An Architecture for an ATM Network Continuous Media Server Exploiting Temporal Locality of Access

Kneale J Rothwell

September 1999

A thesis submitted to the University of Kent at Canterbury in the Subject of Computer Science for the degree of Doctor of Philosophy

“Nothing worthwhile is ever easy…”

F:\Users\kjr\Thesis\Abstract\Abstract.doc
C:\Users\mw380\Downloads\Table of Contents.doc
F:\Users\kjr\Thesis\Acknowledgements\ACK.doc
F:\Users\kjr\Thesis\Chapter_1\Chapter1.doc
F:\Users\kjr\Thesis\Chapter_2\chapter2.doc
F:\Users\kjr\Thesis\Chapter_3\Chapter3.doc
F:\Users\kjr\Thesis\Chapter_4\chapter4.doc
F:\Users\kjr\Thesis\Chapter_5\Chapter5.doc
F:\Users\kjr\Thesis\Chapter_6\Chapter6.doc
F:\Users\kjr\Thesis\Chapter_7\Chapter7.doc
F:\Users\kjr\Thesis\Bibliography\Bibliography.doc
F:\Users\kjr\Thesis\Chapter_5\Ch5AppA.doc
F:\Users\kjr\Thesis\Chapter_5\Ch5AppB.doc

ABSTRACT

With the continuing drop in the price of memory, Video-on-Demand (VoD) solutions that have so far focused on maximising the throughput of disk units with a minimal use of physical memory may now employ significant amounts of cache memory. The subject of this thesis is the study of a technique to best utilise a memory buffer within such a VoD solution. In particular, knowledge of the streams active on the server is used to allocate cache memory.

Stream optimised caching exploits reuse of data among streams that are temporally close to each other within the same clip; the data fetched on behalf of the leading stream may be cached and reused by the following streams. Therefore, only the leading stream requires access to the physical disk and the potential level of service provision allowed by the server may be increased.

The use of stream optimised caching may consequently be limited to environments where reuse of data is significant. As such, the technique examined within this thesis focuses on a classroom environment where user progress is generally linear and all users progress at approximately the same rate – for such an environment, reuse of data is guaranteed.

The analysis of stream optimised caching begins with a detailed theoretical discussion of the technique and suggests possible implementations. Later chapters describe both the design and construction of a prototype server that employs the caching technique, and experiments that use of the prototype to assess the effectiveness of the technique for the chosen environment using ‘emulated’ users. The conclusions of these experiments indicate that stream optimised caching may be applicable to larger scale VoD systems than small scale teaching environments. Future development of stream optimised caching is considered.

Acknowledgements

The work on this thesis would never have been possible without the support of JISC and their New Technologies Initiative programme. My sincere thanks to them for grant NTI-207, and further to Peter Linington, Alan Ibbetson and Gill Waters for giving me the opportunity to participate in the project.

Particular emphatic thanks to Peter and Gill for their unending support of me in writing this thesis, and their unwavering faith in me completing it. I couldn't have done this without them. Similarly, for having worked with them over the last three years – a very instructive experience that has given much strength to my future career.

A 'thousand thanks' to Andrew Smith, my final year project supervisor while I was an undergraduate at university, for pointing me at the advertisement for a research assistant on the SERVICE project. Without any doubt, had Andrew not brought the post to my attention, I would firstly never have continued at the university and secondly would never have entered for a PhD.

My deepest thanks go to my parents, for having put up with me for the past few years in my efforts to gain my doctorate and find my career. I know I've been demanding and you've always been there for me; I can never thank you enough. I hope that I've been worth all the pain!

Sincere gratitude must be expressed to my small army of proof-readers without whom this manuscript would not be as flawless as it is: Peter Linington, John Salter, Egil Tverberg, and Gill Waters. Further thanks in this respect go to John for acting as my proxy in the UK whilst I completed my write up in Oslo.

Finally, my thanks to all those in the Computer Science department at the university who made it such a great place to work, and to my many friends who kept me going and encouraged me to finish. You all know who you are.

"Chance makes the casual momentous" – Iain M Banks

Table of Contents

1. Background & Direction of Work

With the increasing use of computers as a tool for teaching, inevitable progress has led to a demand for the use of video material as a means to enhance the learning process. The size of video files has itself been an obstacle to this end, in terms of both storage requirement and display capabilities. While the use of CD-ROMs is a solution for off-the-shelf multimedia 'learnware', it is not a solution for locally produced material where there is a need for each teaching workstation to access the same material; to duplicate the material across all the workstations would require a large amount of disk on each, and would additionally consume large amounts of network bandwidth in preparing each workstation. If there are several different courses running over the same term, the set-up time required before each class may itself prohibit such a solution, and giving each workstation a larger disk is not only expensive, but may give rise to multiple points of maintenance in the system. While a broadcast solution is possible, it ties all of the users to the same rate of progress and is very restrictive to the learning process; each student should be allowed to progress at their own pace. What is more appropriate here is a central video server that supplies the video in a real-time on-demand basis. This eliminates the need for the workstations to store any part of the source media files, hence reducing their disk requirement, and possibly enabling the use of diskless workstations. More specifically, what is needed is a networked video file server allowing simultaneous access by a group of users in a classroom, yet giving the illusion that each individual has all of the video that they view on their own workstation.

For each class, the users will progress through the learning material at around the same rate of advance. Many of these users will therefore be using the same media clip at approximately the same instant in time, perhaps separated by a time difference varying from a few seconds up to a few minutes. It is likely that the users will begin close together, with a number of user groups forming as the class progresses, each with similar learning ability. Thus, given the potentially small temporal difference between the accesses of a number of users, a memory cache might be used to reduce access demands on the underlying storage medium. That is, a small section of clip is loaded from storage into the cache for the first user to request it. Other temporally close users' requests are then served by the cached section of clip, and hence access is not needed to the storage device. The cached data can then be discarded once all temporally close users have accessed it, making way for other data with predicted reuse. The concept of *stream optimised caching* is derived from this potential reuse phenomenon.

The accesses of a classroom of users to a centralised media server is clearly a case of Video-on-Demand; each user has independent and immediate control over the media they view. However, the accesses of each user across the class will have a great deal of

similarity. The users will look at the same set of clips, and these clips will usually be viewed in the same order. To add to this, what one user views is likely to affect the behaviour of those around them, and vice versa. In short, the users are highly predictable. Stream optimised caching exploits the predictability of users by maintaining a cache with the set of data that is most likely to be accessed by its users. Thus, although a Video-on-Demand solution designed for library archive clips or a video subscriber service might be used, it may be unnecessarily complicated for the task. These two examples do not share the users' near-synchronisation property of the classroom environment. Furthermore, such large scale Video-on-Demand solutions are very expensive.

The subject of this thesis is the investigation of the feasibility of stream optimised caching in practice. The aims of this work were to produce a low-cost Video-on-Demand solution tailored to the classroom environment. In this environment, the learning task was considered to be generally textual in nature, being reinforced by audio and video material. The assumption is that such media are embedded within some other learning application, such as hypertext; the user does not view continuous media all of the time. Thus, given that (at the time of development) a typical class size is of up to 32 client machines, which is the largest of the teaching rooms at Kent, a target of 20 simultaneous users of the media server was set. To minimise system costs, the prototype server was to be built from off-the-shelf components, and use standard low-cost disks.

From these ideas, the "Server Enabling Retrieval of Video Information in a Classroom Environment[1]", or "SERVICE" project was born. The work carried out on this project forms the basis of that presented in these pages. The SERVICE project therefore prescribed the components from which the prototype server would be built. Namely, it would be constructed from TRAnsputer Modules (TRAMs), and would use an ATM network adapter developed in-house at UKC [Tri95]. Transputers were chosen because of their known and predictable real-time behaviour, and the department's in-house development experience with the processor.

1.1 Aims

This thesis will:

- Introduce the concept of stream optimised caching, explain how it works and how it might be implemented
- Document the construction of a continuous media ATM network file server intended to be capable of supporting a classroom of 20 simultaneous users, each with reasonable quality of service and asynchronous independent control

[1] New Technologies Initiative (NTI) project number NTI-207, run under the Joint Information Systems Committee (JISC)

- Show the structure of the server's file format, which was designed not to be specific to any data type; it will support any kind of audio, video or other continuous media type. Furthermore, the file format will be shown to have been effective in practice.
- Detail the hardware comprising the video server, which is from low-cost off-the-shelf components – more specifically, TRAnsputer Modules (TRAMs).
- Show the design options considered during development, and give the software solutions taken in construction of the server. In particular, it will show the design of:
  - the system as an NFS file server, with extensions to allow real-time continuous media streams
  - a flexible continuous media file system, which is capable of efficiently supporting simultaneous multi-user real-time reads and writes
  - ATM Adaptation Layer 5 (AAL5) on the ATM TRAM and other server TRAMs
  - Internet Protocol (IP) over AAL5 on the server to enable communication with other network end-systems
  - a custom VCR-style control protocol for the control of the streams
  - a cache management structure that enables the implementation of a stream optimised cache
  - a simple stream optimised caching algorithm

- Show that the hardware components are capable of the demands that are made of them, and additionally that the server software components met these demands as the media server was constructed
- Demonstrate a stable and reliable continuous media server appropriate for use within other projects. This stability will be supported by actual usage examples
- Demonstrate the effectiveness of stream optimised caching for the chosen environment, supported by performance analysis
- Show that stream optimised caching can be a useful technique in Video-on-Demand systems in general

1.2 Synopsis of Chapters

Chapter 2 introduces the general field of Video-on-Demand. It begins with a discussion of the nature of both disk devices and continuous media. The fundamental problems involved with the provision of continuous media from a disk storage medium are then presented. A range of solutions to Video-on-Demand is examined, with highlights of the many techniques. The scale of solutions considered range from the small to the very large. The chapter ends by considering work reported in the literature that is similar to stream optimised caching, and how it differs.

The idea of stream optimised caching is presented in detail in Chapter 3, including both the theory and suggestions for implementation. The terminology and abstractions of

both media streams and streams caching are described, and the main caching techniques compared in the context of an environment with reuse amongst the users.

The design and implementation of the prototype server are given in Chapter 4. It begins with an architecture for the prescribed components, and gives a simple feasibility calculation for this architecture. Each subsequent section then discusses implementation options and choices, together with a brief post-implementation analysis to show that the initial calculation is satisfied.

Chapter 5 describes the experimental process used to analyse server performance, and presents the results. The results are discussed, and the effectiveness of stream optimised caching for the experiments is determined. Practical results from a client project are also recorded, showing the actual performance of the server from a user's viewpoint.

Potential future extensions of the streams caching work are given in Chapter 6, which considers both options for the prototype server, and for stream optimised caching. The inclusion of other continuous media provision techniques are considered for addition to streams caching, together with alternative technologies for the server hardware.

The thesis is concluded in Chapter 7 with a discussion of the degree of success of both stream optimised caching and its prototype implementation. The exploitation of temporal locality of access is also discussed in the context of the Video-on-Demand field in general

1.3 Disclaimer

All of the work presented in this manuscript is that of the author unless explicitly stated otherwise. In particular, the idea of stream optimised caching was originally proposed by Peter Linington, together with the initial performance simulation used in Chapter 3. Linington was also the author of the user-behaviour script-generator used when gathering the main results of Chapter 5.

2. Introducing the State of the Art

> "Education will come in packets, or personalised doses, so that we can get the right education, at the right time, in the right amount"
>
> – Prof. John M. McCann [McC95]

With the recent arrival of powerful computers into the marketplace, there is now a proliferation of machines that are capable of high quality video with audio. There has similarly been an increase in the performance of server class machines, which are now being considered as the source of various multimedia data for the many workstations within their domains, most notably video material. Of course, it is not only the offices and teaching rooms that will be the clients of such data; a huge market has been identified in the home. Indeed, the largest potential market to be exploited is that of entertainment, replacing the existing broadcast programmes with those giving freedom of control to the individual user. According to McCann[McC95], the market for multimedia will be at least $55 billion per year by the end of the decade, and most of this market involves entertainment. Not surprisingly then, many large companies are involved in the development of large scale 'Video-on-Demand' systems, so called because users can play movies at a time of their choosing, not a time chosen by the broadcaster. McCann also suggests that the methods by which education is provided will evolve to take advantage of this new technology, such that the individual can acquire knowledge at the point in time that it is required. That is, video will soon play an important role in education.

At the time of writing, there has been no complete solution to the provision of Video-on-Demand to the masses. There have however, been many developments in building individual media servers to provide video or audio to a large number of users, of between five and a few thousand users. Each of these systems is at most at a prototype stage, and field trials of the technology are still at an early stage.

The remainder of this chapter begins with an outline of the properties of continuous media, such as video and audio, as compared to normal data traffic. It then discusses the properties of the underlying hardware technology and the reasons why it is itself a problem in the provision of continuous media. Following this introduction of the basic issues involved, the chapter examines the current efforts in providing a solution to video-on-demand. This coverage is by no means exhaustive – the field of Video-on-Demand is a rapidly expanding one that has the attention of many companies and institutions.

