video servers which may not be based on network striping are also included.

3.5.1 General Purpose Servers

Swift

Swift [**Cabrera91**] [**Long94**] is an I/O architecture to support high data rates in general purpose distributed systems by using network striping. It provides fault tolerance using parity in the same manner as RAID levels 4 and 5. Data transport is based on UDP, and one thread per client per file is used initially in server nodes [**Cabrera91**]. [**Long94**] re-implements Swift using one process per file, and a distributed transfer plan executor model is proposed to manage all communication. The executor is implemented as a distributed event-driven finite state machine that decodes and executes a set of reliable data-transfer operations. It is in fact simulating co-routines and RPCs at a level lower than application programs. The Swift architecture is shown in Figure 3.1.

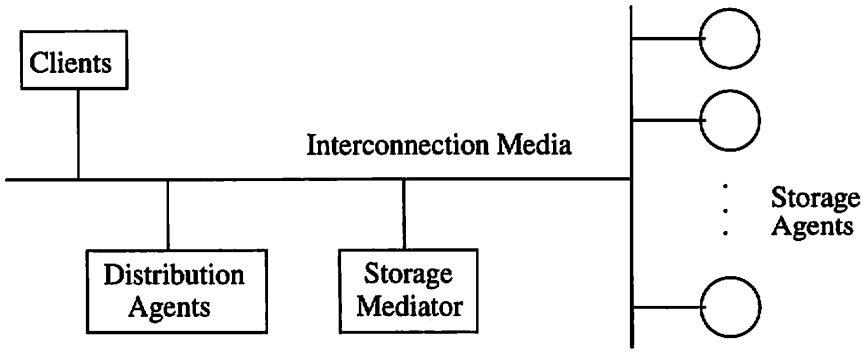

Figure 3.1: Components of the Swift Architecture

A *distribution agent* acts on behalf of its client, the data producer and the consumer, and in practice will be co-resident with the client. Its primary task is to implement striping

of data over several storage agents and to perform all necessary redundancy computation to provide fault tolerance. The *storage mediator* establishes and administers storage and communication resources of the system. It determines SUS, reserves system resources in storage agents, and computes transfer plans, the information needed for distribution agents to access data from storage agents. Finally, *storage agents* administer all aspects of secondary storage media, including data layout, optimisation, and off-line data alignment.

Meta-data, such as directory entries, is managed by the storage mediator. But how it is stored is not specified, and the prototype implementation does not have a storage mediator at all for simplicity reasons. In general, the Swift architecture is flexible because it does not provide detailed specifications. Although it is aimed at multimedia applications, how resources are actually reserved and how QoS can be enforced are not dealt with. Providing high average data rates and supporting QoS guarantees for CM data are not equivalent issues.

Zebra

Zebra [**Hartman95**] is a network striped file system that incorporates ideas from log-structured file systems and RAID. It does not stripe on a per-file basis but on a per-client basis. Each client machine forms its new file data into a single append-only log using caching and stripes this log across storage servers. RAID 4- or 5-like parity is used for data availability, and parity is computed for client logs, not for individual files. The rationale is to provide high performance for writing small files. The structure of Zebra is shown in Figure 3.2, in which squares represent individual machines and circles represent logical entities.

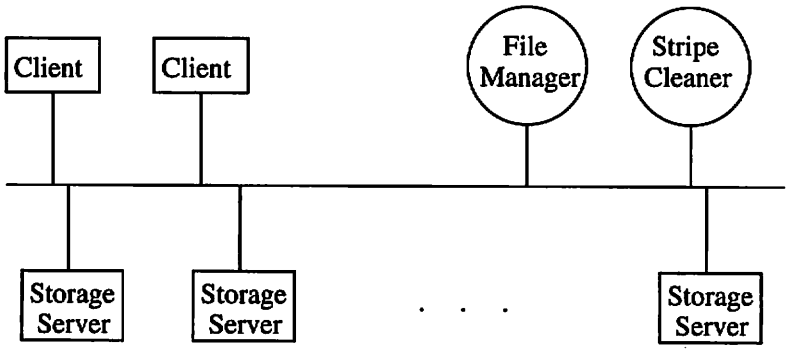

Figure 3.2: The Structure of Zebra

A *client* is a machine that runs application programs. It contacts the file manager to obtain information as to where to store or retrieve data. It is a client's responsibility to compute parity or reconstruct data. *Storage servers* are just repositories for stripe fragments, which are all of the same size, e.g., 512 KB. Storage servers consider a stripe fragment as a large block of bytes with a unique identifier (ID). The *file manager* is responsible for all the meta-data in the file system such as file attributes, directories, symbolic links, and special files for I/O devices. Finally, the *stripe cleaner* is used to reclaim unused space on storage servers.

The file manager does not manipulate any file data except block pointers. A block is a collection of data from a file, and a block pointer contains a fragment ID and the offset of the block's storage location. A client requests block pointers from the file manager and then reads data directly from storage servers. The meta-data from the file manager machine is treated exactly like normal client data and is striped across storage servers, so the file

manager can run on any machine in the network. Data and system state consistency is maintained using logging and checkpoints. Individual component crash recovery is also considered but not implemented.

HPSS

HPSS (High-Performance Storage System) [**Watson95**] is a network centred architecture based on the IEEE Mass Storage Reference Model Version 5 [**Garrison94**]. It assumes a high-speed network for data transfer and a logically separate network for control. The control mechanisms use OSF's DCE RPC. HPSS supports both parallel and sequential I/O, as well as standard interfaces for communication between processors (parallel or otherwise) and storage peripherals. Data is transferred directly between clients and network attached peripherals (NAPs) (e.g., disks, tapes, and frame-buffers) [**Meter96**] under control of HPSS servers. The network protocol for NAPs in HPSS is TCP/IP, although in principle channel protocols or ATM could also be used.

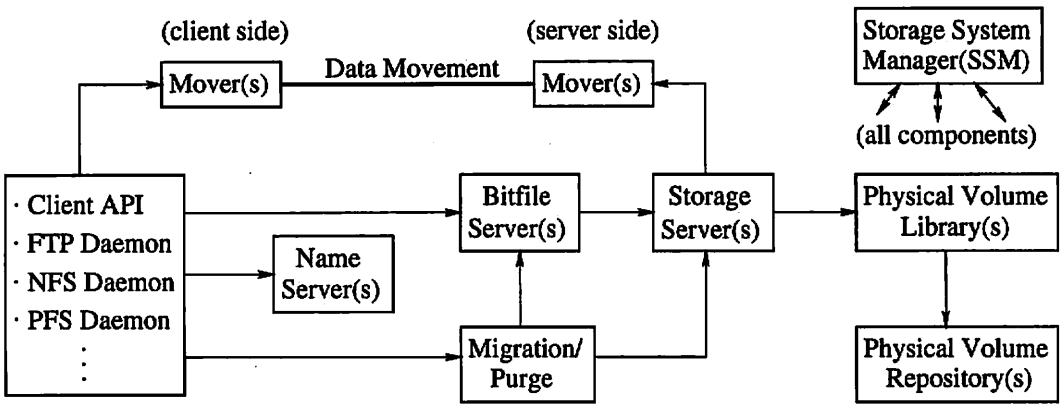

Figure 3.3: The HPSS Software Architecture

The HPSS components are shown in Figure 3.3. The *storage server* provides a hierarchy of storage object abstractions: logical storage segments, virtual volumes, and physical volumes. One storage server is responsible for all storage segment allocation on virtual and physical volumes, but it can also be specialised to manage volumes of the same type, e.g., all disks in the system. It translates references to storage segments into references to the corresponding virtual volumes and finally into physical volume references and peripheral addresses.

A *bitfile* is an uninterrupted bit data stream and has a unique ID associated with it. The *bitfile server* is responsible for mapping requests with bitfile IDs to I/O descriptors containing storage segment IDs, while the *name server* translates user POSIX path names to HPSS objects, such as files or virtual volumes. *Movers* provide components and protocols to transfer data from source devices to sink devices, and there are movers for each type of peripheral and network. Finally, the *physical volume library* manages all physical volumes such as NAPs, and the *physical volume repository* manages all HPSS-supported robotics and their media, such as cartridges.

Parallel I/O in HPSS is enabled through data striping over multiple virtual volumes. SUS and stripe widths are determined by the bitfile server according to client requirements and system specifications. A Parallel Transport Protocol (PTP) [**Berdahl95**] sits above the transport layer in a network architecture and provides data exchange between heterogeneous systems and devices. During parallel I/O, the storage server forks off multiple DCE threads

to handle I/O for each physical volume and associated I/O peripheral. The storage server threads then pass on the expanded I/O descriptor structure with address information to movers associated with the specified peripherals to carry out data transfer.

It can be observed that HPSS and PTP can achieve very high performance for parallel transfer of single large files, which are required by most scientific I/O intensive application programs [**SIOI95**]. In contrast, video servers tend to serve multiple streams with comparatively low bandwidth requirements. This scenario will incur high communication overhead between clients and HPSS servers. In addition, as the storage server in HPSS manages all physical devices of the same type in the system, it could be a potential bottleneck when a large number of I/O requests are present simultaneously. Also HPSS does not provide redundancy information to recover from failures of physical volumes over which data are striped.

xFS

xFS [**Anderson95b**] is a serverless network file system developed as part of the NOW (Networks of Workstations) project [**Anderson95a**]. In place of a centralised server (or set of replicated servers), client workstations in xFS cooperate in all aspects of the file system such as storing data, managing meta-data, and enforcing protections. Data is distributed across storage server disks by implementing a soft RAID using per-client log-structured striping similar to that of Zebra. Control processing is dynamically distributed across the system on a per-file granularity using a serverless management scheme. In addition, xFS eliminates central server caching by using cooperative caching to manage portions of client memory as a large, global file block cache. Cache consistency is maintained on a per-block basis using shared-memory multiprocessor-style schemes such as the write-back ownership protocol similar to the ones used in Sprite [**Nelson88**] and AFS [**Howard88**].

Four types of entities exist in xFS: *clients*, *storage servers*, *managers*, and *cleaners* (cf. the Zebra structure in Figure 3.2). Clients and storage servers have similar functionalities to those in Zebra. However, xFS supports multiple managers and cleaners, scatters cache consistency operations, and can stripe data on distinct subsets of all storage servers. The key to xFS is its ability to dynamically alter the mapping from a file to its manager, which is achieved by maintaining extra indirect mapping information to resolve a file name to where the data is stored on disks. The UNIX Inode structure is used for data block addressing and is interpreted by managers. Data blocks are transferred directly between storage servers and clients by requests from managers.

Although the extra indirection may incur extra load, xFS works well under the traditional workload. This is because of its load balancing ability on both control processing and data retrieval, as well as its use of cooperative caching to take advantage of temporal locality of data. However, these may bring unpredictability for CM applications because caching CM data is not very useful [**Jardetzky92**], and the large sizes of CM files will require a manager to read in extra indirect blocks before a data block can be resolved. In addition, the strategy of one block per request from a manager also creates a high communication overhead for large sequential access.

Petal

Petal [**Lee96**] is a distributed storage system using network striping. It consists of a collection of network connected *servers* that cooperatively manage a pool of physical disks. It takes a different approach by providing large abstract containers called *virtual disks* with

block-level interfaces to its clients, so that it can be easily integrated into any existing computer systems and can transparently support most existing file systems and databases. A replication based redundancy method, specifically chained declustering is used for data availability in virtual disks. The Petal prototype structure is shown in Figure 3.4.

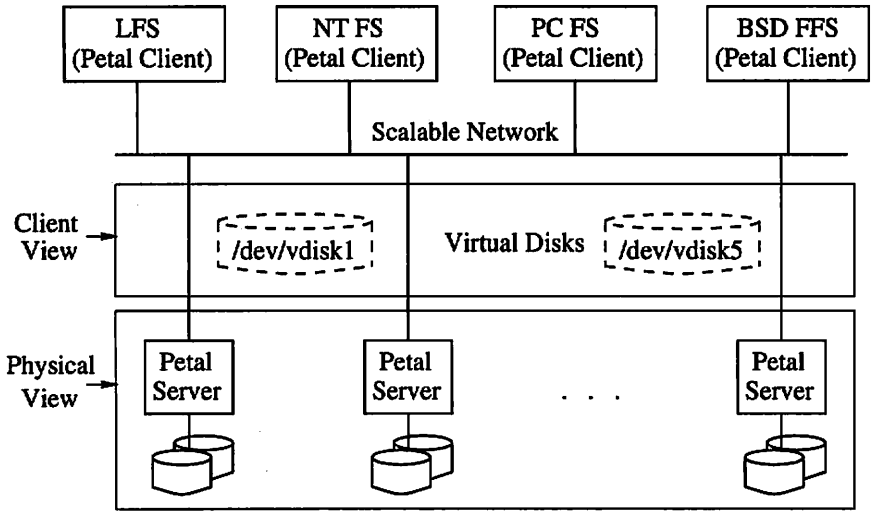

Figure 3.4: The Petal Prototype

Data on a virtual disk is striped over multiple Petal servers. A client request in the form of <virtual diskID, offset> is interpreted by one server initially to find the server where the data is actually stored. If the two servers are not the same, the request is forwarded to the latter, and data is transferred between the server and the client directly. System level global state information is replicated across all Petal servers using the "part-time parliament" algorithm [**Lamport89**], which ensures correctness in the face of arbitrary combinations of server and communication failures and recovery. Other issues such as reconfiguration and data recovery are also considered. However, if applied to video server applications, Petal suffers the same disadvantages as block-based file systems do.

3.5.2 Scalable Video Servers

ISS

The Image Server System (ISS) presented in [**Tierney94**] is not a video server, but it is an image server exploiting network striping. Like HPSS, ISS is motivated by single user applications with high bandwidth requirements. Its main components are *disk servers* which are essentially block servers that are distributed across a wide area network. Data organisation is determined by an application as a function of data types and access patterns, and is implemented during the data load process. Then at run-time all disk servers can send data in parallel to an application. This is useful for applications such as terrain visualisation, which is of interest to the U.S. Army. No redundancy information is provided for data availability in case of server failures.

There is also a *name server* in the system that maps a client request to the physical location (<server, disk, offset>) of the requested data. Requests are sorted on a per server basis and are sent to individual disk servers. Three priority levels exist for block requests, so that a lower priority block can be pre-fetched to a disk server cache but not sent. Priorities are set by client applications using a prediction algorithm to anticipate which part of the data, e.g., corresponding to part of a terrain image, will be needed in the

near future. Data may be discarded if it cannot be sent or did not arrive in time. It is alleged that ISS can be used for video servers, but the claim has not been verified.

MARS

The MARS (Massively Parallel And Real-time Storage) project [**Buddhikot95**] is aimed at the design and prototyping of a high performance large scale multimedia server. A MARS server consists of three basic building blocks: *ATM interconnect*, *storage nodes*, and a *central manager* (Figure 3.5). The central manager performs AC, sets up data flow between storage nodes and clients, and manages associated meta-data in the system. However, how the meta-data is managed is not clear, and no redundancy information is provided. Also only CBR videos are considered.

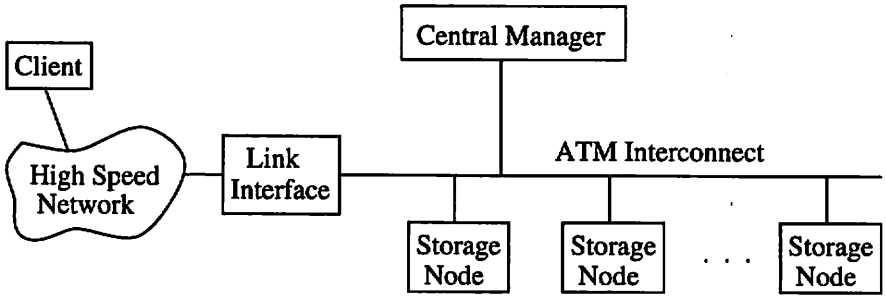

Figure 3.5: The MARS Prototype Architecture

In MARS, data is striped over storage nodes using a staggered distributed cyclic layout to facilitate fast forward operations by skipping storage nodes. The QoS enforcement scheme is cycle based, and cycles are enforced by ATM control cells. A dual-buffer scheme is used to separate disk retrieval and network transmission in a cycle. A cycle is further divided into sub-cycles according to the number of storage nodes in the system to balance network activities in each sub-cycle. How QoS is actually guaranteed in storage nodes is not described. MARS, as presented in [**Buddhikot95**], is a prototype based on analytic modelling.

SPIFFI

The SPIFFI scalable VOD system [**Freedman95**] is based on SPIFFI [**Freedman94**], a high performance, scalable parallel file system that is developed and implemented on an Intel Paragon. SPIFFI assumes a set of *compute nodes* and *disk nodes* connected by a tightly coupled network. Data is striped across all disks in the system in a round-robin fashion without redundancy information. A fragment of a file which is stored on one disk is laid out contiguously. The file system at each disk node is responsible for mapping logical file blocks to physical file blocks and for recording each fragment's size, thus keeping file meta-data to a minimum. Each compute node caches a file's meta-data and calculates which disk contains which blocks of the file so that an application can request data blocks directly from a disk node. A control thread in some node accepts file open/close operations and manages file meta-data. 8 disk request threads are used at each disk node to receive data requests, do the requested disk I/O, and then reply to the compute nodes. Large buffer pools are maintained in each disk node for use by read-ahead and write-behind.

The SPIFFI VOD system described in [**Freedman95**] suggests the use of inexpensive commodity components, and *video terminals* are used in place of compute nodes. Video data

is explicitly requested by video terminals, and each request bears deadline information. This dynamic deadline based QoS enforcement scheme naturally leads to the use of a deadline based disk scheduling algorithm, in which requests are classified according to their urgency and SCAN is used in each prioritised class. Some pre-fetch schemes which engineer the deadlines of future requests are proposed to maximise the performance of the disk scheduling algorithm. The results are presented from a simulation study, and no consideration has been given to AC. Implementing the system on a cluster of Sun workstations is suggested as future work.

Clustered Video Servers

[**Tewari96c**] develops an analytical model for clustered video servers, which is composed of *front-end delivery nodes* and *back-end storage nodes* connected by a switch. A front end node explicitly requests data blocks from back end nodes, and transmits data to clients at regular intervals from its read-ahead buffer. An underlying software layer makes the storage nodes appear as a virtual shared disk to the front ends. Deadlines are associated with requests and EDF disk scheduling is used in storage nodes. The QoS criterion is to minimise the number of block requests missing their deadlines. A result similar to [**Reddy94**] is reported — that more read-ahead buffers but with smaller block sizes may give better performance than the dual-buffer approach.

Only CBR is considered and data is striped on disks using a random placement algorithm, so that it is easier to redistribute data when additional disks are added into the system. [**Tewari96a**] compares mirroring and the software RAID approach for data availability in clustered video servers of the same architecture using an analytic model. A parity based scheme is selected as a better choice, and a random parity placement method is proposed to equally utilise space and bandwidth of all disks during both failure and normal operations.

SCAMS

The aim of the SCAMS (SCAlable Multimedia Servers) project [**Lougher96**] is to produce a distributed hierarchical VOD storage architecture that will scale to tens of simultaneous streams, and will be able to operate over a widely distributed area. The SCAMS hierarchy consists of three levels: disk striping in each *storage node*, node striping in each *domain*, and file replication between different domains. Based on network-link capacity and file popularity, algorithms are proposed to compute optimal configuration of domains and to determine when to replicate files between domains. The system and its behaviour in a variety of network configurations have been investigated through simulation. No redundancy information is provided in the node striping level because of replication in upper layers.

In follow-on research, [**Pegler97**] proposes to use dynamic file component replication to address the scalable limitations of network striping, file replication, and hierarchical techniques by exploring the use of scalable compression. As bandwidth and storage requirements of each compressed component are much less than the original CM file, when overload occurs, the copying overhead incurred during migration or replication is kept low because only the most highly loaded components need to be moved or copied. Separate compressed components are combined using *mixing agents* connected to networks. AC, file replication or migration, and the selection of a mixing agent are handled by a *central manager*.

3.5.3 Industrial Video Servers

Microsoft

Microsoft's Tiger video file server [**Bolosky96**] is a prototype built out of a collection of computers connected by a high speed network. Each of the computers is called a *cub*, and every cub has some number of disks dedicated to storing video data and a disk used for running the file system. Tiger stripes its file data across all disks in the system in a round-robin fashion, and the striping block size, which is typically in the range of 64 KB to 1 MB, is the same for every file. Fault tolerance is provided by replication, specifically interleaved-declustering is used. To avoid retransmission and retain RT performance when data is lost, UDP over ATM is used as the communication channel between cubs and clients. The hardware layout of Tiger is shown in Figure 3.6.

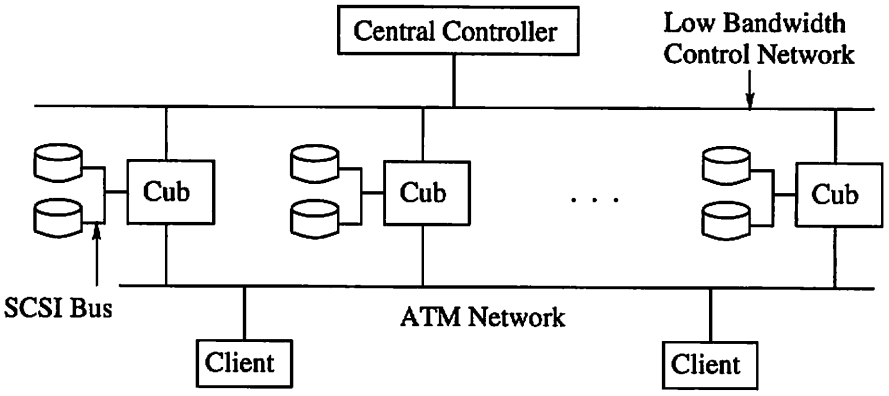

Figure 3.6: The Tiger Hardware Layout

In addition to the cubs, the Tiger system has a *central controller* machine. This controller is used as a contact point for clients of the system, as the system clock master, and for some bookkeeping activities. Data does not pass through the controller; and once a file playback has started, the controller takes no action until the end of the file is reached or the client requests termination of the stream. Tiger makes no assumption about the content of the files it serves, but it assumes that all files on a particular server have the same bit rate. This greatly simplifies AC and QoS enforcement in the server.

The QoS enforcement scheme in Tiger is similar to the cycle based approach. Tiger uses a single *system schedule* to coordinate all disk services. Assume a Tiger server can support N streams of the same rate, then the system schedule has N single entry slots, with an active stream's service request appearing in exactly one slot. In each slot, a disk will retrieve one block for a stream if the slot is not empty. In the prototype, each block has a playback time of 1 second, and the block service time, which can be seen as the cycle length, is 221 ms. Every disk in a Tiger system walks down the schedule, processing a schedule slot every block service time (i.e., every cycle). However, different disks will not process the same slot at the same time, and successive disks are offset from one another in the schedule by a block playback time. The prototype system has 15 disks, and around 15 * 1 second / 221 ms $\approx$ 68 streams can be supported when no disk fails. The prototype is implemented on a collection of PCs running Microsoft Windows NT.

Oracle

The Oracle Media Server [**Laursen94**] [**Laursen95**] is a product supporting access to all types of conventional data stored in the Oracle relational and text databases. In addition, it has an RT stream server that supports storage and playback of CM data. All access into the RT section of the Media Server goes through scheduling dispatchers for AC and scheduling based on CPU, disk, and memory resources. Variability caused by VCR commands is also factored into an RT scheduler in order to provide continuous guaranteed services. QoS enforcement is cycle based in that the entire network is treated as a time division multiplexed space, and each node has a set of time slots in which it can write to another node. The RT stream server is built on a micro-kernel and raw disk and network I/O and unreliable messaging. Only CBR is supported and peak rate reservation is used for VBR playback.

The RT server can identify and correct disk failures without interrupting RT data flow. But the exact redundancy scheme used is proprietary, as well as scheduling mechanisms in the RT server. Communication channels are provided through the Oracle Media Net, an implementation of layers 3 (network) and 4 (transport) of the OSI reference model, with additional ability to deal with network topologies and quantities of traffic common to consumer-based networks. The Media Net is used to build reliable mechanisms such as RPC, distributed object invocation, and NRT data transport, while video data is transferred through separate isochronous channels. However, the Oracle Media Server relies on massively parallel machines such as the nCUBE to serve large amounts of requests [**Pearson94**], which is contrary to the approach taken by this research which uses network striping and loosely coupled components.

Silicon Graphics

The ITV system from Silicon Graphics [**Nelson95**] is based on servers running on its Power Challenge Series, which are RISC multi-processor systems with 256-bit wide data buses [**Boxer95**]. The Challenge Servers run IRIX, SGI's version of UNIX. They are connected via an ATM network to set-top computers, and themselves are connected by FDDI. Scalability and availability are achieved by service replication rather than network striping. Three techniques that are crucial to the successful development of the system are identified as: distributed objects for client/server interactions; leveraging the name service to support replication and recovery; and designing availability into all services. The QoS enforcement scheme is not clear, and only CBR videos are supported.

Hewlett-Packard

The HP MediaStream server [**Natarajan95**] uses specialised I/O hardware to deliver compressed video and audio. The specialised hardware, a *video transfer engine*, simply pulls data out of disks and transmits it at precise data rates. Three hardware modules in an engine are: *data sources*, a *stream controller* and a *stream router*. A data source module accesses MPEG-2 encoded video from disks under requests of the stream controller, and delivers data to a Sonet network through the stream router. The QoS enforcement scheme, as suggested in [**Chen96**], may be a dynamic deadline based one. It is not clear whether data is striped across multiple engines to serve multiple streams.

Online Media

There are two phases for the Cambridge ITV trial [**Vincent95**] led by Online Media. In phase one, an ORL discbrick [**Chaney95**] is used to serve up to 25 MPEG-1 streams of 2 Mbps each. Phase two uses an ICL PrimServer (Parallel Interactive Media Server) which is based on the massively parallel Goldrush system of ICL, and can support around 2,000 simultaneous streams when fully populated. However, the data management and the QoS enforcement scheme in the parallel server are not clear.

3.6 Summary

This chapter has surveyed work on some important aspects of video server design, namely, disk scheduling, data placement, QoS enforcement, and redundancy schemes. It also examined closely related work on scalable storage systems using network striping. General purpose scalable servers are either focused on scientific applications demanding high I/O bandwidth, or are designed for traditional file usage patterns, and neither provides QoS to CM applications. Scalable video servers from academia are more concerned with analytic modelling or simulation of storage services, rather than integrated resource management to guarantee QoS. Industrial video servers, except the Microsoft Tiger, rely on high bandwidth internal interconnection, instead of network striping. Also most existing scalable video servers only support CBR videos.

Chapter 4

System Architecture

This chapter first identifies the requirements imposed on the software system framework for network striped video servers. Then it presents the Cadmus architecture, which is scalable, flexible, and QoS aware. Two important aspects of Cadmus, the meta-data management and the redundancy scheme, are described, along with some other architectural issues.

4.1 Requirements

The storage system for network striped video servers is not identical to traditional distributed file systems. The requirements for such a system can be deduced by examining the shortcomings of the network striped storage servers surveyed in the previous chapter when they are applied in a video server environment.

First, the structure of some of the servers is not *scalable*, that is, potential bottlenecks still exist when there are large numbers of simultaneous requests. For example, some of them have a single control machine to accept and analyse all client requests in the whole system, while some others use a single server to manage all system meta-data, including that for physical devices. Some of them have modules to explicitly send block based requests to storage servers, which may be overwhelmed by the request overhead when many such modules exist.

Second, the design of some of the servers is not *flexible*. For instance, several of them only support fixed SUS and a fixed striping group for all objects in their systems, and most of them do not explicitly provide VBR support or consider VBR smoothing. While ZBR disks are becoming prevalent, none of them have considered the case of AC with variable disk transfer rates in storage servers. Explicit support for NRT activities in storage servers is not common. Moreover, some do not store redundancy information at all, while some others use a parity based scheme, which may increase the complexity of those systems. All these deficiencies will limit the corresponding servers to work only in a special environment with restrictive assumptions.

Finally, some of the servers lack *QoS awareness*. For example, most of the general purpose servers are not oriented to large sequential access, while some of the video servers have no AC considerations. Furthermore, integrated scheduling support in storage servers considering all physical resources is rare.

The conclusion is that the system architecture for a network striped video server should be scalable, flexible, and QoS aware. The above discussion when applied to the network striped storage servers described in the last chapter is summarised in Table 4.1. The aim of the Cadmus architecture is to leave its column blank if it were on the same table.

Table 4.1: Limitations of Some Existing Network Striped Storage Servers

limitations		general purpose servers					video servers				
		Swift	Zebra	HPSS	xFS	Petal	ISS	MARS	SPIFFI	Clustered	Tiger
not scalable	1	✓	✓				✓	✓	✓		✓
	2			✓			✓				
	3	✓	✓	✓	✓	✓	✓		✓	✓	
not flexible	4		✓		✓	✓	✓				✓
	5	✓	✓					✓	✓	✓	✓
	6	N/A					✓	✓			✓
	7						✓	✓	✓	✓	✓
	8						✓	✓	✓	✓	✓
	9							✓			
	10			✓			✓	✓	✓		
	11	✓	✓		✓					✓	
not QoS aware	12		✓		✓	✓					
	13	N/A					✓		✓	✓	
	14						✓	✓	✓	✓	

1. Single control machine.
2. Single server managing all meta-data including that for physical devices.
3. Explicit block-based requests to storage servers.
4. Fixed SUS.
5. Fixed striping group.
6. No VBR support.
7. No VBR smoothing considerations.
8. No ZBR disk considerations in AC.
9. No NRT activity support in storage servers.
10. No redundancy support.
11. Parity based redundancy scheme.
12. Not oriented for large sequential access.
13. No AC considerations.
14. No integrated scheduling support in storage servers.

4.2 The Cadmus Architecture

4.2.1 Entities, Components, and Objects

Entities and Components

Cadmus assumes a hardware structure of many *physical entities* connected by a high-speed switched network. Physical entities are either NAPs with disk device(s), or diskless network computers (NCs). The Cadmus software system framework specifies a set of *logical components* that can run on physical entities. An overview of the Cadmus architecture is shown in Figure 4.1.

There are seven types of (logical) components in the Cadmus system framework: *storage servers* (SSs), *file servers* (FSs), *directory servers* (DSs), *name servers* (NSs), *stream control*

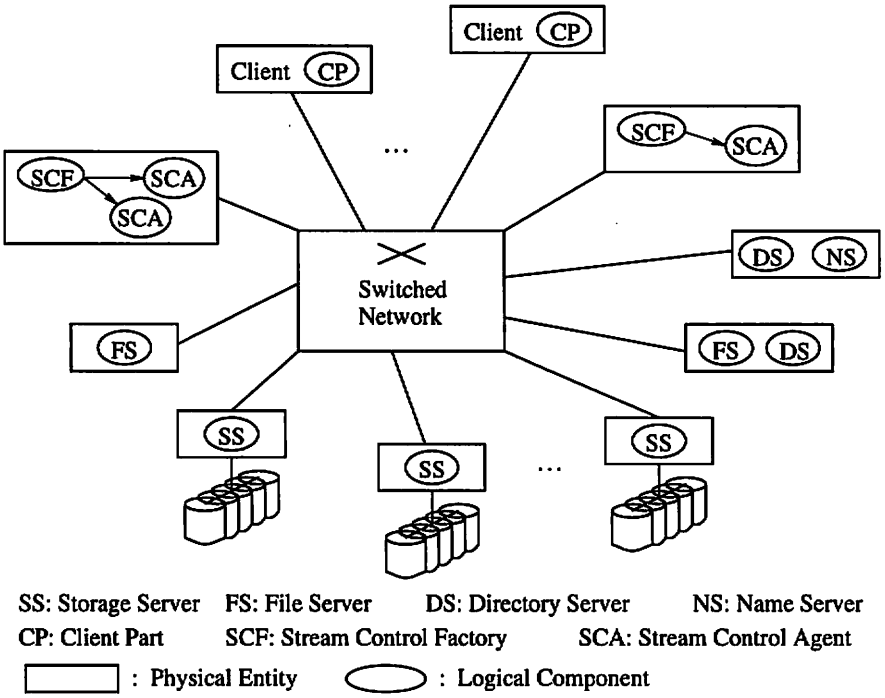

Figure 4.1: The Cadmus Architecture Overview

factories (SCFs), *stream control agents* (SCAs), and *client parts* (CPs). An SS resides in each network attached storage node, while other components can run on any diskless NC. Uninterpreted data objects are managed by SSs and FSs, while a DS maintains a hierarchical directory for its customers. An SCF will create an SCA for a CM stream's playback or recording. However, data is transfered between SSs and CPs directly, where CPs are data aggregating or scattering points. Each server or factory component has a unique ID associated with it, and NSs are used to map component IDs to physical entities. The functionalities of these components will be described in more detail in the next section.

Physical and Logical Objects

The term *object* in Cadmus is only used to refer to stored data. There are two types of data objects managed by Cadmus: *physical objects* (POs) and *logical objects* (LOs). An LO is an uninterpreted byte stream, which can be used to represent a traditional file or directory. An LO has a unique ID, the *LOID*, associated with it. Data of an LO is striped onto a fixed set of SSs in the system in a round-robin manner. However, different LOs may be striped on different sets of SSs and may have different SUS. An LO is managed by one FS only.

A PO is also an uninterpreted byte stream which is managed by an SS. A PO is associated with a unique ID, the *POID*. Normally, the set of striping units from a single LO onto the same SS forms a PO, and a non-zero length LO is always mapped to one or more POs. This relationship between LOs and POs is shown in Figure 4.2. However, a PO may exist on its own and not be contained in any LO. These kinds of stand-alone POs are called *special POs*, and are used to store and replicate important meta-data at the system level. Both LOs and POs can be opened in one of two modes: *RT mode* or *NRT mode*. An object is in RT mode if it is opened for playback or recording, NRT mode otherwise. In the following, RT-LO/NRT-LO and RT-PO/NRT-PO stand for objects opened in the corresponding modes.

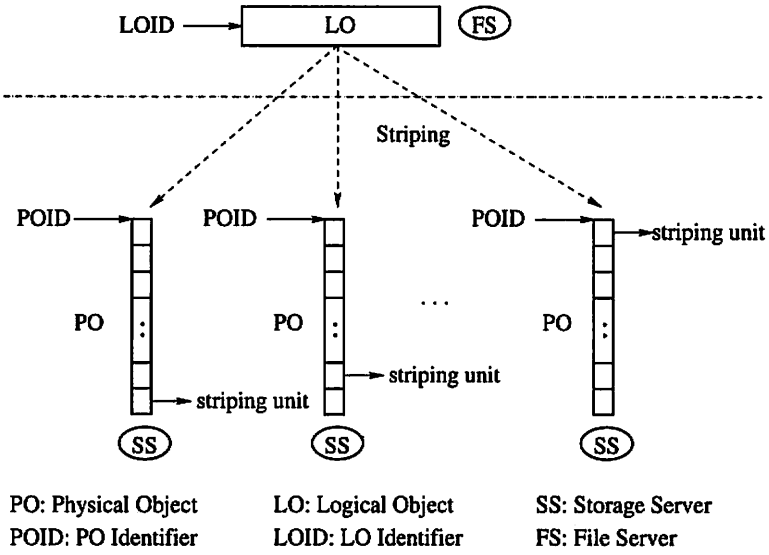

Figure 4.2: Logical Objects and Physical Objects

4.2.2 Component Functionalities

Storage Servers

The four main tasks for an SS are: PO management, data retrieval, AC, and QoS enforcement. To facilitate large sequential access, the extent-based approach is used in SSs to manage POs. However, the Cadmus architecture does not mandate a specific extent-based implementation, nor a data placement strategy for POs. Each SS could have its own policy to store POs on the devices it manages, and its customers can only access data at the PO level using POIDs, not at the device block level. This way, implementation details of an SS can be encapsulated, resulting in a modular system. This is contrary to the fixed block or fragment based approach taken by some existing scalable servers.

Data retrieval in an SS includes data access at both disk and network devices. Cadmus uses the deterministic cycle based approach to guarantee QoS in SSs, and the time in each SS is divided into equal length working cycles. The retrieval schedules for each cycle are dynamically computed at run-time, and they contain admitted access requests from both RT streams and NRT activities. The data of an RT-PO is transferred between a CP and an SS without explicit requests, while read/write operations are defined for NRT-LOs and NRT-POs. NRT data requests are also issued by an SS for its own meta-data management.

AC is performed on all physical resources in an SS on a cycle basis, and resources are reserved if a request, either RT or NRT, is admitted. The geometry of a ZBR disk is understood by an SS and is used in the AC on the disk resource. For an RT playback request of a VBR video, a dynamically computed *read table*, which specifies the amount of data to read in each cycle, is passed from an SCA to each SS where the video is striped over and is used in the AC procedures. Similarly, a *receive table*, which tells how much data to receive in each cycle, is computed by each SS on which a video is to be striped based on peak rate reservation and is used in the AC for VBR recording.

There are two reasons for AC to be done in SSs, instead of a central point. First, an SS is a better place to manage the physical resources under its control and to record their usage. Second, processing costs for AC can be distributed to multiple SSs, making the system more scalable. Integrated scheduling considering both RT and NRT activities is

also provided in an SS to enforce QoS guarantees and to deal with overload situations. AC and QoS enforcement will be considered in more detail in Chapter 6 and Chapter 7.

File Servers

The primary functions of an FS are to create and delete LOs and to maintain the mapping of an LOID to a set of POIDs, i.e., {POID}, which corresponds to the set of POs the LO is striped over. When an LO is created, an FS determines the *striping group*, the set of SSs over which the LO will be striped, and the SUS. Then it will contact the SSs for them to create POs and return the POIDs. When an LO is opened, the {POID} is used to make data connections between a CP and the SSs. Making connections and operations such as AC and VCR control for RT-LOs are delegated to SCAs. However, the Cadmus architecture does not specify where operations such as read and write for NRT-LOs should be implemented. Chapter 7 will describe a scalable approach by co-locating the implementation of NRT-LO operations with CPs.

There may be multiple FSs in a Cadmus system to balance the load imposed on them, although processing overhead of LOs in FSs is kept to minimum because user data does not pass through FSs. Different FSs are independent of each other (because of the meta-data management scheme described in the next section). An LO can be moved to another FS by moving the LOID:{POID} mapping to the latter. However, the LOID should be changed to identify the new FS that contains the LO. Location independence can be achieved by storing a pointer to the new LOID in the old FS. A PO can also be moved to another SS and a new POID should be assigned and the LOID:{POID} mapping be updated in some FS. However, the LOID remains the same in this case. Although a pointer to the new POID could help in some situations, this information is lost when the old SS suffers medium errors because the meta-data managed by SSs is not stored redundantly.

Stream Control Factories and Agents

One of the unique features of Cadmus is the introduction of SCFs and SCAs, which are used when LOs are opened in RT mode. SCFs create and destroy SCAs, and an SCF and the SCAs it manages are located in the same physical entity. The primary functions of an SCA are to coordinate distributed AC, set up data connections between a CP and the SSs, and control the data flow according to VCR type commands from a client. The LOID:{POID} mapping is passed from an FS to an SCF for the creation of an SCA when an LO is opened in RT mode, so that the SCA has access to the corresponding SSs. An SCA is the single point of contact for AC for RT playback or recording of a single stream. It will query the individual SSs and make AC decisions on the information it gathers. If the load on SCF entities is high, additional entities and factories can be added to process more client requests. Thus the scaling of stream control capacity and the scaling of storage performance are independent.

An important objective for SCAs is to deal with VBR video streams. Cadmus distinguishes between the *displayed set* and the *read set* of a video. The displayed set of a video is the set of data actually displayed in a client in each cycle, and the read set is the set of data actually read by the SSs and sent to a CP in each cycle. Every VBR video has a *video index* file associated with it. A video index contains three parts: a header, the displayed set, and an *I-index*, which tells the positions and the lengths of the frames to be retrieved during fast forward or fast reverse, for example, when MPEG is used, these frames could be the I frames.

A video index is interpreted by an SCA to compute the read tables for use by the SSs after employing some smoothing algorithm. With video indexes, the same SCA can be used to playback any format of VBR videos. VBR processing by an SCA for playback is summarised in Figure 4.3. RT recording for VBR videos is achieved using peak rate reservation, and a video index is written by each client as an NRT-LO during recording or it can be extracted afterwards. The receive table used for RT recording is computed by the SSs instead of by an SCA. No index is needed for CBR videos during either playback or recording.

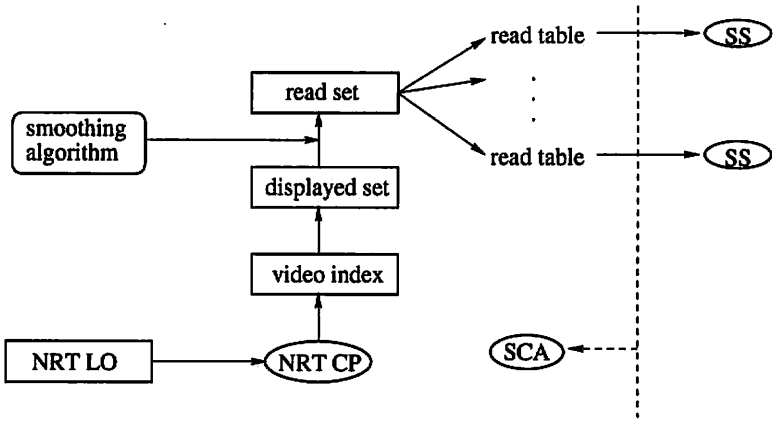

Figure 4.3: VBR Processing in the SCA for Playback

The functionalities of SCAs have evolved since the earliest design stage leading to Cadmus. The following will use playback as an example. In the first pre-Cadmus prototype, an SCA read data from the SSs into the entity where the SCA resides before data was sent to a CP. This puts an SCA into the RT data path, which has the adverse effects of increasing the load of SCF entities and reducing the availability of the system. A second version let SSs send data directly to CPs, but an SCA still had to send individual requests to the SSs with deadlines. Besides the drawbacks of the dynamic deadline based QoS enforcement approach, when one SCA crashes, the relative RT stream controlled by the SCA dies too. Instead, an SCA in Cadmus sends read tables of a VBR video to the SSs during AC, and the tables are retained by the SSs if the admission is successful and committed. From then on, the SCA only responds to interactive commands from a client, and data is transferred between the SSs and a CP without explicit requests from any third party.

Client Parts

A CP is the aggregation or scattering point of the data of an LO from or to the SSs. A CP understands the LO's striping structure and redundancy method. A CP is operating in either RT or NRT mode, depending on the mode of the LO that is being opened. An RT-CP will send/receive data to/from the SSs on a per cycle basis using an unreliable channel to prevent retransmission when error occurs. A CP used for recording also regulates the rate of the data to be sent to the SSs, while CPs used for playback will discard late arriving data. An RT-CP for VBR playback has a copy of the read set so that it can correctly anticipate data arrivals from the SSs without explicit requests. For VBR recording, an RT-CP will send data to the SSs when there is data available from the producer, and enough resources are reserved in the SSs for the unpredictable arrivals of data. An RT-CP for recording will also write redundancy information to the SSs. AC for RT recording will be further described in Chapter 6.

Unlike RT-CPs which are automatic, NRT-CPs will export interfaces with data send and receive operations that can be used to implement read and write operations defined for NRT-LOs. An NRT-CP will transfer data to and from the SSs using reliable channels. It should be emphasised that a CP is only a logical component and can be used wherever data of an LO should be assembled from or distributed to the SSs. For example, in Figure 4.3, an NRT-CP is used in the SCA to read in the video index. As Cadmus uses the replication scheme, as will be shown in later sections of this chapter, there is no computational overhead for operations creating or using redundancies.

Directory Servers and Name Servers

DSs maintain mappings from textual names to LOIDs. The data of one directory with `textual_name:LOID` mapping is stored as an LO, and a UNIX style hierarchical directory can be readily implemented on Cadmus. NSs manage mappings from server or factory component IDs to physical entity addresses. To provide bootstrap at system startup or after component failures, the NS mapping data is directly backed up and replicated among some well known SSs using well known POs. This is feasible because the amount of data managed by NSs is small in nature, and SSs provide the mechanism of special POs. Thus NSs are independent of any FS.

4.2.3 Features

The above discussion can be summarised as a list of major features of Cadmus:

1. Different LOs can have different SUS, and each FS can stripe any of its LOs to any subset of SSs in the network.
2. Each SS could have its own policy to store the POs on the devices it manages, and only PO access interfaces are exported.
3. Directory services are separated from underlying file and storage services. A three-level naming scheme is supported: textual names, LOIDs, and POIDs.
4. All meta-data is stored on SSs where LO data is stored. No additional supporting file systems are needed, and FSs/SCFs/DSs/NSs can run anywhere in the network.
5. Multiple FSs/SCFs/DSs/NSs can be supported if necessary. Each logical component, including an SS, can be added incrementally and independently.
6. NSs do not depend on any FS, and FSs are independent of each other.
7. The system does not depend on any particular video encoding format.
8. NRT activities are supported in the presence of RT stream services.

4.3 Meta-Data Management

The meta-data of an FS is a table of `<LOID:LO_ATTR:{POID}>` tuples, i.e., the *LOID table*, where LO_ATTR represents attributes of an LO including the SUS. An SS can resolve a POID to a list of disk blocks which store the PO. An SS has a `<POID:PO_ATTR:extent_list>`

table, i.e., the *POID table*, where PO_ATTR refers to attributes of a PO, as part of its meta-data (the actual format of this index meta-data is implementation dependent). In addition, an SS has to maintain free block bitmaps for its disk devices. For special POs, each SS reserves a region of continuous entries from the start of its POID table, with each entry corresponding to a potential special PO.

There are several requirements for storing the meta-data. First, it should make the meta-data management of different components as independent as possible. Second, it should make it possible to reconstruct FS meta-data and LO data after one SS is lost. Third, it should support the features listed in Section 4.2.3. Finally, multiple level naming resolution should be optimised since both POID and LOID tables can be very large. To achieve these goals, Cadmus adopts a number of solutions.

For each FS, its meta-data (the LOID table) is just a special kind of LO and is striped across a subset of SSs, with its FSID identifying this special LO. An `<FSID:FS_ATTR:{POID}>` tuple tells where an FS's LOID table is striped, where FS_ATTR refers to the FS attributes. This tuple is created when an FS is constructed and is obtained subsequently to get the LOID table when the FS is rebooted. Although these special LOs are managed by the individual FSs, the collection of the tuples, i.e., the *FSID table*, is replicated in special POs whose entries have the same offset from the start of the POID tables in some SSs. This way, FSs are made independent of each other.

The SS specific meta-data, i.e., POID tables and free block bitmaps, is neither striped nor replicated. When an SS is lost, new SS specific meta-data is created during reconstruction of LO data. This simplifies the management of SS meta-data and achieves the effect of SS abstraction. When a new SS is created to replace a previously lost one, special POs are replicated from another SS first if necessary. Then for each FS in the system (by searching the FSID table from the special PO), if it stripes its meta-data in the lost SS, then the lost meta-data is constructed. Finally, for each FS in the system, for each LO that the FS manages, if the data of the LO is striped in this SS, then the lost LO data is constructed. During the reconstruction, a new bitmap and POID table are created by the SS. Since POIDs are unique among all SSs, they are reused and have exactly the same meaning as before the SS is lost. However, a PO's `extent_list` may now be different from before. For example, as reconstruction is performed on each LO and its PO on the SS in turn, the SS can allocate contiguous disk blocks for reconstructed POs.

To efficiently support multiple level naming resolution, similar ideas from MSSA [**Lo94**] are applied to Cadmus to choose component and object IDs. To facilitate logical component locating from an object ID, an FSID is embedded as the most significant bits in an LOID, while an SSID is embedded in a POID. In addition, the lowest significant bits of an object ID correspond to the offset of the object's entry in the relative LOID or POID table, so that there is no table look up overhead. The bits in the middle can be filled with an epoch number to guarantee the uniqueness of an ID and to facilitate capability based access control.

4.4 The Redundancy Scheme

4.4.1 A Comparison of Redundancy Schemes

This subsection will compare redundancy schemes in order to select the one best suited for Cadmus. To facilitate discussion, the system state is categorised as in either *non-fault mode* or *fault mode*. Non-fault mode is reached when no SS fails, otherwise the system is in fault mode. Assume a video (or LO) is striped across n SSs in the system labelled SS(0),

SS(1), $\cdots$, SS($n-1$). To maintain consistency with Section 3.4, the terms *striping unit* and *block* are used interchangeably.

In theory, parity schemes as used in RAID can be extended to Cadmus to provide data availability during fault mode. However, software RAID3 needs buffers in CPs to store all the data in a parity group before the data can be assembled in both fault and non-fault modes. If SUS is large or videos are widely striped, large buffers are needed. If SUS is small, then more seek and rotational overhead is incurred in each SS and disk bandwidth utilisation is very low. For the RAID5 scheme, extra buffering will be required to reconstruct lost data for each stream served from a parity group where there is an SS failure. And in each cycle, each SS in a parity group has to reserve an equal amount of resources to those needed in non-fault mode to provide immediate availability during fault mode, leading to a 50% waste of resources when there are no SS failures.

From the perspective of the cyclic scheme, mirrored and chained declustering are identical. This is because in both methods, all the backup copies of the primary blocks in one SS, say SS(i), are stored in another single SS, say SS(j), where $i \neq j$. Assume J primary blocks can be retrieved in one cycle in SS(i) during non-fault mode, then SS(j) should reserve resources that could be used to retrieve J backup blocks in the same cycle in case SS(i) fails. This reservation applies to any SS that stores backup blocks for another SS and 50% of the resources are wasted during non-fault mode.

The drawbacks of mirrored and chained declustering suggest that the load of retrieving the backup copies of the J primary blocks associated with the requests in a cycle in one SS be evenly spread into multiple SSs. Both multi-chained and interleaved declustering can achieve this load balancing effect by storing the backup copies of the primary blocks in one SS onto multiple SSs. The following paragraphs will perform simple analysis to compare these two methods.

For simplicity of analysis, assume all the videos in the system are of the same rate with SUS as R, which corresponds to the data retrieved in each cycle, and a declustering degree of d, where $d < n$. For the primary blocks of a video that are stored in SS(i), their backup copies are always stored in the subsequent d SSs starting from SS($i+1 \bmod n$).

Using multi-chained declustering, if the ith primary block of a video, Pi, is stored in SS($ps(i)$), and its backup copy, Bi, is stored in SS($bs(i)$), then:

$$ps(i) = i \bmod n \tag{4.1}$$

$$bs(i) = (i \bmod n + \lfloor i/n \rfloor \bmod d + 1) \bmod n \tag{4.2}$$

A data layout example of $n = 6, d = 4$ using multi-chained declustering with the first 24 blocks of a video file is shown in Table 4.2.

Using interleaved declustering, the same example with the first 6 blocks is shown in Table 4.3, where Pi refers to the ith primary block of a video, and B$i.j$ refers to the jth backup fragment for Pi.

Both schemes have the potential to off-load block retrieval requests in a cycle to multiple SSs when one SS is lost. For instance, in the above multi-chained example, the primary blocks P0, P6, P12, and P18 in SS(0) have backup copies B0, B6, B12, and B18 stored in SS(1), SS(2), SS(3), and SS(4) respectively. If in a cycle the four primary blocks are retrieved in SS(0), then the failure of SS(0) will add 25% extra load to each of its subsequent 4 SSs in the same cycle, instead of 100% extra load to another single SS in the case of mirrored or chained declustering. For interleaved declustering, when SS(0) fails, all its load in any cycle is evenly spread to the subsequent 4 SSs.

Table 4.2: Multi-Chained Declustering: $n = 6, d = 4$

SS(0)	SS(1)	SS(2)	SS(3)	SS(4)	SS(5)	stride
P0	P1	P2	P3	P4	P5	
B5	B0	B1	B2	B3	B4	1
P6	P7	P8	P9	P10	P11	
B10	B11	B6	B7	B8	B9	2
P12	P13	P14	P15	P16	P17	
B15	B16	B17	B12	B13	B14	3
P18	P19	P20	P21	P22	P23	
B20	B21	B22	B23	B18	B19	4

Table 4.3: Interleaved Declustering: $n = 6, d = 4$

SS(0)	SS(1)	SS(2)	SS(3)	SS(4)	SS(5)
P0	P1	P2	P3	P4	P5
	B0.1	B0.2	B0.3	B0.4	
		B1.1	B1.2	B1.3	B1.4
B2.4			B2.1	B2.2	B2.3
B3.3	B3.4			B3.1	B3.2
B4.2	B4.3	B4.4			B4.1
B5.1	B5.2	B5.3	B5.4		

Interleaved declustering can always evenly distribute the load of a failed SS to its subsequent d SSs in any cycle. This may or may not be true for multi-chained declustering. For example, if in a cycle only 2 blocks P0 and P6 are retrieved in SS(0), then the load is only distributed to SS(1) and SS(2) when SS(0) fails. The extreme case is that when one block is retrieved per cycle, 50% of the resources still need to be reserved for fault mode. However, multi-chained declustering could achieve an even load balancing during SS failures when there are multiple primary blocks being accessed per cycle, and the strides of these blocks are evenly distributed. This can be achieved in practice, as will be shown.

Assume that the disk seek time profile constants [**Oyang95**] [**Vin95**] are a and b so that $y = a + b * x$, where x is the seek distance and y is the seek time; the worst case rotational latency is T_{rot}; the disk data transfer rate is constant and is B_D; the cycle length is T_c; the cylinder span of the requests in a cycle is A; each block of size R is stored contiguously on a disk; SCAN scheduling is used in each cycle; and the disk bandwidth is the performance bottleneck in an SS. If the track and cylinder switch time is not modelled, then:

$$a * J + b * A + T_{rot} * J + \frac{R}{B_D} * J = T_c \tag{4.3}$$

$$J = \frac{T_c - b * A}{a + T_{rot} + \frac{R}{B_D}} \tag{4.4}$$

where J is the maximum number of block requests which can be served in a cycle.

Assume $T_c = 1$ second, and the average stream rate is 0.5 MB/s and $R = 0.5$ MB. With the ST12550N [**Seagate93**] disk parameters[1]: $a = 6.6$ ms, $b * A \leq 13.9$ ms, $T_{rot} = 8.3$

[1]Section 8.2.3 will show how the disk seek time profile constants are obtained for the ST12550N drive.

ms, and assuming a constant disk transfer rate of an average $B_D = 4.65$ MB/s (although the ST12550N is a ZBR disk), then $J = 8.05$. If a 5-disk RAID3 is built from ST12550Ns and used in an SS node, then $B_D = 18.6$ MB/s and $J = 23.60$. J is still larger with lower stream rates, for example, MPEG-1 videos have an average rate of about 0.25 MB/s.

If $J > d$, a condition not difficult to meet in practice, then there is a chance that primary blocks retrieved in a cycle have evenly distributed backup copies for multi-chained declustering. In addition, the start stride of a newly recorded video can be selected randomly from 1 to d to reduce the possibility of stride in-phase between the playbacks of different videos. Also because a video is large and it will be striped across the set of n SSs multiple times, with each stripe trying to use a different stride, the chance of stride in-phase between different playbacks of the same video may also decrease. Therefore, load balancing for multi-chained declustering during fault mode could be achieved in practice after taking the above precautions.

Now assume that using multi-chained declustering, the load in an SS can be evenly scattered to the subsequent d SSs during fault mode in any cycle, and that the maximum number of streams that can be served from an SS are X_m and X_i for multi-chained and interleaved declustering respectively when resources have been reserved for fault tolerance. In the multi-chained case, an SS has to serve X_m primary streams in a cycle in addition to reserving disk bandwidth for X_m/d streams in case one of its d predecessors fails, i.e., when an SS fails, X_m/d streams are off-loaded to each of its d successors. Therefore, a total $X_m + X_m/d$ requests with block size R need to be served in a cycle by an SS during fault mode, that is:

$$a * \frac{d+1}{d} * X_m + b * A + T_{rot} * \frac{d+1}{d} * X_m + \frac{R}{B_D} * \frac{d+1}{d} * X_m = T_c \tag{4.5}$$

$$X_m = \frac{d}{d+1} * \frac{T_c - b * A}{a + T_{rot} + \frac{R}{B_D}} = \frac{d}{d+1} * J \tag{4.6}$$

In the interleaved case, an SS has to serve X_i primary requests with block size R. In case one of its d predecessors fails, it has to reserve bandwidth for an additional X_i requests but with size R/d. Therefore:

$$a * 2 * X_i + b * A + T_{rot} * 2 * X_i + \frac{R}{B_D} * X_i + \frac{R}{d * B_D} * X_i = T_c \tag{4.7}$$

$$\begin{aligned} X_i &= \frac{T_c - b * A}{2 * a + 2 * T_{rot} + \frac{d+1}{d} * \frac{R}{B_D}} \\ &= \frac{a + T_{rot} + \frac{R}{B_D}}{2 * a + 2 * T_{rot} + \frac{d+1}{d} * \frac{R}{B_D}} * J \qquad (4.8) \\ &= \frac{a + T_{rot} + \frac{R}{B_D}}{\frac{2d}{d+1} * (a + T_{rot}) + \frac{R}{B_D}} * X_m \qquad (4.9) \end{aligned}$$

$$\frac{a + T_{rot} + \frac{R}{B_D}}{2 * (a + T_{rot}) + \frac{R}{B_D}} * X_m \leq X_i \leq X_m \tag{4.10}$$

When $d = 1$, $X_i = X_m = J/2$, i.e., 50% of the resources need to be reserved. This reduces to mirrored or chained declustering. If $d > 1$, then $X_i < X_m$. Based on the same assumed parameters as above, it is possible to compute the number of primary streams that

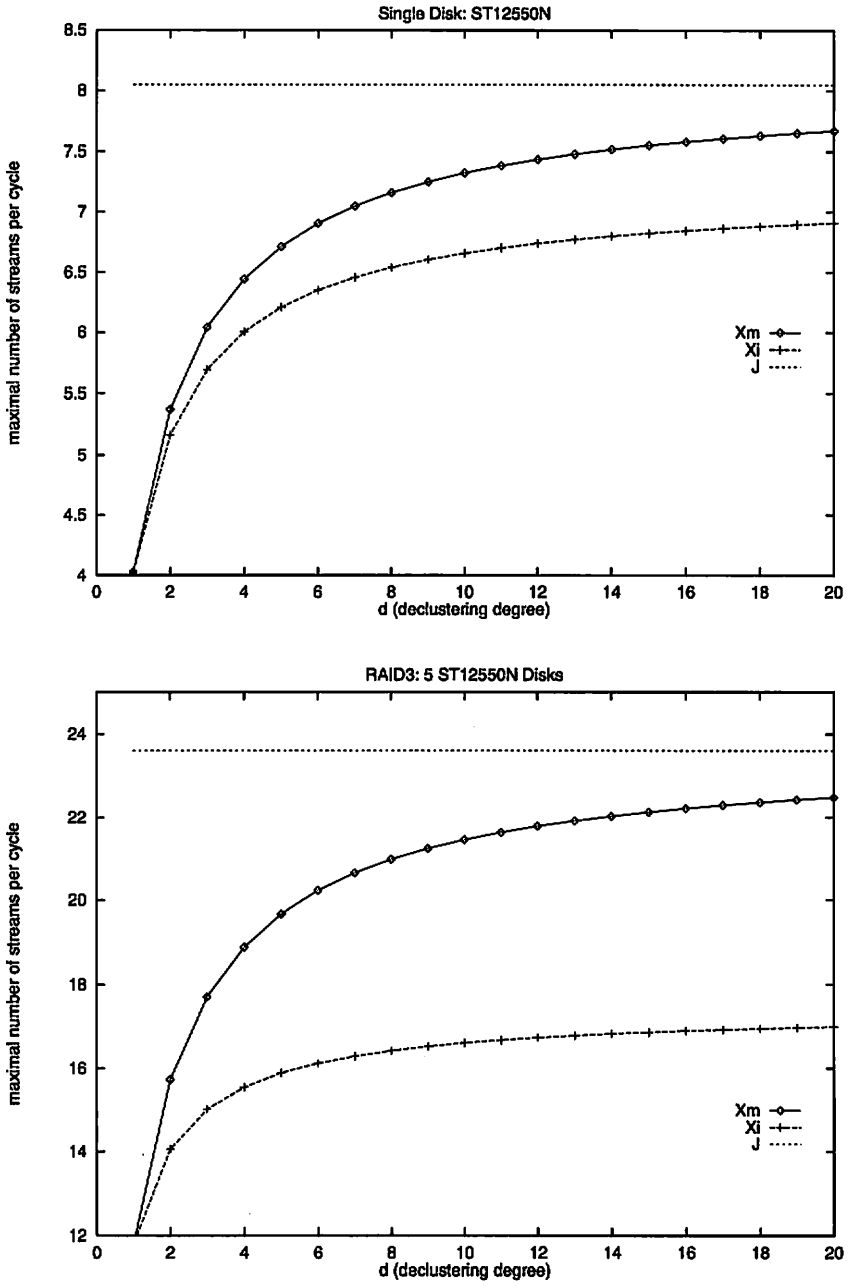

Figure 4.4: Maximum Number of Streams per Cycle vs. Declustering Degree

can be served by an SS, which uses a single disk or a RAID3 as the storage device, as a function of the declustering degree d while reserving resources for fault mode. The results are shown in Figure 4.4.

Interleaved declustering has several disadvantages. If streams of multiple rates are served, then different videos would have different block sizes, for example, lower rate videos could have smaller SUS. If backup copies of low rate videos are declustered widely, the rotational and seek overhead would increase because of small access. Streams with different rates may need different declustering degrees to counter the small access effect, e.g., lower rate videos could have smaller declustering degrees. It should also be noted that during fault mode, interleaved declustering would potentially send data from more SSs for one stream in a cycle than during non-fault mode. All these issues will increase the implementation and management complexity and overhead.

When there are multiple primary block requests per cycle and the blocks' backup copies are evenly distributed to other SSs, multi-chained declustering can serve more streams per cycle than interleaved declustering using the same declustering degree. Besides, multi-chained declustering could use the same declustering degree for all videos, because data is replicated on a block basis instead of in block fragments. Also during fault mode, data is retrieved from at most the same number of SSs for a stream in a cycle as during non-fault mode, because the backup of a lost block is stored in another single SS. Therefore, multi-chained declustering is simpler to implement and manage than the interleaved alternative, and is chosen as the redundancy method for Cadmus.

4.4.2 Fault Mode Resource Reservation

Multi-chained declustering can tolerate any single SS failure. If all videos have the same declustering degree, d, then the failure of any two SSs inside $d+1$ consecutive SSs would break the system. Assume the set of failed SSs is $\mathcal{F}$, then the system can tolerate failure if $\forall i,j$, $(j-i \bmod n > d)$ and $(i-j \bmod n > d)$, and cannot tolerate failure if $\exists i,j$, $(j-i \bmod n \leq d)$ or $(i-j \bmod n \leq d)$, where $i \neq j$, SS(i) $\in \mathcal{F}$, and SS(j) $\in \mathcal{F}$. When d increases, the reliability of the system will drop. A reasonable range of d would be 4–9, and by Equation (4.5), $d/(d+1) = 80\%$–90% of the SS resources can be utilised in non-fault mode compared to a scheme without fault tolerance.

When an SS needs to reserve resources for fault mode in a cycle, it only needs to reserve the *maximum* among d alternatives, with each corresponding to the resource requirements of backup block retrieval in the SS in case one of its previous d SSs fails, and *not* the *aggregate* of the d alternatives. Otherwise 50% of the resources still need to be reserved in each cycle. This can be seen from Table 4.2. Assume at cycle 0, all primary blocks are retrieved by each SS. Then from the perspective of SS(4), it will only need to reserve resources to retrieve B3 *or* B8 *or* B13 *or* B18, not B3 *and* B8 *and* B13 *and* B18 in the same cycle in case any, but not more than one of its 4 predecessors fails.

4.4.3 Replication Management

Cadmus embeds the redundancy method into each LO. Although the native scheme in Cadmus is multi-chained declustering, there is nothing to prevent the basic Cadmus architecture from using other redundancy methods and building proper CPs to interpret the corresponding schemes. For multi-chained declustering, Cadmus records the striping group size, n, and the declustering degree, d, as part of the LO_ATTR. A replicated LO has double entries with different LOIDs, corresponding to the primary and backup LOs respectively in the LOID table of an FS. However, only primary LOIDs are used in the `textual_name:LOID` mappings in DSs, which are unaware of the redundancy scheme used by an LO. In this case, an FS will create two LOs for an LO creation operation, but data consistency of the two copies is maintained by an RT-CP for RT recording and by the write operation of an NRT-LO.

4.5 Other Issues

The SUS of an LO representing a video depends on the video's average rate and rate variability. For a CBR video, the SUS is just the displayed data size per cycle, so that only one striping unit is retrieved from one SS in each cycle. For a VBR video, the SUS is

slightly above the average displayed data size per cycle, so that at most one striping unit is retrieved in most of the cycles after some smoothing algorithm. The problem of determining SUS for VBR videos is examined in more detail in the next chapter.

For VCR functionalities, a client command is issued to an SCA, and if necessary, the SCA will compute a new read set and new read tables for the SSs to perform another AC before the VCR operation is executed. Although this will introduce latency or denial to VCR commands, it is the best way to deterministically prevent QoS degradation to other streams. VCR control will be further described in Section 6.5.

Cadmus does not aim to provide functionalities for all aspects of a real world VOD server. Instead, it strives to provide basic mechanisms upon which a scalable VOD server can be built by adding extra logical components. For example, reconstruction of lost data or reconfiguration of the system when extra SSs are added can be implemented using additional modules exploiting the basic functionalities provided by Cadmus.

Cadmus shares the same design philosophy as MSSA [**Lo94**]: divide-and-conquer, and builds complex abstractions on top of the more basic ones. SSs and POs in Cadmus are roughly comparable to byte segment custodes and byte segments in MSSA. In addition, naming and security issues are addressed more throughly in MSSA, and the solutions can be readily applied to Cadmus as well. Therefore, Cadmus does not consider such issues in order to avoid duplication of work. However, Cadmus is distinguished from MSSA by its network striped approach, redundancy scheme, AC, QoS enforcement method, and VBR considerations.

4.6 Summary

This chapter identified the disadvantages of some existing network striped storage servers when applied to a video server environment. Then it presented Cadmus, a scalable, flexible, and QoS aware architecture for network striped video servers. Data objects in Cadmus are structured into physical and logical types, and are managed by separate logical components. Meta-data for different components is managed in a way which causes little interference between them, so that many of the components can exist independently. VBR videos are supported in Cadmus deterministically by the coordination of various components. Because of its load balancing ability and simplicity, multi-chained declustering is selected as the native redundancy scheme for Cadmus. However, this chapter only gave an outline of the architectural framework. Some of the design issues will be clarified in later chapters.

Chapter 5

Variable Bit Rate Smoothing

This chapter proposes and analyses an algorithm that can be used to smooth the amount of data read in each cycle without starving or overflowing client buffers for VBR videos. Heuristics are introduced to select SUS, and examples are given with some interesting observations. Finally, related work is examined for comparative evaluation.

5.1 Motivation

This chapter is concerned with how to compute the read set for a single video under the deterministic cycle based QoS enforcement scheme. At first sight, it appears to be a trivial problem since it is obvious that one can retrieve the amount of data that is actually needed by the client in a cycle, i.e., the displayed set, which is readily available from the video index. However, a close examination on real VBR traces suggests otherwise. The following discussion uses the MPEG-1 true VBR video traces from [**Trace95**] [**Rose95**] and selects the "MTV" data set as an example. This is because MTV has a higher peak rate and a higher peak/mean ratio than most other video traces. Figure 5.1 shows the frame size trace and Figure 5.2 the data needed in one second interval for a span of 40,000 frames. Table 5.1 shows the statistics of the displayed set with one second cycle length. The whole trace is 120,137 KB and lasts 1600 seconds.

Table 5.1: Displayed Data per Second Statistics for MTV

average (mean) size (KB)	75.1
maximal (peak) size (KB)	300.2
minimal size (KB)	11.0
maximal sum of two consecutive seconds' data size (KB)	586.2
peak/mean ratio	4.0

It can be observed that there are many cycles which only need a small amount of data, for example, well below the average. If the read set is equivalent to the displayed set, then the small requests will lower disk utilisation, as each one would incur a seek and rotational latency without reading much data. This problem is exacerbated when a small scale RAID3 is used as the storage device, since a RAID3 can linearly increase the transfer rate from the disks but without any improvement in reducing rotational or seek overhead. It would

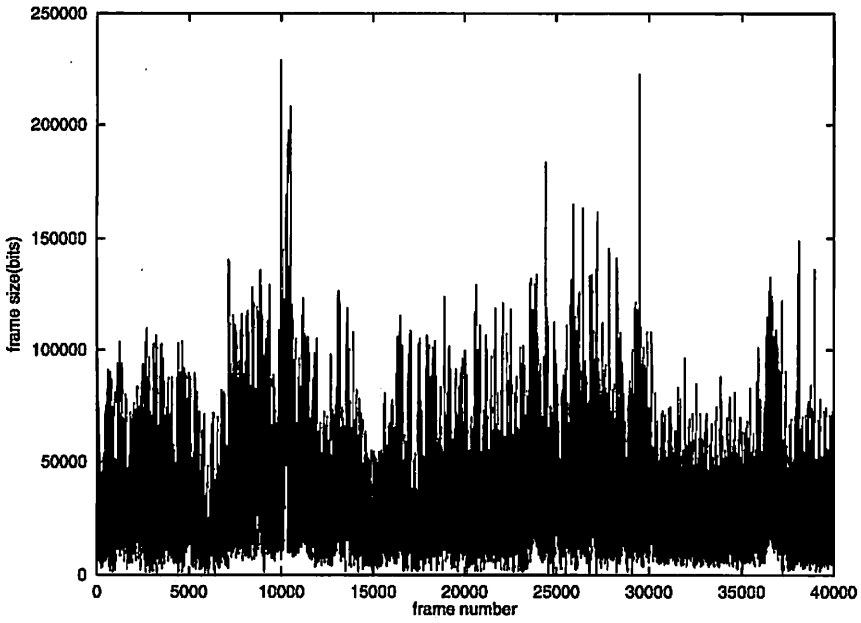

Figure 5.1: Frame Size Trace for MTV

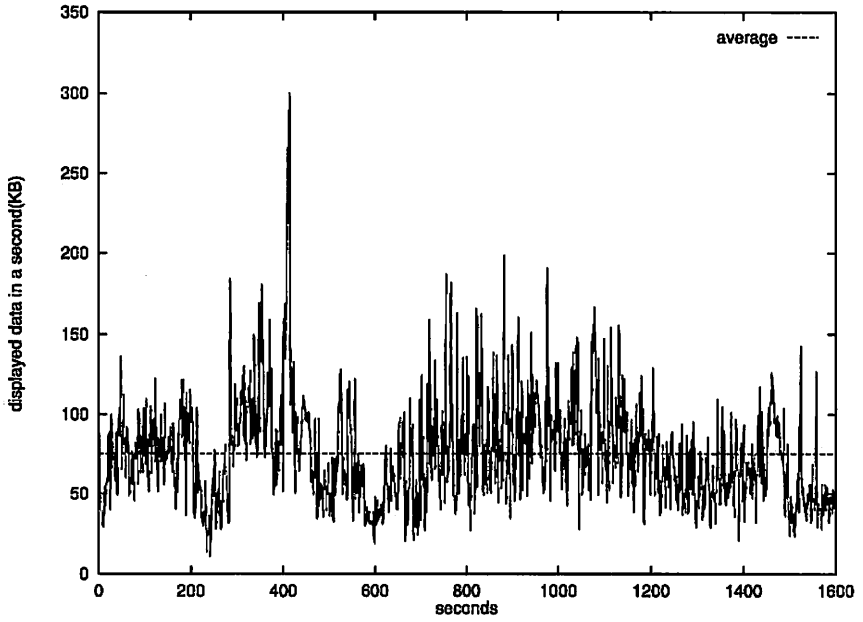

Figure 5.2: Displayed Data per Second Trace for MTV

be beneficial to assemble small reads into larger ones by reading more data in one cycle, and not to read in subsequent cycles if appropriate, where the freed bandwidth can be used by other streams or by NRT operations. This strategy is in fact *read-ahead* in some cycles and *suppressing* in others. During read-ahead cycles, it would also be desirable for an actual read size to always approach a level that is not very small, which can be seen as a *smoothing* procedure to reduce the burstiness of non-suppressed reads. Moreover, if data read in each cycle is always above a level such as the smoothing level, *elimination of small reads* is achieved.

Another concern is *client buffer limits*. If a client has limited buffering, the server read-ahead and suppressing should neither starve nor overflow the client. It would be desirable

to pre-compute a read schedule just based on the video index, instead of doing dynamic flow control at run-time, which would increase the complexity of the server system. Such a schedule can be directly fed into a simple AC algorithm which only needs to sum the aggregate resource requirements of all streams in each cycle and to make sure no overload occurs for a successful admission. If AC succeeds, the server would just abide by the pre-computed schedule without worrying about client buffer utilisation, i.e., implicit flow control is achieved.

Some interesting questions based on the above considerations are: How large is the minimally required client buffer for a stream to be played back continuously? How can small requests be assembled into larger ones while still reserving the continuity requirement yet without starving or overflowing a client? What are the criteria for read-ahead or suppressing in a cycle? Is it possible to find a value so that the request size would always be equal to or larger than it during a read-ahead cycle? If such a value can be found, what other parameters does the largest such value depend on? This chapter will answer all these questions.

5.2 Formulating the Problem

5.2.1 Terminology

For the benefit of formulating the problem discussed above, some symbols are introduced in Table 5.2. T_c is a system parameter, the selection of which is not the topic of this research. However, the rest of the chapter will use a value of 1 second as an example. The smoothing factor (SF), the smoothing scale (r), and the look-ahead steps (h) will be further explained in Section 5.3.

Table 5.2: Symbols Used for VBR Smoothing

symbol	explanation	property
T_c	time length of a cycle	e.g., 1 second
c	cycle index	
T_v	time length of the video	$T_v > 0$
e	total number of cycles of the video	$e = \lceil T_v/T_c \rceil$
P	peak displayed data size per cycle	$P > 0$
V	maximal sum of two consecutive cycles' data size	$0 < P \leq V \leq 2 * P$
M	mean displayed data size per cycle	$0 < M < P$
k	peak/mean ratio of the displayed data size per cycle	$k = P/M > 1$
$D(c)$	displayed data size in cycle c	$\forall c \in [1, e] \quad 0 \leq D(c) \leq P$
$R(c)$	actual data read in cycle c for the video	
SF	smoothing factor	$SF \geq 0$
r	smoothing scale relative to mean	$r = SF/M$ or $SF = r * M$
h	look-ahead steps	$h \geq 1$
CB	total client buffer size for the video	$CB > 0$
$B(c)$	client data balance in cycle c	
$U(c)$	upper limit of the data which can be read in cycle c	
$L(c)$	lower limit of the data which should be read in cycle c	

As a client has limited buffer space for each video, the actual server read size in each cycle ($R(c)$) should not exceed the available client free buffer space ($U(c)$), nor should it be less than the amount of data which should be read ($L(c)$). Otherwise, the client would be overflowed or starved in the next cycle. Because of read-ahead, some unused data may be

left over ($B(c)$) in a client when a cycle ends. The displayed set of a video can be further defined as the set of its $D(c)$, and the read set as the set of its $R(c)$.

5.2.2 Client Buffer Constraints

The buffer usage at a client in consecutive cycles is shown in Figure 5.3. It should be noted that there is delay between the server and client cycles. If both the server and a client use the dual-buffer scheme, then when the server starts a cycle at time t by reading data into the disk buffer, the data will be sent to the client in the cycle starting from $t+T_c$. The client will receive the data in the comparative cycle starting at time $t+T_c+\Delta$, where Δ is the network delay from the server to the client, and the data can be displayed in the next client cycle starting from time $t+T_c+\Delta+T_c$. Delay jitter can be absorbed in a client by extra buffering, which is out of the scope of this research. For simplicity and generality, the following will assume that the server and client cycles can be synchronised.

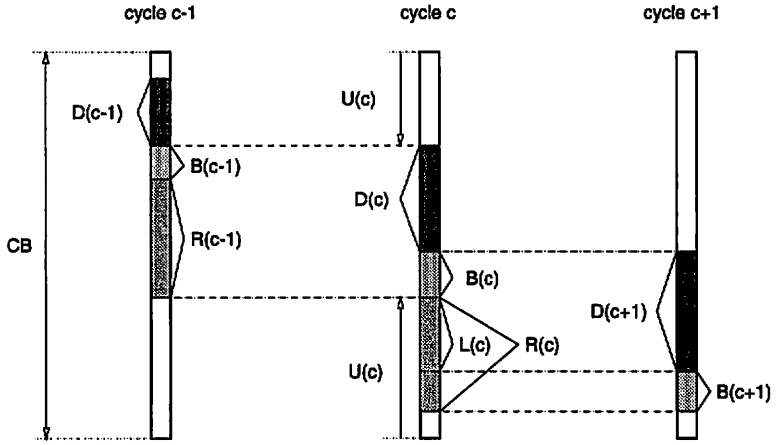

Figure 5.3: Client Buffer Usage

Assume at the end of cycle $c-1$, after a client has displayed data of size $D(c-1)$, the data balance is $B(c-1)$, and the data received by the client (i.e., the data read from the server) in cycle $c-1$ is $R(c-1)$. Then at the start of cycle c, the useful data in the client buffer is $R(c-1)+B(c-1)$. From this, $D(c)$ will be displayed in cycle c, and the rest is the balance for cycle c, i.e., $B(c)$. Therefore:

$$B(c) = R(c-1) + B(c-1) - D(c) \tag{5.1}$$

Apart from the useful data $R(c-1)+B(c-1)$, the free buffer space in the client for cycle c is:

$$U(c) = CB - R(c-1) - B(c-1) = CB - D(c) - B(c) \tag{5.2}$$

However, because in cycle $c+1$ the client will have to display data of size $D(c+1)$, at least data size:

$$L(c) = D(c+1) - B(c) \tag{5.3}$$

should be read in cycle c to prevent client starvation in cycle $c+1$. Therefore, in cycle c, the server read $R(c)$ should fall between $U(c)$ and $L(c)$. That is:

$$L(c) \leq R(c) \leq U(c) \tag{5.4}$$

It should be noted that $L(c)$ may be negative. Because of probable server read-ahead, the balance in one cycle may be more than the data needed for display in the next cycle.

5.3 A Smoothing Algorithm

Two simple methods to compute read sets are either just to read the data needed in the next cycle, i.e., let $R(c) = D(c+1)$, or always to read ahead to fill the client buffer, i.e., let $R(c) = U(c)$. The first approach has the drawback stated above, i.e., many small reads occur (cf. Figure 5.2). For the second simple method, once the client buffer has been filled, the free buffer in each cycle ($U(c)$) is the data just displayed ($D(c)$) in the same cycle, which reduces to $R(c) = D(c)$. This is in fact a phase shift version of the first simple method, and it also has the same small read problem. Neither of these schemes addresses smoothing effects.

To avoid the drawbacks of the above simple retrieval methods, a generic read set computing algorithm is proposed and shown in Figure 5.4. The basic ideas behind the algorithm are *suppressing* and *smoothing*. When the client balance ($B(c)$) in a cycle is more than the data to be displayed in the next h consecutive cycles (where $h \geq 1$), the read in the server can be suppressed (i.e., let $R(c) = 0$) and the continuity requirement will not be broken. This is called *h-step look-ahead suppressing*. In non-suppressed cycles, the algorithm tries to make the amount of the data read ($R(c)$) approach a specific value, i.e., the *smoothing factor (SF)*, if the smoothing factor is within the range of the lower limit ($L(c)$) and the upper limit ($U(c)$). As it is more intuitive to compare SF to the mean (M), SF is set to be $r * M$, where r is called the *smoothing scale* and is a non-negative real number.

---

input: $D(c)$ $c \in [1, e]$; r, h, M, CB;
output: $R(c)$ $c \in [0, e-1]$;

$D(0) = D(-1) = R(-1) = B(-1) = 0$; $SF = r * M$;

for ($c = 0$; $c \leq e-1$; $c++$) {
  $B(c) = R(c-1) + B(c-1) - D(c)$;
  $U(c) = CB - D(c) - B(c)$;
  $L(c) = D(c+1) - B(c)$;
  if $L(c) > U(c)$ { algorithm failed; client needs more buffer; exit; }

ⓐ **else if** $B(c) > \sum_{i=1}^{h} D(c+i)$ $R(c) = 0$; /* h-step look-ahead suppressor */
ⓑ **else if** $0 \leq SF < L(c) \leq U(c)$ $R(c) = L(c)$; /* to prevent client starvation */
ⓒ **else if** $L(c) \leq SF \leq U(c)$ $R(c) = SF$; /* smoother */
ⓓ **else if** $L(c) \leq U(c) < SF$ $R(c) = U(c)$; /* to prevent client buffer overflow */
}

Figure 5.4: The Generic Read Set Computing Algorithm

The algorithm is generic in that if $SF = 0$, then it reduces to the first simple method above. When there is no suppressing and SF is made infinitely large, then it reduces to the second simple method. An interesting alternative is to fill the client buffer while suppressing. However, as the result[1] on the MTV trace shows, this will make the read set very bursty (Figure 5.5). As one cycle has to serve multiple streams at run-time, one video should not hog too many resources, i.e., the data read in a cycle for a stream should not be very large. Another alternative is smoothing without suppressing. However, the result on the MTV trace shows that it cannot eliminate the small read problem (Figure 5.6). The following sections will always assume suppressing and smoothing at the same time.

[1]In all the examples using the MTV trace in this chapter, if not stated otherwise, $CB = V = 586.2$ KB.