2.1 Properties of Data versus Continuous Media

Media types such as video and audio are referred to as 'continuous' because of their temporal property that once they begin, they are played in sequence at a fixed rate, if they are to be usefully understood. It is important that each part of the continuous media be presented to the user one unit after the next in a predetermined timely fashion. Failure to do so results in gaps in the presentation which, if the gaps are even slight, produces clicks and pops in audio, and/or jitter in a video. This contrasts with normal data files, such as text, where there is no temporal nature involved; normal data does not need to be presented to a user in discrete pieces with hard real-time constraints. Thus, the first major property of continuous media is its real-time nature.

The second difference from normal files, is the very size of continuous media files; they are very large. To put this into perspective, first consider a few pure textual examples. Firstly, the Bible contains less than 2Mbytes of data. Secondly, the Encyclopædia Britannica, whose 32 volumes each contain almost 1.4 million words – approximately 9Mbytes per volume assuming an average word length of 7 characters, or 300Mbytes for all 32 volumes. High quality audio, on the other hand, is typically 192kbps for MPEG audio, which for a mere five-minute clip produces a file of 7Mbytes. In further contrast, low/medium quality MPEG-1 compressed video is around 1.4Mbps (0.1 for audio), making a five minute clip a file of approximately 50Mbytes. Thus, while audio files may be large compared to text, they are small relative to video data, which requires a significant amount of space for its storage. It is easy to see that video data is substantially larger than pure textual data.

For broadcast quality MPEG-2, the video stream is around 4 to 9Mbps; taking a 4Mbps stream with a one-hour movie, that produces a file size of around 1.7Gb! Table 2.1 gives some further examples of the size of compressed video clips for various lengths of clip and some typical bit-rates. This table is a simple multiplication table of file bit-rate against length, but gives a better feel for the size of variously coded video streams.

Table 2.1: Examples of Continuous Media File Sizes with varying length and bandwidth

	Bandwidth of Clip			
	MPEG-1	**MPEG-2**		
Length	*1.5Mbps*	*4Mbps*	*6.5Mbps*	*9Mbps*
5 minutes	50Mb	140Mb	230Mb	320Mb
10 minutes	110Mb	290Mb	460Mb	640Mb
30 minutes	320Mb	860Mb	1.4Gb	1.9Gb
1 hour	640Mb	1.7Gb	2.8Gb	3.8Gb
2 hours	1.3Gb	3.4Gb	5.6Gb	7.7Gb

In a network environment, once a connection has been established with a client, the connection must not be dropped or otherwise suffer from network congestion, such that the client's display becomes jumpy with poor picture quality or disturbed audio. That is, sufficient bandwidth must be reserved within the video server, the network connection, and the client's end system such that quality of service is maintained during the lifetime of the connection. Alternatively, if the connection *cannot* be or *is not* maintained, it must be renegotiated with the client system, as discussed for instance by Zhang [ZK97]. For example, the quality of the video is reduced such that it occupies less bandwidth; the user should not be aware of these negotiations however. Congestion issues are not covered further here; emphasis is placed on the provision of Video-on-Demand within the server.

Providing for one user is relatively easy, but providing for many is complicated by the underlying technology used to hold the media, namely disks. Although tape systems may be used to store thousands of large high quality videos, they do not allow random access to the media in real-time. Pure memory based storage systems *would* provide a solution to the multi-user issues, but are not cost effective – the price per MByte of disk is far cheaper than random access memory. A solution allowing multiple users to share a single disk must therefore be dealt with since one user per disk is clearly not economically viable either.

2.1.1 Compression and Continuous Media

In order to reduce the storage and transmission requirements of media, compression can be employed. For still images, the JPEG (Joint Picture Experts Group) [PM92] compression algorithm is often used, which exploits sensitivities of the human eye system. Namely, the eye is more sensitive to low spatial frequencies, so JPEG will discard the higher frequencies from the coding in order to concentrate on coding the lower frequencies. In addition, the eye is more sensitive to changes in light and darkness levels than to colour, so JPEG uses more effort in encoding light levels (luminance) than the colour (chrominance) components. Clearly, once compression is applied, the resultant file size is dependent upon the image resolution, type of scene and the scene's complexity. The quality of the compressed image may be varied, such that more components that the eye is less sensitive to are discarded. However, if compression is applied too harshly, compression 'artefacts' are visible in the result. The most obvious JPEG artefact is scene 'blocking'.

To further complicate the problem of continuous media provision, media data flows can be categorised as either 'Constant Bit Rate' (CBR) or 'Variable Bit Rate' (VBR). CBR streams are typically uncompressed streams, for example CD audio data. For video data however, the data *needs* to be compressed in order to achieve the necessary retrieval rate from the disk and reduce demands on the network; the compression effectively increases the perceived data transfer speed of the disk and the capacity of the network.

This assumes each client can decompress the data faster than it is transmitted, be it with its CPU or with hardware assists. Each image in a video has varying complexity and if each is compressed with JPEG, for example, each frame has a different size, i.e. is VBR. For Motion JPEG, the size of each frame will be similar except, for example, over a scene change.

2.1.2 MPEG Compression in Continuous Media

For MPEG [Motion Picture Experts Group] compression, *intra*-frame coding is used using essentially the same algorithm as JPEG, that is, per picture compression. These frames are referred to as I-Frames in MPEG terminology. Being designed for moving pictures however, MPEG also exploits similarities between successive images. These *inter*-frame coded images are called **b**i-directionally predicted and **p**redicated frames, or B- and P-Frames. B and P- frames contain only the differences, possibly motion compensated, from their reference frame(s). A P-Frame always uses the previous I-Frame as its reference frame. B-Frames use the nearest adjacent P- or I-Frames, one in the future, one in the past, as the references frames. An example MPEG group of pictures (GOP) is shown in figure 2.1. The reference frames used for P and B-frames in the encoding are shown as dependencies.

Since B- and P-Frames encode only frame differences, they are typically significantly smaller than I-Frames. A consequent limitation arises, because B- and P-Frames can only be decoded if their reference I-Frames and intermediate P-Frames are first presented to the decoder. In figure 2.1, the numbers in the frames refer to the order that the frames are viewed. However, the frames are stored in the order of presentation to the decoder, namely [0, 3, 1, 2, 6, 4, 5, 7]. This is because both frames P3 and P6 need frame I0 to allow them to be decoded. Similarly, B1 and B2 need both I0 and P3, whilst B4 and B5 require P3 and P6 to be decoded. Thus, attempting to provide fast reverse play and fast forward play is problematic because dependency frames must always be presented to the decoder beforehand.

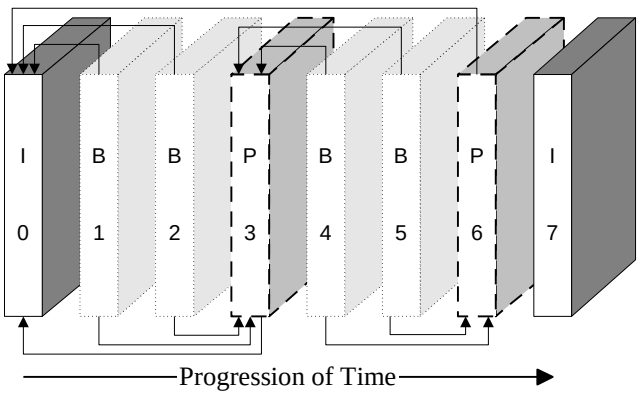


Figure 2.1: Example sequence of frames in an MPEG Group of Pictures (GOP), with dependencies shown. Numbers refer to frame viewing order

The simplest solution to fast playback is to play only the I-Frames, which potentially results in fewer frames per second, and will likely require greater bandwidth than at normal playback speed. The reason lies in that I-Frames are significantly larger than B- and P-Frames. If, for example, the increase in the playback rate is the frequency of I-Frames (8× in figure 2.1), then the actual frame rate remains that of normal playback speed (usually 25 or 30Hz). Bandwidth demands are therefore much higher than at normal playback speed.

For more detailed information on the MPEG standard, the reader is referred to Mitchell et al [MPF96], and Haskell et al [HPN96].

2.2 Properties of Disks

A standard disk typically comprises one of more platters, with one or more adjacent read/write heads per surface of each platter. The surfaces are divided up into concentric tracks, which each are divided into sectors. The tracks from all the surfaces that are at the same radius (or adjacent radii if multiple heads per surface are used) from the disk spindle are referred to as a cylinder group. This is illustrated in figure 2.2. The heads within a disk are generally moved over the disk surfaces on an arm and are thus located within the same cylinder group; the heads cannot read from or write to more than one cylinder at a time. In addition, the sectors within a track are numbered and are generally always read from the first sector requested within a track as the disk rotates. Given this physical construction, it is easy to see where the timing delays occur.

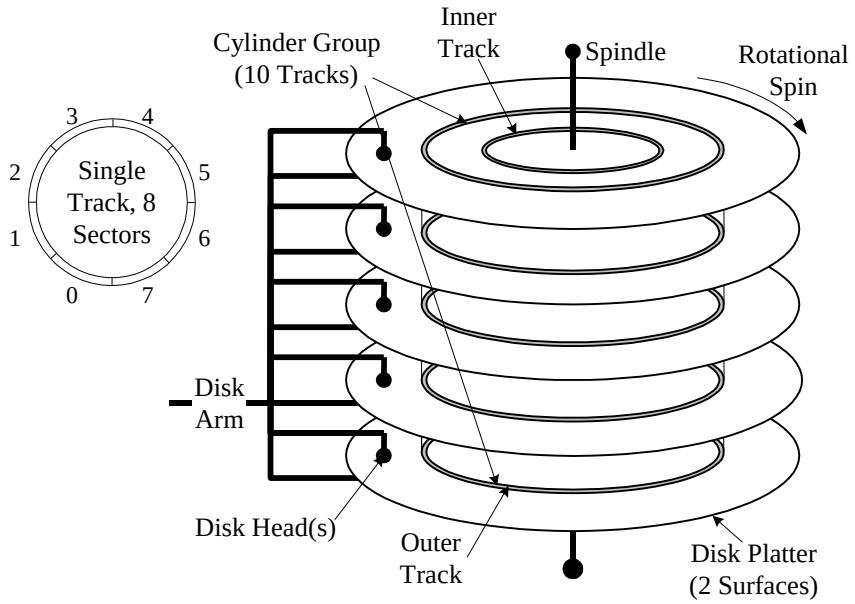


Figure 2.2: Construction of a disk device

The disk spindle is rotating in a typical high-speed disk at 7200 rpm, [WDC97, QC98] for example, with today's fastest disk having a rotational speed of 10000-rpm

[IBM97, FCP97, SC97]. This combined with the density of data on the disk surface determines the actual data transfer speed of the device, which for the example devices given here is between 8 and 16 Mbytes/s. However, the data on a disk might be laid out across the disk, so two consecutive transfers may be on opposite sides of the disk. In this case, after the first transfer is complete, the disk heads will move across the disk to the location of the next data to be read. The time it takes to move the disk heads from after the end of one transfer to the cylinder group of the next is referred to as the 'seek time', which is typically around 8ms for today's fastest devices. Given that the time between two frames in video is 40ms for video at 25 fps, the disk seek time is significant. To add to this, once the heads have reached the target transfer area, the disk must rotate until the sector to be read comes underneath the head such that it can be read. This is called the disk 'rotational latency'. Clearly, this latency time is dependent upon the rotational velocity of the disk. The average latency for the disks above is approximately 3ms, so the most serious delay when using disks is from seeking track to track.

2.3 Properties of Networks

In terms of continuous media, the demands on a network are that it has enough bandwidth to support the desired number of users in precisely the same way as a disk. This similarity in terms of requirements is also shared in respect of a need for constant and very low latency in the transfer of data.

The most common type of network is without doubt IEEE802.3 (Ethernet). With a shared bandwidth of only 10Mbps it is of limited use in multi-user continuous media environments for two reasons. Firstly, there is simply not enough bandwidth to support more than a few users. Secondly, Ethernet is a shared bus media, which means contention on the bus, and consequently, highly variable transmission latencies to a network destination.

Ethernet's successor, IEEE802.12 (100Base VG AnyLAN), or 100Mbps Ethernet, addresses the first issue involved with continuous media over Ethernet. 100Base Ethernet uses a packet switched hub to connect the network nodes; 100Mbps Ethernet cannot use a shared bus architecture like its predecessor. However, the bandwidth provided by the hub is still shared between the nodes; only one Ethernet packet may be switched through the hub at any time. Consequently, the 100Mbps bandwidth is shared between all network nodes directly attached to the hub. Like its predecessor, therefore, 100Base Ethernet is prone to variable network latency if the hub is congested. In any case, 100Base Ethernet is appropriate only for Local Area Networks (LANs) and is inappropriate for large-scale commercial multi-user environments.

In the larger scale, the telecommunications and cable networks will be the carriers of such data. In recent years, there has been an enormous effort in the communications sector to standardise the 'Broadband Integrated Services Digital Network' (B-ISDN)

interface. B-ISDN is intended as a standard worldwide replacement for the current multiplicity of existing public communications systems. For B-ISDN, the data is transmitted using a specific implementation of cell switching techniques known as Asynchronous Transfer Mode (ATM) [Kya95]. For this reason, B-ISDN and ATM may be viewed as synonymous.

ATM uses a fixed size 53-byte cell as the physical unit in contrast to Ethernet's variable length packets of maximum 1518 bytes. In further contrast to Ethernet, ATM's specification does not include the physical media interface; a number of separate specifications enable ATM to be carried over a number of physical implementations including both electrical and optical interfaces. The fixed size cell also has a number of advantages against packet switching techniques.