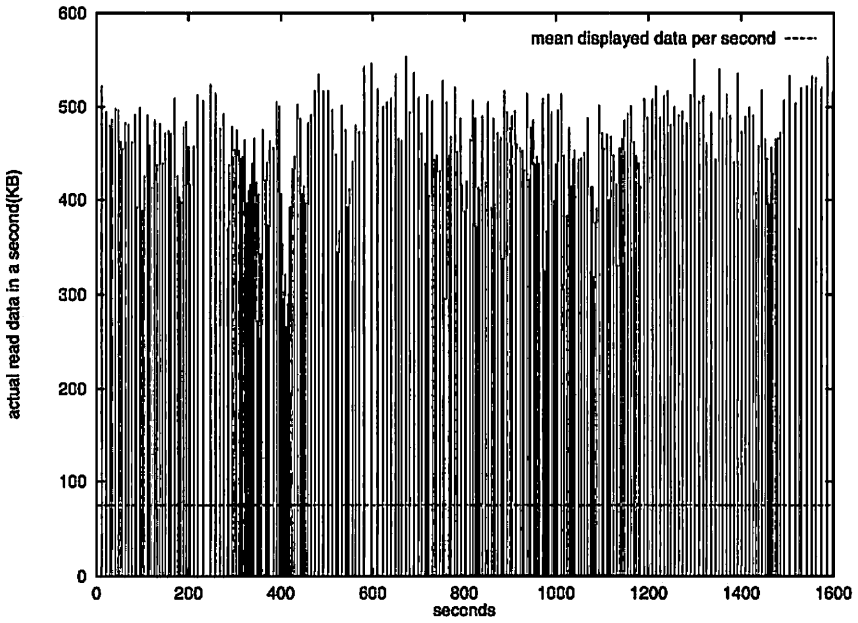

Figure 5.5: Suppressing and Filling Client Buffer ($h = 1$)

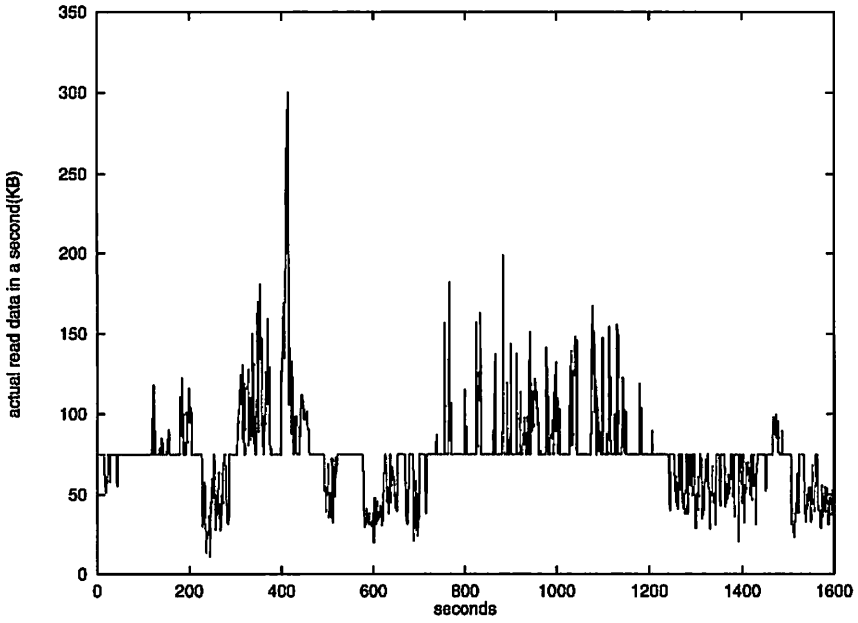

Figure 5.6: Smoothing without Suppressing ($r = 1$)

The read set computed from the above algorithm with smoothing to average rate ($r = 1$) using 3-step look-ahead suppressing ($h = 3$) for the MTV trace can be found in Figure 5.7, and Figure 5.8 shows the details of a specific area. Also Table 5.3 presents the read percentage for the displayed set and the read set, where S represents the size of a read. The effects of smoothing and suppressing are clear and convincing: the algorithm has smoothed most reads to the average; it also reduces the percentage of the reads that are larger than the average and happens to eliminate the small reads below the average; and there are spare cycles that can be used by other streams or NRT requests.

An important question is whether or not the algorithm can eliminate small reads for proper smoothing factor (SF) and look-ahead steps (h), though it happened to achieve this for $SF = M$ and $h = 3$ in the above example. This transforms to the simple question of

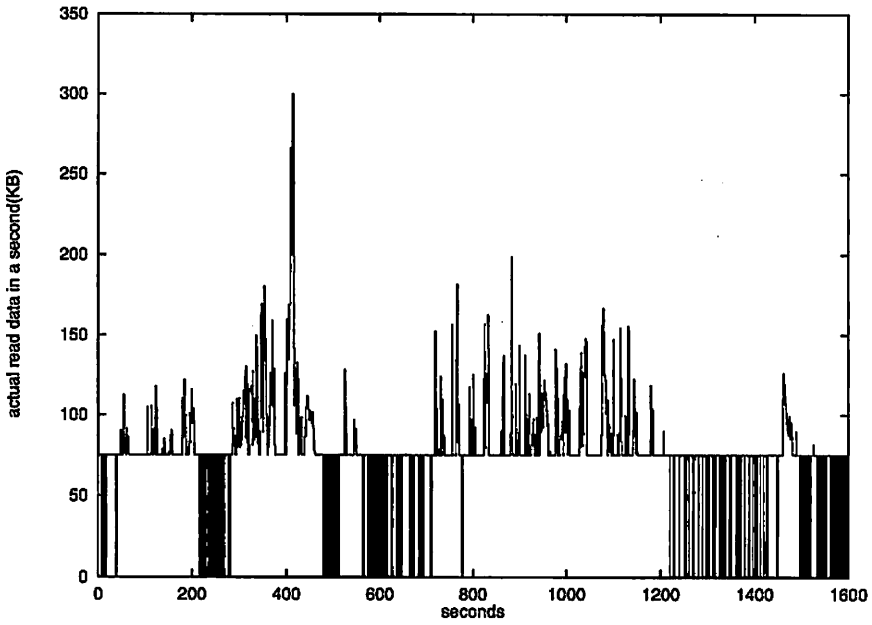

Figure 5.7: Smoothing and Suppressing ($r = 1, h = 3$)

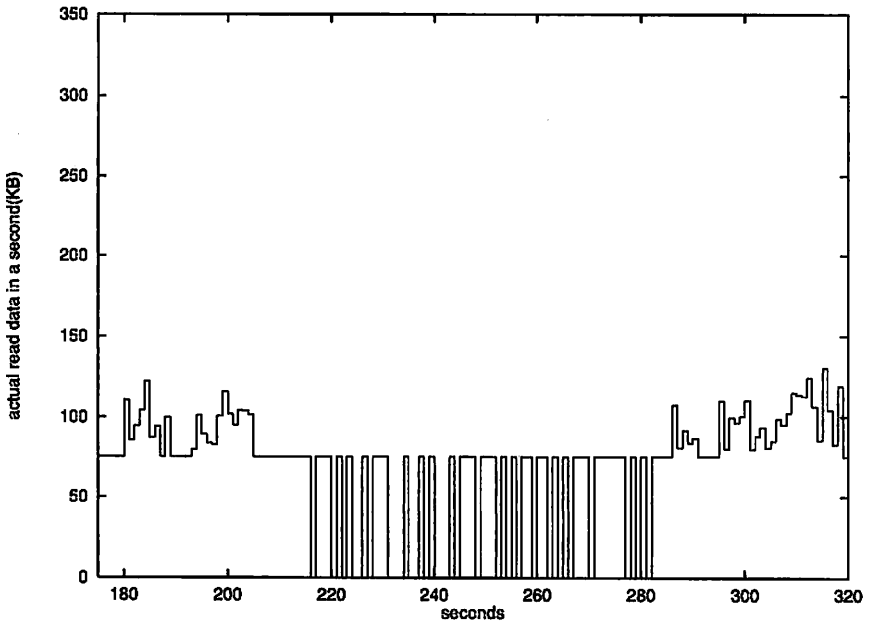

Figure 5.8: Smoothing and Suppressing: Details

whether it is possible to find a maximally allowable SF, such that the data read in each cycle ($R(c)$) is either zero or at least SF under client buffer constraints, i.e., branch @ of the algorithm never gets executed. The analysis of the algorithm in the next section will answer the question.

Table 5.3: Read Percentage of the Displayed and Read Sets

sets	$S > M$ (%)	$S = M$ (%)	$0 < S < M$ (%)	$S = 0$ (%)
displayed set	45.63	0.00	54.37	0.00
read set	26.40	62.77	0.00	10.83

5.4 Analysis and Verification

This section gives theorems and their corollaries which can be used to characterise the algorithm proposed in the last section. Lemmas used to prove the theorems are also included. The proofs of the lemmas, theorems, and corollaries are presented in Appendix A. This section also verifies the formal results using real traces.

5.4.1 Characteristics of the Algorithm

Theorem 1. *The algorithm succeeds if and only if* $\forall c \in [0, e-1] \quad CB \geq D(c)+D(c+1)$.

Corollary 1. *The algorithm will succeed if and only if* $CB \geq V$.

Corollary 2. *The algorithm will succeed if* $CB \geq 2 * P$.

If the algorithm fails, then there is no way that the server can satisfy the continuity requirement during the playback lifetime of a video. The following results will assume that the algorithm will always succeed, i.e., $\forall c \in [0, e-1] \quad CB \geq D(c) + D(c+1)$.

Theorem 2. $\forall c \in [0, e-1] \quad 0 \leq B(c) \leq CB,\ 0 \leq U(c) \leq CB,\ 0 \leq R(c) \leq U(c)$.

Lemma 1. $\forall c \in [0, e-1] \quad$ *if* $B(c) > \sum_{i=1}^{h} D(c+i)$, *then* $B(c) - \sum_{i=1}^{h} D(c+i) \leq SF$.

Lemma 2. *Given* $l \in [2, h]$, *if* $\forall c \in [0, e-1]$

$$\begin{cases} B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l+1) * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l) * SF & \text{if } B(c) \leq \sum_{i=1}^{h} D(c+i). \end{cases}$$

then $\forall c \in [0, e-1] \quad B(c) - \sum_{i=1}^{l-1} D(c+i) \leq (h-l+1) * SF$.

Lemma 3. $\forall l \in [1, h]$ *and* $\forall c \in [0, e-1]$

$$\begin{cases} B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l+1) * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l) * SF & \text{if } B(c) \leq \sum_{i=1}^{h} D(c+i). \end{cases}$$

Corollary 3. $\forall c \in [0, e-1]$

$$\begin{cases} B(c) - D(c+1) \leq h * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - D(c+1) \leq (h-1) * SF & \text{if } B(c) \leq \sum_{i=1}^{h} D(c+i). \end{cases}$$

Theorem 3. $\forall c \in [0, e-1] \quad B(c) \leq h * SF$.

Corollary 4. $\forall c \in [0, e-1] \quad 0 \leq B(c) \leq \min(CB, h * SF)$.

Theorem 4. *If* $CB \geq \max((h+2)/(h+1) * P,\ V)$ *and* $SF \leq CB/(h+2)$, *then* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \geq SF$.

It should be noted that the condition $CB \geq (h+2)/(h+1) * P$ in addition to $CB \geq V$ will not mandate large buffers in a client. This is because normally V is approaching $2 * P$, while $(h+2)/(h+1) * P \leq 3/2 * P < 2 * P$. It indicates that in most situations, it is possible to find a value selected from a range such that the amount of data read in each cycle is either zero or no less than the selected value. Then compared to the value selected, small reads are eliminated. The maximal such value is only determined by h and CB, whose minimal value in turn depends on h, P, and V, where P and V are very simple characteristics of a stored VBR video.

Corollary 5. *For* $X > 0$, *if* $CB \geq \max((h+2)/(h+1) * P,\ V,\ (h+2) * X)$, *then* $\exists SF \geq X$, *such that* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \geq SF$.

Corollary 6. *If* $CB = 2 * P$ *and* $SF \leq CB/(h+2) = 2/(h+2) * P$, *then* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \geq SF$.

Corollary 7. *If* $CB = 2 * P = 2 * k * M$ *and* $r \leq 2 * k/(h+2)$, *then* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \geq r * M$.

5.4.2 Verification of Formal Results

It should be noted that the condition in Theorem 4 is sufficient but not necessary. This is because its proof uses Corollary 4, and $\min(CB, h * SF)$ is not a tight upper bound of $B(c)$. However, to verify that they do not differ very much, maximal $B(c)$ has been computed by fixing h and varying r from 0.1 to 4.6 for the MTV trace. The results for h from 1 to 6 are shown in Figure 5.9. It can be seen that $\min(CB, h * SF)$ is a simple and good estimate as an upper bound for $B(c)$.

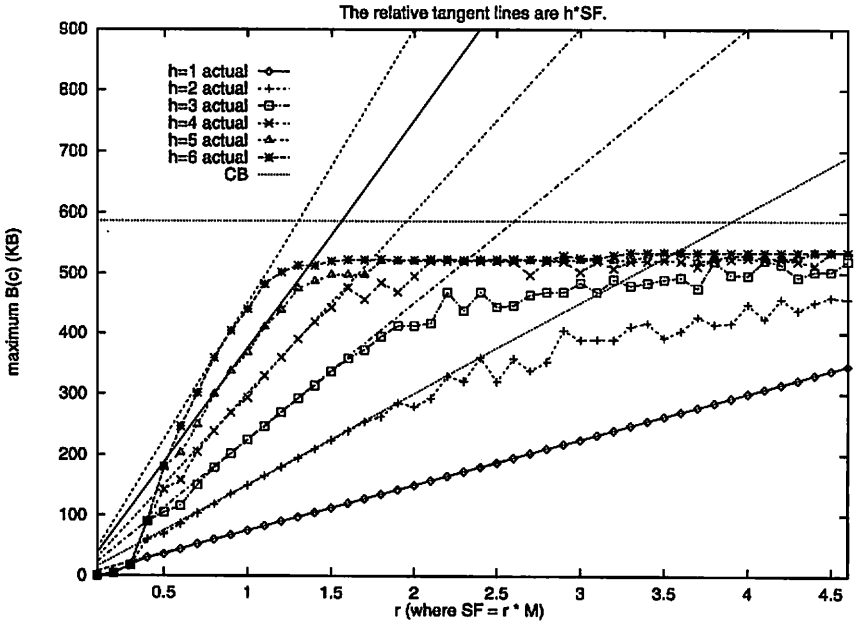

Figure 5.9: Maximal Balance Upper Bound

In the MTV trace, the peak/mean ratio is $k = 4.0$. Assume a client has a buffer size of $2 * P = 600.4$ KB. Then by Corollary 7, when $r \leq 2 * k/(h+2)$, the data read in each cycle is either 0 or above SF. Indeed this is the case when results for a large range of h are computed. Computation has also been done to determine how big r should be for the read set to have read sizes smaller than SF. For example, when $h = 3$, if $r \leq 2*k/(3+2) = 1.60$, then the element of the read set is either 0 or at least SF. It is only when r reaches 1.63 that reads with size smaller than SF start to appear. Similar results are observed for a large range of h starting from 1, which further indicates that the sufficient condition in the formal results is not very far from the necessary condition in reality.

5.5 Effects of Changing Parameters

As the algorithm and the formal results do not show the effects of varying r or h on the computed read set, this section compares different retrieval strategies based on different r or h using real VBR traces. For a given SF and the computed read set, a *large-read* is defined as a read with size larger than SF, a *smoothed-read* as a read equal to SF, a *small-read* as a read smaller than SF but not zero, and a *suppressed-read* as a read with size zero.

The percentage of the above four types of reads are used as parameters for comparison. A good (r, h) pair would result in small large-read and small-read percentages, but with large smoothed-read and suppressed-read percentages. The measures also include the mean and the standard deviation of all reads, and of those reads that are non-suppressed. Small standard deviations in both cases would make for better results. Comparisons have been performed on most of the video traces in [**Trace95**]. As the results are similar, only the MTV trace is shown as an example here. CB is set to $V = 586.2$ KB $\approx 2 * P$ in all cases.

5.5.1 Fixed Look-Ahead Steps, Varying Smoothing Scale

This subsection looks at the effects of fixing h while varying r from 0.1 to 4.6. The same analysis has been repeated for h from 1 to 6, but as all results have similar trends of the above parameters, only cases of $h = 2$ and $h = 5$ are shown in Figure 5.10 and Figure 5.11.

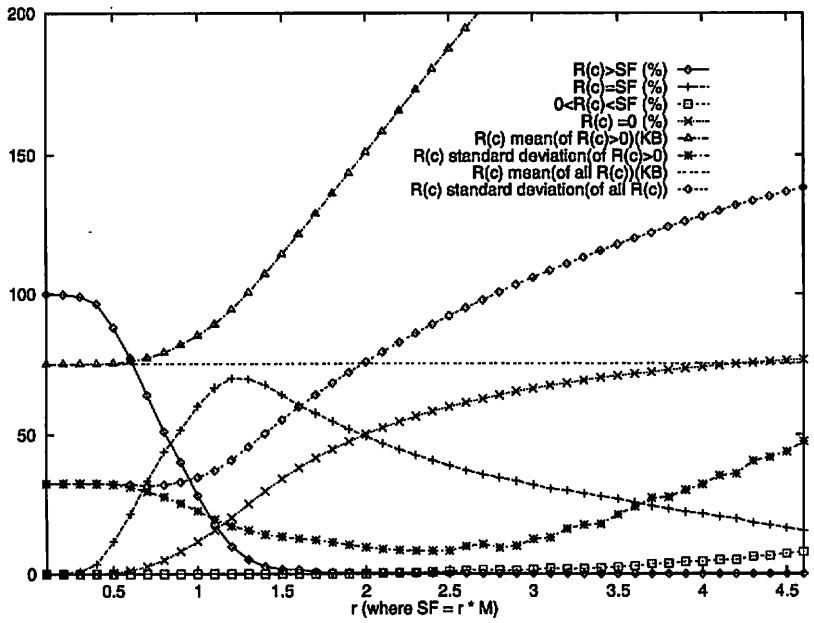

Figure 5.10: Read Set Statistics: $h = 2$, $r \in [0.1, 4.6]$

In both cases, when r increases, the percentage of suppressed-reads increases. This is because as the algorithm is trying to read more data each time into the client buffer, it can suppress reads more often in subsequent cycles. However, the large-read percentage always decreases. This has two causes: one is that suppressing occurs more often; the other is that more reads can be smoothed to SF, which is around the average (M). However, the smoothed-read percentage increases until around $r = 1.2$, and then will decrease. The reason for the percentage decrease of smoothed-reads is that more suppressing occurs first, then roughly when $r > 2 * k/(h + 2)$, which is $r > 2$ and $r > 1.14$ for $h = 2$ and $h = 5$ respectively, small-reads start to appear and to increase.

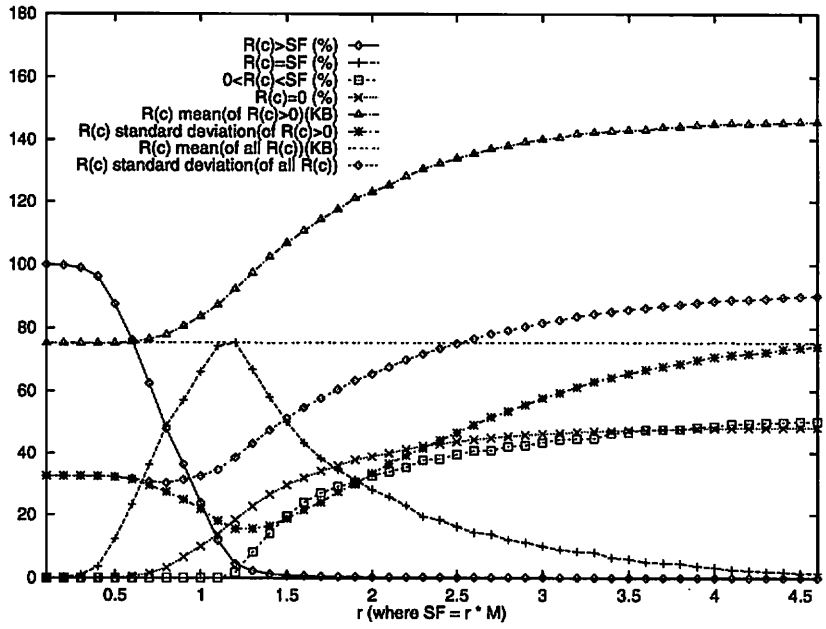

Figure 5.11: Read Set Statistics: $h = 5$, $r \in [0.1, 4.6]$

The standard deviation for all reads increases, because of more suppressing. However, the standard deviation of non-suppressed-reads drops first, because of smoothing, but it will increase later, because small-reads start to appear to counter the smoothing effect. The mean is constant for all reads and increases for non-suppressed-reads because of increased suppressing.

5.5.2 Fixed Smoothing Scale, Varying Look-Ahead Steps

This subsection fixes the value of r, and looks at the above parameters as h varies from 1 to 10. The same procedure has been repeated for values of different r. As the results have similar characteristics, the cases of $r = 1$ and $r = 1.5$ are shown in Figure 5.12 and Figure 5.13.

In both cases, when h increases, small-reads will appear and increase from $h \approx 2*k/r-2$, which is $h = 6$ and $h = 3$ for $r = 1.0$ and $r = 1.5$ respectively. The smoothed-read percentage will increase first, and then decrease when small-reads appear. The increase of the smoothed-read percentage is because when the algorithm looks ahead further, it tends to read in data more often to smooth the burstiness in later cycles in the displayed set. A side-effect of this is that the algorithm also tends to perform suppressing less often, which is why the suppressed-read percentage is always dropping. The large-read percentage always decreases because of better smoothing first and small-reads later.

When the algorithm looks ahead more, the standard deviation of all reads will drop, because of smoothing first, and less suppressing later. The mean of non-suppressed-reads drops because of less suppressing. However, the standard deviation of non-suppressed-reads will drop first because of smoothing, and will start to increase when small-reads occur.

5.5.3 Discussion and Recommendations

When h is fixed, although large r will cause a large suppressed-read percentage, the smoothed-read percentage tends to drop, and the result is more bursty, e.g., the mean of

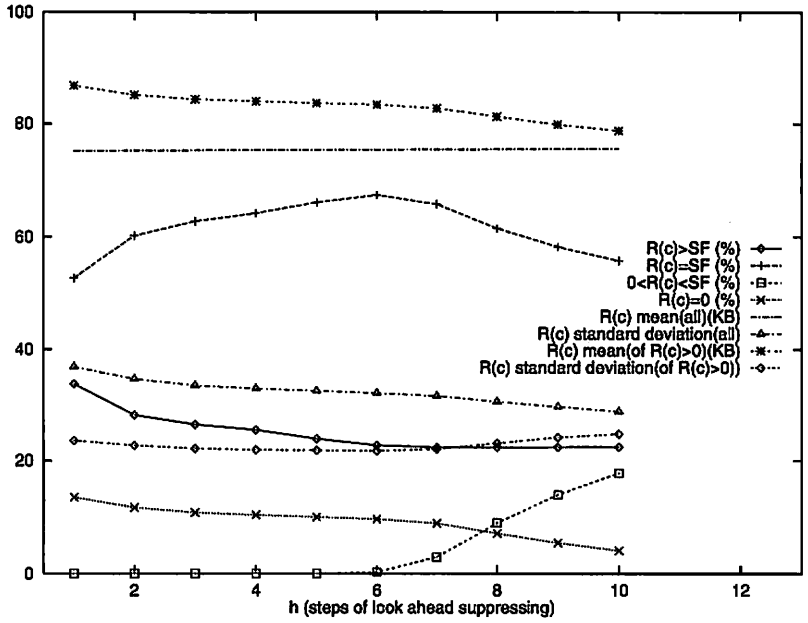

Figure 5.12: Read Set Statistics: $r = 1.0$, $h \in [1, 10]$

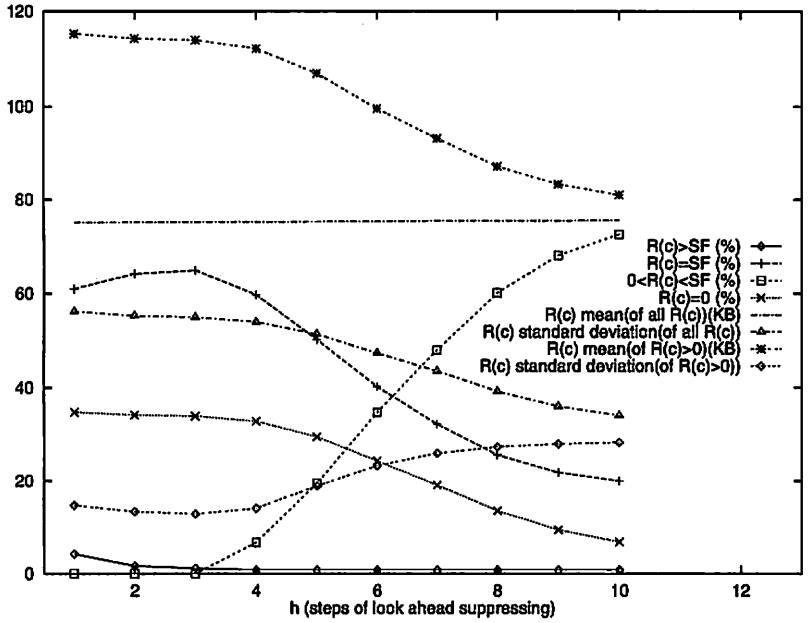

Figure 5.13: Read Set Statistics: $r = 1.5$, $h \in [1, 10]$

non-suppressed-reads and the standard deviations of both all reads and non-suppressed-reads increase when r is large. The situation is more manageable when $r \in [1, 2]$.

When r is fixed, the standard deviation of non-suppressed-reads has roughly the same turning point as the smoothed-read percentage, where the former is minimised and the latter is maximised. As the point, i.e., at $2 * k/r - 2$, is determined by r, r can be seen as a more important parameter than h. Once a reasonable r has been selected, the maximal $h = \lfloor 2 * k/r - 2 \rfloor$ can be computed for use in the algorithm.

If the aim is to maximise the smoothed-read percentage, a real trace can be analysed by increasing r when fixing h to a small value, for example, $h = 2$. Otherwise if h is large, then the range of r allowing each read to be at least $r * M$ is small, as $r \leq 2 * k/(h + 2)$.

Such an analysis has been performed on most of the traces in [**Trace95**], and most of them reach the maximal smoothed-read percentage when $r \approx 1.2$. On the other hand, a small large-read percentage would also be desirable. Because the large-read percentage will drop when r becomes large, larger r can be used for videos with higher peak/mean ratios.

As a rule of thumb, r can be recommended to be near 1.2 when k is not large (e.g., $k \in (2.4, 2.8]$), and near 1.4 when k is medium (e.g., $k \in (2.8, 4]$), and near 1.6 when k is large (e.g., $k \in (4, 6]$). When $k \in (1, 2.4]$, r can be set to either $2 * k/3$ or $k/2$ subject to the look-ahead steps desired ($h = 1$ or $h = 2$). If a small value of r is not desirable (e.g., when $k < 1.5$, $r < 1$ if $CB = 2 * P = 2 * k * M$), CB can always be increased to boost r, since $r * M \leq CB/(h + 2)$.

5.6 Examples and Observations

Though the above comparison and examples use unit of KB for the length of the displayed and read sets, it should be noted that the algorithm itself is independent of the length unit chosen, which could be KB, disk block, fixed-size system block, or adjustable striping unit. This section will give examples for the latter two, and make some interesting observations. Here again the MTV trace is used as the example.

5.6.1 Examples

Fixed-Size System Blocks

As have been shown, some scalable storage servers only provide access to fixed-size system wide blocks or fragments. As an example, assume that the size of the system block is 64 KB. First the displayed set with 64 KB block as the length unit is computed and the results are shown in Figure 5.14 with statistics in Table 5.4.

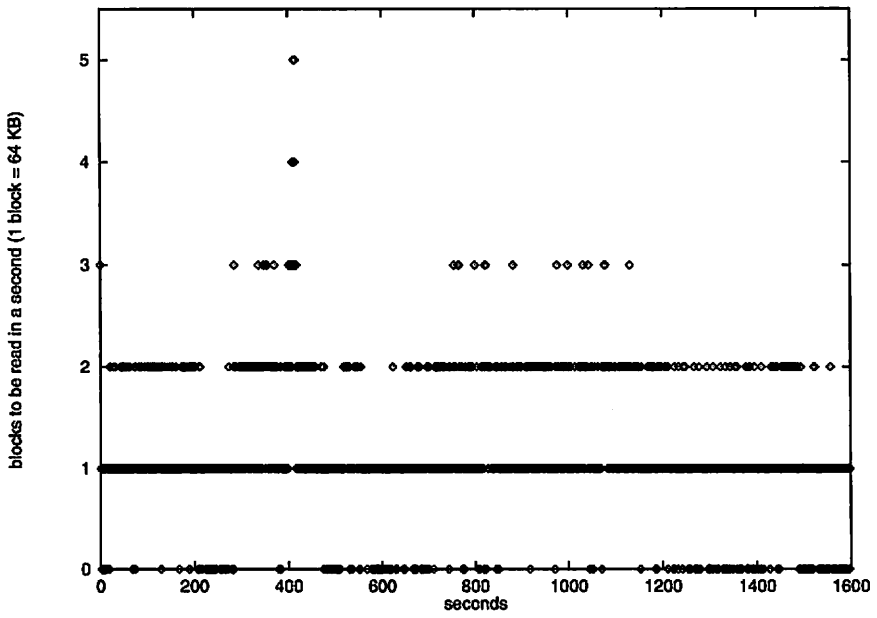

Figure 5.14: Displayed Set with 64 KB Blocks

Since $k = 4.26$ is large, r can be selected as 1.6. As $1.17 * 1.6 = 1.87$ blocks, SF is chosen as 2 blocks. In this case the adjusted $r = 2/1.17 = 1.7$, then $h \leq 2 * 4.26/1.7 - 2 = 3.01$.

Table 5.4: Displayed Data per Second with 64 KB Blocks

more than 2 blocks access (%)	2.13
2 blocks access (%)	23.69
1 block access (%)	63.12
0 blocks access (%)	11.06
average (mean) size (blocks)	1.17
maximal (peak) size (blocks)	5
minimal size (blocks)	0
maximal sum of two consecutive seconds' data size (blocks)	9
peak/mean ratio	4.26

So 3-step look-ahead suppressing is used. Using a client buffer with 10 blocks ($2 * P$), the algorithm is applied to the displayed set and the result can be seen in Figure 5.15 with an area detailed in Figure 5.16. It can be observed that each cycle either reads at least 2 blocks or does not read at all. The percentages of the read in the read set which is larger than 2 blocks, equal to 2 blocks, and zero are: 0.44%, 57.95%, and 41.61%. The mean of all reads is 1.17 blocks with a standard deviation of 1.00, while the mean of non-suppressed-reads is 2.02 blocks with a standard deviation of 0.18. It is interesting to see how small the large-read (more than 2 blocks) percentage is after smoothing and suppressing.

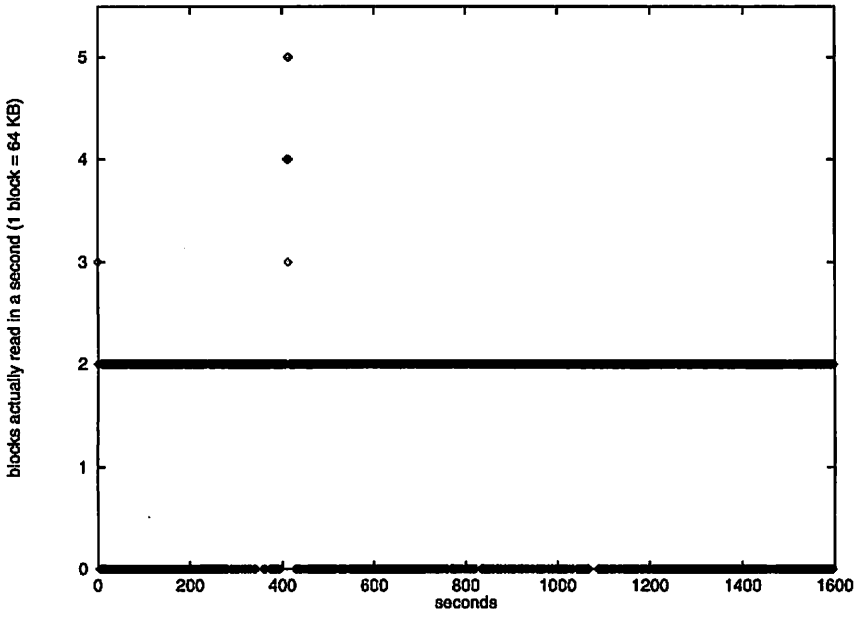

Figure 5.15: Read Set with 64 KB Blocks

Adjustable Striping Units

Cadmus supports striping units (or blocks) of different sizes for different videos. It is desirable to read at most 1 block in most cycles, i.e., to make the smoothing factor to be 1 block. Although the proposed algorithm is not oriented to compute a block based displayed set, compared to the algorithm in Figure 5.4, the block based displayed set computation actually performs one step look-ahead suppressing with SF set to be the block size, and reads to the nearest multiples of SF in branch ⓑ, without considering client buffer usage.

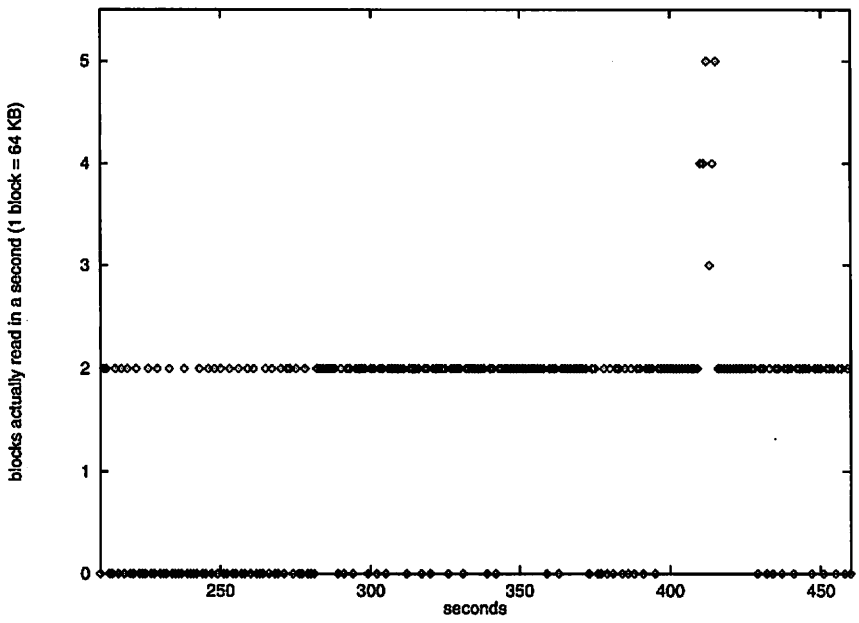

Figure 5.16: Read Set with 64 KB Blocks: Details

For each trace in [**Trace95**], the block based displayed set has been computed for different block sizes ($r * M$) by varying r, and it is found that the trends of the read percentages are similar to the fixed h, varying r analysis in the last section. For example, most traces have the maximal 1 block read percentage when the block size is $1.1 * M$. Therefore, the same rule of thumb can be used as heuristics to select SUS in order to achieve large percentage for 1 block access and small percentage for access with more than 1 block.

Since for the MTV trace $k = 4.0$, r is set to be 1.4, and $1.4 * M$ is chosen as the block size, which is 106 KB (note the cycle length is still 1 second, though on average one block can be displayed for 1.4 seconds). The block based displayed set is shown in Figure 5.17 with statistics in Table 5.5.

Table 5.5: Displayed Data per Second with 106 KB Blocks

more than 1 block access (%)	3.25
1 block access (%)	64.19
0 blocks access (%)	32.56
average (mean) size (blocks)	0.709
maximal (peak) size (blocks)	3
minimal size (blocks)	0
maximal sum of two consecutive seconds' data size (blocks)	6
peak/mean ratio	4.23

The proposed algorithm can be applied again on the block based displayed set with a smoothing factor of 1 block. As there are no small-reads now, only the read-ahead and smoothing functionalities of the algorithm will take effect, while considering the client buffer limit. Because for the block based display set, $r = 1/0.709 = 1.4$ and $2k/r - 2 = 4.04$, 4-step look-ahead suppressing is used with a client buffer of 6 blocks. The result is shown in Figure 5.18. In this case, the percentages for the read in the read set that is larger than 1 block, equal to 1 block and zero are: 0.63%, 69.63%, and 29.74% respectively, where the large-read percentage is further reduced.

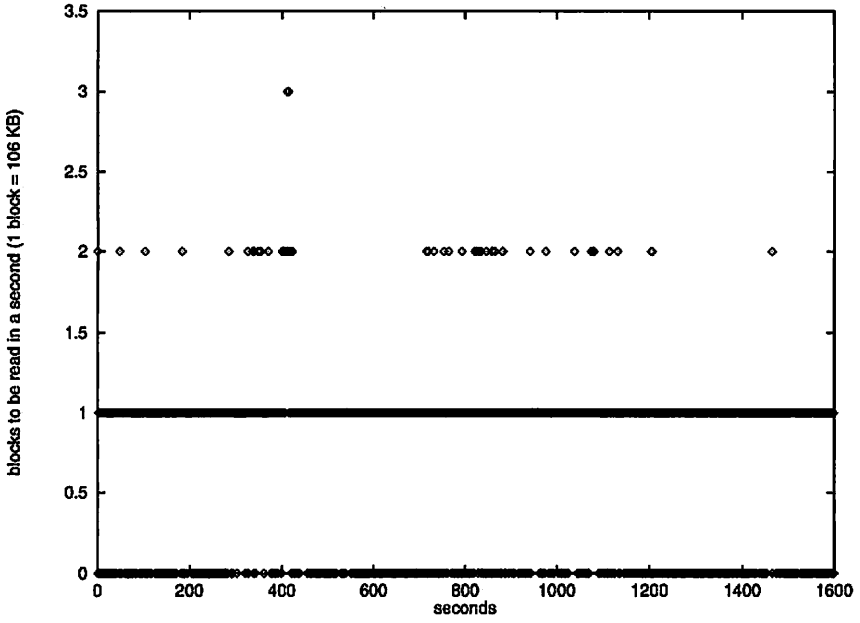

Figure 5.17: Displayed Set with 106 KB Blocks

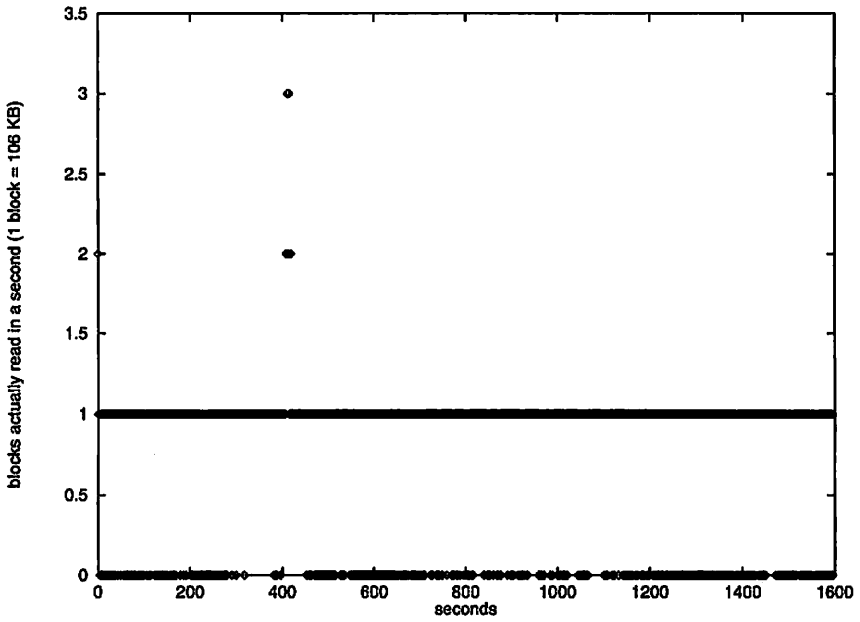

Figure 5.18: Read Set with 106 KB Blocks

5.6.2 Some Observations

One observation from the above examples is that large-reads still exist, though their percentage is small. If a server cannot temporarily store extra data, then it should not pre-fetch data in suppressed cycles in the hope that it can smooth out large-reads in the future. This is because the server then has to send pre-fetched data in advance, which would probably overflow a client. If a server could do *set-aside buffering* to pre-fetch data for the large-read cycles, then these buffers could be shared by multiple streams. This is because the large-reads of a stream tend to aggregate after smoothing and suppressing, and a server can try to avoid multiple large-reads of different streams occurring at the same time during AC. Another method to smooth large-reads is to take advantage of *multiplexing* and use

the suppressed cycles of other streams to accommodate the large-reads of a new stream. This is because the suppressed-reads of a stream tend to aggregate as well. In Figure 5.18, consecutive suppressed periods as long as 4 cycles (4 seconds) are observed. In practice, the above two techniques can be combined.

Another very interesting observation is that for the MTV trace, smoothing and suppressing make the large-read percentage very small. If those large-reads can be dealt with by a server using set-aside buffering or multiplexing, then most of the time the stream is of pseudo *constant block rate*, yet there are suppressed cycles as well. The constant rate needs not to be the peak rate at all. For example, though the peak size per cycle is 3 (106 KB) or 5 (64 KB) blocks in the above examples, the constant size is 1 (106 KB) or 2 (64 KB) blocks per cycle, which is just $1.4 * M$ or $1.7 * M$ compared to the peak size of $4 * M$. This suggests that some stored VBR videos may not be that bursty when an algorithm such as the one proposed here is applied and if there are no adverse effects to reading more than the average data size in a cycle.

All the video traces in [**Trace95**] have been analysed with r set to a value using the above rule of thumb. The adjustable block size is set to $r * M$ and the block based displayed set and read set's read percentages are compared. The results are shown in Table 5.6, which verify the above conjecture to some extent. D and R in the table represent the number of blocks read in a cycle for the displayed set and for the read set respectively. It should be noted that h is computed from the values of k and r of the displayed set. It is observed that any h that is larger than the computed one will not lower the large-read percentage for a read set.