Cell switches do not require as much memory for switching as packet switches; a packet or cell must be fully received before it may be forwarded. The cell itself therefore allows low and steady transmission latency through a switch. An ATM switch may furthermore switch cells in parallel; the bandwidth on an ATM switch is not shared. Indeed, a 100Mbps ATM switch can theoretically supply 100Mbps to all directly attached network nodes; the switching performance of the switch itself is far in excess of the 100Mbps, in this example, that it provides to its clients.

In bandwidth terms, ATM specifies bandwidths from 1Mbps through 155Mbps, then 622 and 2,488Mbps; these latter bandwidths are of course intended for use between network switches. Each ATM cell has a 5-Byte header, which is altered by switches in order to create a connection-oriented path based on a small local identifier. ATM networks may therefore be arbitrarily large. Indeed, this was an important factor for its use within B-ISDN.

Given ATM's use within B-ISDN, giving it a global 'into the home' significance, and its ideal properties of high-bandwidth low-latency, it is a perfect candidate for a network medium with respect to continuous media. For almost all the papers presented in this chapter, an ATM network was used, if not implied or assumed.

Now that the physical constraints of the technology have been outlined, it is possible to discuss the properties of continuous media and the problems introduced through their storage on disk devices.

2.4 Issues in Provision of Continuous Media

Taking the example of a single user viewing a digital video, it is easy to see that the transfer rate of the disk is far greater than the individual consumption rate of a video – 8Mbytes/s versus ½ Mbyte/s. For this example disk then, it should be possible to support 16 users given the maximum transfer rate of the disk. However, if each of those users is viewing a different clip then each clip incurs a seek penalty for each disk access.

Unfortunately, the cumulative seek time across 16 users at 8ms, assuming each frame is accessed as needed, comes to 128ms, which is far beyond the 40ms inter-frame time of each stream. In this simple calculation, assuming an average seek time between users of 8ms, only five users can be supported before the cost of the disk seek becomes too great.

2.4.1 Scalability of Disk Bandwidth

Disregarding for now the issue of seeking, the relatively small number of users of a single disk needs improving far beyond the simplistic example of 16 users above. To achieve this, several disks can be placed together in a number of configurations to exploit the aggregate transfer rates of several disks. This was first outlined by Patterson et al [PGK88] – the so-called 'Redundant Array of Inexpensive Disks' (RAID). In [PGK88], it is said that gains in disk technology with time as compared to CPU and memory technology are very modest. Without the application of RAID, there will soon be a performance crisis caused, relative to CPU and memory speeds, by the very slow storage devices. [PGK88] considers RAID levels 1 to 5; a detailed study of the 7 RAID levels (0 to 6) can be found in [CLG94].

The essential principle in RAID is that the data is placed (or 'striped') across several disks in an array and retrieved in parallel, hence providing an increase in bandwidth that is directly proportional to the width of the array. In addition, by distributing the data over a number of disks, hotspots on accesses are avoided that otherwise result in a few saturated disks whilst the others remain largely idle.

Reliability in RAID systems is of great concern, since the failure rate of an array is multiplied by the number of disks in it; a 100-disk array is 100 times more likely to fail. Consequently, RAID systems employ techniques to add redundancy into the data without having any significant performance impact, thereby adding reliability to the array.

RAID systems can be distinguished by two features: i) the granularity of data interleaving; and ii) the method and pattern in which redundant information is computed and distributed onto the disk array. The data interleave can be either fine or coarse gained. In fine grain, the data interleave is small enough such that all I/O requests will span all the disks. This gives high I/O rates, but only one logical I/O may be in progress at one time. In coarse grain, the data interleave unit is bigger, which results in small I/O accesses using only some of the disks, while large accesses use all the disks. In redundancy terms, there are two possibilities, either concentrate the redundancy data onto one or a few disks, or distribute it over all disks in the array. The latter option is better, since it avoids access hotspots and load balancing problems.

A summary of the seven RAID levels is given in table 2.2. As remarked in [CLG94], there has been much confusion over what a 'RAID level' means and of what each level consists. In this discussion, only RAID types 3 and 5 are of particular interest, which are

byte striped and block striped, respectively. The other types of RAID are more intended for storage reliability and crash recovery than for high performance.

Table 2.2: Summary of the seven RAID levels (0 to 6)

Level	Name	Description
0	Non-redundant	Has no redundancy data. Data is striped over disks with arbitrary block size. A single disk failure will result in data loss.
1	Mirrored	Each disk has a copy disk. If one disk fails, the lost data can be recovered from the mirror disk. Data is striped over disks with arbitrary block size.
2	ECC	Based on error correction in memory systems. Data is bit-wise striped over disks. Uses Hamming codes on multiple parity disks to enable recovery of data.
3	Bit-Interleaved Parity	Similar to Level-2, but since a failed disk controller can be identified (unlike memory controllers), a single parity disk is used. The data is bit (or byte) striped over all disks in the array.
4	Block-Interleaved Parity	Similar to Level-3, but the unit of striping is an arbitrary sized block. Thus, an access that is less than the striping unit size will access only a single disk. As in Level-3, writes must update the data disk(s) and the parity disk. Since all writes must update the parity disk, this can become a bottleneck.
5	Block-Interleaved Distributed Parity	Similar to Level-4, but the bottleneck on the parity disk is removed by distributing the parity data uniformly over the disks in the array. This has the additional advantage of distributing the actual data over all the disks, not *n-1*.
6	P + Q Redundancy	Similar to Level-5, but uses a stronger redundancy coding than parity; parity is capable of identifying only a single failure. This level uses Reed-Solomon codes with a minimum of 2 redundancy disks “P + Q”.

In [CLG94] it is said that levels 2 and 4 are viewed as inferior to the other levels and are thus discounted from any decision of which RAID level to use. Further discounted is level 0, which uses no redundancy at all; level 0 cannot therefore be used in environments that require fault tolerance.

RAID level 3 can only allow one access at any instance because the data is bit-wise striped over all the disks in the array. In contrast, several small reads may be performed in parallel within a level 5 system. During reads, the parity disk is not involved unless a disk has failed, so read performance of level 3 is not optimal.

RAID level 5 is the fastest of all RAID levels in respect of read performance because it distributes the data over all, not *n-1* disks in the array. In fact, RAID level 5 is the fastest solution for small and large reads and large writes. Furthermore, levels 3 and 5 are equivalent if the stripe unit of level 5 is small enough such that all data accesses will always uses all the disks in the array, though 5 distributes the parity data. Level 5's weakness is small writes, since these require a read-modify-write cycle on the parity disk. However, for a continuous media server, small writes are not an issue.

2.4.2 File System Interfaces

Naturally, with the progression from normal file systems to supporting continuous media, the protocol mechanisms driving the data flow were first driven by standard I/O calls, namely *open*(), *read*(), *write*(), *seek*() and *close*(). This is however, not the nature of continuous media, which is governed by the logical time at which a 'frame' is due. This is easier to see when considering how a standard video cassette recorder (VCR) is controlled – the basic operations are play, fast reverse play, fast forward play, pause, and stop. An additional primitive comes out of the random access capability of Video-on-Demand, that is, to reposition the media to a specified time on the timeline of the media. In this approach, there is no explicit read request for each frame to transfer. Instead, consecutive frames are sent continuously according to a local clock at the data source.

The problem with the former data I/O interface is that it requires *read*() control exchanges with the server for each frame of video transferred; in the latter case, the server continues sending successive frame data to the client after only a single 'play' exchange. The standard I/O mechanisms therefore suffer the disadvantage of additional latency in the receipt of the media data at the client end. With the threat of increased delays in the data pipeline comes a need for more buffering to cope with the variable delay in receipt of data. If the data transmission is driven by the server supplying the data, the latencies in the system are reduced to the time to cross the network and the delays within the two end systems. Consequently, buffer requirements in the client to reduce jitter in the playback are reduced, which minimises ultimate cost.

The price that must be paid by driving transmission from the server is that it must model each playback stream. That is, it must maintain the current position in the media, work out when each frame is due, and respond to user requests. Typically, this requires an index over each media file using as primary key the logical timestamp. Thus, the server can determine which portion of the media is due at any given time within the model of the playing stream. An index is required whether the stream is modelled at the

client or server in any case, since which data blocks belong to which frames still needs determining if VCR operations are desired. A client fetching data with *read()* requests, however, will need to maintain its own stream model.

The Video-on-Demand solutions to date have largely concentrated on minimising buffer usage within the server in order to keep the server cost down. In these systems, a large cache is not seen to have any benefit given the large size of files and *apparently* random user access patterns; a very large cache would be needed to exploit any reuse using standard file systems caching techniques, which is prohibitively expensive because of the sheer amount of memory required.

2.4.3 Video-on-Demand (VoD) .vs. Near Video-on-Demand (NVoD)

The many solutions that have been attempted vary in their degree of flexibility in user control. The reason is that there are fundamental problems inherent in the hardware technology used to provide the service, namely, with the disks. The ideal in Video-on-Demand is to give each user the impression that they are the only user of a server system, that is, that the system responds immediately to their requests, and that the quality of playback is always very good. In practice, this is difficult to achieve and the user is instead constrained in some way. For example, the user might have to wait a minute before their chosen movie begins to play, and furthermore, that 'pause and resume' functionality is similarly subject to such start-up delays. Such service can therefore be referred to as 'Near Video-on-Demand'. Other systems are query based, such that a delay might be experienced before playback begins, and then the playback cannot be controlled once it is playing. Such a system may be appropriate, for example, in the provision of stock market news reports requested by brokers.

Fast forward and fast reverse playback causes additional disk seeking per unit time, because although all data is being read, not all of it is used because frames are being skipped in the playback. The rate of seeking increases directly proportionally to the rate at which disk blocks are consumed. Thus, for the increased bandwidth demands on the disks, effective utilisation of this bandwidth is reduced because more time is spent seeking. In addition, from the network's point of view, there is also a likely increased bandwidth requirement, especially if the stream is VBR. Taking MPEG as an example, only the I-Frames and future adjacent P-Frames can be played back; they are typically far larger than the P- or B-Frames. VCR operations are consequently problematic since a further increase in bandwidth may be required, which might exceed the maximum during normal playback. If the bandwidth reserved for any one stream is breached, this can have a knock on effect on other streams resulting in playback jitter.

2.4.4 Programme Generation and Editing

The discussion above has concentrated solely on the *playback* of multimedia material. A related issue is the recording and editing of media, video in particular. This issue is not covered in any depth here where the concentration is very much on providing entertainment or education, not on providing the facilities to produce the materials for broadcast. The recording/editing issue has slightly different requirements to the playback of continuous media. In particular, buffering is required to build up a quota of data in order to make disk writes efficient during recording. The scheduling of these writes are not real-time critical however, unless buffer memory is at a premium. Further, recording systems are not generally required to be multi-user; editing is usually performed on a single-user workstation and uploaded to a multi-user server once edited. Additionally, editing requires jitter-free playback of material that may be scattered arbitrarily over one or more file systems.

Now that a background to the difficulties involved in continuous media has been set out, the next sections cover the current efforts of the field in providing a video-on-demand file server. The first section looks at the individual server, and the second is concerned with scaling up the solutions to provide Video-on-Demand to the masses. This coverage is by no means exhaustive, as there are many reported solutions. An effort has been made however, to cover the full spectrum of the kinds of solutions attempted.

2.4.5 Finding a Solution to Video-on-Demand

As said above, the seek time and latency involved in making a disk access are significant factors in the provision of Video-on-Demand. Disks give their best performance when transfer sizes are as large as possible – when transferring an entire cylinder say, before making any attempt to seek to another cylinder group. In so doing, the time spent in useful transfer is maximised and the time spent seeking and doing no data transfer minimised. This is indeed the solution taken by all systems, though there is a balance to be struck between the size of a transfer and the size of the buffers required. The rate of data consumption of the clients is much lower than the rate of transfer of the disk, so reading a lot of data from the disk does not cause undue delays to client transmission. However, it then takes time for the memory buffer to be consumed and thereafter the next section of media read from the disk. Across multiple users, the buffer requirement can be significant if the time between disk accesses for a single stream is a long period of time, for example, in the order of seconds. Gemmel at al[GVK95], provide an excellent tutorial covering the problems relating to buffer management for multiple users, in addition to discussing the most common data layouts and scheduling strategies on disk and of editing, recording and interactive control of continuous media data.

2.4.5.1 Basic Techniques

2.4.5.1.1 Buffering Techniques

Double buffering is an often-used technique, where while one buffer contains data being transmitted to the network, a second is being filled from the disk. This is shown in figure 2.3. Once a stream's transmit buffer becomes empty, the buffers are switched and the transmitter continues without any delay; the new disk buffer is then filled from the disk as the new transmit buffer is consumed. This cycle repeats once the transmit buffer again becomes empty. Clearly, the disk must always have placed enough data into the disk buffer before the transmitter empties its buffer, else jitter is seen in the resulting stream. In figure 2.3, the disk fills an empty buffer before the owning stream consumes its play-out buffer. For example, between times (a) and (b), the first stream switches its play-out buffer from 0 to 1. An extension of the idea is to use a single buffer, which is cyclic; if the transmitter reaches the end of the buffer, then it wraps around and continues at the start. In this case, the disk reader merely keeps ahead of the transmitter and ensures that the stream does not run dry.

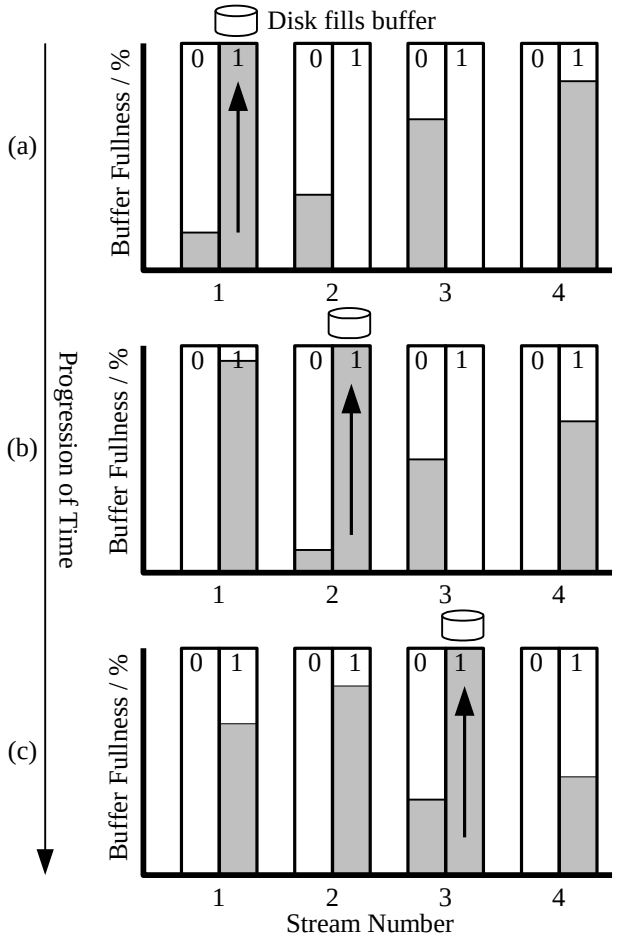


Figure 2.3: Example of double buffered play-out technique

The ideal in handling buffers for multiple users is to fetch only as much data as is needed for each stream, given the time to the stream next being serviced. If too much data is fetched, there will still be a large amount of data available for a stream, perhaps to the point that no disk fetch is needed; the expense here is simply that buffer memory has been used up and that it ought to have been allocated elsewhere, or not been needed at all so reducing system cost. The buffers of several users therefore need to be serviced by the disk in a cycle such that each receives a quota of disk time and all streams use the minimum amount of buffer space per scheduler cycle.

2.4.5.1.2 Disk Scheduling Techniques

The maximisation of the transfer size is not the only way of maximising effective disk transfer utilisation. Where several disk requests are pending at any one moment, the order in which those requests are serviced can be varied such that the overall time spent seeking is minimised. Steinmetz[Ste95] gives a very good survey which covers the main techniques used in disk scheduling, both within traditional file systems and within continuous media systems. These are summarised here.

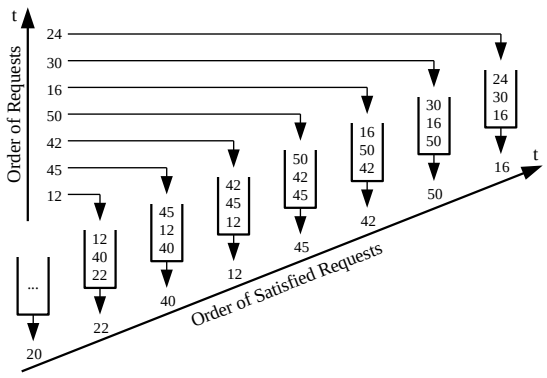


Fig. 2.4: First Come First Served (FCFS)

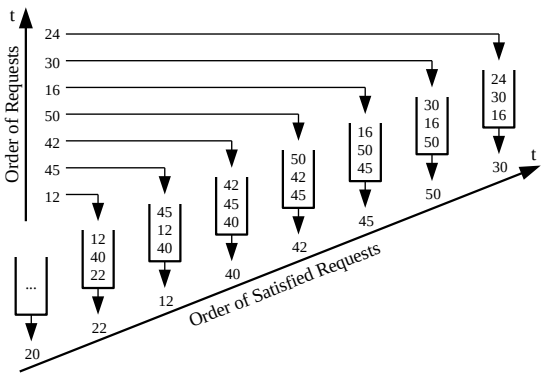


Fig. 2.5: Shortest Seek Time First (SSTF)

The simplest way to honour a number of requests is in First Come First Served (FCFS) order as illustrated in figure 2.4[Ste95]. In all of these examples, the disk scheduler considers only the three most recent requests in the transfer queue with the given algorithm. FCFS can unfortunately result in high seek overheads because the requests may be ordered in such a way that each alternate access is on the opposite side of the disk. A way to minimise the seek overheads and maximise disk utilisation is to honour the request with the Shortest Seek Time First (SSTF) as shown in figure 2.5[Ste95]. However, this can lead to the starvation of service for any requests that are on the inside or outside of the disk, since all those in the centre of the disk are serviced. A further technique is therefore to sweep the disk head back and forth across the disk, servicing requests in the direction of travel until there are no more requests in that direction, such as illustrated in figure 2.6[Ste95]. At this point, the direction of travel is

reversed. This Scan algorithm has a variant, C-Scan, which sweeps the head in only one direction during an I/O cycle, then seeks the head directly back to the sweep start point for the next cycle. This is shown in figure 2.7[Ste95]

The above SSTF, SCAN and C-SCAN techniques attempt to minimise seek overheads and therefore maximise disk utilisation. The problem is that they do this without any real-time constraints on any of the I/O requests; the disk scheduling may be termed 'best effort'. For real-time materials, such as continuous media, the time in which each request is serviced is important. For Scan based algorithms, a stream can starve if not enough data was read in the last cycle and the current cycle takes longer than the play-out duration of the buffer. Consequently, a common technique used in continuous media scheduling is Earliest Deadline First (EDF). That is, the stream that will run out of playback buffer first is given disk I/O first. As with FCFS however, this algorithm can lead to excessive seek overheads and resulting poor disk utilisation. Scan-EDF tries to address this problem by using the Scan scheduler for requests with the same deadline, but its effectiveness depends upon the number of requests with identical deadlines. Quantisation of the requests' deadlines is one method of ensuring that this occurs.

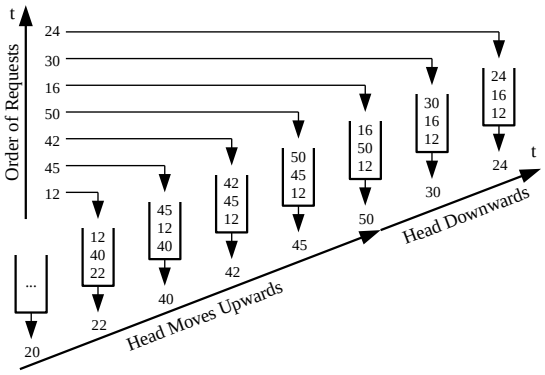


Fig. 2.6: Sequential SCAN

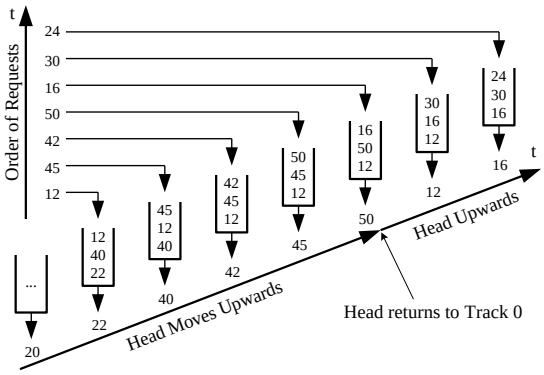


Fig. 2.7: Cyclic-SCAN (C-SCAN)

Another common technique in disk scheduling for continuous media is Round-Robin (RR), where the streams are serviced in a fixed order within a cycle. The cycle then continuously repeats, with each stream always being serviced at the same point in the cycle. Whereas FCFS services each transfer request to completion, RR is fairer in that each participant within a scheduling cycle is given the same upper bound on time to complete a transfer. Thus one of the participants does not hog the disk and starve the others of disk bandwidth. As with FCFS, RR can lead to high seeking costs if the streams are not ordered in the cycle by their relative positions on the physical disk(s), that is, a Scan-RR. Indeed, the Grouped Sweeping Scheduling (GSS) scheme uses the RR policy for its groups, where each group is a number of localised streams. The groups are then serviced using a Scan policy.

In GSS, if all streams are placed into the same group, the technique degenerates into a Scan scheduler. Similarly, if each stream is placed into its own group, GSS degenerates into a RR scheduler. The technique is discussed and further optimised by [CKY93].

2.4.5.2 Buffering Strategies

2.4.5.2.1 Supporting random-access in Retrieval of Continuous Media

The work of Liu et al [LDS95] is concerned with providing video-editing facilities within a single workstation and does not qualify as 'Video-on-Demand'. However, the techniques used do apply to service provision within such servers, hence its inclusion here. Namely, buffer arrangements for jitter-free playback are covered – two-buffer and k-buffer schemes. The double buffering scheme has already been covered above, and is used in Lui's paper only as a means to explain the k-buffer technique. In both of these, the unit of transfer is a frame: "It is our belief that the frame-level model is more proper for a video medium than an audio medium, because usually a video frame cannot be displayed until all the data which composes one frame has been transferred completely". This requirement is clearly important for a video editing application.

The random access capability is constrained to the groups of pictures (GOPs) within a movie, where a GOP is a continuous sequence of frames bound to within a single cylinder. The access within each block of frames is limited to sequential access. This constrained random access is required in order to satisfy real-time constraints during the playback of the media, that is, to minimise disk seek overheads. At the same time, since the GOPs within a cylinder may be placed in any order, the scheme gives disk placement flexibility during the editing process.

The k-buffer approach is born out of a 'transfer anomaly' during playback. Namely, when the disk is finished transferring data into the secondary buffer, the display will have consumed at least part of the primary buffer. Thus, once the disk is finished with the secondary buffer, it could fill the section of the primary buffer that the display has consumed. Unfortunately, in practice it is not possible to determine how much of that buffer the display has used. Thus, the reserved buffer space can never be fully utilised. The k-buffer approach therefore uses a number of buffers such that those buffers the display has consumed may be filled with new data from the disks. In so doing, the utilisation of buffers is increased and the slack time left at the end of each scheduler cycle is increased. The more buffers that are used, the smaller the GOP size can be (the minimum group size is one frame in one group of frames) because the group of pictures is the unit of buffer transfer. Buffer requirements are minimised by minimising the slack time, since that reduces the occupancy time of any data in the buffers. An example of k-buffer strategy is given in figure 2.8.

Decreasing the size of groups has the additional advantage of improving disk space utilisation since larger GOP sizes leave more unused sectors in each cylinder, i.e. those remaining that are smaller than the GOP size. Furthermore, since the GOP is the unit of transfer, the GOPs within a cylinder may be retrieved out of order and avoid the additional re-sequencing latency that a single read buffer would require. Such a re-sequencing is required because the display buffer must be decoded sequentially.

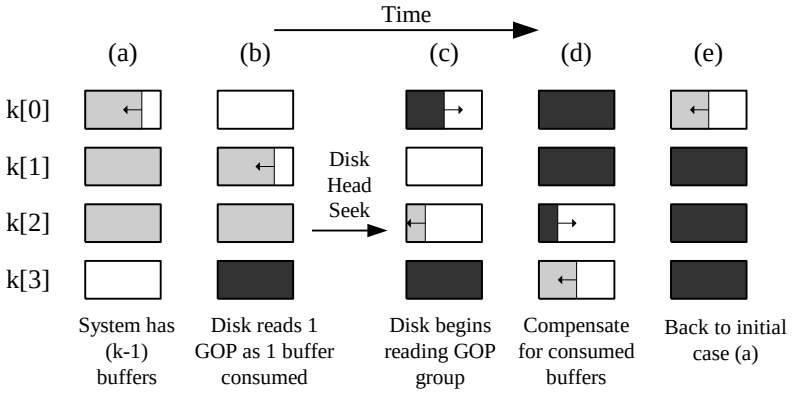


Figure 2.8: Compensation in the k-buffer compensation scheme with k=4 [LDS95]

The k-buffer technique is intended primarily to allow jitter-free playback during data transmission to a display screen, though the space requirements of k-buffer are shown to be markedly lower than the simple double-buffer scheme. The authors believe that the flexibility of placement gained from the k-buffer technique will enable editing support with jitter free playback.