Table 5.6: Read Percentage for All the Video Traces

title	M (KB)	k	r	block (KB)	$D > 1$ (%)	$D = 1$ (%)	$D = 0$ (%)	h	$R > 1$ (%)	$R = 1$ (%)	$R = 0$ (%)
asterix	68.2	3.59	1.4	96	3.62	63.81	32.56	2	1.38	68.42	30.21
atp	66.8	2.76	1.2	80	6.00	71.50	22.50	4	0.88	82.03	17.09
bond	74.2	3.01	1.4	104	2.31	66.69	31.00	4	0.44	70.77	28.87
dino	39.9	3.37	1.4	56	2.88	65.50	31.62	4	0.44	70.63	28.93
lambs	22.3	4.60	1.6	36	4.62	52.44	42.94	4	2.44	56.98	40.58
mrbean	53.8	4.01	1.6	86	3.12	56.19	40.69	4	1.00	60.74	38.26
mtv	75.1	4.00	1.4	106	3.25	64.19	32.56	4	0.63	69.63	29.74
news	63.1	3.61	1.4	88	4.44	62.70	32.86	4	0.95	69.93	29.12
race	93.8	3.54	1.4	132	2.19	66.69	31.13	4	0.19	70.95	28.87
simpsons	56.7	3.20	1.4	80	2.94	64.94	32.13	4	0.25	70.57	29.18
star2	28.4	3.92	1.4	40	6.19	58.13	35.69	4	2.63	65.50	31.87
talk	44.4	2.74	1.2	54	4.44	73.19	22.38	4	1.19	79.90	18.91
term	33.3	2.52	1.2	40	4.44	74.38	21.19	2	0.56	82.26	17.18

Another question presents itself: what is the smallest r that will make a smoothed and suppressed read set have no large-reads when the block size is $r * M$. Computation has been performed by incrementing r from 0.1 to 4.6 with step 0.1 to find such r for all the video traces, and the results are shown in Table 5.7. Unfortunately it seems there is no good r for all, and the results depend on the individual traces. But for most traces, large-reads can be eliminated without using peak rate retrieval (i.e., $r < k$). However, the relative r is large, for example, in most cases $r > k/2$. Therefore, eliminating large-reads by increasing r may not give much gain. Thus it is still better to use smaller r and use multiplexing or server set-aside buffering to absorb the small large-read percentage as was suggested above.

Table 5.7: Zero Large-Read Percentage for All the Video Traces

title	M (KB)	k	r	block (KB)	$D>1$ (%)	$D=1$ (%)	$D=0$ (%)	h	$R>1$ (%)	$R=1$ (%)	$R=0$ (%)
asterix	68.2	3.59	1.9	129	0.88	51.12	48.00	2	0.00	52.97	47.03
atp	66.8	2.76	2.0	133	0.31	49.63	50.06	2	0.00	50.34	49.66
bond	74.2	3.01	1.7	126	0.56	57.81	41.63	2	0.00	59.04	40.96
dino	39.9	3.37	2.1	83	0.31	47.50	52.19	2	0.00	48.22	51.78
lambs	22.3	4.60	3.2	71	0.25	30.94	68.81	2	0.00	31.52	68.48
mrbean	53.8	4.01	2.9	156	0.19	34.19	65.63	2	0.00	34.65	65.35
mtv	75.1	4.00	2.5	187	0.12	39.94	59.94	2	0.00	40.28	59.72
news	63.1	3.61	3.3	208	0.00	30.32	69.68	1	0.00	30.40	69.60
race	93.8	3.54	1.7	159	0.62	57.81	41.56	2	0.00	59.16	40.84
simpsons	56.7	3.20	2.0	113	0.31	49.56	50.13	2	0.00	50.28	40.72
star2	28.4	3.92	2.9	82	0.19	34.31	65.50	2	0.00	34.77	65.23
talk	44.4	2.74	2.8	124	0.00	35.81	64.19	1	0.00	35.88	64.13
term	33.3	2.52	1.4	46	2.00	68.38	29.63	2	0.00	72.48	27.52

5.7 Related Work

Most VBR video server designs using deterministic cycle based schemes [**Dey94**] [**Asai95**] [**Chang94a**] just convert the actually displayed data into blocks and read the relative number of blocks needed in a cycle. They do not consider the effects of read-ahead, probable smoothing, and client buffer limits.

In [**Neufeld96a**], the block based displayed set is used as a basis for AC, and read-ahead is performed during AC for a new stream subject to server buffer availability. However, it does not consider client buffer limitations, nor does it consider smoothing for a single stream. A later article [**Neufeld96b**] proposes a dynamic server credit based flow control mechanism taking into account client buffer availability at run-time. But, if a read set can be pre-computed based on client buffer utilisation, then there is no need to perform explicit flow control at all. On the other hand, if dynamic flow control is performed, then the statically computed read set may not be observed, and the AC criteria that were satisfied may be broken at run-time.

There is research work on designing smoothing algorithms from the perspective of network transmission of VBR videos. [**Feng95**] focuses on reducing the number of rate increases required during a stream's transmission. The approach is to divide the video into segments in which piecewise CBR transmission is possible and there are no rate increases between the sub-segments. However, substantial rate increases will occur during segment boundaries, although it can be amortised to some extent by pre-fetching. The algorithm's applicability to network striped video servers is questionable because much rate variability, although constrained, still exists, and large buffers are needed at client sides to reduce the number of rate increases to an acceptable level.

[**McManus96**] takes the approach of building up enough data in a client before an actual playback starts, so that the subsequent transmission rate from a server can be constant. Analysis is performed to determine the relationships between the client buffer size, the built-up data size, and the constant transfer rate. Piecewise CBR transmission for VBR videos is also considered, because it requires less client buffer as the built-up operations are scattered. This latter approach is further studied in [**McManus95**], whose focus is to identify a transmission schedule which minimises client buffer requirements for a specific number of CBR transmission segments for a video.

[**Salehi96**] presents an optimal smoothing algorithm for achieving the greatest possible reduction in rate variability when transmitting a stored video to a client with a given buffer size. The algorithm relies strictly on work-ahead, and introduces no delay into client playback. The result is a piecewise CBR transmission schedule with minimum variance and peak rate among all feasible schedules for the video. This is achieved by constructing a CBR transmission segment as long as possible, but changing the rate as early as possible when the transmission rate must be increased or decreased to ensure feasibility.

The fundamental difference between the work in this dissertation and the related research in smoothing for network transmission is the design philosophy. The related work tries to smooth a VBR video into multiple CBR segments with different rates. When they are applied to a deterministic cycle based network striped video server, the size of the data retrieved in a cycle will vary from segment to segment, which is not very helpful to the selection of SUS. Also data may be retrieved from multiple SSs in a cycle if the SUS is fixed for a video. In contrast, this research tries to maintain the size of transmission in a cycle to always be above a certain level, with probable suppression of the data retrieval. This has several benefits. First, it can eliminate small reads in the server. Second, it can be used to select SUS. Third, it can smooth the block based retrieval so that only one striping unit is retrieved in most of the cycles. Finally, it only needs reasonable client buffer space.

5.8 Summary

This chapter proposed and analysed an algorithm that could be used to compute the read set from the displayed set, while considering client buffer limitations, performing read-ahead and suppressing, smoothing the read set, and eliminating small reads. The analysis was verified using real VBR traces, and the effects of varying the parameters of the algorithm were identified. The results can be used to choose the SUS of a VBR video, and the criteria only depend on simple video characteristics. The applicability of the algorithm was shown by giving examples of retrieving fixed or adjustable striping units. An interesting observation is that some VBR videos may not be that bursty for block based retrieval if these videos are read slightly above their average rates. Finally, related work in smoothing was evaluated to show the strength of this research.

Chapter 6

Admission Control

This chapter describes the AC aspects of Cadmus, especially the admission criteria in SS entities. The activities and fundamental resources in an SS node are classified. Then the disk service time is estimated with ZBR considerations. AC criteria for both RT and NRT activities are given based on their usage of SS resources, including shared ones. Finally, the Cadmus view on supporting VCR functionalities with AC considerations is presented.

6.1 Admission Control Consideration

6.1.1 The Admission Control Sequence

The distributed nature of Cadmus suggests a two phase commit protocol for AC when there are RT playback or recording requests. For a VBR playback request, an SCA will retrieve the video index and compute the read set. From the read set, a read table is computed for each SS which the video file is striped onto and the read tables are sent to the corresponding SSs. An SS will perform AC based on the read table, the extent list, zone information, and a *cycle table*, which records resource usage in current and future cycles. Each SS will return to the SCA a *start set* of cycles that can be used to start the new stream. The SCA will examine the start sets and find the earliest cycle that can be used by all the SSs. If such a cycle is found, then a commit command is sent to the SSs to actually reserve resources and start data retrieval. Otherwise, an abort command is sent to the SSs and the AC fails.

The AC sequence for RT recording or CBR playback is similar but much simpler because there is no video index or read set involved. For CBR recording or playback, an SS can compute receive tables or read tables respectively just based on LO attributes such as the striping group, the SUS, the redundancy method, and the data rate. For VBR recording, the peak data rates should be known and are used as the basis for AC and resource reservation in conjunction with other LO attributes. The subject of RT recording is further discussed in Section 6.3.3.

6.1.2 Contracts

AC and QoS enforcement are two distinct concepts, but between them there are invisible *contracts*. For an SS, the set of contracts is the resource management policies in the physical

entity where the SS resides. AC is to test whether resource requirements of a request can be satisfied in the future. Without detailed knowledge of the resource scheduling methods, AC would be inaccurate and lead to either resource under-utilisation or QoS degradation. On the other hand, QoS enforcement should faithfully implement the scheduling policies assumed by the AC procedures, otherwise resource waste or service breakage would still occur. The contracts for SSs will be incrementally introduced in this chapter when presenting the AC criteria, while contract enforcement in the Cadmus prototype implementation will be described in the next chapter. In the following, *C*-**i** refers to contract number *i*. A summary of the contracts will be listed in Appendix B.

6.1.3 Resources and Activities

A resource is a commodity necessary to get work done. There are generally two types of resources in a computer system [**Finkel88**] [**Levin75**]: *fundamental resources* and *virtual resources*. Fundamental resources, such as memory, CPU cycles, bus bandwidth, and I/O capability, are related to the basic concepts of space, time, and transport; while virtual resources, such as files and inter-machine or inter-process communication channels, are introduced by the software layer above the bare hardware. To guarantee QoS for RT streams, AC and QoS enforcement are needed because the capacity of any such resource is limited.

This work only concentrates on fundamental resources because the influences of virtual resources can be made negligible in CM applications by minimising single virtual resource sharing among different streams [**Needham92**]. It also does not try to abstract resources, as done in [**Zhao87b**] [**Zhao87a**], because the fundamental resources in an SS entity with a contemporary architecture are not numerous: CPU, memory, bus(es), disk(s), and network interface (Figure 6.1). The primary task of an SS, from the perspective of utilising the fundamental resources, is to move CM data between the disk(s) and the memory, and between the memory and the network interface through the bus(es). This work assumes data cannot be transferred directly between disk(s) and a network interface.

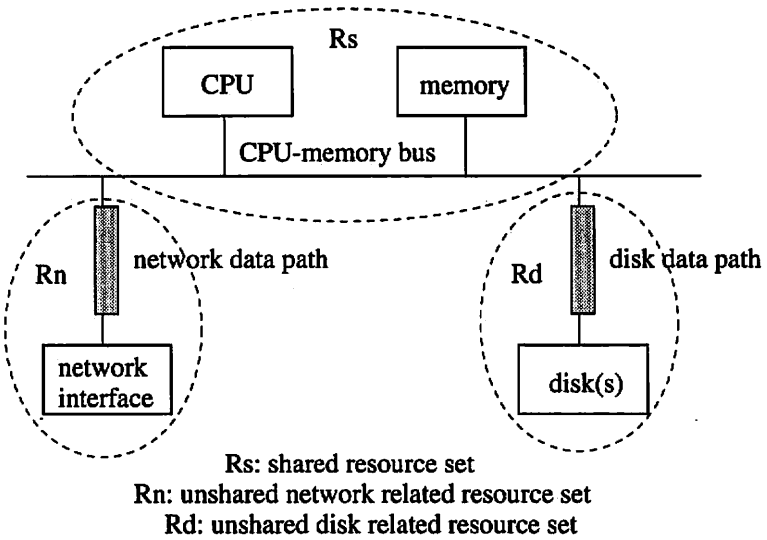

Figure 6.1: Fundamental Resources in an SS Entity

The activities in an SS entity can be classified as *disk read/write*, *network send/receive*, and *pure CPU processing*. Each category can be further divided into *RT* and *NRT* activities, with RT ones related to CM stream playback or recording, the rest being NRT. The resource

usage of the activities is highly dependent on the entity structure. For example, the devices, either the disk(s) or the network interface, may or may not have DMA capability, which will affect whether the CPU is used for data transfer. Further, without DMA, data transferred between a device and the memory will pass through the CPU-memory bus twice if a RISC CPU is used. Thus it is very difficult to give an accurate set of AC criteria without assuming a specific SS entity architecture.

This work will assume there is no DMA support in the devices and does not consider the CPU-memory bus resource for the following reasons: First, AC for programmed I/O is more difficult than the DMA based case. This is because with programmed I/O, the CPU becomes a shared resource for both disk and network activities; while using DMA, the CPU is out of the way of data movement, and the AC on different resources is more independent. Also, an entity using programmed I/O is cheaper than a DMA based one. Second, the CPU-memory bus is ignored because it is also a shared resource and the considerations of the CPU under programmed I/O apply to it as well. In addition, the bus is normally not a bottleneck in most contemporary machine architectures. Although pure CPU processing will use the bus for memory access, corresponding bandwidth could be reserved for this if the bus resource were to be considered. Finally, the above assumptions are true for an SS entity used in the prototype implementation to be described in the next chapter.

This chapter will therefore focus on RT and NRT disk, network, and pure CPU processing activities, as well as their AC on the disk, network, CPU, and memory resources in an SS entity.

6.1.4 General Admission Control Criteria

Network, Disk, and CPU

Receive activities are by far the most difficult to predict, and receive is important because an SS has to obtain information from SCAs, record RT streams, and accept data for video updating. Also receive activities in an interrupt driven system should not lead to the live-lock state **[Ramakrishnan93]** when there are malicious clients, which will make an SS appear to be temporarily lost. Cadmus addresses these problems by reserving a fixed amount of resources for NRT receive (***C*-1**) and by regulating receive activities in each cycle (***C*-2**). It also makes the receive interrupt run at a higher priority than the send and disk I/O (***C*-3**) to reduce the probability of losing data. All these contracts are assumed by the AC criteria below.

Assume if there are no other activities, the time needed to receive the reserved amount of NRT data and move it to the memory is $T_{NRTrecv\_net}$, while the time required on the CPU for moving the data is $T_{NRTrecv\_CPU}$. Because $T_{NRTrecv\_net}$ should be estimated using the component with the lowest capacity along the data path from the network interface to the memory, including the CPU, to reflect the bottleneck effect, $T_{NRTrecv\_CPU} \leq T_{NRTrecv\_net}$. Then the AC criterion for the network resource in cycle c is:

$$[c].send\_net + [c].RTrecv\_net \leq T_c - T_{res\_CPU} - T_{NRTrecv\_net} \tag{6.1}$$

where $[c].send\_net$ and $[c].RTrecv\_net$ represent the aggregate time required by send and RT receive activities in cycle c respectively, and they are estimated in the same way as $T_{NRTrecv\_net}$ is. T_{res_CPU} is the reserved CPU time for activities such as cycle boundary processing, and it in effect shortens the cycle length and is deducted from T_c in all AC criteria.

Because receive activities have a higher priority and are unpredictable, their CPU occupation time should be factored into the disk service cycle even though there is disk seeking time which will not use the CPU. This is because potential receive activities can defer disk transfer at any time in a cycle, as both activities need the CPU to proceed. The AC criterion for the disk resource in cycle c is:

$$[c].disk \leq T_c - T_{res\_CPU} - T_{NRTrecv\_CPU} - [c].RTrecv\_CPU \tag{6.2}$$

where $[c].disk$ and $[c].RTrecv\_CPU$ represent the aggregate disk service time for disk activities and the CPU time for RT receive in cycle c. $[c].disk$ is composed of the disk seek time, $[c].disk\_seek$, and the data transfer time, $[c].disk\_xfer$, which is also estimated using the component with the lowest capacity along the data path between the disk platter and the memory, including the CPU.

The above criteria assume that network send and disk activities are known at the start of a cycle (**C-4**), so that a simple First-Come-First-Served (FCFS) scheduling will suffice for these activities (**C-5**). For the AC criterion on the CPU resource, the CPU time requirements from device related activities should be included because the CPU is a shared resource under programmed I/O. The CPU AC criterion conforming to the above two is:

$$[c].disk\_CPU + [c].send\_CPU + [c].RTrecv\_CPU \leq T_c - T_{res\_CPU} - T_{pure\_CPU} - T_{NRTrecv\_CPU} \tag{6.3}$$

where $[c].disk\_CPU$ and $[c].send\_CPU$ refer to the aggregate CPU time needed by disk data transfer and send activities in cycle c, while T_{pure_CPU} represents the aggregate RT and NRT pure CPU processing time needs to be reserved in a cycle. Note that:

$$[c].RTrecv\_CPU \leq [c].RTrecv\_net \tag{6.4}$$

$$[c].send\_CPU \leq [c].send\_net \tag{6.5}$$

$$[c].disk\_CPU \leq [c].disk\_xfer \tag{6.6}$$

Assumptions and Conditions

The above three criteria are only valid under certain assumptions. Criterion (6.1) assumes that no other activities interfere with network related ones, while Criterion (6.2) assumes that no other activities, apart from network receive, interfere with disk related ones. Both assumptions may not be true because both types of device activities require some shared resources such as the CPU or the CPU-memory bus to proceed. Criterion (6.3) further assumes that a disk transfer or a network send activity known at the start of a cycle should be able to start immediately when there is available CPU resource for it. This may not be true because a disk data transfer cannot start until a potential seek is accomplished. All these assumptions bear a common theme: activity dependency is eliminated if they are true.

As was suggested in Section 2.6.2, RT scheduling problems with precedence and resource constraints are generally NP-complete. However, the simple and specialised functionalities of an SS do not warrant a more complex solution. A natural approach is to find the conditions when these assumptions can be held true, so that a simple FCFS combined with priority based scheduling can achieve the effects of QoS enforcement.

As receive activities run at the highest priority and have been factored into all three criteria, they will not interfere with any other activities because they are also regulated.

Pure CPU processing can be made to run at a lower priority than other activities (*C*-**6**), and RT CPU processing can run at a higher priority than the NRT one (*C*-**7**). Then pure CPU processing will not interfere with device related activities in any cycle. Thus the only interference should be considered is the one between network send and disk activities.

For simplicity of analysis, assume there are only disk and network send activities in a cycle. A disk activity has three time parameters: $disk\_seek$, $disk\_xfer$, and $disk\_CPU$, with $disk\_CPU \leq disk\_xfer$; while a send activity has time requirements: $send\_net$ and $send\_CPU$, with $send\_CPU \leq send\_net$. An important observation is that disk seeks can happen parallel to network and CPU activities, and that a disk transfer cannot start until a disk seek has finished. If any disk transfer is delayed by network send, then the seek of a subsequent disk activity will be delayed as well, which will further delay its data transfer. A better solution is to make disk transfer have a higher priority than network send (*C*-**8**), which has several benefits. First, disk activities are not interfered with by network send. Second, disk utilisation is increased because there is no more delay to disk seeks. Finally, network send can grab the CPU while disks are doing seeking.

However, disk activities will still interfere with network send. A simple example would be a 4 time unit cycle with one disk and one network send activity, with $disk\_seek = 1, disk\_xfer = 3, disk\_CPU = 2, send\_net = 4$, and $send\_CPU = 2$. Although both activities will be admitted by all three AC criteria, the network send will not be finished inside the cycle. This is because the disk transfer will delay the later part of the network send. Nevertheless, a closer examination shows that this interference can be eliminated in two situations.

First, if for any network send activity p and disk activity q, $p.send\_CPU = p.send\_net$ and $q.disk\_CPU = q.disk\_xfer$, then the delay to network send will not pose any problem because Criterion (6.3) guarantees there is always time left over for it. One example for this is where the CPU is the bottleneck on both disk and network data paths (cf. Figure 6.1). Second, if the shared CPU has an equivalent bandwidth not less than the sum of the I/O requirements of disk transfer and network send, then both types of activities can be multiplexed on the CPU without delay to each other. These two situations can be further generalised as is described below.

In Figure 6.1, assume the set of non-shared resources from the disk to the memory is Rd, the set of non-shared resources from the network interface to the memory is Rn, and the set of shared resources such as the CPU-memory bus, CPU, and memory is Rs. If $Bmin(Rn)$ and $Bmin(Rd)$ represent the bandwidth of the slowest resources in Rn and Rd, and Br_n and Br_d the bandwidth of resource r on network and disk data transfer respectively, where $r \in Rs$, then the above two situations are equivalent to:

$$\exists r \in Rs, \quad p.send\_r = p.send\_net \text{ and } q.disk\_r = q.disk\_xfer \tag{6.7}$$

$$\forall r \in Rs, \quad \frac{Bmin(Rn)}{Br\_n} + \frac{Bmin(Rd)}{Br\_d} \leq 1 \tag{6.8}$$

where p and q represent any network send and disk activity respectively, and $send\_r$ and $disk\_r$ the time needed for the data transfer on resource r by the corresponding activity. Also note that: $p.send\_r \leq p.send\_net$ and $q.disk\_r \leq q.disk\_xfer$.

The three AC criteria only work when either of the above conditions is true and when the activity priorities described above are enforced. Condition (6.7) can be interpreted in two ways: either there is a shared resource which is indeed the bottleneck; or $p.send\_r$ and $q.disk\_r$ are raised purposefully to $p.send\_net$ and $q.disk\_xfer$ to make the condition true. Condition (6.8) will hold in most current commercial machines because of the fast CPU and the high speed bus and memory. As the two conditions can greatly simplify AC and QoS

enforcement in an SS entity and are not difficult to satisfy in practice, it is doubtful that exploring the more complex problem space outside these conditions would be beneficial. This work will assume that one or the other of the conditions is true (*C*-**9**).

Five levels of priority for different activities, i.e., network receive, disk transfer, network send, RT CPU processing, and NRT CPU processing, have been introduced in an SS entity. However, there will be no priority inversion problem because each type of activity can be preempted by tasks of higher priorities over shared fundamental resources. This is in contrast to the using of shared virtual resources, where data integrity should be maintained, and a task already occupying a shared virtual resource may not be preempted by another task requiring the same resource, which is the necessary condition for priority inversion to occur.

Memory

The dual-buffer scheme, with one buffer for RT disk activities and another for RT network activities to achieve pipelining, is well established and is used in the SSs of Cadmus as well. Indeed, for special cases such as all streams are of CBR and disks are assumed to have constant data transfer rates, buffers can be reused on a per stream basis. However, for VBR streams on ZBR disks, buffer space is better reused on a per cycle basis. This is because although how much data to read in a cycle is known in advance, the amount to read for different streams and in different cycles varies. Memory allocation and deallocation would incur run-time costs when reusing buffers before a cycle ends, which would also introduce activity dependency and further complicate resource scheduling in an SS entity.

In contrast, cycle based buffer reuse makes buffer management simple and efficient. Two buffers each with size K for RT playback can be allocated when an SS starts, and such a buffer is called a *K-buffer* in Cadmus (*C*-**10**). Whenever a cycle c starts, buffers are allocated for all playback requests from the start of an empty K-buffer sequentially for disk activities to read data into. When cycle $c+1$ starts, the same K-buffer is passed to network activities for sending the data to clients. At cycle $c+2$, this same buffer can be passed back to disk activities as an empty one, and there is no need to perform any deallocation. For recording, because peak rate reservation is used, a dual-buffer is allocated and fixed for each stream (*C*-**11**). If the aggregate buffer requirements of all RT playback activities in cycle c is $[c].buf$, then the AC criterion for the memory is:

$$[c].buf \leq K \tag{6.9}$$

It was intended to use a three-buffer scheme in Cadmus to absorb disk anomalies. However, the scheme was finally dropped. This is because the three-buffer scheme would use read-ahead and write-behind which are at odds with deterministic QoS guarantees. Also 50% of the resources have to be reserved in the AC procedures because it is impossible to know in advance exactly which cycle's data is being retrieved at run-time as disk anomalies are unpredictable.

6.2 Disk Service Time Estimation

6.2.1 Discussion

While service time for other activities can be linearly deduced from measured system parameters, disk service time is more difficult to estimate because it is not proportional to

the amount of data transferred. There are three origins for this non-linear effect. First, disk devices continue to become smarter. They have their own CPUs, buffer caches, scheduling mechanisms, and disk anomaly processing methods. It would be desirable to make a disk device dumber (*C*-**12**), e.g., to ask it not to spend much time recovering from an error or perform unpredictable caching, so that an estimation of its service time could be more accurate.

Second, disk anomalies will produce unpredictable disk service time and will potentially overload a cycle. To guarantee QoS in subsequent cycles and to maintain system wide cycle synchronisation, cycle overload should be dealt with while preserving the semantics of both RT and NRT activities (*C*-**13**).

Finally, disk head and track switch time as well as ZBR will contribute to transfer rate variability from a disk. For example, the ST12550N has 24 zones, with the outer-most zone's transfer rate 5.82 MBps and the innermost zone 3.48 MBps. This gap is quadrupled for a 5-disk RAID3. If the worst case (innermost) transfer rate is used, then much disk bandwidth would be wasted. To better utilise the disk resource, the switch time and ZBR effects should be considered in disk service time estimation, which will be described in the next subsection. Considerations of ZBR in AC and overload processing in the case of disk anomalies are unique features of Cadmus.

6.2.2 Service Time Estimation

Assume in cycle c, there are $J(c)$ disk read/write requests, each with data size $R(j,c)$, $j \in [1, J(c)]$. Using the extent list of a PO (*C*-**14**), it is possible to find the number of extents on which a request $R(j,c)$ resides. Let the number be $X(j,c)$. Assume $R(j,c)$ has data of size $P(j,c,x)$ in each extent, where $x \in [1, X(j,c)]$. Such a contiguous segment $P(j,c,x)$ is called a *piece* of $R(j,c)$, and $\sum_{x=1}^{X(j,c)} P(j,c,x) = R(j,c)$. As one piece is stored discontinuously from another on a disk, each piece will incur a seek and rotational latency, and there are altogether $\sum_{j=1}^{J(c)} X(j,c)$ such pieces in a cycle.

In practice, if large extents are used, $R(j,c)$ is much smaller than an extent, and $X(j,c)$ is 1 most of the time and 2 occasionally when $R(j,c)$ crosses an extent boundary. However, optimisation can be used to make $X(j,c)$ always be 1 to reduce seek and rotational overhead. For example, a request can be based on block access and an extent can be guaranteed to be the multiples of such blocks. While specific optimisation is not the topic of this work, the extent-based file system, the above AC criteria, and the following disk service time estimation do not prohibit such optimisation at all.

The disk service time of a piece is composed of seek and rotational latency and data read/write. Because seeks between the pieces in a cycle can be optimised using SCAN (*C*-**15**), the upper bound of the seek time in cycle c can be estimated as:

$$a * \sum_{j=1}^{J(c)} X(j,c) + b * A(c) \leq a * \sum_{j=1}^{J(c)} X(j,c) + bA \tag{6.10}$$

where $A(c)$ is the cylinder span of all the pieces in cycle c, while a and b are the disk seek time profile constants (cf. page 44), and $bA = b * \mathit{disk\_total\_cylinder\_span}$. Although read and write have different such constants, as they do not differ very much, the write seek constants, which are the worse of the two, can be used in the above estimation.

In practice, it is very hard to optimise rotational latency without modifying disk controller microcode. This is because there is a gap between individual SCSI requests issued

from a device driver to a disk controller, and the driver can hardly estimate where the disk has spun to at run-time. If batch requests were issued, then an intelligent disk controller would use its own seek optimisation which cannot be controlled by a user and would introduce unpredictability into the system. Thus it is better to use worst case rotation time to estimate the rotational overhead in a cycle, that is: $T_{rot} * \sum_{j=1}^{J(c)} X(j,c)$, where T_{rot} is the worst case rotational latency.

There are three factors contributing to the read/write time of a contiguous piece: data transfer, track switch, and cylinder switch. To estimate read/write time on a ZBR disk for a piece $P(j,c,x)$, first it is necessary to find the zone $z(j,c,x)$ where the piece resides using the zone information maintained by an SS (**C-16**). If a piece crosses zones, then the inner zone can be used as $z(j,c,x)$ because adjacent zones do not have radically different transfer rates or track or cylinder switch time. Assume at zone z, the track switch time is $TS(z)$, the cylinder switch time is $CS(z)$, the size of a track is $SN(z)$, the size of a cylinder is $CN(z)$, and the transfer rate is $B(z)$, which can be computed from the disk rotation speed and the size of a track. Then for a piece of size $P(j,c,x)$ at zone $z(j,c,x)$, its total read/write time is the sum of the data transfer time, the track switch time, and the cylinder switch time, which is:

$$\frac{P(j,c,x)}{min(B(z(j,c,x)), Bmin(Rd), Bmin(Rs\_d))} + \left\lceil \frac{P(j,c,x)}{SN(z(j,c,x))} \right\rceil * TS(z(j,c,x)) + \left\lceil \frac{P(j,c,x)}{CN(z(j,c,x))} \right\rceil * CS(z(j,c,x))$$

where $Bmin(Rs\_d)$ is the bandwidth of the slowest resource in Rs on disk data transfer. It should be noted that the data transfer time is estimated using the I/O bandwidth of the slowest component in the data path between the disk and the memory, including the CPU.

Consequently, the time required to service $J(c)$ disk requests using SCAN in cycle c can be estimated as:

$$\begin{aligned}[c].disk = {} & (a + T_{rot}) * \sum_{j=1}^{J(c)} X(j,c) + bA \\ & + \sum_{j=1}^{J(c)} \sum_{x=1}^{X(j,c)} \left\{ \left\lceil \frac{P(j,c,x)}{SN(z(j,c,x))} \right\rceil * TS(z(j,c,x)) + \left\lceil \frac{P(j,c,x)}{CN(z(j,c,x))} \right\rceil * CS(z(j,c,x)) \right\} \\ & + \sum_{j=1}^{J(c)} \sum_{x=1}^{X(j,c)} \frac{P(j,c,x)}{min(B(z(j,c,x)), Bmin(Rd), Bmin(Rs\_d))} \end{aligned} \quad (6.11)$$

Naturally, Criterion (6.2) can be refined as:

$$\begin{aligned} & (a + T_{rot}) * \sum_{j=1}^{J(c)} X(j,c) + \sum_{j=1}^{J(c)} \sum_{x=1}^{X(j,c)} \frac{P(j,c,x)}{min(B(z(j,c,x)), Bmin(Rd), Bmin(Rs\_d))} \\ & + \sum_{j=1}^{J(c)} \sum_{x=1}^{X(j,c)} \left\{ \left\lceil \frac{P(j,c,x)}{SN(z(j,c,x))} \right\rceil * TS(z(j,c,x)) + \left\lceil \frac{P(j,c,x)}{CN(z(j,c,x))} \right\rceil * CS(z(j,c,x)) \right\} \\ & \leq T_c - T_{res\_CPU} - bA - T_{NRTrecv\_CPU} - [c].RTrecv\_CPU \end{aligned} \quad (6.12)$$

If $R(j,c)$ also denotes the request itself, then its seek time and data transfer time can be estimated as:

$$R(j,c).disk\_seek = (a + T_{rot}) * X(j,c) + \sum_{x=1}^{X(j,c)} \left\{ \left\lceil \frac{P(j,c,x)}{SN(z(j,c,x))} \right\rceil * TS(z(j,c,x)) + \left\lceil \frac{P(j,c,x)}{CN(z(j,c,x))} \right\rceil * CS(z(j,c,x)) \right\} \quad (6.13)$$

$$R(j,c).disk\_xfer = \sum_{x=1}^{X(j,c)} \frac{P(j,c,x)}{min(B(z(j,c,x)), Bmin(Rd), Bmin(Rs\_d))} \quad (6.14)$$

respectively, and the disk AC criterion can be further refined as:

$$\sum_{j=1}^{J(c)} (R(j,c).disk\_seek + R(j,c).disk\_xfer) \leq T_c - T_{res\_CPU} - bA - T_{NRTrecv\_CPU} - [c].RTrecv\_CPU \quad (6.15)$$

6.3 Admission Control for Real-Time Activities

6.3.1 Example Data Structure

For playing back a VBR video, an SCA will compute the displayed set from the video index, then the read set from the displayed set. An example read set of a video file with the first 15 cycles is shown in Table 6.1. It lists in each cycle the number of blocks needed to be retrieved and the SS number where the primary blocks and the backup copies reside using the placement example in Table 4.2.

Table 6.1: An Example Read Set

cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
blocks needed	2	1	0	1	1	1	0	1	0	1	0	4	1	1	1
block number	0,1	2		3	4	5		6		7		8,9,10,11	12	13	14
primary SS	0,1	2		3	4	5		0		1		2,3,4,5	0	1	2
backup SS	1,2	3		4	5	0		2		3		4,5,0,1	3	4	5

From the read set and the redundancy scheme, a read table can be constructed for each SS on which a video file resides. A read table tells in which cycle to retrieve which block of a video in non-fault and fault modes. For example, the read table for SS(1) of the video file in Table 6.1 is shown in Table 6.2, where iBj means backup block Bj should be retrieved in the relative cycle when SS(i) fails.

Table 6.2: An Example Read Table: SS(1)

cycle	0	9	11	13	15	...
blocks	0B0,P1	P7	5B11	P13	4B16	...

Each SS has a *cycle table* for use by AC and resource reservation. A cycle table will record resources committed so far in each cycle (**C-17**). An example cycle table for SS(1)

after the stream in Table 6.1 is admitted is shown in Table 6.3, in which $ct[c].res$ refers to the amount of resources reserved in cycle c. $ct[c].res$ is composed of the resource requirements for non-fault mode, i.e., $ct[c].non\_fault$, and the extra resource requirements for fault mode, i.e., $ct[c].fault$. The cycle table of an SS also contains d entries for each of the previous d SSs in each cycle, with each entry recording the aggregate resources needed if the SS corresponding to the entry fails, i.e., $ct[c].fail[SS(i)]$ represents the amount of resources required to retrieve backup copies when SS(i) fails at cycle c. It should be noted that $ct[c].fault$ is the maximum of the d entries, not their sum (cf. Section 4.4.2).

Table 6.3: An Example Cycle Table: SS(1)

$ct[c]$	cycle 0	1	...
non_fault	$ct[0].non\_fault = ct[0].non\_fault$ + resources needed for P1	...	...
$fail[SS(3)]$	$ct[0].fail[SS(3)]$	...	...
$fail[SS(4)]$	$ct[0].fail[SS(4)]$	...	...
$fail[SS(5)]$	$ct[0].fail[SS(5)]$	...	...
$fail[SS(0)]$	$ct[0].fail[SS(0)] = ct[0].fail[SS(0)]$ + resources needed for B0	...	...
$fault$	$\max(ct[0].fail[SS(3)], ct[0].fail[SS(4)], ct[0].fail[SS(5)], ct[0].fail[SS(0)])$	...	...
res	$ct[0].res = ct[0].non\_fault + ct[0].fault$	...	...

For CBR recording, a receive table can be computed by an SS using the attributes of the LO to be recorded. Assume that in each cycle, a CBR data source will produce 2 blocks. Then with the placement method from Table 4.2, the receive table for SS(1) of the video to be recorded is shown in Table 6.4. Unlike a read table, the backup blocks in Table 6.4 should always be received and written.

Table 6.4: An Example Receive Table for CBR: SS(1)

cycle	1	4	6	7	9	10	11	...
blocks	P1, B0	P7	B11	P13	B16	P19	B21	...

Cadmus assumes it is impossible to predict how much data a source will produce in each cycle during VBR recording except the peak rate. To provide deterministic services for VBR recording, peak rate reservation is used in AC. Assume a video source will produce at most 4 blocks of data in a cycle. With the striping structure in Table 4.2, each SS will receive at most 1 primary and 1 backup blocks in each cycle. The corresponding receive table is shown in Table 6.5, in which P? and B? means that a primary or backup block may be received in a cycle but the block number is unknown.

Table 6.5: An Example Receive Table for VBR

cycle	1	2	3	4	5	...
blocks	P?, B?	P?, B?	P?, B?	P?, B?	P?, B?	...

6.3.2 Playback

An SS will perform AC for RT playback based on read tables passed from SCAs. There may be multiple primary or backup block access requests in a read table entry. Assume the kth block request in the ith entry of a read table is $rt[i].b[k]$, and the cycle index of the entry is $rt[i].cycle$, which is relative to the start of the video (cf. Ta-

ble 6.2). Let $rt[i].b[k].disk\_seek$, $rt[i].b[k].disk\_xfer$, $rt[i].b[k].disk\_CPU$, $rt[i].b[k].send\_net$, $rt[i].b[k].send\_CPU$, and $rt[i].b[k].buf$ be the corresponding resource requirements of the block access. Then from a cycle c in the cycle table, the AC will succeed only if all the entries in the read table satisfy the AC criteria. This is shown in more detail in Figure 6.2.

```
var: cycle index: j; temporary cycle table: sct[];

sct = ct;
for all read table entry rt[i] do
  j = rt[i].cycle;

  for all block request rt[i].b[k] do
    if primary block access then
      sct[c + j].non_fault.disk += rt[i].b[k].disk_seek + rt[i].b[k].disk_xfer;
      sct[c + j].non_fault.CPU += rt[i].b[k].disk_CPU;
      sct[c + j].non_fault.buf += rt[i].b[k].buf;
      sct[c + j + 1].non_fault.net += rt[i].b[k].send_net;
      sct[c + j + 1].non_fault.CPU += rt[i].b[k].send_CPU;
    else /* backup block access in case SS(x) fails. */
      sct[c + j].fail[SS(x)].disk += rt[i].b[k].disk_seek + rt[i].b[k].disk_xfer;
      sct[c + j].fail[SS(x)].CPU += rt[i].b[k].disk_CPU;
      sct[c + j].fail[SS(x)].buf += rt[i].b[k].buf;
      sct[c + j + 1].fail[SS(x)].net += rt[i].b[k].send_net;
      sct[c + j + 1].fail[SS(x)].CPU += rt[i].b[k].send_CPU;
    end if
  end for

  /* The following four are short-hands for sets of operations on individual resources. */
  sct[c + j].fault = max(sct[c + j].fail[] array);
  sct[c + j + 1].fault = max(sct[c + j + 1].fail[] array);
  sct[c + j].res = sct[c + j].non_fault + sct[c + j].fault;
  sct[c + j + 1].res = sct[c + j + 1].non_fault + sct[c + j + 1].fault;

  if  sct[c + j].res.disk ≤ T_c − T_res_CPU − bA − T_NRTrecv_CPU − sct[c + j].RTrecv_CPU
     && sct[c + j].res.CPU ≤ T_c − T_res_CPU − T_pure_CPU − T_NRTrecv_CPU
     && sct[c + j].res.buf ≤ K
     && sct[c + j + 1].res.net ≤ T_c − T_res_CPU − T_NRTrecv_net
     && sct[c + j + 1].res.CPU ≤ T_c − T_res_CPU − T_pure_CPU − T_NRTrecv_CPU
   then
    continue; /* AC succeeds for entry rt[i]. */
  else
    return; /* AC fails from cycle c. */
  end if
end for

/* AC succeeds for read table rt[] from cycle c. */
if need to reserve resources then
  ct = sct; /* commit. */
end if
```

Figure 6.2: AC for RT Playback

6.3.3 Recording

To achieve RT recording, the storage requirements of a video to be recorded are first examined by the SSs to see if there is enough free space. If so, an FS and the SSs will create LOs and POs respectively for both primary and backup copies. In subsequent AC, the SSs will pre-allocate extents for both primary and backup POs based on estimated storage requirements. For VBR recording, each extent is made up of multiples of the SUS so that Equation (6.13) and Equation (6.14) can be used to estimate the disk service time of the P? and B? in a receive table using the slowest zone parameters from all pre-allocated extents. The extents will be de-allocated if AC fails.

Pre-allocation of extents for recording has several benefits. First, because receive tables for CBR recording can be computed precisely, extent lists make it possible to estimate disk service time more tightly using Equation (6.13) and Equation (6.14). Second, although data arrivals are unpredictable for VBR recording and which block to write cannot be determined in each cycle, parameters of the slowest zone among those occupied by pre-allocated extents can be used in the same equations to provide a better estimation than the one using the slowest zone of the whole disk. Finally, run-time scheduling complexity is reduced because the situation when RT disk write has caught up with extent allocation activities, which are NRT, will not occur.

AC on the memory is done on a per stream basis instead of a per cycle basis, because a fixed dual-buffer is used for RT recording. Assume the kth block request in the ith entry of a receive table is $rt[i].b[k]$, and the cycle index of the entry is $rt[i].cycle$, which is relative to the start of the video (cf. Table 6.4 and Table 6.5). Let $rt[i].b[k].disk\_seek$, $rt[i].b[k].disk\_xfer$, $rt[i].b[k].disk\_CPU$, $rt[i].b[k].recv\_net$, and $rt[i].b[k].recv\_CPU$ be the corresponding resource requirements of the block access. Then from a cycle c in the cycle table, the AC will succeed only if all the entries in the receive table satisfy the AC criteria. This is shown in more detail in Figure 6.3.

6.4 Admission Control for Non-Real-Time Activities

6.4.1 Introduction

Cadmus supports NRT activities as well, but NRT requests should not disturb admitted RT services. Apart from NRT receive, other NRT requests are subject to run-time AC before they are issued to the disk or network device (*C*-**18**). Their AC is simplified because each request is either disk or network bound but not both, but it is also more complex because of performance requirements. A naive approach would be to put every NRT request into an *NRT list* when it arrives and to perform AC for it on later cycles in order to preserve QoS of admitted RT and NRT activities in the current cycle. However, this will bring severe performance problems especially when an SS is lightly loaded, as will be shown in the next chapter. Cadmus addresses the problem using a three-level AC for NRT requests: *current*, *next*, and *delayed* (*C*-**19**).

When an NRT request arrives, resources in the current cycle are examined first: this requires detection of how much time is left in the current cycle and how many resources will be used in the time left. If there are not enough resources in the current cycle, resources from the next cycle, which are recorded in the cycle table, are used for the AC. This step of look-ahead is safe because in the worst case the NRT request will defer the schedule of

```
var: cycle index: j; temporary cycle table: sct[];

if there is not enough memory for the fixed dual-buffer requirement then
  return; /* AC fails. */
end if

allocate extents for both the primary and backup POs based on the estimated
    storage requirements;

sct = ct;
for all receive table entry rt[i] do
  j = rt[i].cycle;

  for all block request rt[i].b[k] do
    sct[c + j].RTrecv_CPU += rt[i].b[k].recv_CPU;
    sct[c + j].non_fault.net += rt[i].b[k].recv_net;
    sct[c + j].non_fault.CPU += rt[i].b[k].recv_CPU;
    sct[c + j + 1].non_fault.disk += rt[i].b[k].disk_seek + rt[i].b[k].disk_xfer;
    sct[c + j + 1].non_fault.CPU += rt[i].b[k].disk_CPU;
  end for

  /* The following two are short-hands for sets of operations on individual resources. */
  sct[c + j].res = sct[c + j].non_fault + sct[c + j].fault;
  sct[c + j + 1].res = sct[c + j + 1].non_fault + sct[c + j + 1].fault;

  if  sct[c + j].res.net ≤ T_c − T_res_CPU − T_NRTrecv_net
     && sct[c + j].res.disk ≤ T_c − T_res_CPU − bA − T_NRTrecv_CPU − sct[c + j].RTrecv_CPU
     && sct[c + j].res.CPU ≤ T_c − T_res_CPU − T_pure_CPU − T_NRTrecv_CPU
     && sct[c + j + 1].res.disk
         ≤ T_c − T_res_CPU − bA − T_NRTrecv_CPU − sct[c + j + 1].RTrecv_CPU
     && sct[c + j + 1].res.CPU ≤ T_c − T_res_CPU − T_pure_CPU − T_NRTrecv_CPU
   then
    continue; /* AC succeeds for entry rt[i]. */
  else
    de-allocate the extents for both the primary and backup POs;
    return; /* AC fails from cycle c. */
  end if
end for

/* AC succeeds for receive table rt[] from cycle c. */
if need to reserve resources then /* commit. */
  reserve the dual-buffer;
  ct = sct;
end if
```

Figure 6.3: AC for RT Recording

the next cycle, which is delayable; otherwise the NRT request should not be admitted in the first place. If the current and the next cycles cannot accommodate an NRT request, it is delayed and added to the NRT list for later AC after the RT schedule is computed. The next subsection will discuss these scenarios in more detail.

6.4.2 Three-Level Admission Control

Current

Assume an NRT request arrives at a point with a period of *passed* from the start of the current cycle and *left* from the end of the cycle, and $passed + left = T_c$. For the request to be admitted in the current cycle, an SS should measure the actual resource usage of several activities in the period *passed* (**C-20**), so that resources needed in the period *left* can be deduced and used in the AC for the NRT request. The symbols used for these purposes and their explanations are listed in Table 6.6.

Table 6.6: Symbols Used for NRT AC: Current

resources used in the period *passed* (measured)	
recv_CPU_used	CPU time used by receive activities
ds_CPU_used	CPU time used by disk and send activities
recv_net_used	time spent on the network data path by receive activities
send_net_used	time spent on the network data path by send activities
resources needed in the period *left* (deduced)	
disk_todo	time needed by disk activities
net_todo	time needed on the network data path by network activities
recv_CPU_todo	CPU time needed by receive activities
ds_CPU_todo	CPU time needed by disk and send activities
pure_CPU_todo	CPU time needed by pure CPU processing activities

Assume the current cycle has index c and its reserved resources are recorded in $ct[c].res$. Then resources needed in the period *left* can be computed as shown in Figure 6.4.

input: $ct[c].res, passed, left, recv\_CPU\_used, ds\_CPU\_used, recv\_net\_used, send\_net\_used$;
output: $disk\_todo, net\_todo, recv\_CPU\_todo, ds\_CPU\_todo, pure\_CPU\_todo$;

$disk\_todo = \max(0, ct[c].res.disk + bA + recv\_CPU\_used - passed)$;
$net\_todo = ct[c].res.net + T_{NRTrecv\_net} - recv\_net\_used - send\_net\_used$;
$recv\_CPU\_todo = ct[c].res.RTrecv\_CPU + T_{NRTrecv\_CPU} - recv\_CPU\_used$;
$ds\_CPU\_todo = ct[c].res.CPU - ct[c].res.RTrecv\_CPU - ds\_CPU\_used$;
$pure\_CPU\_todo = \max(0, T_{pure\_CPU} + recv\_CPU\_used + ds\_CPU\_used - passed)$;

Figure 6.4: Computing Resources to be Used: Current

If an NRT request is admitted when it arrives, it should be appended to the end of the current schedule or sent to the device directly if previously admitted requests have been finished in the current cycle. This is necessary in order to prevent a newly admitted NRT request from delaying admitted RT requests in overload situations. When overload does occur in the current cycle, newly admitted NRT requests are appended to the end of the next cycle's schedule (**C-21**) in the hope that there *may be* spare resources left. This will further increase the chance of a newly admitted NRT request being executed. Although no AC is performed for this NRT request movement, no harm will be done to admitted RT requests in the next cycle. This is because if the next cycle does not have enough resources, these NRT requests will be put back to the NRT list at the end of the next cycle. Other unfinished NRT requests in the current cycle are moved to the NRT list (**C-21**), because they are not entitled to participate in the new cycle any more.

To admit an NRT disk request in the current cycle, the request's worst seek time should be used instead of the amortised seek time computed from Equation (6.13). This is because when an admitted disk request is appended to the end of a schedule, it will be out of the SCAN order. Assume an NRT disk request q only accesses one *piece* of data and has a worst seek time of $q.disk\_worst\_seek$, with $q.disk\_worst\_seek = q.disk\_seek + bA$. Also assume its data transfer and CPU time are $q.disk\_xfer$ and $q.disk\_CPU$ respectively. Then the AC procedure in the current cycle for q is shown in Figure 6.5.

```
if disk_todo + q.disk_worst_seek + q.disk_xfer ≤ left − T_res_CPU − recv_CPU_todo
   && ds_CPU_todo + q.disk_CPU + recv_CPU_todo ≤ left − T_res_CPU − pure_CPU_todo
then
  /* AC succeeds. Update resource usage. */
  ct[c].res.CPU += q.disk_CPU;
  if disk_todo > 0 then
    ct[c].res.disk += q.disk_worst_seek + q.disk_xfer;
  else /* disk_todo == 0 */
    ct[c].res.disk = passed + q.disk_worst_seek + q.disk_xfer − bA − recv_CPU_used;
  end if
end if
```

Figure 6.5: AC for an NRT Disk Request: Current

Similarly, Figure 6.6 shows the AC procedure in the current cycle for an NRT network send request q with resource requirements $q.send\_net$ and $q.send\_CPU$.

```
if net_todo + q.send_net ≤ left − T_res_CPU
   && ds_CPU_todo + q.send_CPU + recv_CPU_todo ≤ left − T_res_CPU − pure_CPU_todo
then
  /* AC succeeds. Update resource usage. */
  ct[c].res.net += q.send_net;
  ct[c].res.CPU += q.send_CPU;
end if
```

Figure 6.6: AC for an NRT Network Send Request: Current

Next

When an NRT request cannot be admitted in the current cycle c, resources from the next cycle, recorded in $ct[c+1].res$, are examined to admit the request. If admitted, the request is treated exactly as if it were admitted using resources of the current cycle. Worst seek time should be used in NRT disk request AC as well. However, when overload occurs, these NRT requests are moved to the next cycle (**C-21**) because there *are* spare resources in the next cycle, although the effects of moving NRT requests admitted using resources of either the current or the next cycle are the same. Figure 6.7 shows the AC procedure for an NRT disk request using resources in the next cycle, while Figure 6.8 shows the procedure for an NRT network send request. Both requests are represented by q.

```
if ct[c + 1].res.disk + q.disk_worst_seek + q.disk_xfer
      ≤ T_c − T_res_CPU − bA − T_NRTrecv_CPU − ct[c + 1].RTrecv_CPU
   && ct[c + 1].res.CPU + q.disk_CPU ≤ T_c − T_res_CPU − T_pure_CPU − T_NRTrecv_CPU
 then
  /* AC succeeds. Update resource usage. */
  ct[c + 1].non_fault.disk += q.disk_worst_seek + q.disk_xfer;
  ct[c + 1].res.disk += q.disk_worst_seek + q.disk_xfer;
  ct[c + 1].non_fault.CPU += q.disk_CPU;
  ct[c + 1].res.CPU += q.disk_CPU;
end if
```

Figure 6.7: AC for an NRT Disk Request: Next

```
if ct[c + 1].res.net + q.send_net ≤ T_c − T_res_CPU − T_NRTrecv_net
   && ct[c + 1].res.CPU + q.send_CPU ≤ T_c − T_res_CPU − T_pure_CPU − T_NRTrecv_CPU
 then
  /* AC succeeds. Update resource usage. */
  ct[c + 1].non_fault.net += q.send_net;
  ct[c + 1].res.net += q.send_net;
  ct[c + 1].non_fault.CPU += q.send_CPU;
  ct[c + 1].res.CPU += q.send_CPU;
end if
```

Figure 6.8: AC for an NRT Network Send Request: Next

Delayed

If both the above two steps fail for an NRT request, then it is added to the NRT list waiting for AC in later cycles. A schedule for a later cycle x will be dynamically computed for admitted RT requests first. If there are spare resources left in $ct[x]$, then requests in the NRT list are examined for AC. The procedures are similar to those shown in Figure 6.7 and Figure 6.8, except $ct[x]$ is used instead of $ct[c+1]$, and *disk_seek* is used instead of *worst_disk_seek*, as NRT disk requests can now be inserted in the newly computed schedule according to SCAN order.

6.4.3 Problems and Solutions

Pure CPU Processing Starvation

Admitting newly arrived NRT requests using the next cycle's resources has the potential of disrupting QoS for pure CPU processing activities. This is because device related NRT activities have higher priorities and they are appended to the schedule of the current cycle once admitted. There are chances that pure CPU processing will never get executed if NRT requests keep arriving and there are always spare resources in the next cycle. A simple solution employed in Cadmus is to measure the CPU time used by disk and network send activities in the current cycle c, and to disable both activities once the following condition is true (*C*-**22**):

$$ds\_CPU\_used \geq T_c - T_{res\_CPU} - T_{pure\_CPU} - T_{NRTrecv\_CPU} - ct[c].res.RTrecv\_CPU \tag{6.16}$$

Receive Re-activation

To regulate receive activities and to prevent the live-lock state, Cadmus also measures the CPU time used by receive activities in the current cycle c, and disables them whenever the following condition is true:

$$recv\_CPU\_used \geq ct[c].res.RTrecv\_CPU + T_{NRTrecv\_CPU} \tag{6.17}$$

However, it is overkill to disable receive when all the other activities have finished and there is time left in the current cycle. Figure 6.9 shows the Cadmus approach to the problem by re-activating receive after all device related activities have finished (**C-23**). To prevent starving pure CPU processing, the algorithm is only executed when the CPU is idling. Three aspects of the algorithm should be noted. First, resources needed in the period *left* still have to be checked because there may be some NRT requests that are admitted using the current or next cycle's resources but are not on any schedule, i.e., they may arrive and be admitted after the current disk and send schedules have finished. Second, the resource usage should be updated during receive re-activation because there may be NRT requests arriving in the future. Also it makes further receive regulation according to Equation (6.17) possible. Third, pure CPU processing activities may be ready after some receive events. Therefore, receive activities should leave some time for them to execute. The two parameters $T_{extra\_recv\_CPU\_threshold}$ and $T_{extra\_recv\_CPU}$ are introduced for this purpose.

```
if CPU idling
   && disk schedule finished
   && network send schedule finished
   && receive disabled
 then
  /* Disk can be deferred at most X_disk. */
  X_disk = left − T_res_CPU − disk_todo;
  /* Network can be deferred at most X_net. */
  X_recv_net = left − T_res_CPU − net_todo;
  /* CPU can be deferred at most X_CPU. */
  X_CPU = left − T_res_CPU − ds_CPU_todo;

  deduce X_recv_CPU from X_recv_net;

  /* Extra receive can consume CPU time at most Y. */
  Y = min(X_disk, X_CPU, X_recv_CPU);

  if Y > T_extra_recv_CPU_threshold then
    /* Update resource usage and re-activate receive. */
    Z_recv_CPU = min(Y, T_extra_recv_CPU);
    deduce Z_recv_net from Z_recv_CPU;
    ct[c].res.RTrecv_CPU += Z_recv_CPU;
    ct[c].res.net += Z_recv_net;
    ct[c].res.CPU += Z_recv_CPU;
    re-activate receive;
  end if
end if
```

Figure 6.9: Receive Re-activation: Current

6.5 VCR Functionalities

VCR functionalities refer to the stop, pause, resume, slow forward, slow reverse, fast forward (FF), and fast reverse (FR) of a video being played. This section introduces the Cadmus view on supporting these functionalities. However, because of the limited research time scale, they are not implemented in the Cadmus prototype.

Stopping during normal playback will make an SS recompute the resources needed in each left over cycle of a stream and deduct them from the cycle table, i.e., reclaim the reserved resources. Also the stopped stream is deleted to prevent its inclusion in future schedule computing. Pause is different in that a stream may be resumed later on. A simple approach would be for pause to reclaim reserved resources while resume will incur additional AC to prevent disturbance to other streams. However, if new streams are admitted using the reclaimed resources, the paused stream may wait for a long time before it could be resumed because of scarce resources.

To make a paused stream have a better chance of resumption, it is better to reserve a uniform resource usage estimated using mean stream rate and mean disk transfer rate for the paused stream in subsequent cycles. A mark in each cycle will indicate these uniformly reserved resources cannot be used by new stream admission, but they can be used by NRT requests. Though these reserved resources are wasted because otherwise new streams could be admitted, it is reasonable if a server continues to charge the client if a stream is paused. Stopping during pause will reclaim the uniformly reserved amount of resources and delete the stream.

Since slow forward and slow reverse, FF and FR are symmetric pairs, only slow forward and FF are considered. A new read set, with more suppressed cycles, is computed for slow forward and AC is done just as for a normal playback request. Also additional AC is needed for the resumption of normal playback. The same philosophy as behind pause-resume applies: a uniform amount of resources is reserved in the extra suppressed cycles to gain a better chance of resumption after slow forward.

FF is more difficult to deal with because it would potentially increase server load by retrieving and sending more data in a cycle. However, before giving a solution, some properties of FF can be observed from the usage pattern of a VCR. First, most of the time it is operated in normal playback mode. Second, most of the time the purpose of FF is to let a tape go to a specific point of the video sequence and there is no need to watch the video in between. This is called FFNV (FF no viewing). Finally, in the occasions when FF is used with viewing (FFV), most of the time it is for finding a scene where normal playback can be resumed instead of viewing the fast video on purpose, since it is difficult for human perception to receive all the information when a video is shown faster than the normal playback rate. More details of the scenes can be viewed using slow forward like the replay in sports events.

FFNV can be dealt with by a video server without introducing extra load since it is actually a random access problem. The same pause-resume sequence can be used, but now resumption goes to a different place instead of the point where a video is paused. A client can request a fast forward to any point relative to the start of a video or skip any number of frames relative to the current position of normal playback. This better FFNV accuracy than that provided by a tape based VCR will further reduce the need for FFV, although FFV is useful to find a particular event in a programme. All these suggest FFV is a minor problem faced by video server design and should be treated simply. Storing an extra FF version of a video [**Heybey96**], using scalable streams [**Shenoy95**], or optimising data placement just for FF [**Chen94a**] may not be justified. They also make a video server more complex.

Based on the above observations, it would be better to simply skip frames when FFV is really desired by a user. For example, considering a 24 fps MPEG video sequence with 12 frames per GOP (Group of Picture) and a GOP structure of IPBB, i.e., each GOP has 1 I frame, 3 P frames, and 8 B frames. Then a 1 second cycle will have to retrieve 2 GOPs in average during normal playback. When FFV is requested, a server could retrieve and send the I frames only. I frames are independently decodable and a user can also notice where a video is by watching only I frames. However, if the I frames of a video are not stored contiguously, each I frame retrieval will incur a rotational latency, although seeks can be optimised by SCAN.

Assume a speed up factor of u, then each cycle has to retrieve $2 * u$ I frames with $(2 * u - 1) * T_{rot}$ extra rotational latency compared to normal playback if the 2 GOPs of a cycle are stored in one piece. The load of data transfer depends on the size of the I frames. If the I/P/B frame size ratio is around 4/2/1, then a speed up factor of u will retrieve extra data with size $2 * u * 4 - 2 * 18$ per cycle compared to normal playback for the above settings if the B frame size is normalised to 1. This may or may not incur extra data transfer load depending on the value of u. To reduce the rotational latency for FFV, the I frames in a cycle can be grouped together. For example, if the 2 I frames can be stored contiguously before the P and B frames of the 2 GOPs in a cycle, only $(u - 1) * T_{rot}$ extra rotational latency is incurred for FFV and this extra overhead may be distributed to different SSs.

6.6 Summary

This chapter presented AC procedures for both RT and NRT activities on the network, disk, CPU, and memory resources in an SS entity, in which the CPU was modelled as a shared resource for disk and network activities. General AC criteria were given with assumptions and conditions which would minimise interference between different types of activities and lead to less QoS enforcement efforts. Considerations were given to ZBR in AC and overload processing in case of disk anomalies. RT AC was based on the cycle lifetime of a video, while a three-level AC scheme was proposed for NRT requests. Special attention has been paid to pure CPU processing starvation and receive regulation and re-activation. This chapter also identified and introduced the set of contracts assumed by the AC procedures which must be fulfilled by QoS enforcement. Finally, the Cadmus view on supporting VCR functionalities was described.

Chapter 7

Implementation

This chapter describes a prototype implementation of the Cadmus system in the following aspects: implementation environment, system and some extra components, software structure and functionalities in storage nodes, PO management, contract enforcement, and resource reclamation.

7.1 Implementation Environment

A prototype of the Cadmus architecture has been implemented using the ORL direct peripheral infrastructure and ATM interconnect [**Chaney95**]. Different system components interacted through CORBA interfaces using the locally developed OmniORB1 [**Gilmurray95**], which was a multi-threaded implementation of the ORB and ran on Solaris, Digital UNIX, Linux, Windows NT, and ATMos, a message based micro-kernel. The thread package used in ATMos was based on P-Threads [**Pthreads93**]. CORBA object communication and NRT data transfer went through reliable TCP channels, while RT data was transferred using AAL5 over MSNL [**McAuley90**], a light weight virtual circuit protocol.

An SS normally ran on an ATMos card with an attached disk or disk array. An ATMos card is a simple processing unit with a direct ATM connection and runs the ATMos micro-kernel. A client ran on a video tile, which was composed of an ATMos card and a colour LCD display with VGA quality[1]. An SCF could be configured to run on either a Sun SparcStation under Solaris 2.5 or on an ATMos card for performance reasons. As there are no RT requirements for an FS, it normally ran on a Sun SparcStation. It should be noted that components running on any Solaris machine only used its CPU and memory, without referring to its file system.

As Cadmus relies on the cycle based QoS guarantee scheme, accurate and synchronised time is of paramount importance. This problem was addressed very simply with the help of some timing hardware and software. The prototype used two satellite GPS (Global Positioning System) receivers and an NTP (Network Time Protocol) package [**Chiang96**] to provide a global and synchronised time in the whole system. SSs and some clients also had add-on profiler boards [**Addlesee95**], each of which had a 0.5–2 MHz free-running counter and several hardware timers with the same frequency. With these settings, the

[1] More information of Smart ATMos Modules (SAM) can be found in [**SAM**].

time in different physical entities could be synchronised to within 1 ms [Mills94], which was deemed as acceptable for experimental purposes.

The Cadmus components of the prototype system consisted of about 25,000 lines of C++ and CORBA IDL code, which was about 10% of the code of the whole system, such as the kernel, device drivers, OmniORB1 library, and other ATMos processes (not including stub code), which were written in a mixture of C/C++/Assembly.

7.2 System and Extra Components

This section presents the implementation of the Cadmus system components described in Chapter 4. Some *extra components*, such as ports, connections, points, units, and unit factories, are also introduced. The extra components were used to implement the system components or to extend the Cadmus system. CORBA IDL interfaces of the components are listed in Appendix C along with the important methods. Some components, such as ports, points, CPs, and POs, also have unlisted implementation specific public methods manipulating buffer pointers in order to avoid data copying in the same physical entity.

Although the Cadmus architecture specifies the functionalities of DSs and NSs to make the system self-contained, they were not implemented in the prototype. However, they can be readily built on other components. To achieve the effects of NSs, physical entity addresses were embedded in the relative components which needed to know them. In addition, the experiments reported in the next chapter used LOs directly, instead of textual names, thus doing away with DSs. Some considerations to support VCR functionalities in CPs are also presented below, although these functions were not implemented.

7.2.1 Ports and Connections

An important aspect of the implementation is the separation of data and control paths. While control goes through distributed object invocation, audio and video data goes through *connections* established between *ports*. A port is merely a passive end-point for data transport. There are two kinds of ports: *source ports* and *sink ports*. A source port connects while a sink port listens. There is no restriction that data should always flow from a source port to a sink port. This bi-directional nature of ports is exploited by NRT-CPs and NRT-POs[2].

A connection has a pair of source/sink port references and will obtain addresses of the ports and make a data connection between them. After that, data is sent and received through the ports only. Both ports and connections have associated *transport attributes*, which can be used to specify a set of network related QoS parameters, although in the Cadmus implementation they were only used to select either reliable TCP or raw AAL5 transport.

7.2.2 Client Parts

Each CP is associated with an LO to scatter or gather its data and is the single point to maintain the state of the LO_ATTR for the LO. It is also notified of relative SS failures or recovery in order to activate or deactivate fault mode data retrieval. Because a CP cannot be used on its own, its buffers are provided by a third party for simplicity and modularity

[2]A PO can represent either the data object itself or the implementation used to access the data object when it is opened in a certain mode. The context will make the meaning clear. The same is true for an LO.

reasons. Each CP also has a number of ports corresponding to the number of SSs the LO is striped over. A CP sends/receives data through the ports without contacting the SSs.

There are two types of CPs: *RT-CPs* and *NRT-CPs*. An NRT-CP can perform both send and receive and provides an implementation of the NRT-LO, while RT-CPs are further divided into uni-directional *source* and *sink RT-CPs*. A single thread is used in an NRT-CP and a source RT-CP to receive and send data respectively, while multiple threads are used in a sink RT-CP to receive data from the SSs. Each data block contains information such as the block number and whether the block is a primary or backup one. This facilitates data de-multiplexing in both CPs and SSs and also reduces the number of connections.

Sink RT-CPs

Assume the normal playback read set of a video has a peak rate of p blocks per cycle and $p < n$, where n is the striping group size. Then during non-fault mode, in each cycle, a sink RT-CP will receive data blocks from at most p SSs simultaneously in normal playback mode, $\lceil p/o \rceil$ SSs in slow forward mode with a slow down factor of o, and $u * p$ SSs in FFV mode with a speed up factor of u. Thus the CP should have at most p, $\lceil p/o \rceil$, and $u * p$ active receiving threads anticipating data from the SSs in the corresponding modes.

In each cycle, the CP will receive at most one data block from a single SS in normal playback or slow forward mode. Although in FFV mode, I frames are sent in a cycle for MPEG streams, the aggregate I frame size in a block is less than the block size. So at most one data block needs to be received by the CP in FFV mode from a single SS if $u * p < n$, although data is retrieved by more SSs in a cycle. During fault mode, the CP will receive data from at most the same number of SSs as during non-fault mode, because the backup of a failed block is retrieved from another single SS for multi-chained declustering.

The algorithm in Chapter 5 suggests that a CP buffer with $2 * p$ blocks will suffice for normal playback or slow forward during non-fault or fault mode. For FFV, $2 * u * p$ blocks worth CP buffers will be needed. Because the read set computing algorithm reads ahead and smoothes the displayed set, in practice, one or zero blocks are received by a CP in most of the cycles in normal playback or slow forward mode. Read sets are passed from SCAs to sink RT-CPs for the latter to correctly anticipate data arrivals in each cycle for VBR videos.

Source RT-CPs

Because AC in SSs for RT recording makes assumptions about source data rates, a source RT-CP will have to regulate the actual rate to conform to the assumptions. This is to enforce constant rates for streams admitted as CBR videos and to ensure that peak rates are not exceeded for VBR recording. For each primary block to be written in a cycle, a CP will also send the same data to the SS where the backup should be stored through a corresponding port in the same cycle. Both primary and backup blocks to the same SS are sent through the same connection.

NRT-CPs

An NRT-CP uses the same connection to send or receive either primary or backup blocks to or from the same SS. Its send/receive operations are primitive in that they only retrieve data for either the primary or backup LO but not both. This way, the implementation of

the read/write operations for NRT-LOs can be made more flexible. For example, an NRT-LO can ask an NRT-CP to read either primary or backup blocks depending on the load of the corresponding SSs. However, it also becomes an NRT-LO's responsibility to maintain consistency between primary and backup blocks during writing.

7.2.3 Points, Units, and Unit Factories

Points, units, and unit factories are not parts of the Cadmus architecture, but are supplementary components primarily used by clients of Cadmus. There are two kinds of *points*: *end points* and *network points*. End points deal with client end devices such as camera sources, tile or X sinks, or UNIX file sources/sinks which are ultimate sources or sinks of data. A network point sends/receives data in some buffers through its port which is associated with one connection. The buffers of a point are also provided by a third party such as a unit. Each point can be either the source or sink of data flow, but not both. A point is active in that it has a thread, which can be started or stopped, to move or to process data.

A *unit* is an intermediate or client end component which can be inserted into data paths from a server to clients for data striping, concentrating, capturing, displaying, or processing. Units are managed by *unit factories* which reside in client or intermediate entities. A unit has a semaphore guarded circular buffer with a size of multiples of the SUS of the LO it is dealing with. The buffer is shared by a pair of points, CPs, or point and CP in the unit (Figure 7.1), while one of them is filling one slot of the buffer, the other consumes data in another slot of the buffer.

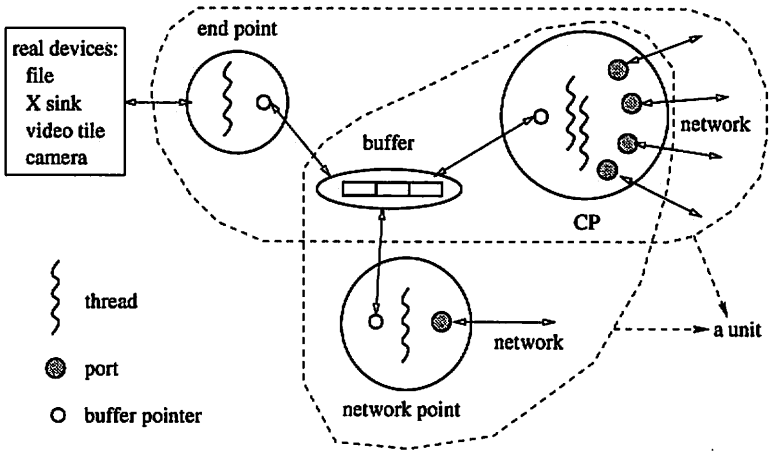

Figure 7.1: Points, CPs, and Units

Different combinations of points and/or CPs will result in units with different functionalities. For example, a tile unit with a tile sink end point and a sink RT-CP can be used to stream data from multiple SSs and display the data on a tile; file units with file end points and NRT-CPs can be constructed to transfer NRT data between Cadmus and a UNIX file system; and a unit with a network source point and a network sink point can form a data processing pipe.

7.2.4 Physical Objects and Storage Servers

Based on the mode in which a PO can be opened, three kinds of PO implementation are provided: *playback RT-POs*, *record RT-POs*, and *NRT-POs*. Each PO has a port and

a buffer with a size of multiples of the SUS (Figure 7.2), although a playback RT-PO's buffer is dynamically allocated from a K-buffer. Each PO implementation will open both the primary and backup POs if the PO is not a special one. For playback, the backup PO is used in case of other SS failures; for recording and NRT write, both the primary and backup POs will be written; also the backup PO may be used for NRT read to achieve load balancing.

RT-POs do not have any threads associated with them, as RT data retrieval is managed by SSs. However, an NRT-PO has a thread to do receive and can act as both data source and sink. An SS creates/erases and opens/closes POs. Special POs are treated as NRT-POs and there is no separate interface for them, although then only operations on a primary PO are valid. An SS is notified of other SS failures for fault mode data retrieval. The next two sections will describe SS and PO implementation in more detail.

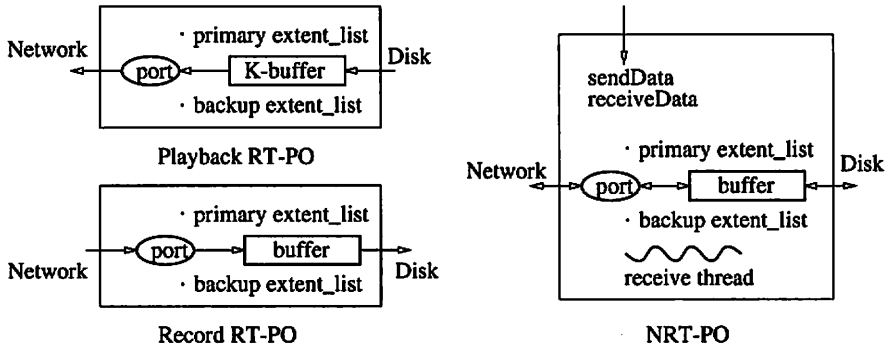

Figure 7.2: Physical Object Implementation

7.2.5 Logical Objects, Stream Control Agents, File Servers, and Stream Control Factories

Two types of LO implementation exist: *RT-LOs* and *NRT-LOs*. LOs and SCAs are closely related because an SCA is regarded as an RT-LO. All LOs are opened/closed through FSs in order to resolve concurrent access conflicts on the same LO. But an RT-LO (or SCA) is actually implemented by an SCF, and an NRT-LO by an NRT-CP. The decision to implement an NRT-LO in an NRT-CP is an implementation optimisation. It improves the performance for NRT data retrieval and prevents an FS from becoming a bottleneck for NRT data access, which may be true if FSs were to implement NRT-LOs.

The design criterion for LOs is that they hold references of data sources and sinks, but stay outside of data paths. Each LO, including an SCA, consists of references to a CP and a number of POs. It will make connections between the ports of the CP and the ports of the POs, and control the data flow, such as VCR control for SCAs and read/write for NRT-LOs. There are two kinds of SCAs: *playback SCAs* and *record SCAs*. The structure of a playback SCA for VBR videos is shown in Figure 7.3, in which an NRT-LO is used to read in the video index.

Although a replication based redundancy scheme is used in Cadmus, a design strategy is to hide this fact at the LO level in component interfaces by only referring to primary LOs. However, LO_ATTR will record the mirrored LOID, so that given one LO, its replica can always be found. Upon LO creation, an FS will actually create two LOs, but only attributes of the primary LO are returned. The functions of an SCF are very simple: it just creates and deletes playback and record SCAs upon requests from FSs. SCFs cannot be directly contacted by clients.

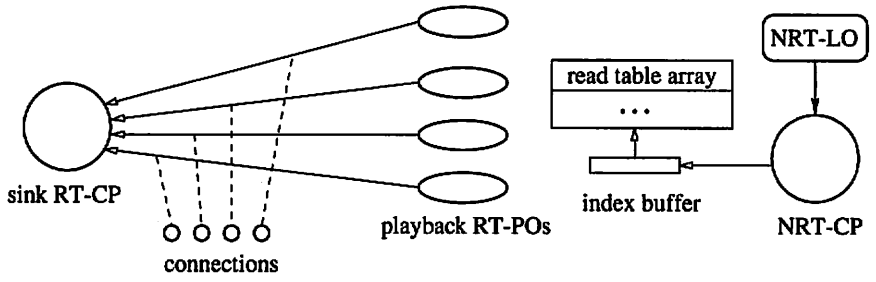

Figure 7.3: Playback SCA Implementation

Both FSs and SCFs will be notified of SS failures and recovery. This information will be propagated from an FS to the LOs it manages and the corresponding CPs. When an FS fails, SCFs may provide additional channels to notify SS failures and recovery to SCAs and then RT-CPs. However, SS failure detection was not implemented in the prototype.

7.3 The Storage Node

7.3.1 Software Structure

The storage node where an SS resides was built on an ATMos card which ran the ATMos micro-kernel, and its software structure consisted of a number of processes communicating via the message passing mechanism. Generally there are three kinds of processes in an ATMos system: device drivers, system processes, and user provided ones. The prototype implementation retained all these processes so that tasks that can be performed by the original ATMos system can also be executed in the Cadmus implementation. This strategy, rather than writing a new kernel, proved vital because it facilitated rapid prototyping. It also helped experimenting because the ATMos system already provided a rich set of tools and system services.

ATMos is the small and propriety kernel introduced in Section 2.6.3. Because the kernel, the thread package, and most device drivers and system server processes did not satisfy the QoS enforcement requirements for the Cadmus system, they have been modified for use by the Cadmus prototype. In addition, a process has been added to implement most of the functionalities of an SS identified in Section 4.2.2, and the process is also named as the SS process. However, the full functionalities of an SS are provided by the SS process in conjunction with other elements in a storage node, such as the file process, the kernel, and the device drivers. The storage node software structure is shown in Figure 7.4, in which an oval represents a process.

The main feature of the structure is that apart from receive requests originating from system processes such as the tcp process, other network and disk related requests are directed to the SS process, instead of to the relative device drivers as in the original ATMos system. NRT receive requests do not go through the SS process because they should not be delayed, as the atm driver will discard data if there is no receive buffer. Another consideration is that it is impossible to regulate individual receive requests, because newly arrived data can be de-multiplexed only after it has been received and examined. Thus receive requests are sent to the driver directly by their originators, which also simplifies the implementation. Other device related requests will generate corresponding NRT activities and have to undergo admission tests in the SS process before being issued to the device drivers.

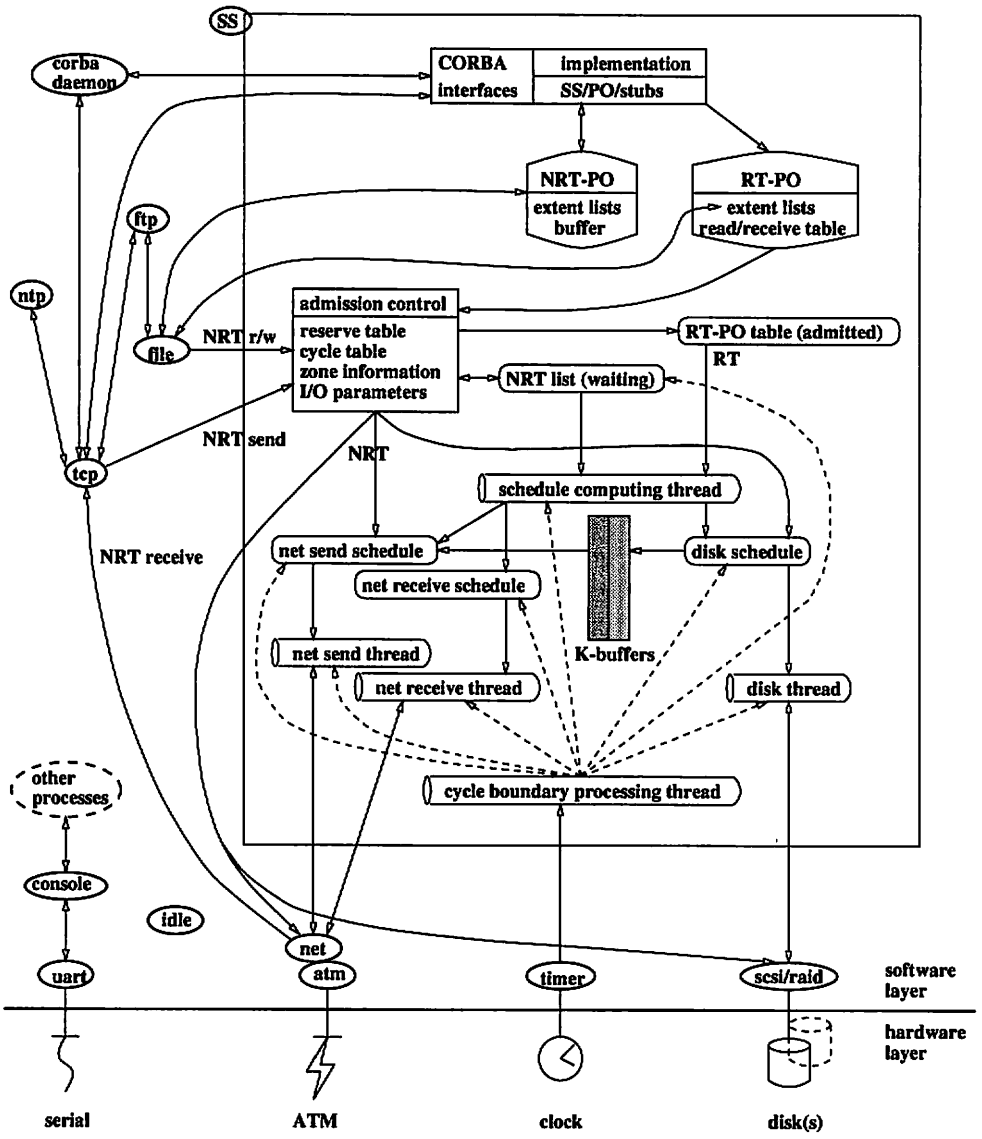

Figure 7.4: The Storage Node Software Structure

Only the SS process will generate RT activities, while other processes are recognised as sources of NRT pure CPU, disk, or network related activities only. The file process manages extent-based PO storage on disk devices, SS meta-data, data retrieval (read-ahead and write-behind) for NRT-POs, caches for SS meta-data and NRT data, and locks on the access of NRT-POs. The ftp process implements the FTP protocol, while the ntp process implements the NTP protocol. The tcp process implements the TCP/UDP/IP protocol suite and provides NRT communication channels for the ntp and the ftp processes as well as for the CORBA daemon and implementation.

The idle process has the lowest priority and will only run if there are no other CPU bound activities. The uart, atm, timer, and scsi/raid are device driver processes. However, device related activities will run at the interrupt level if the CPU is required, and the communication overhead of sending requests to the device drivers is kept to a minimum by a mechanism which is equivalent to allowing other processes to access the request queues in the drivers. Because an ATMos image runs in a single address space, excessive data copying between the kernel and the processes is avoided by passing buffer pointers.

7.3.2 The Storage Server Process

The structure of the SS process is also shown in Figure 7.4. The SS process fulfils four major tasks: SS and PO interface implementation, AC, device related RT and NRT request scheduling, and cycle boundary processing.

SS and PO Interface Implementation

SS and PO interfaces are centred around POs, whose storage is managed by the file process. Thus the creation and deletion of a PO need participation from the file process. Also when opening a PO, the SS process will obtain the extent list from the file process. For RT-POs, their corresponding disk and network related requests are processed inside the SS process. However, for NRT-POs, the read/write requests are delegated to the file process, while the send/receive requests are delegated to the tcp process. These NRT requests, apart from NRT receive, will eventually come back to the SS process for AC, although they will access data at a lower level such as disk blocks or AAL5 packets.

Admission Control

The AC procedures proposed in Chapter 6 for both RT and NRT activities have been implemented in the SS process. An RT playback or recording request originates from an RT-PO, and its resource requirements are computed from the corresponding read or receive table into a *reserve table* using disk zone information and I/O parameters of the storage node. AC is then based on the reserve table and the cycle table. An admitted RT-PO is added to an *RT-PO table*, which is used to compute device related RT requests in each cycle at run-time.

It should be noted that both the RT-PO table and the cycle table in an SS node have stable sizes even when the system scales by increasing the number of SSs. A cycle table only needs to record entries for cycles up to the longest video, which normally is 2 hours, i.e., 7,200 entries are needed if the cycle length is one second. Although an RT-PO table records the read or receive table for each RT-PO, when more SSs are added, the size of the read or receive table of an RT-PO for each SS decreases proportionally because less blocks are stored in each SS. However, the number of RT-POs served by an SS increases proportionally at the same time, and thus the size of the RT-PO table in an SS would not vary very much.

A newly arrived network or disk NRT request from either the file or the tcp process has three destinations: the NRT waiting list if it fails the AC tests using the current and the next cycles' resources; a schedule in the current cycle if the request is admitted and the schedule has not finished; or directly to a driver if the request is admitted but the schedule has already finished. NRT requests on the waiting list will undergo AC test again in later cycles.

Device Related Request Scheduling

The SS process maintains three schedules for each of the three consecutive cycles starting from the current cycle. The three schedules are: a *net send schedule*, a *net receive schedule*, and a *disk schedule*. Each schedule records a list of device related RT and NRT requests to be sent to a device driver in the corresponding cycle. The requests are issued to a driver in the order they appear on a schedule. At current cycle c, the schedules for cycle $c + 1$ are

ready, while the schedules for cycle $c + 2$ are being computed. This makes it possible to admit newly arrived NRT requests using the next cycle's resources.

When computing a schedule, the RT-PO table is examined first to see if there is data to be read or received in that cycle. This is achieved by looking at the read or receive table for each admitted RT-PO. The corresponding RT read or receive requests are added to the relative schedules. Then the NRT list is examined and an NRT request is added to a schedule if it satisfies the AC test. When a disk request is added to the disk schedule, it is inserted according to SCAN order. Schedule computing is accomplished by a *schedule computing thread.*

The SS process implementation does not take the approach of using one thread for the data access of each RT-PO, which would introduce much contention and scheduling jitter. Instead, dedicated threads are used to process all RT and NRT requests to be sent to the device drivers. A *disk thread* is responsible for sending read and write requests in the disk schedule to the scsi/raid driver, while a *net send thread* sends send requests in the net send schedule to the atm driver. Because of the special nature of receive activities, a *net receive thread* is used to send all RT receive requests in the net receive schedule to the atm driver at the start of a cycle and wait for the replies. In contrast, a disk or send request is sent to the corresponding driver only after a previous one has finished.