2.4.5.2.2 The Continuous Media File System – CMFS

Anderson et al. [AOG92] uses an extension of standard I/O primitives to provide hints to the server on the media consumption rate. The server is supplied at connection with the rate value R and a 'cushion' value Y, by which the server will guarantee to stay ahead of the client. The server itself does not use any meta-file to determine the timestamps of data to be transmitted. The media streams are timed using the rate R given during connection. Consequently, the system cannot properly handle VBR streams, though it does allow streams to use different rates. The file system in fact uses a contiguous file layout, though the exact structure of the file system is not dictated by the paper. Instead, the file system only needs to be parameterised by two variables: U, the time taken to read *n* consecutive blocks of a file, regardless of the starting point of the disk heads and the first file block to be read; and V, the time taken to read *n* blocks from starting block *i*. Resource in the server is reserved for each client session, each with fixed work load and delay bounds. Since both of these values are parameterised by R, Y, U and V, resources can *work-ahead* of any real-time streams. The disk scheduler is cyclic and

reads consecutive blocks for each serviced stream, though the order in which streams are serviced may change from cycle to cycle.

For the purpose of new session acceptance, a schedule of disk accesses is searched for that would allow the rate requirements of all active streams to be met – a *workahead-augmenting set* (WAS). However, the WAS is only *feasible* if the time it takes to execute does not violate the buffer guarantees of any playing stream, and that no buffers overflow. The operations within a WAS do not specify the execution order. The operation sequence Ω considers all permutations of the streams in the WAS. A single operation sequence in Ω is only feasible if the WAS is feasible. An Ω sequence is safe if it does not cause any of the streams to starve in a scheduler cycle.

An algorithm for deriving the *minimal* feasible WAS is given and proved in the paper. The algorithm essentially enumerates all operation sequences Ω for a feasible WAS, which is safe relative to Ω. The disk scheduling policy defines a concept of slack time which is derived from the minimal feasible WAS. The slack time can be used for non-real-time operations provided enough time is available, but is otherwise used among the stream clients to increase their workahead.

In the paper's conclusion, an area for improvement is the use of buffer space. If two sessions are considered to have opposite phases in the scheduling cycle, then the sum of their buffer space usage at any instant will always be less than the sum of the maximums used by each. Thus, the two can share buffer space, which might possibly improve performance and increase the number of sessions that can be accepted.

2.4.5.2.3 Maximising Buffer and Disk Utilisation for News On-Demand

Ng and Yang [NY94] develop the idea in [AOG92] above. The phases of several streams are distributed across the cycle of the server such that the buffer consumption of all streams at any instance is only part of the sum of the buffers allocated to the streams. Consequently, as illustrated by figure 2.9, the total buffer space required is much less, possibly by up to 50%, and the streams are said to share buffers. Another way to view this is that as every stream consumes its data, the used space is freed back to the free-pool for immediate reuse by the stream that next accesses the disk. It is not necessary to allocate each stream an independent buffer because each buffer would then only be full for a fraction of its lifetime and therefore the buffer space under-utilised.

The work assumes a query based 'News-on-Demand' system, where effort is given to maximising the number of queries through the system. Queries are accepted if this can be done without adversely affecting existing streams. Otherwise, they are placed into a queue of waiting queries. For the active queries, the disk scheduler prefetches data for the streams. However, it takes account of the fact that too much prefetching does not aid performance because the data within the buffer is not consumed over the single cycle, thus ineffectively occupying memory during the cycle. An 'Intelligent Prefetch' (IP)

therefore considers the queries waiting in the queue and fetches data ahead of the activation of the query. Therefore, once the query is admitted, it can begin play-out immediately and maximise system query throughput.

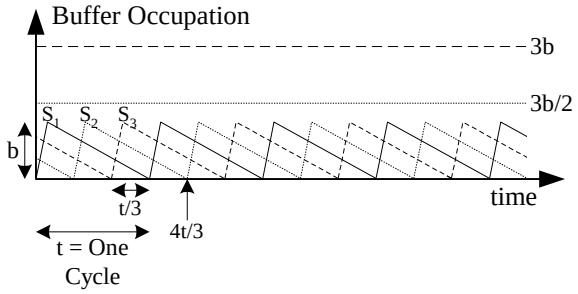


Figure 2.9: Buffer sharing for three streams with identical consumption rates [NY94]

2.4.5.2.4 Piggyback Merging Policies for Video-on-Demand

Aggarwal et al [AWY96] have found that users do not perceive the difference between a movie played at normal speed, and the same played back 5% more quickly or slowly. Consequently, for a number of users whom are viewing the same piece of material at around the same point, the users can be made to converge on the same point in the video. The technique simply slows a leading stream and speeds up a trailing stream, as illustrated in figure 2.10.

Once the two streams meet, they are merged into one – the so-called 'piggyback' event. The paper discusses the most effective policy for the merging of streams in order to save bandwidth. The findings of the report suggest that most gains are to be made in a 'window' over the start of a movie. This is because slowing a stream already well in progress for purposes of merging can only last until the end of the clip, so the efficiency gain here is far less compared to a merge early on in the material. Its snapshot algorithm for determining which streams to merge is found to be the most efficient of the piggyback merging policies in this field of work. The origin of the piggybacking concept can be found in [GLM95]. The paper ends with a question of whether MPEG based material could take advantage of the piggyback technique. However, given that the major gains in the technique occur early on in material, a fast version of the start of each movie could be encoded at the cost of disk space. The result would be a system more in the spirit of Video-on-Demand since the end users would not experience any latency in start-up as found in the similar batching (phase based) technique, and might be a good companion technique. What should be done with the audio track whilst using the technique is not discussed; a user would very likely notice a change of around 5% as a change in pitch. Similarly, the omission of some samples to allow the audio to catch-up would be obvious as clicks in the audio playback.

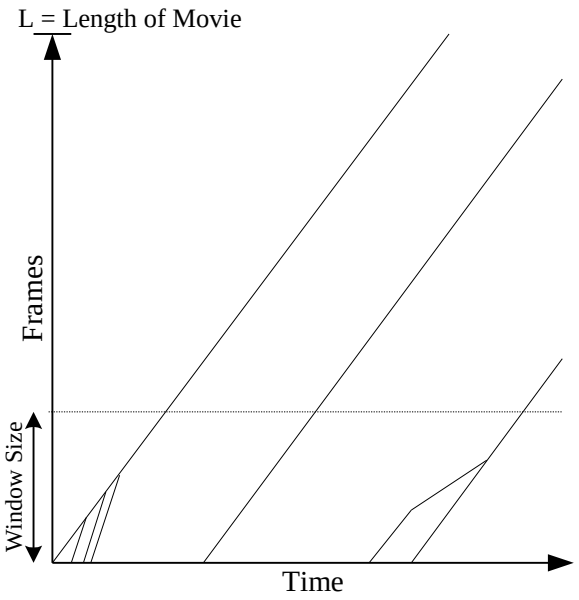


Figure 2.10: Generalised simple merging policy (piggybacking) [AWY96]

2.4.5.3 Contiguous File Systems

2.4.5.3.1 The Tiger Shark File System

The 'Tiger Shark' file system of Haskin and Schmuch [HS96] is IBM's commercial multimedia file server and is based on a RAID structure. With a 256Kbyte block size the disk throughput is maximised, together with a file striping across the disks to achieve high aggregate bandwidth and load balancing across the disks. The interface to the file system is through standard AIX file system calls, with the server recognising stream access by its sequential access and playback rate. Such a recognition results in bandwidth reservation and deadline disk scheduling. Tiger Shark is particularly concerned with system reliability given the loss of earnings due to system failure in a commercial Video-on-Demand operation. Component failure tolerance and replication are considered.

2.4.5.3.2 The Olivetti "Disk Brick" – a RAID-3 Multimedia File Server

Chaney et al [CWH95] discuss the implementation of Olivetti's multimedia file server, or "Disk Brick", which considers the consumer clients to be directly connected network objects rather than end systems. The aim here is to reduce end system contention on internal buses. [CWH95] argues that any attempt to use intelligent disk data layouts is rendered irrelevant because all factors conspire against it. For example, SCSI disks map bad sectors out and provide a continuous model to the user; the underlying structure of the disk cannot be accessed. Trying to take advantage of disk geometry is therefore rendered impossible. Given that the most up-to-date disks use techniques to eliminate rotational latency effects and are optimised for large sequential accesses, [CWH95] chose to use these characteristics when designing a continuous media file system. The result is a flexible contiguous file system that allocates large I/O buffers (and therefore maximises

transfer size and speed) and minimises seek overheads. A RAID-3 Byte striping is used for the data to achieve high transfer rates together with a cache for the file meta-data such that disk I/O is dedicated to data transfer. [CHW95] makes the claim that such a caching strategy is vindicated because the very size of video files is guaranteed to flush any kind of data cache.

2.4.5.3.3 A Storage Server for Continuous Media

Lougher and Shepherd [LS93] use a similar technique to the Olivetti disk brick with their log based continuous media server, which again uses contiguous files on the disk. The log based file system is an 'append-only' system that writes blocks sequentially over the surface of a disk. Deleted file space is reclaimed by a compaction process during system idle-time that maintains file contiguity. The block size used for transfer is dependent upon the media and the point of access to the stream; the VBR nature of streams is taken into account such that the logical unit of transfer on disk is a unit of time rather than of physical size. This arrangement leads to a simpler disk scheduler, which is round robin, and prevents any one stream from accumulating too much data in the memory buffers. Unlike all of the above solutions however, the server drives the data transmission using a VCR style RPC for the client control. The physical layout of the data over the RAID is that the logical blocks are then striped over the disks with dedicated directory, metadata and meta-index data files on a separate disk.

2.4.5.3.4 The Stony Brook Video Server

The work of Vernick et al [CVV97] is unusual in its use of 10Mbps Ethernet as the network medium. Here, a Real-time Ethernet network protocol, RETHER, is described which allows bandwidth reservation and the use of an Ethernet network in a real-time environment. The intended target is within cable TV networks where entertainment material is requested, and after a short delay of possibly minutes, the item begins playback. Similarly with many other systems, a session is not started until the server has sufficient resources for it.

One of the major design goals of the project was to use standard equipment in the construction of the server, hence presumably the use of Ethernet. A related paper [VVC96], is a development of the work in [CVV97] and describes a higher performance server SBVS-2 (Stony Brook Video Server) that uses 100Mbps Ethernet. Client control is VCR based with a meta-index on the video data to allow random access. The scheduling strategy is cycle based, but the service order is governed by the use of a Scan scheduler in order to minimise seek overheads. New stream admissions are allowed only if the new stream can slot into the scan schedule without violating service agreements. The file data itself is striped across a RAID-5 disk array such that each data unit on each disk does not cross any cylinder boundaries and therefore incur a seek penalty. The units

of each video are laid out contiguously on the disk with the most frequently accessed videos on the outside cylinders where transfer rate is greatest. The buffer staging is a simple double-buffer technique in favour of easier implementation.

Both papers describe *client* buffering techniques to reduce jitter in the playback, though this issue is not considered here where the concentration is more on providing the video server. The article makes an interesting comment that "because it is easier to add memory incrementally to a system than to replace disks, extending buffer space is a more popular technique to enhance the performance of the disk subsystem". This comment is likely aimed at lengthening the cycle time of a video server to further reduce seek overheads; the aim is to reduce the seek costs. However, if significant volumes of memory are to be used within video servers, it does raise the question of whether the fundamental buffering technique used within most systems is the most appropriate. Indeed, this is the question raised within this thesis. [CVV96] also covers the issues of fast forward and rewind operation with MPEG based clips, including usage of dedicated fast-forward and reverse play files and lower quality images. The dedicated files are derived from the original file by skipping frames and thus produce files that are played at normal speed (25 or 30 fps), but present a fast-review to the viewer. Lower quality images are suggested, since otherwise the dedicated files are typically produced by skipping B and P frames, thus producing a high-bandwidth media file in comparison to the original file.

2.4.5.4 Disk Scheduling Strategies

2.4.5.4.1 A Variable bit-rate Continuous Media Server

Neufeld et al [NMH95] attempt to deal with the problem of VBR streams, and variable rates across streams, even if each is CBR. The disk scheduler also works in cycles, but each slot in the cycle is of fixed time length, of the order of half a second. Consequently, the size of the units on the disk varies for VBR material, which is similar to the work of Lougher and Shepherd[LS93]. The user control primitives are open, prepare and read, and the system does not support variable rate playback or repositioning. However, this inflexibility is important in the admissions control. Each stream can be written down as the sequence of its unit sizes, and hence is described by a vector. The maximum number of blocks that can be read within a scheduler cycle must be known; this requires a calibration step. The maximum number of blocks that the server can read within the duration of a slot is measured by reading equally spaced blocks over the entire surface of the disk, so taking account of the maximum seek time.

The admissions control algorithm tracks the costs of the current disk schedule by summing the vectors of all active streams into the future, up to the end of the stream to be admitted. An example is given in figure 2.11. When a new stream is requested, if any

element of the resultant vector is greater than the maximum found in the calibration, then the session is rejected. Thus, in figure 2.11, if the maximum vector was 10 units, the new stream would be rejected. However, in some cycles the number of blocks to read in a cycle will be less than the maximum possible, giving an idle disk at the end of a cycle. The slack time can be used to read ahead and effectively reduce the future slot totals by the size of the read-ahead. Consequently, a slot total of more than the maximum enabled by the disk can be allowed, since it merely empties or reduces the amount of pre-cached data. The latter approach therefore enables the acceptance of requested sessions, such as that in the example, where the former more conservative method would have denied the session.