Cycle Boundary Processing

Cycle boundary processing is performed by a *cycle boundary processing thread*, which deals with finished and unfinished requests in the current schedules and sets up the boundary for the next cycle according to the time which is constantly adjusted by the ntp process. Cycle overload occurs when there are unfinished requests at cycle boundaries.

During cycle boundary processing, unfinished RT requests are discarded, while finished RT send and write requests are deleted. Finished RT read requests are converted to RT send requests and are added to the net send schedule of the next cycle, and finished RT receive requests are converted to RT write requests and are inserted into the disk schedule of the next cycle according to SCAN order. These RT request conversions do not require additional AC, because it has already been taken into account during AC for the corresponding RT-POs as shown in Figure 6.2 and Figure 6.3. Unfinished NRT requests are added to the end of the next cycle's schedules if they were admitted using the current or next cycle's resources, and to the NRT list otherwise.

The cycle boundary processing thread will then start a new cycle by making the schedules of the next cycle the current ones and by activating the net send, the net receive, the disk, and the schedule computing threads if necessary. Concurrency control problems among the several threads dealing with RT requests are avoided because of the following solutions: the cycle boundary processing thread runs in critical section with interrupts disabled; the request processing threads have different current schedules; the schedule computing thread will compute schedules for the third cycle starting from the current; and AC for an RT playback or recording request starts from the fourth cycle.

7.3.3 Physical Object Management

PO management is concerned with how PO data is organised on a disk device, and it is implemented by the file process. Because the file process in the prototype is a modification from the one used in the original ATMos system, the following will only briefly describe disk organisation and PO representations.

A disk is partitioned into adjacent *regions* of equal size, and each region has its own bitmap to record the usage of the disk blocks in that region. When an extent is being allocated for a PO and the current region does not have enough free space, other regions are searched in an order corresponding to their distance from the current region, so that data of a PO is stored as close together as possible.

Each PO has a *header disk block*, corresponding to a UNIX inode, associated with it. A header block contains PO attributes, direct extent entries, and pointers that can be used to reach the single, double, and triple indirect disk blocks which are used to store more extent entries. Because of the voluminous nature of CM data, a disk block has a size of multiples of the disk sectors. This also has the benefit of enabling a header disk block to store hundreds of direct extent entries, which reduces the possibility and overhead to retrieve extra indirect blocks when opening a PO.

7.4 Resource Management

7.4.1 Contract Enforcement

Chapter 6 identified a list of contracts that are assumed by the AC criteria but should be implemented by QoS enforcement in an SS. Most contracts can find their implementation in the last section. The following will briefly describe or summarise the not-so-obvious implementation for some contracts. The label and content of each contract listed below are the same as those shown in Appendix B.

C-2 Regulate receive activities in each cycle.

The receive interrupt processing routine will count the time it spends from the start of a cycle using the free running counter of the profiler board. It will disable receive interrupt whenever Condition (6.17) is true.

C-4 Disk and network send activities (not including late arriving NRT activities) are known at the start of a cycle.

The disk and the net send schedules are known at the start of a cycle. Requests in these schedules will generate corresponding device related activities after being sent to the device drivers.

C-5 Use FCFS scheduling for the disk and network send activities.

Requests in the disk and the net send schedules are sent to the drivers according to the order they appear on the schedules. Also a newly admitted NRT request is appended to a current schedule if the latter has not finished.

C-7 RT pure CPU processing has a higher priority than the NRT one.

The SS process has a higher priority than other processes. The schedule computing, the net send, the disk, the net receive, and the cycle boundary processing threads have increasing priorities, and all of them have higher priorities than other threads in the SS process.

C-9 Either Condition (6.7) or Condition (6.8) should be true or made true.

There are two types of storage node in the prototype: one with a single disk using programmed I/O and the other with a RAID3 disk array using DMA for disk data transfer. Condition (6.7) is made true in both situations with the shared resource r as the CPU.

C-12 Reduce the cleverness of a disk device.

Some parameters of the disk drives used in the prototype have been adjusted by sending *Mode Select* SCSI commands to the drives. The following is a summary of the parameters, their values, and the meanings for the ST12550N disk used in the prototype.

Automatic Read Reallocation : 0 : the disk drive shall not automatically relocate bad blocks detected during read operations.

Enable Early Recovery : 0 : allows the disk drive to apply ECC correction as soon as possible, before the retry count is exhausted.

Read Retry Count : 0 : the maximum number of times the drive attempts its read recovery algorithm. In this setting, there are still 3 hidden retries.

Write Retry Count : 3 : the number of times the drive attempts its recovery algorithm during write operations. This value is not alterable for the ST12550N.

Connect Time Limit : 0 : the drive is allowed to remain connected indefinitely until it attempts disconnection. This is valid because each SCSI bus only attaches one drive in the prototype.

Read Cache Disable : 1 : the SCSI *Read* command must access the disk media.

Write Cache Enable : 1 : the SCSI *Write* command may return status and completion message bytes as soon as all data has been received from the host.

Caching Analysis Permitted : 0 : caching analysis is disabled.

Disable Prefetch Transfer Length : 0 : prefetch is disabled for any SCSI *Read* command whose requested transfer length exceeds this value.

Force Sequential Write : 1 : multiple block writes are to be transferred over the SCSI bus and written to the media in an ascending, sequential, logical block order. This prevents the drive from using its own scheduling method for write operations.

Disable Read-Ahead : 1 : the disk drive shall not read into the cache any logical blocks beyond the addressed logical block(s).

Number of Cache Segments : 1 : the number of segments into which the host requests the drive should divide the cache.

C-16 Detect and record zone information for ZBR disks.

The zone information of a SCSI disk can be obtained by issuing *Mode Select* SCSI commands with the *Notch and Partition Page*. "Notch" is another name for "Zone".

C-20 Measure the actual resource usage in the current cycle.

The receive, the send, and the disk interrupt processing routines will count the CPU time they spend from the start of a cycle using the free running counter. The time spent on the network can be deduced from the CPU time usage, which will be shown in Section 8.2.3.

C-22 Regulate disk and network send activities to prevent pure CPU processing from starvation.

The send and the disk interrupt processing routines will ask the net send and the disk threads not to issue any more requests to the device drivers whenever Condition (6.16) is true. If there are requests left on the corresponding schedules, then cycle overload occurs.

C-23 Re-activate receive activities after they have been disabled and when conditions allow.

The algorithm listed in Figure 6.9 is executed by the idle process in a storage node. To prevent race conditions, it is run in critical condition with interrupts disabled.

7.4.2 Resource Reclamation

In Figure 6.2, resources are reserved for retrieving backup blocks during fault mode. However, a video server may be operating in non-fault mode most of the time. Even when there are SS failures, some functioning SSs may not be affected if they do not store the backups of the primary blocks stored on the failed SSs. Thus it is desirable to reclaim some resources reserved for fault mode and to make them usable by NRT requests.

Assume the current cycle index is c, then the resources are reclaimed for cycle $c+2$ by the schedule computing thread. The reclamation is performed after computing the schedules using the RT-PO table but before admitting NRT requests from the NRT list. Figure 7.5 shows the fault mode resource reclamation procedure. The reclaimed resources can also be potentially used by NRT requests which will arrive at cycle $c+1$ or $c+2$.

```
/* A resource item has fields: buf, disk, CPU, and net. */
var: resource item: retained, reclaimed;
var: current cycle index: c; cycle table: ct[];

if there is SS failures
   && the failure of SS(x) affects this SS
 then
   retained = ct[c + 2].fail[SS(x)];
else
   retained = 0;
end if

reclaimed = ct[c + 2].fault − retained;

ct[c + 2].fault = retained;
ct[c + 2].res − = reclaimed;
```

Figure 7.5: Fault Mode Resource Reclamation

7.5 Summary

This chapter has described the implementation of a Cadmus prototype system. The implementation environment was based on distributed object computing and peripherals directly connected to an ATM network. A global clock was available in the entire system through the use of a GPS/NTP package. The important logical components of the Cadmus architecture were implemented, as well as some extra components used to support or extend the system. The software structure in storage nodes and the implementation of SSs and POs were presented and have the following features: NRT requests are re-directed to a single process for AC; dedicated threads are used to compute the schedules, to process all the I/O requests of the same category, and to deal with overload situations at cycle boundaries; the contracts identified in Chapter 6 have been enforced; and resources reserved for fault mode are reclaimed for use by NRT requests when conditions allow.

Chapter 8

Evaluation

This chapter evaluates the prototype implementation described in the last chapter through experiments and measurement. The experimental configuration is outlined, and parameters needed by the AC procedures are measured. Then performance results of NRT data transfer and multiple stream playback are presented. Finally, the performance bottleneck in one SS node is analysed.

8.1 Experimental Configuration

The experimental configuration is shown in Figure 8.1. Each networked physical entity has a name associated with it, such as *cirlbunting* or *jura*. Each of the SSs and the test clients was equipped with an ATMos card with a 32 MHz or 25 MHz ARM610 processor and 8 MB memory. Although Cadmus supports multi-chained declustering, only two SSs were available and were attached to the network, and it reduces to the chained setting. One SS (*cirlbunting*) used a Seagate ST12550N disk, while the other (*jura*) used a RAID3 disk array with 5 Conner CFP2105E disks. *Cirlbunting* used programmed I/O for both disk and network data transfer, while *jura* used DMA for disk read/write and programmed I/O for network activities. Although finding the optimal cycle length is outside the scope of this research, the experiments selected it to be 1 second for reasons of simplicity and to conform to the examples in previous chapters.

The test video sequences were recorded Medusa [**Wray94**] video clips. Although they are CBR, the test clips were treated exactly like VBR: a video index was generated for each video and was interpreted by an SCA; and read tables were sent to the SSs for AC. Each clip was 166 seconds long with 12 fps and a resolution of 88×64 pixels per frame, and had a size of 21.40 MB. Because the sequence was uncompressed, it had a rate of 1.03125 Mbps. Uncompressed versions of the videos were used because it was desirable to look at results on a video tile, which did not have a hardware MPEG decoder and was not capable of decoding MPEG in software in real time. 16 such video sequences were striped to the two SSs with a block size of 132 KB, which is the average size per second. Data was sent to the SSs using NRT file transfer. It was found at run-time that the file process in each SS node stored each PO, which was about 10 MB long, with 3 to 10 extents.

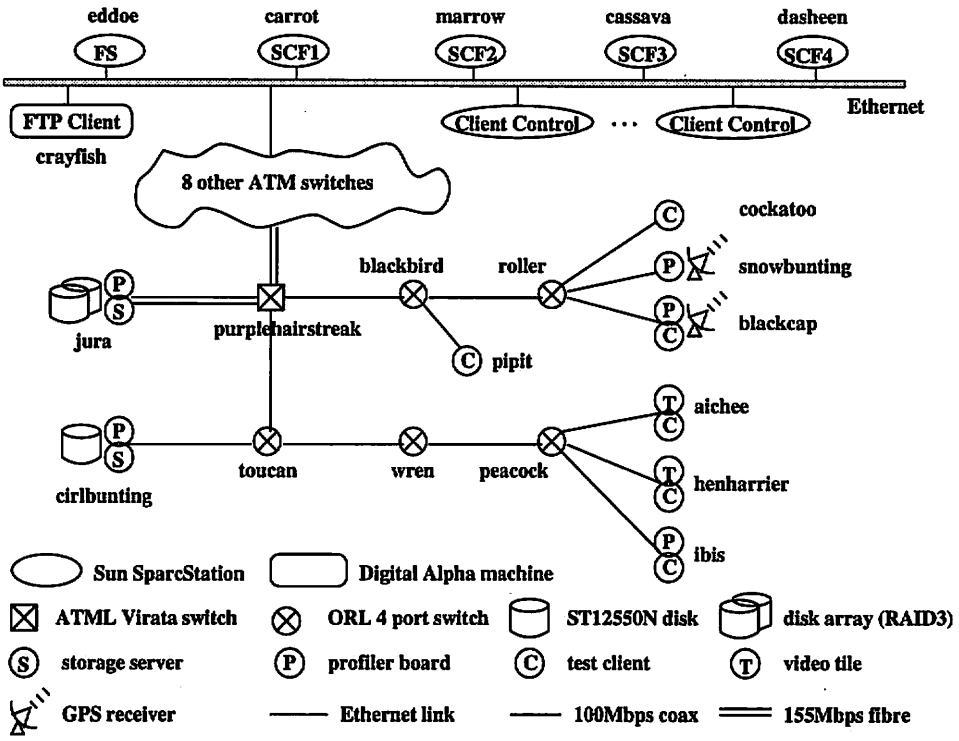

Figure 8.1: Equipment Deployment

8.2 Admission Control Parameters

This section will present values for parameters needed by the AC procedures in Chapter 6. These parameters include I/O bandwidth of the resources in an SS node on network and disk related activities. Because *jura* seemed to be a faster device than *cirlbunting*, only I/O parameters of the latter were tested and used in the AC procedures in both SSs. It was hoped that any workload *cirlbunting* had admitted could be undertaken easily by *jura* in each cycle. However, it turned out not to be the case, and the implications will be discussed later. The experiments in this section were performed using the original ATMos system but with a more accurate timer driver written by the author.

8.2.1 Network Side Parameters

The data path from the network interface to the memory in *cirlbunting* is shown in Figure 8.2. Two 4 KB FIFO buffers were used for send and receive activities. The 32-bit CPU-memory bus was clocked at 16 MHz, resulting in a transfer rate of 64 MB/s. The ATM hardware connection could support up to 100 Mbps. But as data send and receive were performed by software, the bandwidth of the CPU and the unshared network data path should be measured.

Figure 8.3 shows the results of sending AAL5 packets of different sizes from *cirlbunting* to *ibis* without intervals between individual send requests. The process percentage represents the CPU time occupied by activities other than the idle process or interrupt processing. The average rate was computed by dividing the size of the data sent by the iteration period. However, as there were idle time and other CPU processing activities, the effective rate for the CPU to send data was computed by dividing the size of the data sent by the aggregate time spent on the send interrupts. It can be observed that some parameters, such as the

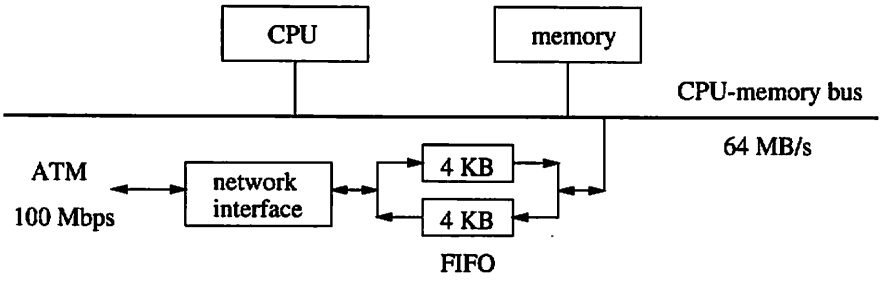

Figure 8.2: Network Side Data Path

effective rate, the average one cell processing time, and the number of cells processed by each send interrupt, are quite stable. This can be further verified from Figure 8.4, in which the average rate of sending 32 KB packets from *cirlbunting* to *ibis* was adjusted by inserting intervals between individual requests. All the parameters have included interrupt latency and messaging overhead if applicable.

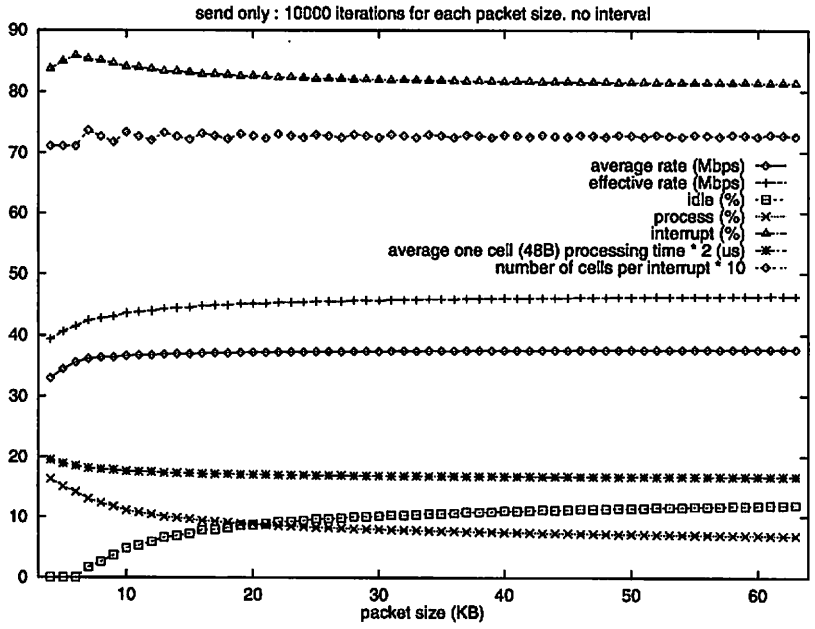

Figure 8.3: Network Send with Increasing Packet Size

In Figure 8.3, the average rate can be characterised as 37.19 ± 0.77 Mbps, where 37.19 Mbps is the mean and 0.77 Mbps the standard deviation. Similarly, the effective rate is 45.23 ± 1.43 Mbps. The discrepancy of the average and the effective rates can be explained by the different capacity on network send by the unshared network data path and the CPU. While the average rate is limited by the bandwidth of the unshared network data path, the effective rate is limited by the bandwidth of the CPU on network send activities. If this were not the case, then the percentage of the idle CPU time would not have increased and been approaching a stable level when the packet size increases in Figure 8.3.

Measurements have been performed for receiving in *cirlbunting* from two different sources: one from *blackcap* with a 25 MHz CPU, the other from *ibis* with a 32 MHz CPU. The results are shown in Figure 8.5 and Figure 8.6. To correctly receive data in each iteration in *cirlbunting*, the sources had to insert intervals between individual sends. However, the effective rate, the average cell processing time, and the number of cells per interrupt are also near constant in either case. The difference is that *cirlbunting* will process

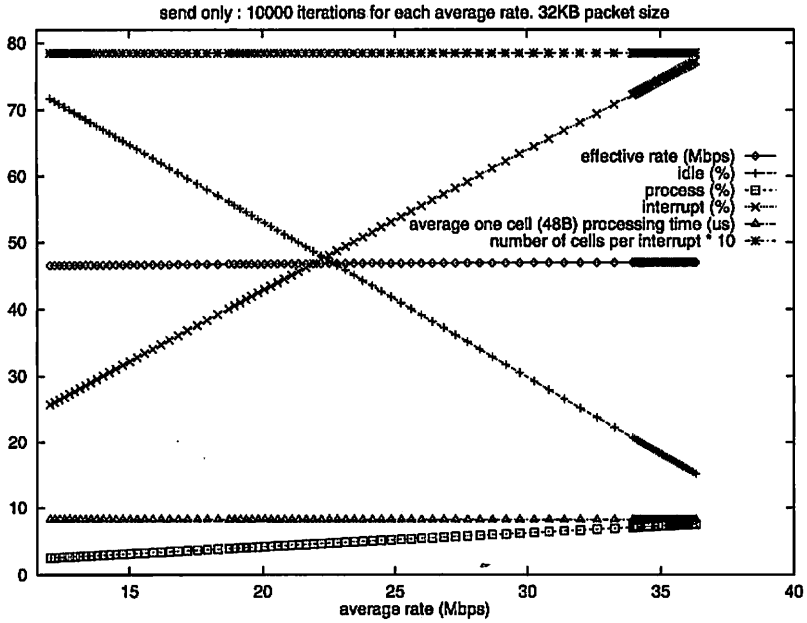

Figure 8.4: Network Send with Increasing Average Rate

more cells (21.68 ± 1.28 cells) per interrupt from the faster *ibis* than those (9.75 ± 0.12 cells) from the slower *blackcap*. This results in a higher effective receive rate (48.26 ± 1.92 Mbps) from *ibis* than that (44.66 ± 1.75 Mbps) from *blackcap* because of less processing overhead per cell. For simplicity reasons, the capacity of the unshared network data path and the CPU on receive activities were selected as the same as that on network send for use in the AC procedures.

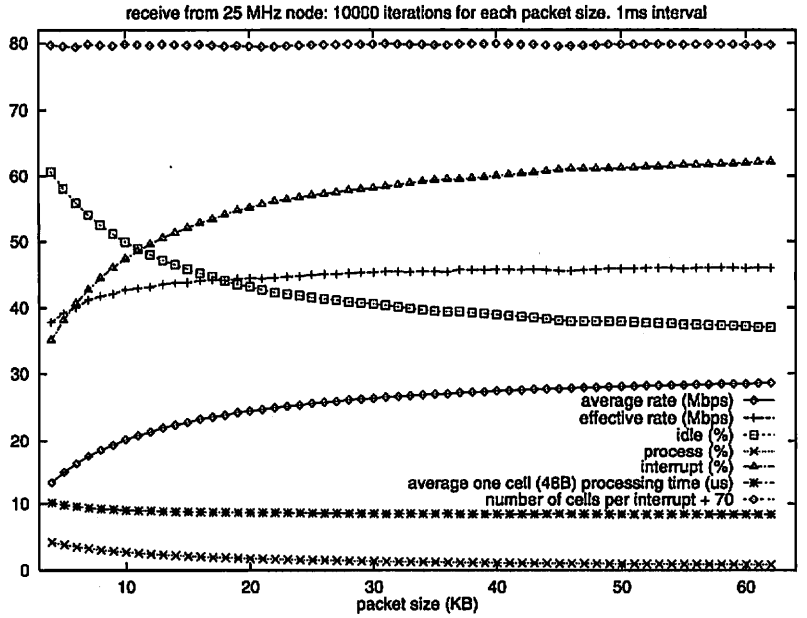

Figure 8.5: Network Receive from Slow Source

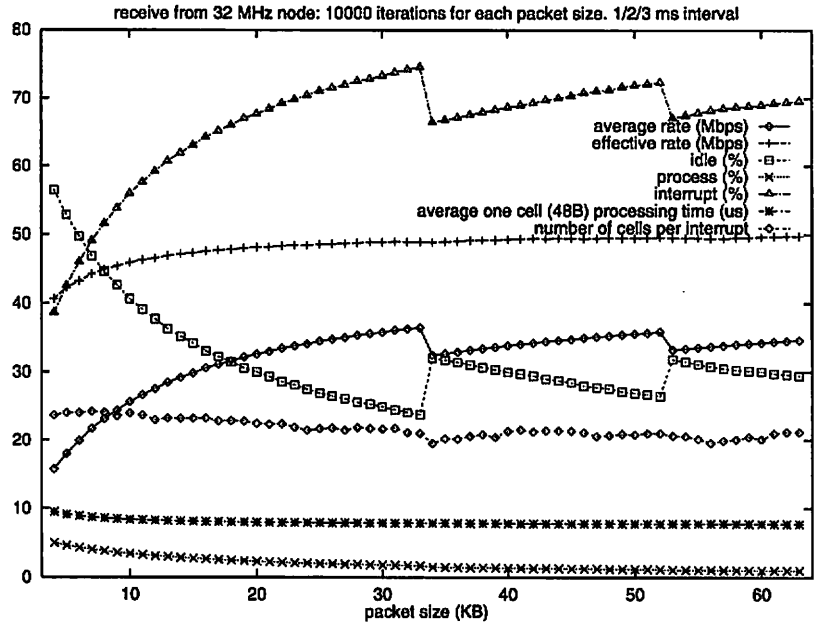

Figure 8.6: Network Receive from Fast Source

8.2.2 Disk Side Parameters

The data path from the disk to the memory in *cirlbunting* is shown in Figure 8.7. An XSI interface was used to connect the ATMos card with the SCSI bus, to which an ST12550N disk was attached. The SCSI-1 bus could support 4 MB/s, while the XSI bus could support 20 MB/s. The zoned disk had a transfer rate of 27.84–46.56 Mbps from its platter. The effective rate of the CPU on read/write data transfer had to be measured.

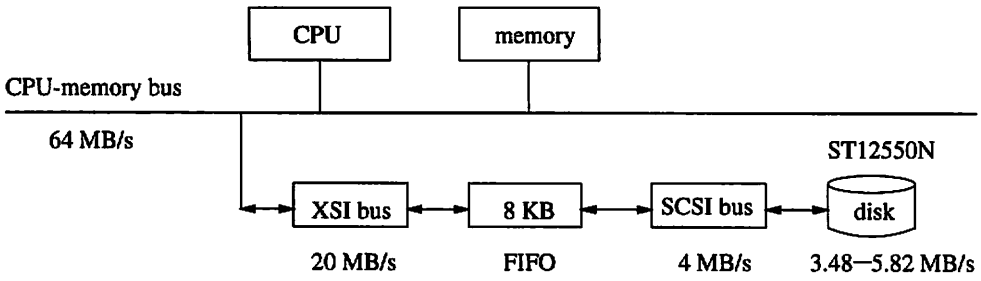

Figure 8.7: Disk Side Data Path

Figure 8.8 shows the results when increasing the size of read requests to the disk. Each iteration performed SCAN scheduling once. Similarly, the effective rate was computed using the aggregate time spent on the read interrupts, while the average rate on the whole iteration interval. However, the average rate cannot be used to characterise the bandwidth of the unshared disk data path. First, there is seek time that does not contribute to data transfer. Second, the disk has different transfer rates on different zones. Hence, the bandwidth of the XSI bus, the SCSI bus, and the disk itself combined to determine the capacity of the unshared disk data path.

The average one disk sector of 512 B processing time (135.17 ± 0.69 μs) and the effective rate (30.27 ± 0.13 Mbps) are also quite stable. This can be verified from Figure 8.9, in which the average rate for reading 128 KB data was adjusted. When the request size increases, the average rate is approaching the effective rate because of less disk seek overhead. The same

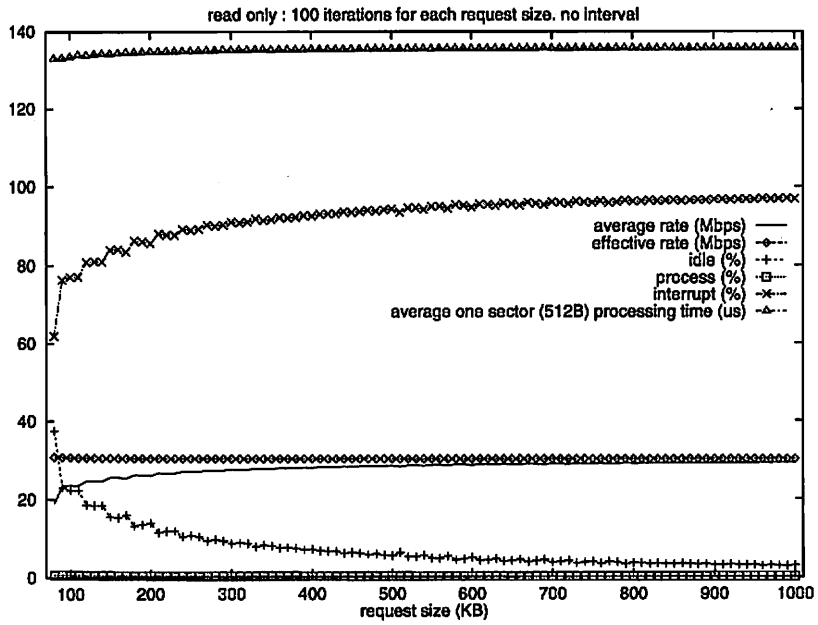

Figure 8.8: Disk Read with Increasing Request Size

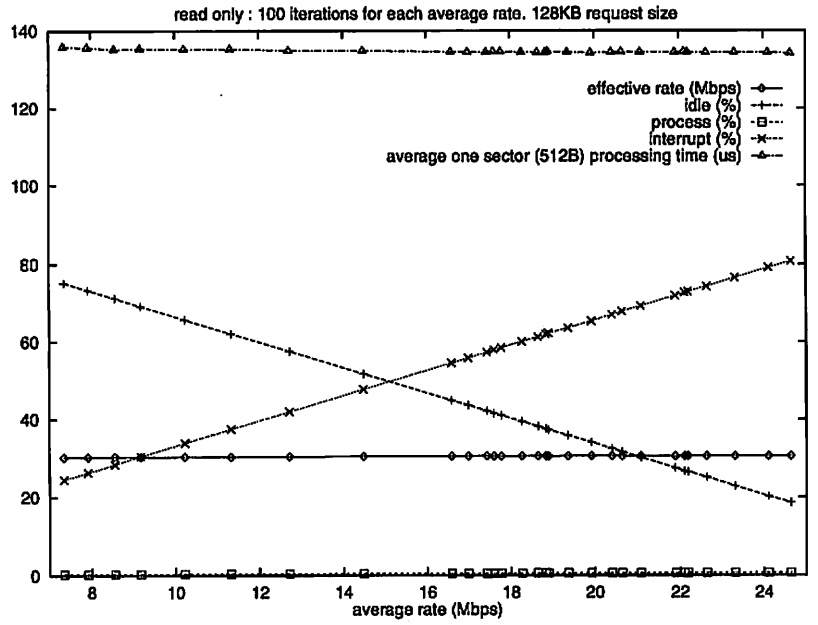

Figure 8.9: Disk Read with Increasing Average Rate

measurement was performed on write requests, and the results are shown in Figure 8.10 and Figure 8.11. Disk write has an average one disk sector processing time of 142.38 ± 3.32 μs and an effective rate of 28.78 ± 0.71 Mbps, both of which are less than that of disk read. To reduce the effective rate of read to that of write for use in AC would decrease server performance, because a video server will generally perform more reads than writes. Therefore, read and write had different effective rates in the AC procedures.

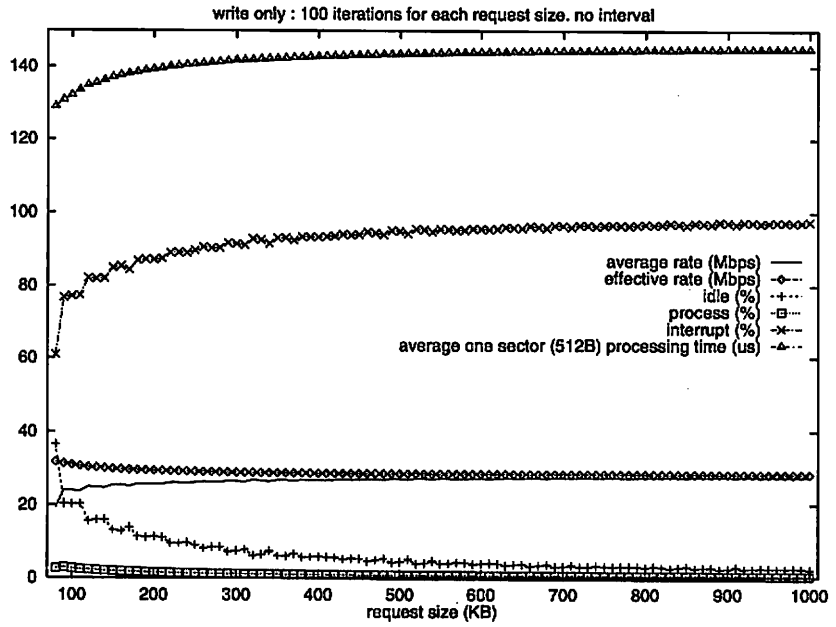

Figure 8.10: Disk Write with Increasing Request Size

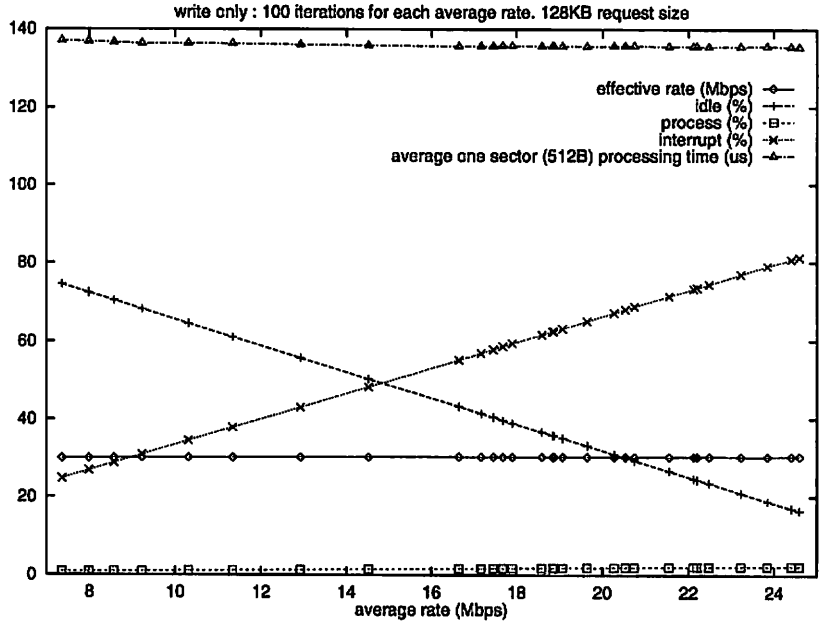

Figure 8.11: Disk Write with Increasing Average Rate

8.2.3 Parameters Used

Concurrent Activities

The previous measurement was based on activities of the same type without interference from activities of other types. Figure 8.12 shows the results of various numbers of threads doing both read and send. The parameters that were stable for single type activities are still stable when there are concurrent read and send activities. Particularly, the effective network send rate is 46.80 ± 0.01 Mbps, and the effective disk read rate is 30.52 ± 0.02 Mbps. Both of them are very near to the ones measured from Figure 8.3 and Figure 8.8. The average rate is 17.48 ± 0.38 Mbps.

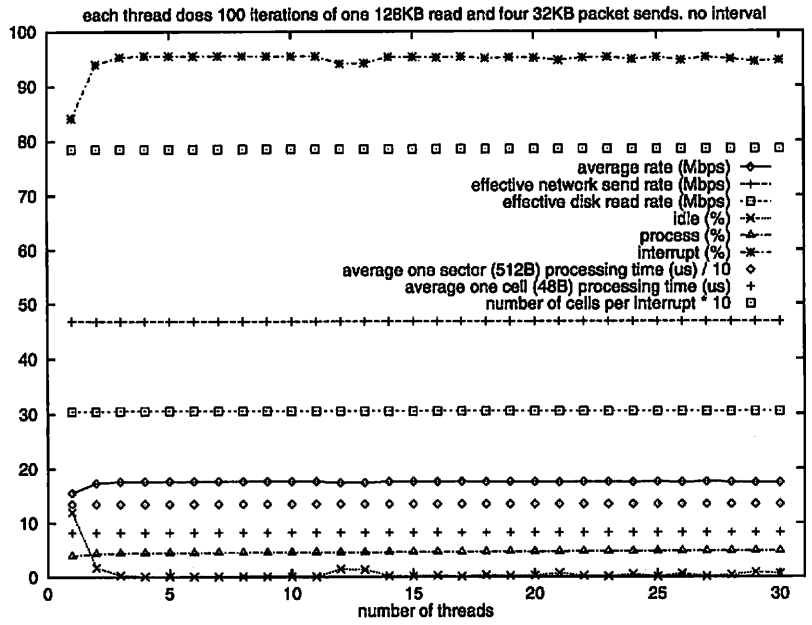

Figure 8.12: Concurrent Disk Read and Network Send

Parameter Analysis

The results from Figure 8.3, Figure 8.8, and Figure 8.10 were used in the AC procedures. This is because they were measured using different packet or request sizes and also conform to the results from Figure 8.12. Table 8.1 summarises the capacity of the unshared data paths and the CPU on the various types of activities.

Table 8.1: Resource Bandwidth on Different Activities

activity	bandwidth (Mbps)			
	unshared data path		CPU	
	measured rate	lower bound	effective rate	lower bound
send	37.19 ± 0.77	36.42	45.23 ± 1.43	43.80
receive	37.19 ± 0.77	36.42	45.23 ± 1.43	43.80
read	XSI: 80 SCSI: 32		30.27 ± 0.13	30.14
write	disk: 27.84–46.56		28.78 ± 0.71	28.07

Contract 9 from Section 6.1.4 required either Condition (6.7) or Condition (6.8) should be true or made true. The shared CPU-memory bus clearly is not a bottleneck and was ignored. However, Condition (6.8) will not hold for the above parameters when considering the shared CPU resource. This is because:

$$\frac{36.42 \text{ Mbps}}{43.80 \text{ Mbps}} + \frac{27.84 \text{ Mbps}}{30.14 \text{ Mbps}} = 1.76 > 1$$
$$\frac{36.42 \text{ Mbps}}{43.80 \text{ Mbps}} + \frac{27.84 \text{ Mbps}}{28.07 \text{ Mbps}} = 1.82 > 1$$

Therefore, Condition (6.7) was made true as shown below. Assume a network send request p and a disk request q have data size $p.size$ and $q.size$ respectively. Then:

$$p.send\_CPU = p.send\_net = \frac{p.size}{36.42 \text{ Mbps}} \tag{8.1}$$

And from Equation (6.14), if the disk request q is optimised to access data on one contiguous piece, and $z(q)$ refers to the zone of the piece, then:

$$\begin{aligned} q.disk\_CPU &= q.disk\_xfer \\ &= \frac{q.size}{min(B(z(q)), Bmin(\text{XSI, SCSI, disk}), Bmin(CPU\_d))} \\ &= \frac{q.size}{min(B(z(q)), Bmin(\text{XSI, SCSI}), Bmin(CPU\_d))} \\ &= \frac{q.size}{min(B(z(q)), 32 \text{ Mbps}, Bmin(CPU\_d))} \\ &= \frac{q.size}{min(B(z(q)), Bmin(CPU\_d))} \end{aligned} \tag{8.2}$$

where $Bmin(CPU\_d)$ is 30.14 Mbps for a read request and 28.07 Mbps for a write request. Note that $Bmin(CPU\_d) < 32$ Mbps, which is the bandwidth of the SCSI bus.

However, for a receive request g with data size $g.size$, if the time required on the network data path and on the CPU are $g.recv\_net$ and $g.recv\_CPU$ respectively, then:

$$g.recv\_net = \frac{g.size}{36.42 \text{ Mbps}} \tag{8.3}$$

$$g.recv\_CPU = \frac{g.size}{43.80 \text{ Mbps}} \tag{8.4}$$

$$g.recv\_net = 1.20 * g.recv\_CPU \tag{8.5}$$

Equation (8.5) was used in Figure 6.9 to convert between the time spent on the network data path and the time on the CPU for any receive activity.

Other Parameters

Table 8.2 is a summary of other AC parameters and their values which have been used. The meanings of the parameters are as shown in Chapter 6. The zone parameters needed in Equation (6.13) and Equation (8.2) can be deduced from the disk rotation speed (7,200 rpm) and the disk geometry shown in Table 2.3. Although 8 MB memory was installed in *cirlbunting*, only 2112 KB was available for the SS process to use as the K-buffers because of the space occupied by the image and used up by other processes. Some of these parameters, as well as the bandwidth related ones measured above, may be changed dynamically according to run-time requirements and measurement. However, this was not implemented in the prototype and all the parameters were fixed in the AC procedures.

Table 8.2: Other AC Parameters and Values

parameter	value	parameter	value
$T_{NRTrecv\_net}$	60 ms	a	6588 μs
$T_{NRTrecv\_CPU}$	50 ms	bA	13911 μs
T_{pure_CPU}	80 ms	T_{rot}	8334 μs
T_{res_CPU}	20 ms	$T_{extra\_recv\_CPU\_threshold}$	5 ms
K	1056 KB	$T_{extra\_recv\_CPU}$	50 ms

The disk seek time profile constants, i.e., a and b (cf. page 44), were computed from Figure 8.13, which shows the write seek time profile for the ST12550N disk. Each seek

time for a specific seek distance was the average of 30 write seeks on different places of the disk platter. The profile was approximated using standard 2-parameter linear regression [**Press88**], and the result is $y = 6.58734 + 0.00514065 * x$ ms. So $a = 6588$ μs, and $bA = 0.00514065 * 2706$ ms $= 13911$ μs. T_{rot} is a full rotation time, which is 8.333 ms for a rotation speed of 7,200 rpm.

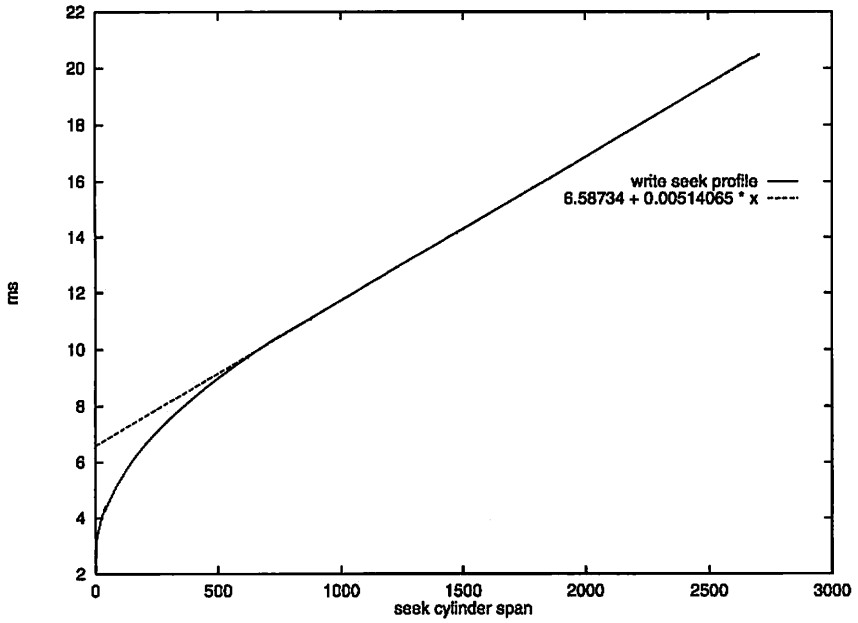

Figure 8.13: ST12550N Write Seek Time Profile and Approximation

8.3 Measured Results

This section presents results from experiments performed when the Cadmus components were up and running in the physical entities as shown in Figure 8.1. The AC parameters used were as described in the last section. Because fault mode operations and RT recording were only partly implemented, they were not tested.

8.3.1 File Transfer

Transferring files to an SS using the FTP protocol will test the SS's NRT processing capability. Experiments were done to send 10 MB files from *crayfish*, a DEC Alpha machine running Digital UNIX, to *cirlbunting*. Before the three-level AC for NRT activities was developed, only delayed AC was implemented. The resulting transfer rate was merely 0.29 KB/s. After implementing the three-level AC, the rate increased dramatically to around 480 KB/s. Similar rates were achieved by sending the same files from *crayfish* to *jura*. This performance is reasonably high because 480 KB/s was also the average rate of transferring data from *crayfish* to an ATMos file server equipped with a RAID3 disk array with 5 ST12550N disks.

File transfer when there were RT activities was also tested. When there were 2 active playback streams of 1.03125 Mbps, with each SS serving one stream in each cycle, the rate of transferring a 10 MB file from *crayfish* to *cirlbunting* became 410 KB/s. When there were 4 playback streams present, with each SS serving 2 streams per cycle, the transfer rate dropped to 210 KB/s. This clearly shows that RT activities were delaying the NRT ones in SS nodes.

8.3.2 Multiple Stream Playback

With fault mode resource reservation turned off, the system could support up to 16 Medusa videos of 1.03125 Mbps, and the bottleneck was the 2112 KB buffer in *cirlbunting*. The following will present the results of playing back 14 streams, because the effect of serving different streams per cycle by an SS can then be shown (8 in one cycle and 6 in the other; while in the 16 streams case, each cycle will serve 8 streams). The experiment was done by first playing 1 stream in the test client *ibis*, which would time-stamp and discard each incoming 132 KB block. Then 2 streams were sent to the two video tiles: *aichee* and *henharrier*, for visual testing. Finally, 11 other streams were admitted and sent to the other three test clients: *pipit*, *cockatoo*, and *blackcap*, which would just discard any data received. All four SCFs were active and were employed in this experiment. Special test units and unit factories were used in the test clients.

Figure 8.14 shows the CPU usage in *cirlbunting*. These are the results of several runs of the experiment when 8 and 6 streams were read in each cycle alternately. These cycles were divided into two groups according to the number of 132 KB blocks read in each cycle. It could be seen in both cases read and send time are quite stable, and there is also much idle CPU time left. In some cycles there is less idle time, because there were background NRT activities using the CPU resource as well. Cycle boundary processing would take about 200 μs when there were no RT activities and no cycle overload. But an additional 100 μs was needed for each 132 KB RT block read in the previous cycle, because it was the responsibility of the cycle boundary processing thread to convert an RT read request into several 32 KB AAL5 packet send requests in the prototype implementation.

Figure 8.15 shows the 132 KB block arrival time in *ibis* relative to the start of a GPS second. It is the concatenation of 9 runs, with each run covering the time period of admitting and playing the other 13 streams. It could be noted that no blocks arrive outside the 1 second cycle length starting from a GPS second. Also the arrival pattern is hard to predict, although in general the arrival time increases when more streams are admitted and being played. This is because blocks were sent from an SS according to the retrieval order from the disk schedule in the previous cycle. With multiple streams active, and each stream's data being stored in different places in the disks, the retrieval order of multiple streams may not be preserved between cycles when SCAN is used. In all these runs, the displays of the two video tiles were observed, and no disruption was noticed.

8.3.3 Discussion

Something which was not expected was that sometimes network send overload would occur in *jura*, the SS node with a RAID3 disk array and DMA for disk data transfer, if 8 streams were sent in one cycle. These runs have not been taken into account in the above results. In contrast, such overload *never* happened to *cirlbunting*. This is probably because *jura* had a different atm driver which would delay network send or message passing. Because of time limitations, this was not further investigated. But two things could be observed: one is that a seemingly fast device might not guarantee QoS if its I/O parameters were not obtained correctly; the other is that the system was still stable during overload. When occasional cycle overload occurred in *jura*, pictures in the video tiles either suffered a temporary glitch or were not disrupted at all because the RT data discarded in *jura* was not to be sent to the tiles.

The state of live-lock was also tested. A client was put in *ibis* which would loop to send 32 KB packets to *cirlbunting* without any interval between individual sends. When receive

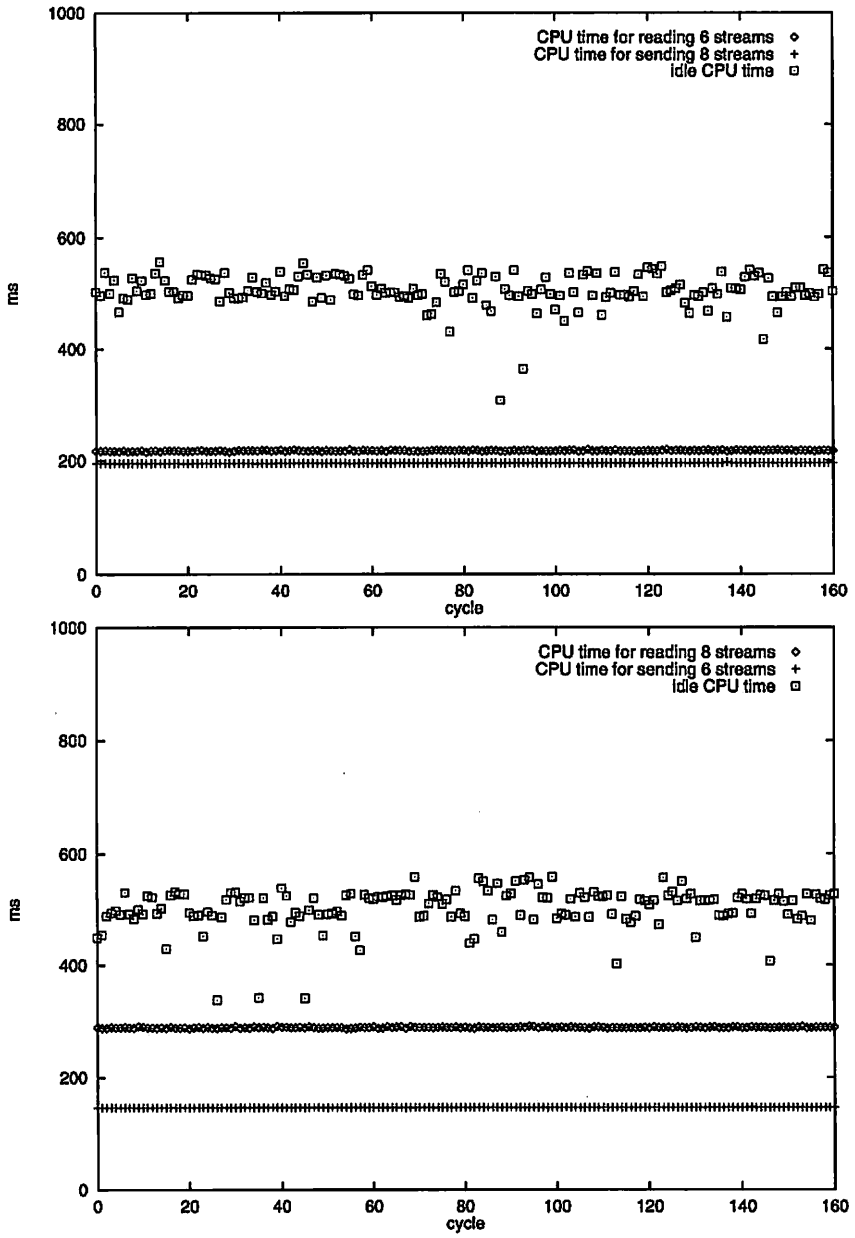

Figure 8.14: SS Node CPU Usage (*cirlbunting*)

was not regulated, whenever such a client was active, the console of *cirlbunting* just hung. Either the client in *ibis* had to be killed or *cirlbunting* had to be reset to recover from such a live-lock state. After receive was regulated, such a state did not exist any more.

8.4 Bottleneck Analysis

The upper limit of 16 streams of the 1.03125 Mbps Medusa video was constrained by the buffer size in *cirlbunting*. This can be simply analysed based on the AC criteria from Chapter 6 and the parameters from Section 8.2.3. Assume in each cycle *cirlbunting* can support up to $J.net$, $J.disk$, $J.CPU$, and $J.buf$ streams of the 132 KB/s video without reserving resources for fault mode, and these numbers are solely limited by the bandwidth of the corresponding single resource. Also assume there are no RT receive activities. Then,

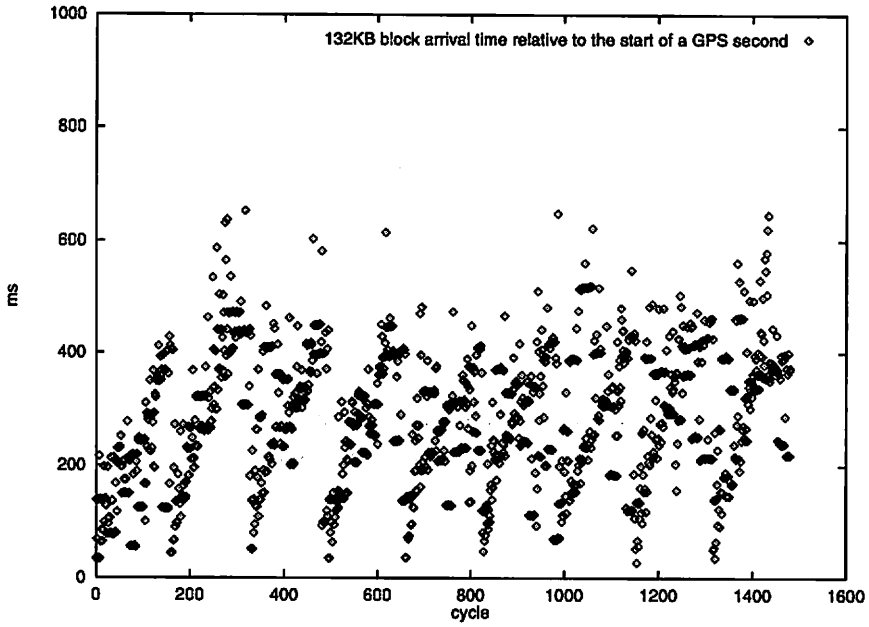

Figure 8.15: Block Arrival Time in the Client (*ibis*)

from Equation (6.1) and Equation (8.1):

$$\frac{J.net * 132 \text{ KB}}{36.42 \text{ Mbps}} = 1 \text{ second} - 20 \text{ ms} - 60 \text{ ms} \tag{8.6}$$

$$J.net = 32.50 \tag{8.7}$$

Assume each 132 KB block is stored contiguously on the disk, which was true in the previous experiments, and the blocks are stored in the slowest zone without crossing track or cylinder boundaries. Then from Equation (6.13), Equation (8.2), and Equation (6.15):

$$J.disk * (6588\ \mu s + 8334\ \mu s) + \frac{J.disk * 132 \text{ KB}}{27.84 \text{ Mbps}} = 1 \text{ second} - 20 \text{ ms} - 13911\ \mu s - 50 \text{ ms} \tag{8.8}$$

$$J.disk = 17.63 \tag{8.9}$$

From Equation (8.1), Equation (8.2), and Equation (6.3):

$$\frac{J.CPU * 132 \text{ KB}}{27.84 \text{ Mbps}} + \frac{J.CPU * 132 \text{ KB}}{36.42 \text{ Mbps}} = 1 \text{ second} - 20 \text{ ms} - 80 \text{ ms} - 50 \text{ ms} \tag{8.10}$$

$$J.CPU = 13.00 \tag{8.11}$$

From Equation (6.9):

$$J.buf * 132 \text{ KB} = 1056 \text{ KB} \tag{8.12}$$

$$J.buf = 8.00 \tag{8.13}$$

Therefore:

$$J.buf < J.CPU < J.disk < J.net \tag{8.14}$$

If the memory of *cirlbunting* could be increased, the system should be able to support 26 streams of 1.03125 Mbps. Then the bottleneck would become the CPU of *cirlbunting*. If then a faster CPU is used, the disk may be the next bottleneck, although the above equations should be re-computed using new bandwidth parameters. Another simple observation is that when one SS in the prototype could support only 8 streams per cycle, two SSs could indeed double that capacity to 16 streams when resources for fault mode were not reserved.

8.5 Summary

This chapter performed an evaluation on the Cadmus prototype implementation. Because the AC criteria from Chapter 6 rely on stable bandwidth of the various resources in an SS node on different activities, several experiments have been undertaken to measure these I/O parameters. With the experimental configuration outlined, the 2 SS node prototype could support the playback of up to 16 streams of 1.03125 Mbps video without reserving resources for fault mode. The performance results for NRT file transfer and 14 stream playback were presented. The implications of an anomaly were discussed and the live-lock state elimination was tested. Finally, the bottleneck of a storage node was quantitatively analysed using the AC criteria and measured parameters.

Chapter 9

Conclusion

This dissertation has presented an architecture for scalable and deterministic video servers. This chapter will summarise the work and its conclusions, and make suggestions for further study.

9.1 Summary

The background information from Chapter 2 identified the characteristics of CM data, storage devices, and the basic video server structure. These characteristics have several implications on video server design. First, a large scale video server has to be made up of a number of comparatively slow disk devices and a certain interconnect structure. It was argued that a network striped video server architecture with off-the-shelf components connected by a scalable switched network has the benefits of cost-effectiveness, flexibility, scalability, and high-performance. Second, the temporal property of CM data and the mechanical nature of disk devices require a video server to support AC, QoS, and overload processing. Third, the stored nature of CM data makes it possible for a video server to smooth VBR videos for playback, as well as to provide deterministic services.

It is the observation of this dissertation that the software system architecture for network striped video servers is of equal importance to the scalable hardware structure itself. However, most contemporary distributed file systems do not support network striping, let alone performing AC and guaranteeing QoS at the same time. Although there are many research papers on video server design, which were surveyed in Chapter 3, most of them are only concerned with single storage node video servers and are concentrated on a specific aspect, rather than tackling design problems in an integrated way at the system level. Some related proposals built on network striping were described in more detail. Nevertheless, a close examination reveals that they are either not scalable or not flexible, or lack QoS awareness. Motivated by all these observations, a new system architecture (Cadmus) for scalable and deterministic video servers was developed and presented in Chapter 4.

The Cadmus architecture is built on the concept of logical components manipulating data objects opened in different modes. POs are uninterpreted byte streams stored in a network attached storage node, while LOs, such as files and directories, are uninterpreted byte streams that are meaningful to clients. Usually a PO represents the data striped on one storage node from an LO. Both objects can be opened in either RT mode for playback or recording, or NRT mode otherwise. An SS exists in each storage node to manage POs,

while FSs are used to manage LOs. The data of an LO is scattered and gathered by a CP. In addition, SCFs and SCAs are introduced to control the playback or recording of RT streams. Data is transferred between SSs and CPs directly. Finally, DSs and NSs make the Cadmus architecture self-contained.

The main features of the Cadmus architecture are its flexibility and scalability: mappings from LOs to SSs are not fixed; each SS has its own policy to store the POs it manages; different logical components are largely independent of each other and can be added incrementally; and all meta-data is stored on SSs, enabling other logical components to run anywhere in the network. The deterministic aspect of the architecture is reflected by the adoption of the cycle based QoS guarantee scheme, which processes data according to fixed time length cycles in both SSs and CPs. Video indexes are used for VBR playback so that the amount of data to retrieve in each cycle is known a priori. This also makes Cadmus independent of any video encoding format. Data availability is provided by the adoption of multi-chained declustering, which is deemed superior to other replication based alternatives.

Not only does the Cadmus architecture provide mechanisms to exploit VBR smoothing in SCAs, this dissertation also presented and analysed a new smoothing algorithm in Chapter 5. The algorithm takes into account client buffer limits, performs or suppresses read-ahead, smoothes the data read in each cycle, and eliminates small reads to increase disk bandwidth utilisation. Based on empirical analysis of the algorithm, heuristics were given to select the SUS of an LO using very simple video characteristics. An interesting observation is that some VBR videos may not be that bursty if they are retrieved slightly above their average rates.

The Cadmus architecture is also QoS aware by virtue of its AC criteria and the associated contracts to be fulfilled by QoS enforcement in SS nodes, both of which were described in Chapter 6. The CPU in an SS node is modelled as a shared resource for both disk and network activities. The solution is integrated in that different resources in SS nodes are considered simultaneously and interference between different activities is identified and factored into the AC criteria. AC procedures for both RT and NRT activities were presented, with emphasis on a three-level AC for NRT activities in order to speed up their execution. Other issues of practical value, such as ZBR, live-lock state elimination, pure CPU processing starvation, and receive regulation and re-activation, were also considered.

A prototype implementation of the Cadmus architecture that can support NRT file transfer and multiple stream playback is operational and was described in Chapter 7. A global clock was provided by a GPS/NTP package, and the important logical components were implemented. The software in a storage node was built around a message-based micro-kernel that had been modified to conform to requirements of the AC criteria. A single process was used to act as a monitor for both RT and NRT activities, so that they were subject to AC before being issued to the devices. Dedicated threads were used for activities of the same type, and I/O scheduling was performed on a cycle basis.

The prototype implementation was evaluated in Chapter 8 through experiments and measurement. AC parameters in an SS node were measured and analysed in great detail, because they form the basis for the correctness of the AC criteria and QoS enforcement. Performance results for file transfer and multiple stream playback were presented, and potential bottlenecks in a storage node were analysed using the AC parameters and criteria. Without reserving resources for fault tolerance, the 2 SS node prototype could support up to 16 streams of 1.03125 Mbps video, with each SS node being able to serve up to 8 streams in each cycle. The functioning implementation and its evaluation have verified the feasibility and utility of the Cadmus architecture and its AC criteria.

During the process of implementing and evaluating the Cadmus prototype, the following observations were made:

- It is very hard to provide QoS in the server end without considering the software and hardware environment in storage nodes, such as the kernel, the file system, the communication subsystem, the device drivers, the actual devices and their performance parameters.
- It is also very difficult for QoS enforcement without a good timer device and driver. The original ATMos card only had a very inaccurate timer with a 10 ms resolution, which impeded every effort to guarantee QoS in SS nodes. In the later stage, the add-on profiler board was found very helpful and a new timer driver was written for it.
- Using the cycle based scheme, if each striping unit can be stored contiguously in a disk and SCAN is used, it is doubtful whether any more complex mechanism for laying out video data and scheduling disk heads will actually improve the performance of the system.
- In a dedicated server node, the design could be simplified compared to a workstation environment where multiple users which could not be trusted share resources. The results are less data copying, fewer concurrency issues, no problems with resource protection between different domains, and simple QoS regulation and enforcement.
- However, it is also felt that video server design is in general a very complex problem if the implementation of a functioning system is involved. This is because not as many assumptions could be made in this situation.

In conclusion, this dissertation has proposed a flexible and scalable architectural framework for the construction of scalable and deterministic video servers. Related issues such as VBR smoothing and integrated AC and QoS enforcement were also considered and proper solutions provided. A functioning prototype was built and evaluated, and the utility and feasibility of the ideas were verified.

9.2 Future Work

The Cadmus prototype was only implemented and tested to the extent that time and resources allowed. Some components, such as NSs and DSs, and some functionalities, such as VCR control and RT recording, were not implemented; while some aspects, such as fault mode activation and operations, were not tested. It may also be desirable to run the prototype on more storage nodes if they are available. The prototype implementation may be completed in future work, but the architectural framework should stand.

Although video data availability was provided by multi-chained declustering, more work is needed on failure recovery issues. These include failure detection, component recovery, and lost data reconstruction. Also performance implications of re-striping an LO or reconstructing a lost SS in the presence of RT services need to be investigated. However, Cadmus does provide mechanisms such as background file transfer and RT recording on which data recovery functionalities can be built.

Because an LO in Cadmus can be striped on any subset of SSs and is managed by only one FS, algorithms have to be developed to determine where to stripe an LO and when

to migrate or replicate an LO to a different set of SSs in order to achieve load balancing or to increase the number of streams that can be served from the LO. Also SSs may be partitioned to store different types of LOs in order to reduce interference between RT and NRT services.

The Cadmus architecture can also be used to support personalisation and mobile applications which may negotiate for different types of QoS most suited for the application environment at different times. This can be achieved either by SCAs and CPs which understand the various types of QoS and process the requests similarly to VCR commands, or by extra components that can filter the data from a Cadmus server. The data processing pipe unit from Section 7.2.3 is a good candidate for the latter.

Finally, the weakness of this research is its assumption that the network is not the bottleneck, which may not be true for a large scale video server. Further work is required to consider AC and QoS enforcement in switched networks and to extend the Cadmus architecture to reflect these considerations. Viable ways may be to include switches as one of the types of physical entities and to add QoS aware logical components in the switches.

Appendix A

Proof of Theorems

This appendix gives proofs of the lemmas, theorems, and corollaries listed in Section 5.4.1. For easy of reference, the smoothing algorithm from Section 5.3 is repeated below.

A.1 The Smoothing Algorithm

---

input: $D(c)$ $c \in [1, e]$; r, h, M, CB;
output: $R(c)$ $c \in [0, e-1]$;

$D(0) = D(-1) = R(-1) = B(-1) = 0$; $SF = r * M$;

for $(c = 0;\ c \leq e-1;\ c++)$ {
$B(c) = R(c-1) + B(c-1) - D(c)$;
$U(c) = CB - D(c) - B(c)$;
$L(c) = D(c+1) - B(c)$;
if $L(c) > U(c)$ { algorithm failed; client needs more buffer; exit; }

ⓐ **else if** $B(c) > \sum_{i=1}^{h} D(c+i)$ $R(c) = 0$; /* h-step look-ahead suppressor */
ⓑ **else if** $0 \leq SF < L(c) \leq U(c)$ $R(c) = L(c)$; /* to prevent client starvation */
ⓒ **else if** $L(c) \leq SF \leq U(c)$ $R(c) = SF$; /* smoother */
ⓓ **else if** $L(c) \leq U(c) < SF$ $R(c) = U(c)$; /* to prevent client buffer overflow */
}

A.2 The Proofs

Theorem 1. *The algorithm succeeds if and only if* $\forall c \in [0, e-1] \quad CB \geq D(c) + D(c+1)$.

Proof.
$$\begin{aligned}
&\forall c \in [0, e-1] \quad CB \geq D(c) + D(c+1) \\
\iff &\forall c \in [0, e-1] \quad CB - D(c) - B(c) \geq D(c) + D(c+1) - D(c) - B(c) \\
\iff &\forall c \in [0, e-1] \quad U(c) \geq L(c) \\
\iff &\forall c \in [0, e-1] \quad \text{The algorithm will succeed.}
\end{aligned}$$

□

Corollary 1. *The algorithm will succeed if and only if* $CB \geq V$.

Proof. "$\longleftarrow$": $CB \geq V$
$\Longrightarrow \forall c \in [0, e-1] \quad CB \geq V \geq D(c) + D(c+1)$
$\Longrightarrow$ The algorithm will succeed by Theorem 1.
"$\longrightarrow$": Because $\exists c \in [0, e-1] \ D(c) + D(c+1) = V$, then if $CB < V$
$\Longrightarrow \exists c \in [0, e-1] \quad CB < D(c) + D(c+1)$
$\Longrightarrow$ The algorithm will fail by Theorem 1.
$\Longrightarrow$ If the algorithm will succeed, then $CB \geq V$.

□

Corollary 2. *The algorithm will succeed if* $CB \geq 2 * P$.

Proof. $CB \geq 2 * P$
$\Longrightarrow CB \geq V$
$\Longrightarrow$ The algorithm will succeed by Corollary 1.

□

Theorem 2. $\forall c \in [0, e-1] \quad 0 \leq B(c) \leq CB, \ 0 \leq U(c) \leq CB, \ 0 \leq R(c) \leq U(c)$.

Proof. (i). When $c = 0$
$0 \leq B(0) = 0 \leq CB$
$0 \leq U(0) = CB \leq CB$

If the (control) flow goes through branches @ⓑⓒ, then $R(0) \geq 0$; otherwise branch @ makes $R(0) = U(0) \geq 0$.

(ii). Assume for any $c \in [0, e-2]$, $0 \leq B(c) \leq CB$, $0 \leq U(c) \leq CB$, $0 \leq R(c)$. We prove $0 \leq B(c+1) \leq CB$, $0 \leq U(c+1) \leq CB$, $0 \leq R(c+1)$ at cycle $c+1$ when the flow goes though branches @ⓑⓒ@ in cycle c. The symbol $c.@$ represents that the flow goes through branch @ at cycle c, and $c.@ⓑ$ means the flow goes through either @ or ⓑ at cycle c.

▶ $c.@$:

$R(c) = 0$
$\Longrightarrow B(c+1) \quad = R(c) + B(c) - D(c+1)$
$= B(c) - D(c+1) \leq B(c) \leq CB$

$B(c) > \sum_{i=1}^{h} D(c+i)$
$\Longrightarrow B(c) > D(c+1)$
$\Longrightarrow B(c+1) = B(c) - D(c+1) \geq 0$

Therefore, $0 \leq B(c+1) \leq CB$.

$U(c+1) \quad = CB - D(c+1) - (R(c) + B(c) - D(c+1))$
$= CB - R(c) - B(c)$
$= CB - B(c)$

Because $0 \leq B(c) \leq CB, \ 0 \leq U(c+1) \leq CB$

At cycle $c+1$, if the flow goes through branches @ⓑⓒ, then $R(c+1) \geq 0$; otherwise branch @ makes $R(c+1) = U(c+1) \geq 0$.

▶ $c.ⓑ$:

$R(c) = D(c+1) - B(c)$
$\Longrightarrow B(c+1) = R(c) + B(c) - D(c+1) = 0$
$\Longrightarrow 0 \leq B(c+1) \leq CB$

$$\begin{aligned} U(c+1) &= CB - D(c+1) - B(c+1) \\ &= CB - D(c+1) \\ \Longrightarrow & 0 \le U(c+1) \le CB \end{aligned}$$

This is because we assume the algorithm will always succeed, i.e., $\forall c \in [0, e-1] \quad CB \geq D(c) + D(c+1)$. This condition will not be reiterated and will be assumed as default in the following proofs.

At cycle $c+1$, if the flow goes through branches @ⓑⓒ, then $R(c+1) \geq 0$; otherwise branch @ makes $R(c+1) = U(c+1) \geq 0$.

▶ c.ⓒ@:

$$\begin{aligned} R(c) \le U(c) \quad &= CB - D(c) - B(c) \\ \Longrightarrow B(c+1) \quad &= R(c) + B(c) - D(c+1) \\ &\le CB - D(c) - B(c) + B(c) - D(c+1) \\ &= CB - D(c) - D(c+1) \le CB \end{aligned}$$

$$\begin{aligned} R(c) \ge L(c) \quad &= D(c+1) - B(c) \\ \Longrightarrow B(c+1) \quad &= R(c) + B(c) - D(c+1) \\ &\ge L(c) + B(c) - D(c+1) = 0 \end{aligned}$$

Therefore, $0 \le B(c+1) \le CB$.

$$\begin{aligned} & B(c) \geq 0 \textit{ and } R(c) \geq 0 \\ \Longrightarrow & U(c+1) = CB - R(c) - B(c) \le CB \end{aligned}$$

$$\begin{aligned} R(c) \le U(c) \quad &= CB - D(c) - B(c) \\ \Longrightarrow U(c+1) \quad &= CB - R(c) - B(c) \\ &\ge CB - (CB - D(c) - B(c)) - B(c) = D(c) \geq 0 \end{aligned}$$

Therefore, $0 \le U(c+1) \le CB$.

At cycle $c+1$, if the flow goes through branches @ⓑⓒ, then $R(c+1) \geq 0$; otherwise branch @ makes $R(c+1) = U(c+1) \geq 0$.

At this point, we've proved $\forall c \in [0, e-1] \quad 0 \le B(c) \le CB$, $0 \le U(c) \le CB$, $0 \le R(c)$.

From the algorithm, it is easy to see in all branches either $R(c) = 0 \le U(c)$ or $R(c) \le U(c)$. Therefore, $\forall c \in [0, e-1] \quad R(c) \le U(c)$.

□

Lemma 1. $\forall c \in [0, e-1]$ *if* $B(c) > \sum_{i=1}^{h} D(c+i)$, *then* $B(c) - \sum_{i=1}^{h} D(c+i) \le SF$.

Proof. (i). When $c = 0$, $B(0) = 0 \le SF + \sum_{i=1}^{h} D(0+i)$
$\Longrightarrow B(0) - \sum_{i=1}^{h} D(0+i) \le SF$
whether $B(0) > \sum_{i=1}^{h} D(0+i)$ or not.

(ii). Assume for any $c \in [0, e-2]$, if $B(c) > \sum_{i=1}^{h} D(c+i)$, then $B(c) - \sum_{i=1}^{h} D(c+i) \le SF$. We only need to prove when flow goes through branch @ at cycle $c+1$, i.e., when $B(c+1) > \sum_{i=1}^{h} D(c+1+i)$, then $B(c+1) - \sum_{i=1}^{h} D(c+1+i) \le SF$.

However, at cycle c, the flow can go through @ⓑⓒ@. In the following, the symbol c.@ $\rightarrow c+1$.ⓑ means the algorithm goes through branch @ in cycle c and branch ⓑ in the next cycle.

▶ $c.@ \to c+1.@$:

$$\begin{aligned} & R(c) = 0 \text{ and } B(c) - \sum_{i=1}^{h} D(c+i) \le SF \\ \Longrightarrow\ & B(c+1) - \sum_{i=1}^{h} D(c+1+i) \\ & = R(c) + B(c) - D(c+1) - \sum_{i=1}^{h} D(c+1+i) \\ & = B(c) - \sum_{i=1}^{h} D(c+i) - D(c+1+h) \\ & \le SF - D(c+1+h) \le SF \end{aligned}$$

▶ $c.ⓑ \to c+1.@$:

$$\begin{aligned} & B(c+1) = 0 \\ \Longrightarrow\ & B(c+1) - \sum_{i=1}^{h} D(c+1+i) \\ & = -\sum_{i=1}^{h} D(c+1+i) \le 0 \le SF \end{aligned}$$

▶ $c.ⓒ@ \to c+1.@$:

$$\begin{aligned} & R(c) \le SF \text{ and } B(c) \le \sum_{i=1}^{h} D(c+i) \\ \Longrightarrow\ & B(c+1) - \sum_{i=1}^{h} D(c+1+i) \\ & = R(c) + B(c) - D(c+1) - \sum_{i=1}^{h} D(c+1+i) \\ & \le SF + B(c) - \sum_{i=1}^{h} D(c+i) - D(c+1+h) \le SF \end{aligned}$$

□

Lemma 2. *Given* $l \in [2,h]$, *if* $\forall c \in [0, e-1]$

$$\begin{cases} B(c) - \sum_{i=1}^{l} D(c+i) \le (h-l+1) * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - \sum_{i=1}^{l} D(c+i) \le (h-l) * SF & \text{if } B(c) \le \sum_{i=1}^{h} D(c+i). \end{cases}$$

then $\forall c \in [0, e-1] \quad B(c) - \sum_{i=1}^{l-1} D(c+i) \le (h-l+1) * SF.$

Proof. When $c = 0$, $B(0) = 0$
$\Longrightarrow B(0) - \sum_{i=1}^{l-1} D(0+i) \le 0 \le (h-l+1) * SF$

Then we only need to prove $\forall c \in [0, e-1] \quad B(c+1) - \sum_{i=1}^{l-1} D(c+1+i) \le (h-l+1) * SF.$

▶ $c.@$:

$$\begin{aligned} & R(c) = 0 \\ \Longrightarrow\ & B(c+1) - \sum_{i=1}^{l-1} D(c+1+i) \\ & = R(c) + B(c) - D(c+1) - \sum_{i=1}^{l-1} D(c+1+i) \\ & = B(c) - \sum_{i=1}^{l} D(c+i) \\ & \le (h-l+1) * SF \end{aligned}$$

The last deduction is because of the condition in the lemma: given $l \in [2,h]$, $\forall c \in [0, e-1] \quad B(c) - \sum_{i=1}^{l} D(c+i) \le (h-l+1) * SF$ if $B(c) > \sum_{i=1}^{h} D(c+i)$.

▶ $c.ⓑ$:

$$\begin{aligned} & B(c+1) = 0 \\ \Longrightarrow\ & B(c+1) - \sum_{i=1}^{l-1} D(c+1+i) \le 0 \le (h-l+1) * SF \end{aligned}$$

▶ $c.ⓒ@$:

$$\begin{aligned} & R(c) \le SF \\ \Longrightarrow\ & B(c+1) - \sum_{i=1}^{l-1} D(c+1+i) \\ & = R(c) + B(c) - D(c+1) - \sum_{i=1}^{l-1} D(c+1+i) \\ & \le SF + B(c) - \sum_{i=1}^{l} D(c+i) \\ & \le SF + (h-l) * SF \\ & = (h-l+1) * SF \end{aligned}$$

The last deduction is because of the condition in the lemma: given $l \in [2,h]$, $\forall c \in [0,e-1]$ $B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l) * SF$ if $B(c) \leq \sum_{i=1}^{h} D(c+i)$.

□

Lemma 3. $\forall l \in [1,h]$ *and* $\forall c \in [0,e-1]$

$$\begin{cases} B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l+1) * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l) * SF & \text{if } B(c) \leq \sum_{i=1}^{h} D(c+i). \end{cases}$$

Proof. (i). When $l = h$, by Lemma 1, $\forall c \in [0,e-1]$ if $B(c) > \sum_{i=1}^{h} D(c+i)$,
then $B(c) - \sum_{i=1}^{h} D(c+i) \leq SF = (h-h+1) * SF$.
Also $\forall c \in [0,e-1]$ if $B(c) \leq \sum_{i=1}^{h} D(c+i)$,
then $B(c) - \sum_{i=1}^{h} D(c+i) \leq 0 = (h-h) * SF$.

Therefore, the proposition holds when $l = h$.

(ii). Assume for any $l \in [2,h]$, $\forall c \in [0,e-1]$

$$\begin{cases} B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l+1) * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - \sum_{i=1}^{l} D(c+i) \leq (h-l) * SF & \text{if } B(c) \leq \sum_{i=1}^{h} D(c+i). \end{cases}$$

We only need to prove the proposition is true for $l-1$, i.e., $\forall c \in [0,e-1]$

$$\begin{cases} B(c) - \sum_{i=1}^{l-1} D(c+i) \leq (h-l+2) * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - \sum_{i=1}^{l-1} D(c+i) \leq (h-l+1) * SF & \text{if } B(c) \leq \sum_{i=1}^{h} D(c+i). \end{cases}$$

This is true because of Lemma 2.

□

Corollary 3. $\forall c \in [0,e-1]$

$$\begin{cases} B(c) - D(c+1) \leq h * SF & \text{if } B(c) > \sum_{i=1}^{h} D(c+i) \\ B(c) - D(c+1) \leq (h-1) * SF & \text{if } B(c) \leq \sum_{i=1}^{h} D(c+i). \end{cases}$$

Proof. In Lemma 3, let $l = 1$.

□

Theorem 3. $\forall c \in [0,e-1]$ $B(c) \leq h * SF$.

Proof. When $c = 0$, $B(0) = 0 \leq h * SF$.

Then we only need to prove $\forall c \in [0,e-1]$ $B(c+1) \leq h * SF$.

▶ c.ⓐ:

$$\begin{aligned} & R(c) = 0, \ B(c) > \textstyle\sum_{i=1}^{h} D(c+i), \text{ and Corollary 3} \\ \Longrightarrow B(c+1) &= R(c) + B(c) - D(c+1) \\ &= B(c) - D(c+1) \leq h * SF \end{aligned}$$

▶ c.ⓑ:

$$B(c+1) = 0 \leq h * SF$$

▶ c.ⓒⓓ:

$$\begin{aligned} & R(c) \leq SF, \ B(c) \leq \textstyle\sum_{i=1}^{h} D(c+i), \text{ and Corollary 3} \\ \Longrightarrow B(c+1) &= R(c) + B(c) - D(c+1) \\ &\leq SF + B(c) - D(c+1) \\ &\leq SF + (h-1) * SF = h * SF \end{aligned}$$

□

Corollary 4. $\forall c \in [0, e-1] \quad 0 \le B(c) \le \min(CB, h * SF)$.

Proof. By Theorem 2 and Theorem 3.

□

Theorem 4. *If* $CB \ge \max((h+2)/(h+1) * P,\ V)$ *and* $SF \le CB/(h+2)$, *then* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \ge SF$.

Proof. $CB \ge V$ will guarantee the algorithm will not fail.

$SF \le CB/(h+2)$
$\Longrightarrow h * SF \le (h+2) * SF \le CB$
$\Longrightarrow h * SF$ is a closer upper bound for $B(c)$ than CB is when $SF \le CB/(h+2)$.

If branch @ in the algorithm is never executed, then $\forall c \in [0, e-1] \quad R(c) = 0$ or $R(c) \ge SF$. If in cycle c, $U(c) \ge SF$, then the flow will not go through branch @. We will prove under the conditions in the Theorem, $\forall c \in [0, e-1] \quad U(c) \ge SF$.

(i). When $c = 0$, $SF \le CB/(h+2) < CB = U(0)$

(ii). Assume for any cycle $c \in [0, e-2]$, $U(c) \ge SF$, i.e., flow goes through branches @ ⓑⓒ. We prove in each case, $U(c+1) \ge SF$.

▶ c.@:

$R(c) = 0$ and Theorem 3
$\Longrightarrow U(c+1) = CB - R(c) - B(c)$
$= CB - B(c)$
$\ge CB - h * SF$
$\ge CB - h * CB/(h+2)$
$= 2/(h+2) * CB$
$> CB/(h+2) \ge SF$

▶ c.ⓑ:

$R(c) = D(c+1) - B(c)$
$\Longrightarrow B(c+1) = R(c) + B(c) - D(c+1) = 0$
$\Longrightarrow U(c+1) = CB - D(c+1) - B(c+1)$
$= CB - D(c+1)$
$\ge CB - P$
$\ge CB - (h+1)/(h+2) * CB$
$= CB/(h+2) \ge SF$

▶ c.ⓒ:

$R(c) = SF$ and Theorem 3
$\Longrightarrow U(c+1) = CB - R(c) - B(c)$
$= CB - SF - B(c)$
$\ge CB - SF - h * SF$
$= CB - (h+1) * SF$
$\ge CB - (h+1) * CB/(h+2)$
$= CB/(h+2) \ge SF$

□

Corollary 5. *For* $X > 0$, *if* $CB \ge \max((h+2)/(h+1) * P,\ V,\ (h+2) * X)$ *then* $\exists SF \ge X$, *such that* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \ge SF$.

Proof. In Theorem 4, let $SF = CB/(h+2) \ge X$.

□

Corollary 6. *If* $CB = 2 * P$ *and* $SF \leq CB/(h+2) = 2/(h+2) * P$, *then* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \geq SF$.

Proof. As $CB = 2 * P \geq \max((h+2)/(h+1) * P,\ V)$, Theorem 4 will make the proposition true.

□

Corollary 7. *If* $CB = 2 * P = 2 * k * M$ *and* $r \leq 2 * k/(h+2)$, *then* $\forall c \in [0, e-1] \quad R(c) = 0$ *or* $R(c) \geq r * M$.

Proof. In Corollary 6, let $SF = r * M$.

□

Appendix B

Contract Summary

This appendix summarises the set of contracts introduced in Chapter 6. They are assumed by the AC criteria but should be implemented by QoS enforcement in an SS. C-i refers to contract number i.

C-1 Reserve fixed amount of resources for NRT receive.

C-2 Regulate receive activities in each cycle.

C-3 Receive activities have the highest priority.

C-4 Disk and network send activities (not including late arriving NRT activities) are known at the start of a cycle.

C-5 Use FCFS scheduling for disk and network send activities.

C-6 Pure CPU processing has a lower priority than device related activities.

C-7 RT pure CPU processing has a higher priority than the NRT one.

C-8 Disk transfer has a higher priority than network send.

C-9 Either Condition (6.7) or Condition (6.8) should be true or made true.

C-10 Use a dual K-buffer for all RT playback requests.

C-11 Use a fixed dual-buffer for recording each stream.

C-12 Reduce the cleverness of a disk device.

C-13 Cycle overload processing in case of disk anomalies.

C-14 Maintain an extent-list for each PO.

C-15 SCAN disk scheduling.

C-16 Detect and record zone information for ZBR disks.

C-17 Maintain a cycle table to record resource usage in each cycle.

C-18 NRT requests should be subject to run-time AC.

C-19 Three-level AC for NRT requests.

C-20 Measure the actual resource usage in the current cycle.

C-21 Process unfinished NRT requests according to their natures during cycle overload.

C-22 Regulate disk and network send activities to prevent pure CPU processing from starvation.

C-23 Re-activate receive activities after they have been disabled and when conditions allow.

Appendix C

Implementation Interfaces

This appendix lists the major interfaces with their important methods used in the Cadmus prototype implementation. Both interface and system defined type names start with capital letters. Exception handling and some type definitions are omitted for clarity reasons. The interfaces are written in CORBA IDL.