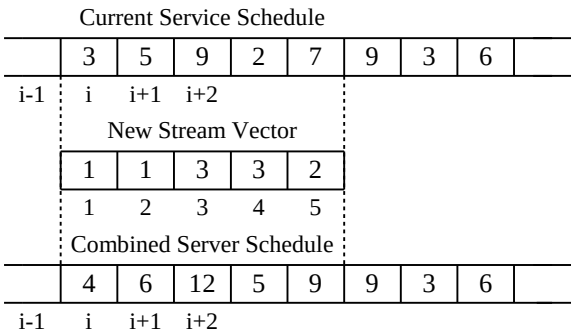


Figure 2.11: Calculation of Disk Scheduler Vectors [NMH95]

2.4.5.4.2 Batched-SCAN Disk Scheduling

Another disk scheduling technique called BSCAN is developed by Kenchammana-Hosekote and Srivastava [KS97]. In this scheme, the playback guarantees of the active streams are maintained whilst responding to VCR operations. The user's expected system response time to such operation requests is used in two techniques called active and passive accumulation. These techniques use the slack time at the end of the scheduler cycle to gather data for a state transition such that once the transition occurs, the playback quality of all streams remains unaffected. At the same time, the transition is attempted as quickly as possible.

The BSCAN algorithm is an extension of SCAN, but the requests of a single stream are batched together into a single continuous stream of disk block sized requests. For each BSCAN scheduler cycle, some or all of these unit requests are fulfilled.

Passive accumulation is where the scheduler cycle length remains the same, and slack time is used to read extra data over all client streams. In active accumulation, the cycle length is made larger such that more data per stream can be gathered. Clearly, the latter approach cannot be taken immediately because it might starve those streams earlier in the cycle, causing them to jitter. The ideal algorithm therefore begins passively, and switches to active accumulation once it is safe to do so, as illustrated in figure 2.12. Here, after a VCR operation is made, the passive phase begins and increases the buffer use with each

scheduler round up to point *x*. The switch to active accumulation lengthens the cycle, thus reducing the buffer content of the streams earlier in the cycle. This can be seen as a knee in the buffer trace to point *x*+1. As demonstrated however, the drop in buffer occupancy does not drop below the minimum required to allow continuous undisturbed play-out. Consequently, the state change of the VCR-operation can be performed as quickly as possible, and does not adversely affect any playback guarantees.

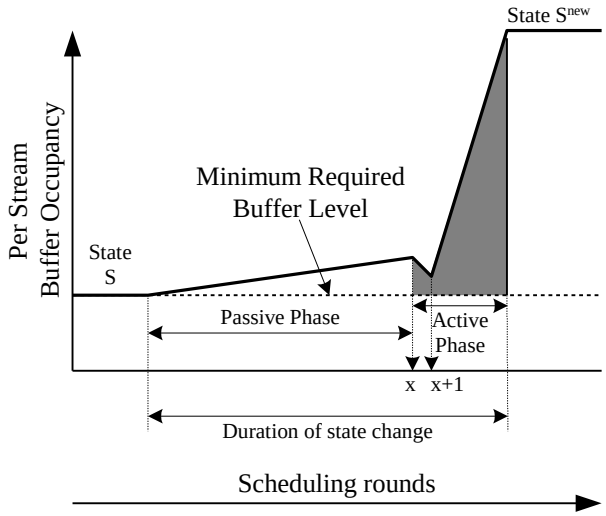


Figure 2.12: Passive and active accumulation of data during scheduling rounds to achieve seamless state change [KS97]

2.4.5.5 Disk Layout Strategies

2.4.5.5.1 A Mass Storage System for Video-on-Demand

The paper by Hsieh et al [HLL96] is an examination of the performance of RAID-3 arrays in a hierarchy to support more users than one RAID may achieve by itself. Clearly, a single RAID-3 would not provide for many users, and so multiple RAIDs are employed to achieve a higher performance and capacity. The performance of various different arrangements is experimentally examined in search of the best way to distribute data across the RAID devices. The paper concludes that application level striping, which block stripes data similar to RAID-5 across the RAID-3 volumes in a volume set, is the most efficient method.

The two arrangements are striping data across several RAIDs at the device level (logical volume striping) and striping data over several RAIDs and/or logical volumes (application volume striping). For logical volume striping, the data stripes retrieved from the RAIDs need to be placed back into sequence before they are passed on from the device driver layer. In the application volume striping case, since the data access is under application control, no reconstruction of data is required and data are accessed in a pipeline fashion, that is, in sequence and not in parallel as a RAID device.

The findings reported are that the application level striping achieved better results in allowing a greater number of clients because it had lower data reconstruction delays from multiple RAID arrays than those required by wide logical volume striping. It concludes however, that the video data needs to be laid out on the disks properly such that the worst case seek times, which are the dominant factor of retrieval latency, are reduced.

2.4.5.5.2 A Novel Video-on-Demand Architecture for VBR Data

Lau and Lui [LL95] tackle the problem of VBR streams by using a database over the media files that maintains the timestamps, locations and sizes of all the frames. The layout of the data on the file system is a round-robin scatter of equally sized fragments, in this case two cylinders, over several disks. The unit of disk transfer is the fragment such that disk-seek and latency costs are minimised. On the server, each fragment may contain one or more frames, but a frame cannot cross a fragment boundary. The disk scheduler finds an optimum execution order for the streams for each disk in the parallel array; since the database records the number of frames in each fragment, the deadline for each stream can be calculated. This is done in such a way as to minimise the buffer requirement of each stream and therefore to handle the buffer build up problem caused by VBR streams. Thus, in this scheme, the disks are used in parallel, but do not constitute a RAID since the scheduler addresses each disk separately. Their scheduler in fact uses the EDF algorithm in scheduling the stream requests of each disk.

2.4.5.5.3 A Server in Support of Movie on Demand

An interesting disk layout structure is suggested in the work of Özden et al [OB94] where streams are phase constrained, and users are mapped to the nearest phase. In this layout, each movie is divided into p portions of size n × block size d, as given in figure 2.13. The phase difference is therefore determined by the row size n × d. Each row of the movie is n blocks, so the movie is a matrix of p × n. The data is stored on disk in major column order from 1 to n, each unit being a contiguous sequence of p blocks. (A drawback of the scheme taken is that it cuts the frames up across the columns – a better approach would be to use a frame as the dividing unit instead of the media block size d).

The basic idea behind the approach is that an entire column of data can be read far more quickly than one block of data can be played back. Therefore, the playback of a stream can be started at any phase that is a multiple of the phase difference. Effectively, *p* copies of the movie can be simultaneously transmitted, each being skewed by the phase difference from the previous stream. The buffering technique uses a circular buffer for a total of *p* buffers; one buffer for each of the concurrent phases. Each buffer is in fact a double buffer – one buffer for the disk read and one buffer for the play-out of the previously read phase.

A problem with the scheme is that the head needs repositioning from column n back to column 1 at the end of the disk file. A suggested solution to this is to use two disks with a submatrix across the two disks in major column layout, effectively striping the data over the two disks. Thus, the penultimate column is read from one disk and the disk begins to seek back to column one. The final column is then read by the second disk whilst the first disk completes its reposition. This approach requires that the phase difference used in the movie is at least twice the duration of the disk repositioning time.

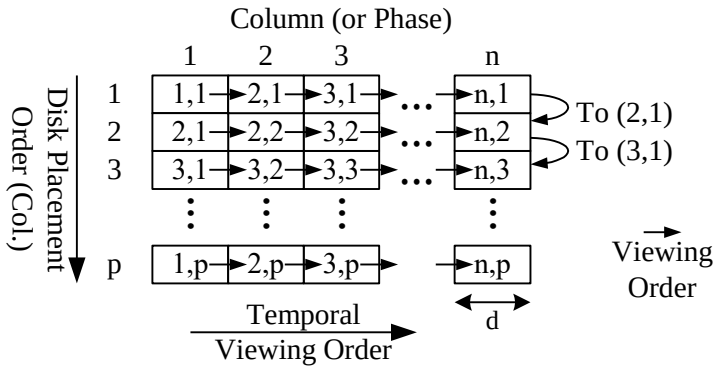


Figure 2.13: A movie described as a matrix. Viewing order is left to right, top to bottom; storage order is top to bottom, left to right [OB94]

A problem with the submatrix solution above is that at any instance one of the disks is always idle. Thus, a further suggestion is to use only one disk and to read *m* columns when column $n - m$ is reached. Therefore, the disk may reposition to column 1 whilst the last m columns are played out. The cost in this option is from the extra cache memory required for reading the last m columns in a single scheduler cycle.

Of course, for VCR operations, the user is confined to playback of one of the phases, which gives a very coarse granularity in terms of frame rate. In this case, a dedicated fast forward version of the movie is suggested, which uses the same phase technique on the physical disk. Other granularity issues are also addressed.

2.4.5.5.4 Design and Analysis of a Video-on-Demand Server

Srivastava et al [SKS97] also the use phase technique. The paper initially describes the problem of arbitrary client control in that it can require a flow control feedback loop. This has a knock-on effect of the scheduling of the disk head in the server, therefore affecting other users. Consequently, the layout of the data on disk has each movie interleaved with itself, in precisely the same manner as Özden[OB94]. An example is given in figure 2.14; in Özden's terminology, figure 2.14 has $p = 3$ and $n = 5$, with one media unit being equivalent to one column. Each media unit is therefore placed a predetermined distance after the previous unit, thus interleaving the media units on the single disk. The users' expected system response (visually), the response time (relatively

slow) and actual response time are used to advantage within the server. For example, after a reposition, the exact location specified by the user is translated into the nearest active playback phase. Furthermore, a slight delay might be added by the system to achieve a closer fit to the user's request.

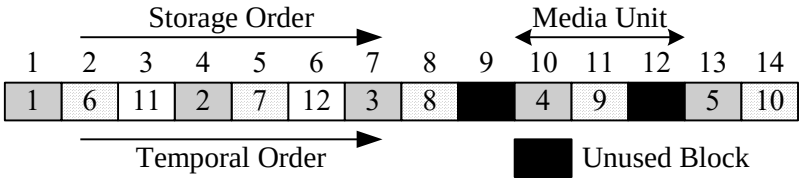


Figure 2.14: Storage layout of a movie with 12 media units [SKS97]

For fast review operation, frames from the adjacent phases are played back to the user, for example (1, 6, and 11); the potentially large gaps between phases are not seen as a problem to the end user. For higher quality playback however, separately encoded fast forward and fast reverse play versions of each movie are suggested using the same interleave and phase technique. In this approach, each user then belongs to a fast review phase, rather than skipping successively between normal play streams.

2.4.5.5.5 Effective Utilisation of Disk Bandwidth for Interactive Video-on-Demand

Cheng et al [WCL96] propose a solution for dealing with fast forward and reverse play through use of disk layout and access strategies that do not require extra bandwidth reservation over those of normal playback. The disk architecture is of two levels: the first level consists of RAID-3 disk arrays; the second of *M* RAID-3s grouped to form a single structure, effectively creating a RAID-5 of RAID-3s. This architecture was originally presented in [OWC95] and is shown in figure 2.15.

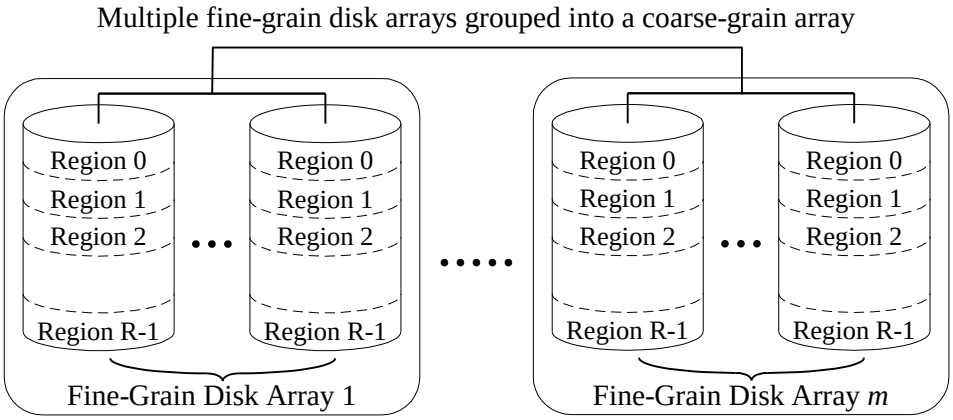


Figure 2.15: Two-level general disk architecture [WCL96]

The 'fine-grain' RAID-3s are divided up into *R* partitions of equal size called 'regions', and all file data are interleaved over all such devices according to a formula. Namely, the file block offsets in each region are *i* + multiples of $2\times R\times M$, where *i* is

distributed over the *M* arrays in a manner illustrated by figure 2.16. When accessing the disks, the streams are divided into groups in the same way as phase based access. Each group accesses a different RAID-3 in any scheduler cycle, so there are at most *M* groups. The streams of each group are therefore synchronised on accesses to the same blocks within a single region. The disk heads sweep over the disk in both directions, region by region for each scheduler cycle, and thereafter the disk groups rotate among the *m* arrays.