C.1 Some Basic Types

```
struct LoAttribute {
  Loid     loid;
  PoidList poidList;
  boolean  isPrimary;     // Primary or backup.
  Loid     replicaLoid;
  long     sus;           // Striping unit size.
  long     n;             // Number of SSs the LO is striped over.
  long     d;             // Declustering degree.
  long     length;
  long     type;          // Directory/meta-data/CM or non-CM data.

  // The following two are only valid for a CM data object.
  boolean  isVbr;         // VBR or CBR.
  long     p;             // Maximal number of displayed striping units.

  ...                     // Other attributes.
};

typedef LoAttribute FsAttribute;
```

```
struct PoAttribute {
  Poid poid;
  long sus;     // Striping unit size.
  long length;
  long type;

  ...           // Other attributes.
};

struct TransportAttribute {
  ... // Requirements on the communication channel.
};

interface FaultNotify {
  void ssLost(in Ssid lostSsid);
  void ssRecovered(in Ssid recoveredSsid)
};
```

C.2 Port and Connection

```
interface Port {
  void setTransportAttribute(in TransportAttribute attr);
  TransportAttribute getTransportAttribute();

  void setAddress(in PortAddress addr);
  PortAddress getAddress();
};

interface SourcePort : Port {
  void connect(in PortAddress addr);
};

interface SinkPort : Port {
  void listen();
};

typedef sequence<SourcePort> SourcePortList;
typedef sequence<SinkPort>   SinkPortList;

/* ____________________________________________________________ */

interface Connection {
  void setTransportAttribute(in TransportAttribute attr);
  TransportAttribute getTransportAttribute();

  void setSourcePort(in SourcePort port);
  void setSinkPort(in SinkPort port);
  void connect();  // Connect source and sink ports.
};
```

C.3 Client Part

```
interface Cp : FaultNotify {
  // The CP is responsible for maintaining the LO attributes.
  void updateLoAttribute();
  LoAttribute getLoAttribute();
};

interface RtCp : Cp {
  void start();
  void stop();
};

interface SourceRtCp : RtCp {
  SourcePortList getSourcePortList();
};

interface SinkRtCp : RtCp {
  SinkPortList getSinkPortList();

  // The sink RT-CP uses the read set to anticipate VBR data arrivals.
  void setReadSet(in ReadSet readSet);

  ... // Other VCR commands.
};

interface NrtCp : Cp {
  SinkPortList getSinkPortList();

  // The startPosition is relative to the start of the LO.
  long receiveData(in long startPosition, in long length,
                   in boolean fromPrimary);
  long sendData(in long startPosition, in long length,
                in boolean toPrimary);

  // An NRT-LO is implemented by its associated NRT-CP.
  NrtLo openNrtLo(in LoAttribute attr, in long mode);
  void closeNrtLo(in NrtLo lo);
};
```

C.4 Point, Unit, and Unit Factory

```
interface Point {
  void start();
  void stop();
};
```

```
interface SourcePoint : Point {};
interface SinkPoint   : Point {};

interface CameraEndSourcePoint : SourcePoint {};
interface TileEndSinkPoint     : SinkPoint   {};
interface XEndSinkPoint        : SinkPoint   {};
interface FileEndSourcePoint   : SourcePoint {};
interface FileEndSinkPoint     : SinkPoint   {};

interface NetSourcePoint : SourcePoint {
  SourcePort getSourcePort();
};

interface NetSinkPoint : SinkPoint {
  SinkPort getSinkPort();
};

/* ____________________________________________________________________ */

interface Unit {
  LoAttribute getLoAttribute();

  void start();
  void stop();
};

interface UnitWithRtCp : Unit {
  SourceRtCp getSourceRtCp();
  SinkRtCp   getSinkRtCp();
};

interface UnitWithNrtCp : Unit {
  NrtCp getNrtCp();
};

interface UnitWithNetPoint : Unit {
  NetSourcePoint getNetSourcePoint();
  NetSinkPoint   getNetSinkPoint();
};

interface CameraSourceUnit : UnitWithRtCp {};
interface TileSinkUnit     : UnitWithRtCp {};
interface XSinkUnit        : UnitWithRtCp {};

interface FileSourceUnit : UnitWithNrtCp {};
interface FileSinkUnit   : UnitWithNrtCp {};

interface NetPipeUnit : UnitWithNetPoint {};

/* ____________________________________________________________________ */
```

```
interface UnitFactory {
  CameraSourceUnit openCameraSourceUnit(in Camera camera, in Loid loid);
  void closeCameraSourceUnit(in CameraSourceUnit unit);

  TileSinkUnit openTileSinkUnit(in Loid loid, in Tile tile);
  void closeTileSinkUnit(in TileSinkUnit unit);

  XSinkUnit openXSinkUnit(in Loid loid, in XDisplay xDisplay);
  void closeXSinkUnit(in XSinkUnit unit);

  FileSourceUnit openFileSourceUnit(in File file, in Loid loid);
  void closeFileSourceUnit(in FileSourceUnit unit);

  FileSinkUnit openFileSinkUnit(in Loid loid, in File file);
  void closeFileSinkUnit(in FileSinkUnit unit);

  NetPipeUnit openNetPipeUnit(in Loid loid);
  void closeNetPipeUnit(in NetPipeUnit unit);
};
```

C.5 Physical Object and Storage Server

```
interface Po {
  PoAttribute getPoAttribute();
};

interface RtPo : Po {
  boolean  commit(in long startSecond);

  void start();
  void stop();
};

interface PlayRtPo : RtPo {
  StartSet admit(in ReadTable readTable);

  SourcePort getSourcePort();
};

interface RecordRtPo : RtPo {
  // CBR and VBR recording are treated differently.
  StartSet admit(in LoAttribute attr);

  SinkPort getSinkPort();
};
```

```
interface NrtPo : Po {
  SourcePort getSourcePort();

  // The startPosition is relative to the start of the PO.
  long sendData(in long startPosition, in long length,
                in boolean fromPrimary);
  long receiveData(in long startPosition, in long length,
                   in boolean toPrimary);
};

/* ____________________________________________________________ */

interface Ss : FaultNotify {
  PoAttribute getPoAttribute(in Poid poid);

  Poid createPo(in PoRequirement requirement);
  void erasePo(in Poid poid);

  PlayRtPo openPlayRtPo(in Poid primaryPoid, in Poid backupPoid);
  void closePlayRtPo(in PlayRtPo po);

  RecordRtPo openRecordRtPo(in Poid primaryPoid, in Poid backupPoid);
  void closeRecordRtPo(in RecordRtPo po);

  NrtPo openNrtPo(in Poid primaryPoid, in Poid backupPoid, in long mode);
  void closeNrtPo(in NrtPo po);
};
```

C.6 Logical Object and Stream Control Agent

```
interface Lo : FaultNotify {
  LoAttribute getLoAttribute();
};

interface RtLo : Lo {
  StartSet admit();
  boolean  commit(in long startSecond);

  void start();
  void stop();
};

interface PlayRtLo : RtLo {
  ...  // Other VCR commands.
};

interface RecordRtLo : RtLo {};
```

```
interface NrtLo : Lo {
  // The startPosition is relative to the start of the LO.
  long readData(in long startPosition, in long length);
  long writeData(in long startPosition, in long length);
};

/* ---------------------------------------------------------------------- */

typedef RtLo       Sca;
typedef PlayRtLo   PlaySca;
typedef RecordRtLo RecordSca;
```

C.7 File Server and Stream Control Factory

```
interface Fs : FaultNotify {
  void setFsAttribute(in FsAttribute attr);
  FsAttribute getFsAttribute();

  void setLoAttribute(in LoAttribute attr);
  LoAttribute getLoAttribute(in Loid loid);

  LoAttribute createLo(in LoRequirement requirement);
  void eraseLo(in Loid loid);

  // The FS will get the SCA implementation reference from an SCF.
  PlaySca openPlayRtLo(in Loid loid, in Loid indexLoid, in SinkRtCp cp);
  void closePlayRtLo(in PlaySca sca);

  RecordSca openRecordRtLo(in Loid loid, in SourceRtCp cp);
  void closeRecordRtLo(in RecordSca sca);

  // The FS will get the NRT-LO implementation reference from the NRT-CP.
  NrtLo openNrtLo(in Loid loid, in long mode, in NrtCp cp);
  void closeNrtLo(in NrtLo lo);
};

/* ---------------------------------------------------------------------- */

interface Scf : FaultNotify {
  PlaySca openPlaySca(in LoAttribute attr, in LoAttribute indexAttr,
                      in SinkRtCp cp);
  void closePlaySca(in PlaySca sca);

  RecordSca openRecordSca(in LoAttribute attr, in SourceRtCp cp);
  void closeRecordSca(in RecordSca sca);
};
```

Bibliography

[Abbott90] R. K. Abbott and H. Garcia-Molina. *Scheduling I/O Requests with Deadlines: a Performance Evaluation.* In IEEE Real-Time Systems Symp., 1990. (p 19)

[Addlesee95] M. Addlesee. *Programmers Model for the XSI Profiler Board Version 1.0.* Olivetti Research Ltd., Nov 1995. (p 89)

[Anderson92] D. P. Anderson, Y. Osawa, and R. Govindan. *A File System for Continuous Media.* ACM Trans. on Computer Systems, 11(4), Nov. 1992. (p 20)

[Anderson95a] T. E. Anderson, D. E. Culler, and D. A. Patterson. *A Case for NOW (Networks of Workstations).* IEEE Micro, 15(1), Feb 1995. (p 27)

[Anderson95b] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. *Serverless Network File Systems.* In Proc. 15th ACM SOSP, CO, USA, Dec 1995. (p 27)

[ANSI94] *X3T11/Project 960D/Rev 4.4.* Working Draft Proposal, ANSI, Oct 1994. (p 12)

[Asai95] M. Asai, K. Shibata, M. Igawa, Y. Kim, Sato M, and Y. Takiyasu. *Essential Factors for Full-Interactive VOD Server: Video File System, Disk Scheduling, Network.* In IEEE GLOBECOM'95, 1995. (pp 22, 66)

[Audsley90] N. Audsley and A. Burns. *Real-time System Scheduling.* Technical Report, Department of Computer Science, University of York, U.K., 1990. (p 16)

[Bach86] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986. (p 14)

[Bacon93] J. Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems.* Addison-Wesley, 1993. (p 15)

[Baker88] T. P. Baker and A. Shaw. *The Cyclic Executive Model and Ada.* In IEEE Real-Time Systems Symp., 1988. (p 16)

[Baker90] T. P. Baker. *A Stack-Based Resource Allocation Policy for Realtime Processes.* In IEEE Real-Time Systems Symp., 1990. (p 16)

[Barham96] P. R. Barham. *Devices in a Multi-Service Operating System.* Technical Report 403, Computer Laboratory, University of Cambridge, Oct 1996. (pp 17, 19)

[Barnsley88] M. F. Barnsley and A. E. Jacquin. *Application of recurrent iterated function systems to images.* In Visual Communications and Image Processing, volume 1001, pages 122–131. SPIE, 1988. (p 7)

[Baruah91] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, D. Shasha, and F. Wang. *On the Competitiveness of On-Line Real-Time Task Scheduling.* IEEE Real-Time Systems Symp., 1991. (p 16)

[Berdahl95] L. Berdahl. *Parallel Transport Protocol Proposal.* Draft Proposal, Lawrence Livermore National Laboratory, USA, Jan 1995. (p 26)

[Bershad94] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. *SPIN — An Extensible Microkernel for Application-specific Operating System Services.* Technical Report 94-03-03, Dept. Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA, Feb 1994. (p 17)

[Berson95] S. Berson, L. Golubchik, and R. R. Muntz. *Fault Tolerant Design of Multimedia Servers.* In ACM SIGMOD, San Jose, CA, USA, 1995. (p 23)

[Birrell84] A. D. Birrell and B. J. Nelson. *Implementing Remote Procedure Calls.* ACM Trans. on Computer Systems, 2(1):39–59, Feb. 1984. (p 13)

[Black95] R. J. Black. *Explicit Network Scheduling.* PhD dissertation, University of Cambridge, Computer Laboratory, April 1995. (p 17)

[Bolosky96] W. J. Bolosky, J. S. Barrera, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. *The Tiger Video Fileserver.* In Proc. NOSSDAV'96, Zushi, Japan, Apr 1996. (p 31)

[Boxer95] A. Boxer. *Where Buses Cannot Go.* IEEE Spectrum, Feb 1995. (pp 11, 32)

[Bryan94] J. Bryan. *Fibre Channel Speeds Up.* Byte, Aug. 1994. (p 12)

[Buddhikot95] M. M. Buddhikot and G. M. Parulkar. *Efficient Data Layout, Scheduling and Playout Control in MARS.* In NOSSDAV'95, Durham, NH, US, April 1995. (p 29)

[Cabrera91] L. F. Cabrera and D. D. E. Long. *Swift: Using Distributed Disk Striping to Provide High I/O Data Rates.* Computing Systems, 4(4), Fall 1991. (p 24)

[Chaney95] A. J. Chaney, I. D. Wilson, and A. Hopper. *The Design and Implementation of a RAID-3 Multimedia File Server.* In NOSSDAV'95, Durham, NH, US, April 1995. (pp 33, 89)

[Chang94a] E. Chang and A. Zakhor. *Admissions Control and Data Placement for VBR Video Servers.* In 1st IEEE Intl. Conf. on Imaging Processing, Austin, TX, USA, Nov 1994. (pp 22, 66)

[Chang94b] E. Chang and A. Zakhor. *Variable Bit Rate MPEG Video Storage on Parallel Disk Arrays.* In 1st Intl. Workshop Community Networking Integrated Multimedia Services to the Home, San Francisco, CA, USA, July 1994. (p 22)

[Chang96] E. Chang and A. Zakhor. *Cost Analysis for VBR Video Servers.* In Proceedings of Multimedia Computing and Networking (MMCN), San Jose, CA, USA, Jan 1996. (pp 21, 22)

[Chen93] H. J. Chen and T. D. C. Little. *Physical Storage Organizations for Time-Dependent Multimedia Data.* In 4th International Conference on Foundations of Data Organization and Algorithms, Evanston, IL, U.S., Oct. 1993. (p 21)

[Chen94a] M. S. Chen, D. D. Kandlur, and P. S. Yu. *Support for Fully Interactive Playout in a Disk-Array-Based Video Server.* In ACM Multimedia'94, San Francisco, CA, USA, Oct 1994. (p 86)

[Chen94b] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. *RAID: High-Performance, Reliable Secondary Storage.* ACM Computing Surveys, 26(2), June 1994. (pp 9, 10)

[Chen95] M. S. Chen, H. I. Hsiao, C. S. Li, and P. S. Yu. *Using Rotational Mirrored Declustering for Replica Placement in a Disk-Array-Based Video Server.* In ACM Multimedia, San Francisco, CA, USA, Nov 1995. (p 23)

[Chen96] S. Chen and M. Thapar. *I/O Channel and Real-Time Disk Scheduling for Video Servers.* In Proc. NOSSDAV'96, Zushi, Japan, Apr 1996. (pp 20, 22, 32)

[Cheng88] S. C. Cheng, J. A. Stankovic, and K. Ramamritham. *Scheduling Algorithms for Hard Real-Time Systems — A Brief Survey.* In J. A. Stankovic and K. Ramamritham, editors, *Tutorial: Hard Real-Time Systems.* Computer Society Press, 1988. (p 16)

[Chervenak95] A. L. Chervenak, D. A. Patterson, and R. H. Katz. *Choosing the Best Storage System for Video Service.* In ACM Multimedia 95, San Francisco, CA, USA, Nov 1995. (p 11)

[Chiang96] H. S. Chiang. *Time Synchronisation on ATM Network.* Olivetti Research Ltd. Internal Talk, 1996. (p 89)

[Clark90] R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD dissertation, Carnegie Mellon University, School of Computer Science, Aug 1990. (p 17)

[Clark92] R. K. Clark, E. D. Jensen, and F. D. Reynolds. *An Architectural Overview of the Alpha Real-Time Distributed Kernel.* In USENIX Workshop on Microkernel and Other Kernel Architectures, Seattle, US, April 1992. (p 17)

[Copeland89] G. Copeland and T. Keller. *A Comparison of High-Availability Media Recovery Techniques.* In ACM SIGMOD, Portland, Oregon, USA, June 1989. (p 23)

[CORBA91] *The Common Object Request Broker: Architecture and Specification.* Revision 1.1 Draft 10 OMG Document Number 91.12.1, Dec 1991. (p 13)

[CORBA94] *The Common Object Request Broker: Architecture and Specification.* Revision 2.0, Object Management Group, Dec 1994. (p 13)

[**Cornhill87**] D. Cornhill and S. Sha. *Priority Inversion in Ada.* Ada Letters, Nov 1987. (p 16)

[**Dey94**] J. Dey, C. S. Shih, and M. Kumar. *Storage Subsystem Design in a Large Multimedia Server for High-Speed Network Environments.* In IS&T/SPIE Symp. Electronic Imaging Science and Technology, volume 2188, San Jose, CA, USA, Feb 1994. (pp 22, 66)

[**Engler94**] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole Jr. *The Operating System Kernel as a Secure Programmable Machine.* In Proc. 6th SIGOPS European Workshop, 1994. (p 17)

[**Engler95**] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole Jr. *Exokernel: An Operating System Architecture for Application-Level Resource Management.* In Proc. 15th SOSP, Dec 1995. (p 17)

[**Feng95**] W. C. Feng and S. Sechrest. *Smoothing and Buffering for Delivery of Prerecorded Compressed Video.* In IS&T/SPIE Proc. of Multimedia Computing and Networking (MMCN), volume 2417, San Jose, CA, USA, Feb 1995. (p 66)

[**Finkel88**] R. A. Finkel. *An Operating Systems Vade Mecum.* Prentice Hall, Inc., 2nd edition, 1988. (p 70)

[**Fisher95**] Y. Fisher. *Fractal Image Compression: Theory and Application.* Springer Verlag, New York, 1995. (p 7)

[**Freedman94**] C. S. Freedman and D. J. DeWitt. *SPIFFI — A Scalable Parallel File System for the Intel Paragon.* Technical Report, Computer Science Department, University of Wisconsin, Madison, WI 53706, USA, 1994. (p 29)

[**Freedman95**] C. S. Freedman and D. J. DeWitt. *The SPIFFI Scalable Video-on-Demand System.* In SIGMOD'95, San Jose, CA, USA, 1995. ACM. (p 29)

[**Gall91**] D. Le Gall. *MPEG: A Video Compression Standard for Multimedia Applications.* Commun. of the ACM, 34(4), Apr. 1991. (p 5)

[**Garrison94**] R. Garrison. *Reference Model for Open Storage Systems Interconnection: Mass Storage Reference Model Version 5.* IEEE Storage System Standards Working Group (Project 1244), Sept 1994. (p 26)

[**Ghandeharizadeh95**] S. Ghandeharizadeh, S. H. Kim, and C. Shahabi. *On Configuring a Single Disk Continuous Media Server.* In SIGMETRICS'95, Ottawa, Ontario, Canada, 1995. ACM. (p 21)

[**Gilmurray95**] D. Gilmurray. *OmniORB1.0 – Distributed Object Computing for ATMos.* Technical Report, Olivetti Research Ltd., May 1995. (p 89)

[**Gopinath89**] P. Gopinath and K. Schwan. *CHAOS: Why One Cannot Have Only An Operating System for Real-Time Applications.* ACM Operating Systems Rev., 23(3), July 1989. (p 17)

[**Grochowski96**] E. G. Grochowski and R. F. Hoyt. *Future Trends in Hard Disk Drives.* IEEE Trans. on Magnetics, May 1996. (p 11)

[**Haritsa91**] J. R. Haritsa, M. Livny, and M. J. Carey. *Earliest Deadline Scheduling for Real-Time Database Systems.* In IEEE Real-Time Systems Symp., 1991. (p 20)

[**Hartman95**] J. H. Hartman and J. K. Ousterhout. *The Zebra Striped Network File System.* ACM Trans. on Computer Systems, 13(3), Aug 1995. (p 25)

[**Hennessy90**] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990. (p 10)

[**Heybey96**] S. Heybey, M. Sullivan, and P. England. *Calliope: A Distributed, Scalable Multimedia Server.* In USENIX Technical Conference, San Diego, CA, USA, Jan 1996. (p 86)

[**Hilton94**] M. L. Hilton, B. D. Jawerth, and A. Sengupta. *Compressing Still and Moving Images with Wavelets.* Multimedia Systems, 2:218–227, Dec 1994. (p 7)

[**Hopkins94**] R. Hopkins. *Digital Terrestrial HDTV for North America: The Grand Alliance HDTV System.* IEEE Trans. on Consumer Electronics, 40(3), Aug 1994. (p 6)

[**Howard88**] J. H. Howard, M. J. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. *Scale and Performance in a Distributed File System.* ACM Trans. Computer Systems, 6(1), Feb 1988. (pp 13, 27)

[**Hsiao90**] H. I. Hsiao and D. J. DeWitt. *Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines.* In Proc. 6th Intl. Conf. Data Engineering (ICDE), 1990. (p 23)

[**Hyden94**] E. A. Hyden. *Operating System Support for Quality of Service.* PhD dissertation, Computer Laboratory, University of Cambridge, June 1994. Technical Report No. 340. (p 5)

[**Ibbett89**] R. N. Ibbett and N. P. Topham. *Architecture of High Performance Computers*, volume II. Macmillan Education Ltd., 1989. (p 11)

[**ISO/IEC93a**] *ISO/IEC 11172-1/2/3: Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s – Part 1/2/3: systems/video/audio*, Aug 1993. (p 6)

[**ISO/IEC93b**] *Working Document for ISO/IEC 13818-1/2/3: Information technology — Generic coding of moving pictures and associated audio information – Part 1/2/3: systems/video/audio*, Nov 1993. (p 6)

[**Jardetzky92**] P. W. Jardetzky. *Network File Server Design for Continuous Media.* PhD dissertation, University of Cambridge, Computer Laboratory, Oct. 1992. Technical Report No. 268. (pp 22, 27)

[**Jardetzky95**] P. W. Jardetzky, C. J. Sreenan, and R. M. Needham. *Storage and Synchronization for Distributed Continuous Media.* Multimedia Systems, 3:151–161, 1995. (p 22)

[Jensen94] E. D. Jensen. *Asynchronous Decentralized Realtime Computer System.* In W. A. Halang and A. D. Stoyenko, editors, *Real Time Computing.* Springer-Verlag, 1994. (p 16)

[Katz92] R. H. Katz. *High-Performance Network and Channel Based Storage.* Proceedings of the IEEE, 80(8), Aug. 1992. (p 11)

[Katzman78] J. A. Katzman. *A Fault-Tolerant Computing System.* In Proc. 11th Hawaii Conf. on System Sciences, Jan 1978. (p 23)

[Kopetz89] H. Kopetz, A. Damm, C. Koza, and M. Mulozzani. *Distributed Fault Tolerant Real-Time Systems: The Mars Approach.* IEEE Micro, pages 25–41, 1989. (p 17)

[Koren92] G. Koren and D. Shasha. *Dover: An Optimal On-Line Scheduling Algorithm for Overload Real-Time Systems.* In IEEE Real-Time Systems Symp., 1992. (p 16)

[Kung92] H. T. Kung. *Gigabit Local Area Networks: A Systems Perspective.* IEEE Commun. Mag., April 1992. (p 12)

[Lamport89] L. Lamport. *The Part-Time Parliament.* Technical Report 49, DEC Systems Research Center, 130 Lytton Ave. Palo Alto, CA 94301-1044, USA, Sept 1989. (p 28)

[Lau95] S. W. Lau and J. C. S. Liu. *A Novel Video-On-Demand Storage Architecture for Supporting Constant Frame Rate with Variable Bit Rate Retrieval.* In NOSSDAV'95, Durham, NH, US, April 1995. (p 22)

[Laursen94] A. Laursen, J. Olkin, and M. Porter. *Oracle Media Server: Providing Consumer Based Interactive Access to Multimedia Data.* In ACM SIGMOD Conference, Minneapolis, MN, U.S., May 1994. (p 32)

[Laursen95] A. Laursen, J. Olkin, and M. Porter. *Oracle Media Server Framework.* In IEEE Compcon, Spring, San Francisco, CA, USA, Mar 1995. (p 32)

[Lawlor81] F. D. Lawlor. *Efficient Mass Storage Parity Recovery Mechanism.* IBM Technical Disclosure Bulletin, 24(2):986–987, July 1981. (p 9)

[Lee95] E. K. Lee. *Highly-Available, Scalable Network Storage.* In IEEE Compcon, Spring, San Francisco, CA, USA, Mar 1995. (p 23)

[Lee96] E. K. Lee and C. A. Thekkath. *Petal: Distributed Virtual Disks.* In ASPLOS VII, MA, USA, Oct 1996. ACM. (p 27)

[Lehoczky89] J. Lehoczky, L. Sha, and Y. Ding. *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior.* In IEEE Real-Time Systems Symp., 1989. (p 16)

[Leslie96] I. Leslie, D. McAuley, R. Black, and T. Roscoe. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE J. Selected Areas Communi., Sept 1996. (p 17)

[Levin75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. *Policy/Mechanism Separation in Hydra.* In Proc. 5th Symp. SOSP. ACM, 1975. (p 70)

[**Liou91**] M. Liou. *Overview of the px64kbit/s Video Coding Standard.* Commun. of the ACM, 34(4), Apr. 1991. (p 5)

[**Liu73**] C. L. Liu and J. W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, 20(1), Jan. 1973. (p 16)

[**Lo94**] S. L. Lo. *A Modular and Extensible Network Storage Architecture.* PhD dissertation, University of Cambridge, Computer Laboratory, Jan. 1994. Technical Report No. 326. (pp 15, 42, 48)

[**Locke86**] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD dissertation, Carnegie Mellon University, 1986. CMU-CS-86-134. (pp 16, 20)

[**Long94**] D. D. E. Long, B. R. Montague, and L. F. Cabrera. *Swift/RAID: A Distributed RAID System.* Computing Systems, 7(3), Summer 1994. (p 24)

[**Lougher92**] P. Lougher and D. Shepherd. *The Design and Implementation of a Continuous Media Storage Server.* In Proc. 3rd Network and Operating Systems Support for Digital Audio and Video, pages 70–80, La Jolla, California, U.S.A., Nov. 1992. Springer–Verlag. (pp 19, 20, 22)

[**Lougher93**] P. Lougher and D. Shepherd. *The Design of a Storage Server for Continuous Media.* The Computer Journal, 36(1), 1993. (p 21)

[**Lougher96**] P. Lougher, R. Lougher, D. Shepherd, and D. Pegler. *A Scalable Hierarchical Video Storage Architecture.* In Proc. Multimedia Computing and Networking, San Jose, CA, USA, Jan 1996. (pp 13, 30)

[**McAuley90**] D. R. McAuley. *Protocol Design for High Speed Networks.* PhD dissertation, University of Cambridge, Computer Laboratory, Jan. 1990. Technical Report No. 186. (p 89)

[**McKeown96**] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz. *The Tiny Tera: A PAcket Switch Core.* Technical Report, Dept. Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-4070, USA, 1996. (p 12)

[**McManus95**] J. M. McManus and K. W. Ross. *Prerecorded VBR Sources in ATM Networks: Piecewise-Constant-Rate Transmission and Transport.* Technical Report, Dept. of Systems Engineering, Univ. of Pennsylvania, Philadelphia, PA, 19104, USA, Sept 1995. (p 66)

[**McManus96**] J. M. McManus and K. W. Ross. *Video on Demand over ATM: Constant-Rate Transmission and Transport.* In IEEE INFOCOM'96, Mar 1996. (p 66)

[**Meter96**] R. V. Meter. *A Brief Survey of Current Work on Network Attached Peripherals.* ACM OSR, 30(1), Jan 1996. (p 26)

[**Mills94**] D. L. Mills. *Improved Algorithms for Synchronizing Computer Network Clocks.* In ACM SIGCOMM'94, London, UK, Sept 1994. (p 90)

[**Natarajan95**] K. Natarajan. *Video Servers Take Root.* IEEE Spectrum, April 1995. (p 32)

[Needham92] R. Needham and A. Nakamura. *An Approach to Real-Time Scheduling – but is it Really a Problem for Multimedia?* In Proc. 3rd Network and Operating Systems Support for Digital Audio and Video, pages 32–39, La Jolla, CA, U.S.A., Nov. 1992. Springer–Verlag. (p 70)

[Nelson88] M. Nelson, B. Welch, and J. Ousterhout. *Caching in the Sprite Network File System.* ACM Trans. on Computer Systems, 6(1), Feb 1988. (p 27)

[Nelson95] M. N. Nelson, M. Linton, and S. Owicki. *A Highly Available, Scalable ITV System.* In Proc. 15th ACM SOSP, CO, USA, Dec 1995. (p 32)

[Neufeld96a] G. Neufeld, D. Makaroff, and N. Hutchinson. *Design of a Variable Bit Rate Continuous Media File Server.* In Proceedings of Multimedia Computing and Networking (MMCN), San Jose, CA, USA, Jan 1996. (pp 22, 66)

[Neufeld96b] G. Neufeld, D. Makaroff, and N. Hutchinson. *Server Based Flow Control in a Distributed Continuous Media Server.* In Proc. NOSSDAV'96, Zushi, Japan, Apr 1996. (p 66)

[OMA90] *Object Management Architecture (OMA) Guide.* Object Management Group (OMA), 1990. (p 13)

[OSF91] *Introduction to OSF DCE.* Open Software Foundation, Dec 1991. (p 13)

[Ousterhout89] J. Ousterhout and F. Douglis. *Beating the I/O Bottleneck: A Case for Log-Structured File Systems.* ACM Operating Systems Review, 23(1), Jan. 1989. (p 14)

[Oyang95] Y. J. Oyang, M. H. Lee, C. H. Wen, and C. Y. Cheng. *Design of Multimedia Storage System for On-Demand Playback.* In Proc. 11th Intl. Conf. on Data Engineering (ICDE'95), 1995. (p 44)

[Ozden96a] B. Ozden, R. Rastogi, P. Shenoy, and A. Silberschatz. *Fault-tolerant Architectures for Continuous Media Servers.* In SIGMOD'96, Montreal, Canada, June 1996. ACM. (p 23)

[Ozden96b] B. Ozden, R. Rastogi, and A. Silberschatz. *On the Design of a Low-Cost Video-on-Demand Storage System.* Multimedia Systems, 4:40–54, 1996. (p 21)

[Patterson88] D. A. Patterson, G. Gibson, and R. H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID).* In ACM SIGMOD, Chicago, IL, 1988. (pp 9, 10)

[Pearson94] D. Pearson. *Multi-Media Client-Server Systems.* Presented in the 4th International EDBT Conference, St. Johns College, Cambridge, U.K., Mar. 1994. (p 32)

[Pegler97] D. Pegler, N. Yeadon, D. Hutchison, and D. Shepherd. *Incorporating Scalability into Networked Multimedia Storage System.* In Proc. Multimedia Computing and Networking, San Jose, CA, USA, Jan 1997. (pp 13, 30)

[Petajan95] E. Petajan. *The HDTV Grand Alliance System.* Proc. of the IEEE, 83(7), July 1995. (p 6)

[**Press88**] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988. (p 110)

[**Prycker93**] M. De Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN.* Ellis Horwood Ltd., second edition, 1993. (p 12)

[**Pthreads93**] IEEE. *Draft Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]*, Apr 1993. P1003.4a Draft 7. (p 89)

[**Ramakrishnan93**] K. K. Ramakrishnan. *Performance Considerations in Designing Network Interface.* IEEE JSAC, 11(2), Feb 1993. (p 71)

[**Ramakrishnan95**] K. K. Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaser, and W. Duso. *Operating System Support for a Video-on-Demand File Service.* Multimedia Systems, 3:53–65, 1995. (p 22)

[**Ramamritham94**] K. Ramamritham and J. A. Stankovic. *Scheduling Algorithms and Operating Systems Support for Real-Time Systems.* Proc. of the IEEE, 82(1), Jan 1994. (p 15)

[**Rangan91**] P. V. Rangan and H. M. Vin. *Designing File Systems for Digital Video and Audio.* ACM Operating Systems Review, 25(5), 1991. (pp 20, 22)

[**Rangan92**] P. V. Rangan, H. M. Vin, and S. Ramanathan. *Designing an On-Demand Multimedia Service.* IEEE Commun. Mag., July 1992. (p 19)

[**Rangan93**] P. V. Rangan and H. M. Vin. *Efficient Storage Techniques for Digital Continuous Multimedia.* IEEE Trans. on Knowledge and Data Engineering, Special Issue on Multimedia Information Systems, Aug. 1993. (p 20)

[**Reddy94**] A. L. Reddy and J. C. Wyllie. *I/O Issues in a Multimedia System.* IEEE Computer, Mar. 1994. (pp 20, 22, 30)

[**Roscoe95**] T. Roscoe. *The Structure of a Multi-Service Operating System.* PhD dissertation, University of Cambridge, Computer Laboratory, Aug 1995. (p 17)

[**Rose95**] O. Rose. *Statistical Properties of MPEG Video Traffic and Their Impact on Traffic Modelling in ATM Systems.* Technical Report 101, Institute of Computer Science, University of Wurzburg, Germany, Feb 1995. (pp 6, 49)

[**Rosenblum92**] M. Rosenblum and J. K. Ousterhout. *The Design and Implementation of a Log-Structured File System.* ACM Trans. on Computer Systems, 10(1), Feb. 1992. (p 14)

[**Sachs94**] M. W. Sachs, A. Leff, and D. Sevigny. *LAN and I/O Convergence: A Survey of the Issues.* IEEE Computer, Dec 1994. (p 12)

[**Sachs96**] M. W. Sachs and A. Varma. *Fibre Channel and Related Standards.* IEEE Communication Magazine, Aug 1996. (p 12)

[Salehi96] J. D. Salehi, Z. L. Zhang, J. F. Kurose, and D. Towsley. *Supporting Stored Video: Reducing Rate Variability and End-to-End Resource Requirements through Optimal Smoothing.* In ACM SIGMETRICS'96, PA, USA, May 1996. (p 66)

[SAM] *ORL Smart ATM Modules.* http:// www.orl.co.uk /modules.html. (p 89)

[Satyanarayanan81] M. Satyanarayanan. *A Study of File Sizes and Functional Lifetimes.* In Proc. 8th SOSP. ACM, 1981. (p 15)

[Satyanarayanan93] M. Satyanarayanan. *Distributed File Systems.* In S. Mullender, editor, *Distributed Systems*, chapter 14. Addison-Wesley, 2nd edition, 1993. (p 15)

[Schroeder93] M. D. Schroeder. *A State-of-the-Art Distributed System: Computing with BOB.* In S. Mullender, editor, *Distributed Systems*, chapter 1. Addison-Wesley, 2nd edition, 1993. (p 13)

[Seagate93] *Seagate ST11950N/ND ST11951N/ND ST12550N/ND ST12551N/ND Product Manual (Volume 1)*, May 1993. (pp 7, 44)

[Seagate96a] *MR Heads: The Next Step in Capacity and Performance.* http:// www.seagate.com/, 1996. Technology Paper. (p 9)

[Seagate96b] *Seagate Introduces Cheetah — the World's First 10,000 RPM Disc Drive Family.* Seagate Press Release, Oct 1996. (p 9)

[Sha90] L. Sha, R. Rajkumar, and J. P. Lehoczky. *Priority Inheritance Protocol: An Approach to Real-Time Synchronization.* IEEE Trans. on Computer, Sept 1990. (p 16)

[Shenoy95] P. J. Shenoy and H. M. Vin. *Efficient Support for Scan Operations in Video Servers.* In ACM Multimedia'95, San Francisco, CA, USA, Nov 1995. (p 86)

[Shi96] F. Shi and A. Hopper. *A Network Striped Storage System for Video on Demand.* In Collected Abstracts NOSSDAV'96, Zushi, Japan, Apr 1996. Also at ftp:// ftp.cam-orl.co.uk /pub /docs /ORL /abstracts.html#67. (p 22)

[SIOI95] *Preliminary Survey of I/O Intensive Applications.* Scalable I/O Initiative Working Paper No. 1, 1995. Applications Working Group of the Scalable I/O Initiative. (p 27)

[SMART96] *DRAM Articles.* Newsletters, SMART Modular Technologies, Inc., Dec 1996. (p 11)

[Spector89] A. Z. Spector. *Distributed Transaction Processing Facilities.* In Sape Mullender, editor, *Distributed Systems*, chapter 10. Addison-Wesley, 1989. (p 23)

[Spuri94] M. Spuri and J. A. Stankovic. *How to Integrate Precedence Constraints and Shared Resources in Real-Time Scheduling.* IEEE Trans. Computers, 1994. (p 17)

[**Staehli93**] R. Staehli and J. Walpole. *Constrained-Latency Storage Access.* IEEE Computer, Mar 1993. (p 22)

[**Stankovic91**] J. A. Stankovic and K. Ramamritham. *The Spring Kernel: A New Paradigm for Real-Time Systems.* IEEE Software, May 1991. (p 17)

[**Stoller95**] S. D. Stoller and J. D. DeTreville. *Storage Replication and Layout in Video-on-Demand Servers.* In NOSSDAV'95, Durham, NH, US, April 1995. (p 11)

[**Tandem87**] *NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL.* In Proc. 2nd Intl. Workshop on High Performance Transaction Systems, Asilomar, CA, USA, Sept 1987. The Tandem Database Group. (p 23)

[**Tanenbaum92**] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall International, Inc., 1992. (p 14)

[**Teradata85**] Teradata Corp., Los Angeles. *DBC/1012 Data Base Computer System Manual, Release 1.3*, Feb 1985. C10-0001-01. (p 23)

[**Tewari96a**] R. Tewari, D. M. Dias, R. Mukherjee, and H. M. Vin. *High Availability in Clustered Multimedia Servers.* In IEEE Intl. Conf. on Data Engineering, New Orleans, USA, Feb 1996. (pp 23, 30)

[**Tewari96b**] R. Tewari, R. King, D. Kandlur, and D. M. Dias. *Placement of Multimedia Blocks on Zoned Disks.* In Proc. Multimedia Computing and Networking (MMCN), San Jose, CA, USA, Jan 1996. (p 21)

[**Tewari96c**] R. Tewari, R. Mukherjee, D. M. Dias, and H. M. Vin. *Design and Performance Tradeoffs in Clustered Video Servers.* In IEEE ICMCS'96, Hiroshima, Japan, June 1996. (p 30)

[**Tierney94**] B. Tierney, W. E. Johnston, H. Herzog, G. Hoo, G. Jin, J. Lee, L. T. Chen, and D. Rotem. *Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers.* In Multimedia'94, San Francisco, CA, USA, Oct 1994. (p 28)

[**Tobagi93**] F. A. Tobagi, J. Pang, R. Baird, and M. Gang. *Streaming RAID — A Disk Array Management System for Video Files.* In ACM Multimedia'93, 1993. (p 23)

[**Tokuda89**] H. Tokuda and C. Mercer. *ARTS: A Distributed Real-Time Kernel.* ACM Operating Systems Rev., 23(3), July 1989. (p 17)

[**Trace95**] *MPEG-1 Frame Size Traces.* ftp-info3. informatik. uni-wuerzburg. de /pub/ MPEG/, Sept 1995. (pp 6, 49, 58, 61, 63, 65)

[**Vin93**] H. M. Vin and P. V. Rangan. *Designing a Multiuser HDTV Storage Server.* IEEE Journal on Sel. Areas in Commun., 11(1), Jan. 1993. (p 22)

[**Vin94a**] H. M. Vin, A. Goyal, A. Goyal, and P. Goyal. *An Observation-Based Admission Control Algorithm for Multimedia Servers.* In Proc. 1st IEEE Intl. Conf. Multimedia Computing and Systems (ICMCS'94), Boston, MA, USA, May 1994. (p 22)

[Vin94b] H. M. Vin, P. Goyal, A. Goyal, and A. Goyal. *A Statistical Admission Control Algorithm for Multimedia Servers.* In Proc. ACM Multimedia, San Francisco, CA, US, Oct 1994. (p 22)

[Vin95] H. M. Vin, A. Goyal, and P. Goyal. *Algorithms for Designing Multimedia Servers.* Computer Communication, Mar 1995. (p 44)

[Vincent95] G. Vincent. *The Superhighway in Action: The Cambridge Trial.* IEE Review, May 1995. (p 33)

[Wallace91] G. K. Wallace. *The JPEG Still Picture Compression Standard.* Commun. of the ACM, 34(4), Apr. 1991. (p 5)

[Watson95] R. W. Watson and R. A. Coyne. *The Parallel I/O Architecture of the High-Performance Storage System (HPSS).* In 14th IEEE Computer Society Mass Systems Symp., Sept 1995. (p 26)

[Wray94] S. Wray, T. Glauert, and A. Hopper. *The Medusa Applications Environment.* In International Conference on Multimedia Computing and Systems, Boston, MA, U.S., May 1994. (pp 6, 101)

[Yu89] C. Yu, W. Sun, D. Bitton, Q. Yang, R. Bruno, and H. Tullis. *Efficient Placement of Audio Data on Optical Disks for Real-Time Applications.* Commun. of the ACM, 32(7), July 1989. (p 20)

[Yu92] P. S. Yu, M. S. Chen, and D. D. Kandlur. *Design and Analysis of a Grouped Sweeping Scheme for Multimedia Storage Management.* In Proc. 3rd Network and Operating Systems Support for Digital Audio and Video, La Jolla, California, U.S.A., Nov. 1992. Springer–Verlag. (p 19)

[Zhao87a] W. Zhao, K. Ramamritham, and J. A. Stankovic. *Preemptive Scheduling Under Time and Resource Constraints.* IEEE Trans. Computers, C-36(8), Aug 1987. (pp 16, 70)

[Zhao87b] W. Zhao, K. Ramamritham, and J. A. Stankovic. *Scheduling Tasks with Resource Requirements in Hard Real-Time Systems.* IEEE Trans. Software Engineering, SE-13(5), May 1987. (pp 16, 70)