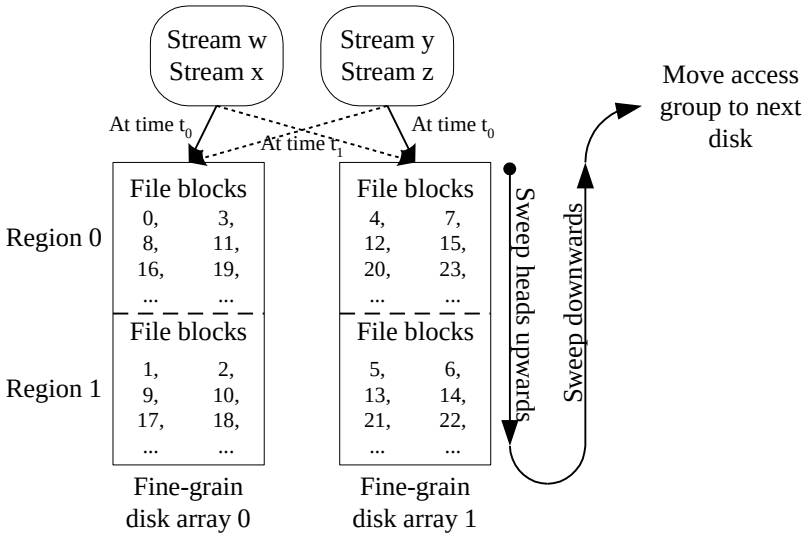


Figure 2.16: Example of region disk block layout and subsequent disk access order (M=2; R=2) [WCL96]

The fast operations are achieved by 'leaping' some data blocks and displaying only some groups of pictures. This is similar to the use of review functions on a CD player, where the audio is always played in the forward direction, but sections of the audio are skipped over in the appropriate direction. Using MPEG as an example, the GOP size is a variable chosen at the point of encoding, and can be the unit of fast review. However, provided the duration of the GOP is short, say less than half a second, this may be a better solution to the provision of the review capability; previous attempts to solve the fast playback problem were based on the analogue equivalent (that is, to increase the frequency of media units in the playback), which might not be appropriate for the digital medium. This solution is better in that it does indeed not require extra disk bandwidth to perform, since all the data fetched are viewed resulting in perfect disk utilisation. The only cost of the scheme is that extra data buffering is required, though only during the review operations. For example, in figure 2.16, if a double-speed fast forward stream is to play GOPs 0, 2, 4, 6, 8, 10, 12, 14 and 16, then those blocks may only be played once GOP 18 is about to be retrieved in the next service cycle. This is because of the retrieval order of these blocks from the regions, for example, 0, 2, 10, 8, 4, 6, 14, and 12. Consequently, there is a delay of several GOP sized gaps in the start-up of the fast

operation since the fast playback buffers need to be filled. The number of buffers required depends upon *M*, the number of RAID-3s, and number of partitions per node *R*. However, though such buffers are needed, the disk seek costs are the same as those of normal play.

2.4.5.5.6 MARS – Massively-parallel And Real-time Storage

Buddhikot et al [BP97] use an internal hierarchy of ATM Port Interconnect Controllers (APICs) [DCP95] and each APIC system is used similarly to a disk within a RAID system. That is, 'chunks' are distributed across the APIC nodes, and each chunk is stored within a RAID within one APIC. This is similar to figure 2.15[WCL97], replacing the coarse grain array with APIC nodes. One of the APICs is used as a controller/manager and is not involved in client data transfer. The unit of chunk is a number of consecutive frames, so the chunks are of variable length. If care is not taken therefore, one of the APICs can end up with all the largest chunks in a clip, and hence cause a load imbalance during normal playback. For example, one chunk might be one MPEG GOP of 16 frames distributed over four APICs. Hence, the first APIC always transmits the I-Frames. The storage policy of the file system within each APIC is up to the individual node. The paper points out the problems caused by fast review access, in that it causes load imbalance across the APICs because some APICs can then be accessed more than others. Their solution is to use 'safe skipping distances' (number of jumps over APICs) during playback such that the load during review operations is evenly balanced. It also comments on the problem of poor disk utilisation during review operations because only a fraction of the data fetched from the disks is actually used in network transmission. A sharing of data between clients is suggested in order to reduce these effects, but the paper points out that Video-on-Demand is typically done on a per stream basis in order to minimise the buffer requirements of the servers. That is, no caching is performed with memory only being used to smooth the accesses of the disk for each stream client over time.

2.4.5.5.7 Prime-Round-Robin (PRR) Disk Layout Strategy

Kwon et al [KCL97] use a disk layout strategy which tries to prevent disk access overload hotspots from occurring during the playback of the media, in particular, during arbitrary rate playback. The Prime Round-Robin (PRR) technique uses a prime number as the divisor when deriving which of the disks should be selected for placements and retrieval. In this approach, the media are broken into segments, which are equivalent to one or more MPEG GOPs. These segments are read in a pipeline for each stream, such that the data is de-clustered. The client control is through VCR primitives with 'immediate' system response to user requests. By eliminating the hotspot areas on the disks, disruption to service caused by VCR operations to the disk schedule are greatly reduced. The delay to the service on a VCR primitive is shown by simulation to be less

than the RR placement technique, and hence potentially enables the support of a larger number of users.

2.4.5.6 Flow-Control in Video-on-Demand

One example in the provision of VCR controlled MPEG is by Rowe et al in [RP94]. In this paper, full variable rate playback, pause and resume, and arbitrary positioning control is allowed. Each target movie uses a metadata file for a time index and a data header; this is generated when the media are placed onto the continuous media server. The index gives the server random access to the MPEG data, whilst the header allows the client to determine the dimensions and rate of the video. Rowe's player software uses a concept of multimedia ropes where segments of video and audio clips are triggered on a logical timeline, which in sequence constitute the rope. The server system performs frame dependency checks such that only a valid sequence of MPEG frames is transmitted. The paper points out however, that such a solution can result in a much higher bandwidth, such as in reverse play because all dependency frames must be transmitted first. The client systems communicate with the server to achieve a flow control mechanism so that if the clients' buffer begins to overrun, the server can stop sending frames and hence save bandwidth. Therefore, no frames are transmitted from the server that are not usefully decoded and displayed. The results of the system are somewhat modest because the MPEG decoder is in software and its output is displayed on an 8-bit colour X-console. Nonetheless, the paper outlines a working system, which with improved decoder and display speeds, should easily handle real-time VCR controlled MPEG-1 material.

2.4.6 The Greater Challenge – Towards a Scalable Solution

All of the solutions presented in the previous section are single server solutions with finite limits that would easily be reached in a commercial environment. The provision of Video-on-Demand to the masses requires solutions that can store thousands to tens of thousands of video and audio clips, and play them to hundreds of thousands of users. All of the solutions to this involve a storage hierarchy, with very large tape archive systems at the bottom, which feed into many Video-on-Demand servers, which then supply the material to the users. Such an architecture is given in figure 2.17. Tape archive systems are extremely cheap per Mbyte of data stored compared to disks, but cannot support users directly. The problem to be solved here is how, if at all, to replicate material across the tape servers and disk servers, and further, which material to store within the disk servers to offer the best level of service. In addition, this must be done in such a way that service is not interrupted, that is, is tolerant to system and network failures and highly reliable. As with the single server solutions, the cost of the system as a whole needs to be kept to a minimum. The above issues are covered by Stoller and DeTreville[SD95], Rowe et al [BR96], and Shepherd et al.

2.4.6.1.1 Scalability Issues for a Networked Multimedia Storage Architecture

Shepherd et al [LPS94, LSP94] also consider the use of network links which are too slow to be used for service provision to clients. The inter-network bridge in figure 2.17 is an example of such a 'slow' link connecting to other high-speed data networks. Here, bulk data transfers are used to ship data between network domains, that is, between high bandwidth network segments separated by slower speed network links. Within domains, such replication may be in real-time to minimise traffic overheads; by not using a bulk file transfer, the replication has a minimal impact on network bandwidth and server loading within the high-speed domain.

The domain formation is a dynamic process based on the capabilities of the network connections between systems. Thus, a domain hierarchy is formed. File priorities and file interests are then used to determine which files should be migrated across the network domains. As a file gets further away from its source, its interest decays such that it is no longer propagated. Shepherd reasons that the network striping of data to allow a greater number of users will also reach a bottleneck, just like RAID end-systems. Therefore, alternative techniques must be found to find a scalable solution. Redundancy replication and scalable compression techniques are suggested.

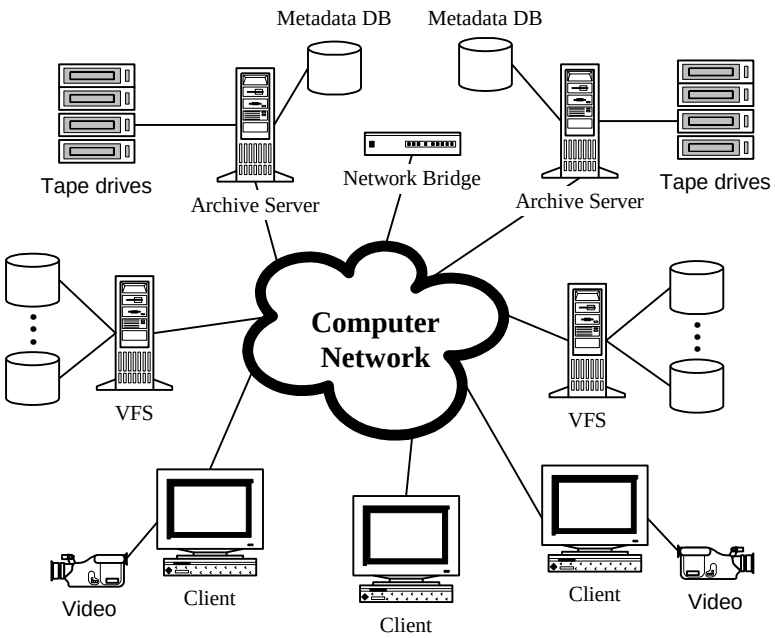


Figure 2.17: Distributed Video-on-Demand System Architecture

2.4.6.1.2 Indexes for User Access to Large Video Databases

Rowe et al [RBE94] examine the issue of user query and access. A movie database, such as those in figure 2.17, contains details of all the available media; the query interface obtains a key to the movie the user wishes to view. This key is then supplied to an appropriate video file server for subsequent playback. Where no video server has the

requested item, it is spooled off of a tape archive system into a video file server. A transcode step is taken during such a transfer, accounting for varying internal data representations across different video file servers. The paper details the design of the underlying database, including the types of data content and user query styles. Administrator 'hints' are also considered such that the system can load material into the file servers for expected or scheduled usage at a given time.

2.4.6.1.3 Hierarchical Storage Management in a Distributed Video-on-Demand System

Brubeck and Rowe [BR96] discuss the management of the storage hierarchy as outlined in the introduction to this section. The video or tape server selected for client playback is done through use of server and network load, and service-wait times. The actual video file servers used in the Berkeley system are developed by various companies, so the architecture is flexible in this respect. Hence, the transcode step is included to take account of the varying storage and access methods used across the servers. The mechanism that determines which movies should be loaded into the video server uses a priority system. Each movie has a priority function, which varies with the time of day. The set of highest priority clips is maintained in the video servers. Two algorithms are given, one for load balancing, the other with additional service-wait minimisation. Rowe is careful to point out that encryption and conditional access mechanisms must be developed if Video-on-Demand using copyrighted material is to be successful.

2.4.6.1.4 Storage Replication and Layout in Video-on-Demand Servers

Stoller and DeTreville [SD95] provide an analysis of the general problem, particularly of balancing system cost against fault tolerance and reliability. Their algorithm for content distribution over a storage hierarchy uses estimates of mean demand for each title. They conclude that for an additional hardware cost of around 13%, server availability is improved by around 75 to 90%.

All of the above systems are either prototypes that are still in development and are at present of modest size, or are purely theoretical. With the deployment of digital television via cable TV networks and pilot ATM networks, sufficient communications infrastructure will soon exist to carry the data of Video-on-Demand into the home and office. Powerful computers already exist in the home, soon to be followed by set-top boxes in the digital TV revolution. Inevitably then, the means to receive Video-on-Demand will soon come into being – the industry needs only to find the technology to exploit the multi-billion dollar market for interactive entertainment from the home.

2.5 Another Piece in the Puzzle – Caching in Video-on-Demand

The above solutions have largely concentrated on provision of commercial servers providing large sets of movies to a huge customer base, or else a very large set of smaller

clips to a smaller populace in the hundreds. Ultimately, all users have been considered behaviourally independent with no significant temporal overlap over whatever media is being viewed. With the recent massive drop in the price of computer memory, it perhaps becomes viable to employ a more significant amount of memory in order to improve the performance and service-level of Video-on-Demand systems. This might be seen to be similar to the way that very fast (and relatively expensive) cache RAM is used between a CPU and its main memory in order to improve the overall throughput of the main memory. Memory is already used between disk systems and network transmission in *all* cases. Thus, if the memory within continuous media servers is made larger, there is a question of how best it might be put to use, and can the gain in performance outweigh the cost of the additional RAM? This question has also been raised by Wen et al [WCL97] of the University of Hong Kong, whose work ran at the same time as that presented within this thesis.

Wen et al [WCL97] address the problem of providing sufficient bandwidth to a large number of between 1500 to 5000 clients. The underlying principle is that if two requests that access the same programme occur closely in time, the data of the first client is forwarded, or *relayed*, to the second. Hence, no disk bandwidth is required by the second client. A memory cache is used to form *relay chains*, such as those in figure 2.18, where the leading stream within each chain relays its data to the following streams in the chain. In the example, the chains relay their data from the top to the bottom of the diagram. As the relay uses cache memory, additional disk bandwidth is not required beyond the leading streams. The memory usage is optimised by allocating it to the shortest relaying distances. Client service requests are admitted if sufficient disk resources exist, otherwise they are placed into a queue. Such queued requests are rejected if they are not serviced within a set timeout.

In an evaluation of the relaying mechanism for a set of short programmes such as cartoons or music videos, the transmit-mechanism without the relay cache requires 56Mbytes. With an additional 230MByte relay, the number of requests serviced increases by an average of 80%, whilst the rejection rate drops from 60% to 13%. The effectiveness of the relaying mechanism is dependent upon the popularity distribution of the video clips; better results are achieved where the clips' popularities follow a Poisson distribution with some clips being far more popular than others. These gains come at the cost of additional memory. However, as Wen et al remark, the continuing drop in the price of memory favours implementation with a relaying type mechanism.

In an extension of the original relay work, Wen et al [WCLW97] develop a generalised relay mechanism, which uses the expected popularity of clips to speculatively form relay chains. A speculative relay chain is headed by an actual access instance and a chain formed from buffer space despite not having any following streams. The most

popular clips are selected to form such speculative chains if memory resources permit. The idea is that since the clip is popular, a new access instance will immediately use such a speculative chain, in which case it is no longer speculative. Their conclusions indicate that the use of popularity indicators on the source material gives a reasonable but significant improvement in the level of service provided and drop in the number of rejections over the original relay mechanism.

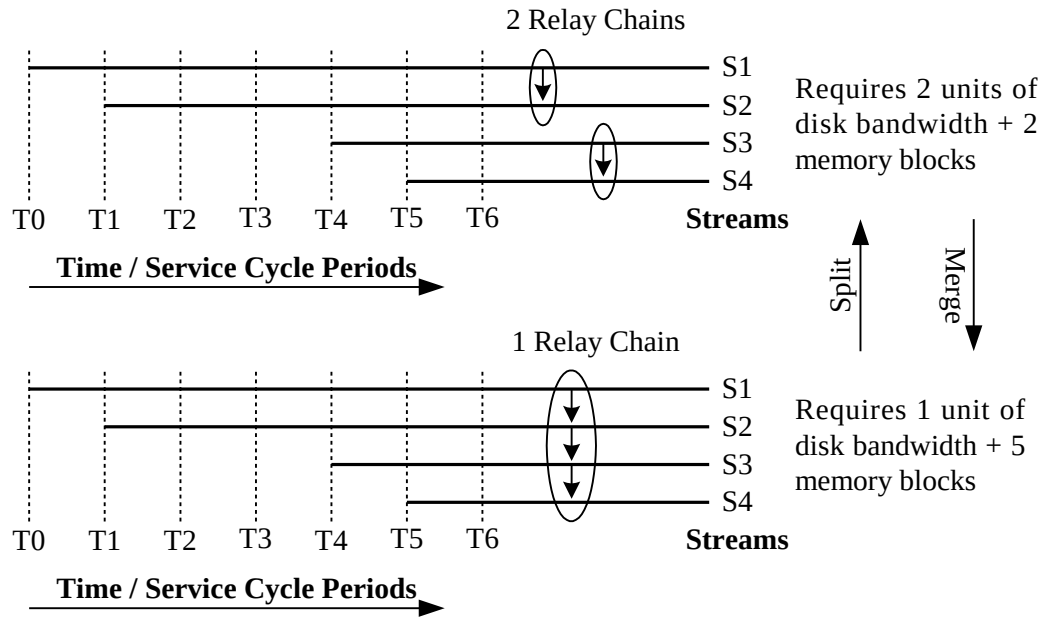


Figure 2.18: Example of Relay Chain organisation with splitting and merging operations [WCL97]

The technique used by Wen [WCL97] and that developed within this thesis has two good analogues: firstly, that of cache memory between a CPU and its main memory; and secondly, that of a cache in a multi-user mainframe's file system to improve data throughput to its users. With careful use of this cache memory, the overall data throughput achieved by the disks is potentially far beyond that which the disks could achieve alone. The next chapter discusses the cache organisation in detail.

2.6 Summary

This chapter has given an outline of the properties of both continuous media and the storage hardware. The problems involved in the provision of continuous media to a user have subsequently been introduced. The chapter has covered a number of notable efforts to produce a solution, in both small and large scale, and finishes with an examination of similar work in the field to that in this thesis.

3. Stream Caching

Given a requirement of 20 simultaneous users of a single networked video file server, there is an aggregate outgoing bandwidth of between 30Mbps (4MBytes/s) up to 100Mbps (12.5MBytes/s), depending upon the material in question. Assuming an average time between successive frames of 40 milliseconds (25 frames per second) for each individual user, and that the average maximum sustainable disk transfer rate is around 4 to 5 Mbytes/s with a seek time of 8 to 10 ms per access, it is not enough to allow direct access to the server's disks. In terms of the disk seek latency alone, this would result in either missed frames or jerky motion, most likely both. Furthermore, increasing the bandwidth of the streams much above the minimum requirement would exceed the maximum performance of such disks in any case.

To enable the use of such disks within a video server requires a memory buffer that removes the effects of disk scheduling latencies from the users' point of view. Such a buffer would eliminate jitter and enable reuse of data among the active users such that the need to physically access the disk is minimised, thus lessening the disk bandwidth bottleneck. To implement such a scheme efficiently however, does require a detailed and accurate knowledge of the activities of all the video streams, and further that this information be kept up to date. In particular, a record of what each stream is doing, and what events may occur in the future is needed.

In this chapter, the stream model used throughout the chapter is first described. Thereafter, caching systems in general are discussed. The theory of "stream optimised caching" is then introduced for the first time in detail, together with a simulation of expected performance of several caching algorithms. The remainder of the chapter suggests possible implementation strategies.

3.1 Operational Model

Continuous media, such as video, can be defined as a sequence of data frames with each frame occurring at a discrete logical point in time on the time scale of the media. For example, videocassette recorders can use an arbitrary time counter scale much like an audio cassette deck, or may use a time relative scale based around hours, minutes and seconds. Each frame can therefore be uniquely identified by the logical time at which the frame is due relative to the start of the media at 0 minutes, 0 seconds; this is known as the frame *timestamp*. Figure 3.1 illustrates the concept of a logical time scale, referred to as a *timeline*, of a five-minute media clip. The current position in the media clip is shown as two minutes 15 seconds.

Given the definition of the media timeline, each stream can be described by the following model, as used by the Palantir [Lin90] project:

Table 3.1: The Stream Model

`Direction`	Indicates the direction of play: forward or reverse
`Playrate`	The rate of playback relative to normal
`Paused`	Indicates whether the media is being played back or is paused
`Position`	The current position of the media on its logical time line in ms.
`TotalTime`	The total duration of the clip in milliseconds.

Each model can be altered by operations that a user may require. These operations are appropriate to the control of continuous media and may be summarised thus:

Table 3.2: Stream Operators

`Connect`	Initiate a playback stream
`Speed`	Change the rate of playback
`Direction`	Change the direction of playback
`Pause/Unpause`	Pause or Continue playback of a stream
`Reposition`	Change position on timeline
`Quit`	Finish playback of a stream

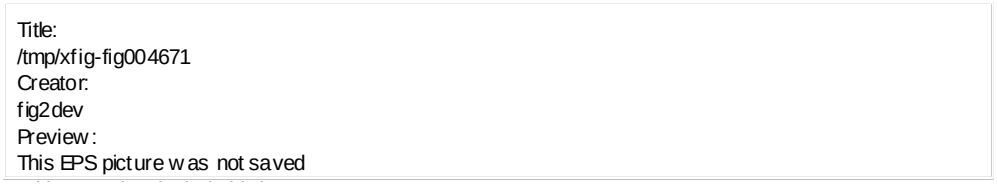


Figure 3.1: An example representation of time in continuous media

3.2 Caching and Typical Approaches

In order to improve the performance of disk file systems it is necessary to introduce a layer of data buffering such that repeated accesses to a file need not cause physical access to the storage medium. In so doing, the response time of the file system is decreased and the access time and transfer rate requirements of the disk are reduced – this is particularly important in a multi-user environment where many users access the file system resource simultaneously.

File systems typically assume that files are accessed in a sequential nature. To improve the responsiveness of the file system therefore, look-ahead caching is performed such that the data is available in the data-cache ahead of when it is actually required; i.e. the data is *pre*-fetched. At the same time, the accessed data is maintained in the cache once accessed, should the user require it again.

Of course, resources are finite; only so much physical memory can be given to the operation of a cache. If the cache is to be effective, it is necessary to determine which data should remain in the cache. Thus, regardless of the algorithm used, all caches are organised such that those blocks of data with the shortest time to reuse have highest

priority. The blocks with lowest priority, i.e. those with the longest time to reuse, are replaced first. In practice, the precise time to reuse for each block cannot be known beforehand. Hence, a *predicted* time to reuse must be used instead, for some suitable predictor; the predictor predicts the expected time before reuse, such that the cache may be organised on *predicted* least time to reuse. If this predictor is sufficiently accurate, a high level of cache hit rate (i.e. reuse) is maintained; if the predictor is perfect, the cache will approach 100% efficiency.

One of the most common cache control policies is Least Recently Used (LRU). Here, each block of the cache is effectively given a last access timestamp and the cache contents are organised on this time of access. Thus, when cache contents need to be discarded to make room for new data, the oldest data is removed first. The prediction with LRU is that the least recently used data has not been accessed recently and presumably will not be accessed within the near future. Consequently, only that data which has recently been accessed will remain within the cache.

3.3 Temporal Locality of Access

With continuous media, the access to the data tends to be sequential from start to finish with little variation. Thus, a simple look ahead cache can eliminate the latency of disk access when a client requests an item of data; removing the disk latency results in a far shorter and steadier latency in transmission of data to the client, and consequently in very low or zero jitter playback. In addition, the behaviour of such media streams is highly predictable; since the current location of a stream, its rate and direction of play, and the properties of the media concerned are known, these can be used to advantage. For example, to prefetch in the direction corresponding to the direction of play and by an amount appropriate to the rate of advance. Of greater importance however, is where several media streams are operating concurrently on the same source file; data can be re-used across streams by consideration of their playback properties. That is, where a stream is predicted to use the same section of data recently used by another stream, that data can be maintained in the cache until it is predicted not to be used by any of the streams. How long and how much data can be maintained between streams is dependent upon the size of the cache. It is possible to organise the cache in such a way as to exploit this *predicted* temporal locality of access.

Clearly, this approach breaks down where no re-use occurs, and it is effective only where such re-use is possible, as for example, in a classroom environment where the users progress through some learning task at roughly the same rate; indeed, it was precisely for this environment that the concept of stream caching was intended. In applications such as video libraries or video subscriber services where users access entirely different files, or reuse is limited to stream intervals of (say) twenty minutes or more, the technique is of

limited use; the size of cache required would need to be enormous to enable the exploitation of reuse and its cost would be prohibitive.

3.4 Modelling Performance

As a first step in the development of the basic technique, an experiment was carried out by Prof. Peter Linington to examine the effectiveness of stream caching by simulation.

In [Lin95], a stream is classified by the identity of the clip being used, and the user's relative offset within that clip. A sample of usage is generated by selecting, for each user, the clip being used, and the user's offset within it; each user plays in the forwards direction at normal play rate until the end of the clip is reached. The simulation takes these active streams and calculates the set of intervals between accesses to each frame by the different users; each frame, once shown to one user, ages in the cache until reused, or discarded. An estimate of expected system performance is generated by running a large number of trials of varying user usage patterns; from these, the mean and deviation of expected cache performance are calculated. Several caching policies were applied to the distribution of accesses and the cache behaviour derived.

3.4.1 Stream Optimisation

For a number of streams, if the pattern of activity is known, it is immediately obvious from the stream positions when an access has no expected successor; either there is no stream approaching the frame, or the incoming stream is too far away in comparison to other intervals. Such frames can be flushed from the cache immediately, reserving space for re-use. The most advantageous use of cache space is for frames with the smallest time before re-use, and so cache space should be allocated to the k smallest inter-user spacings, such that k is the maximum value for which $t \geq \Sigma_k$, where t is the cache length.

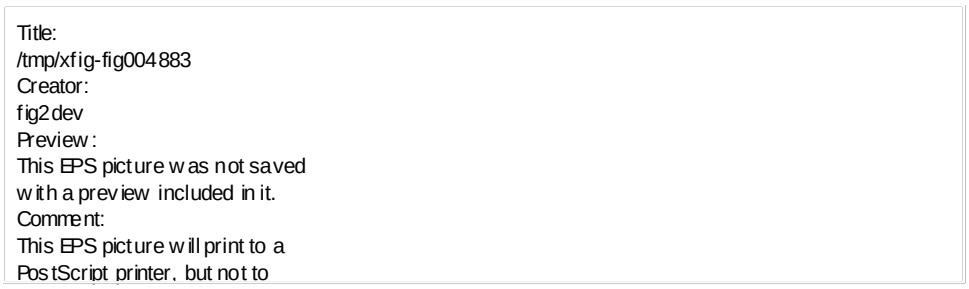


Figure 3.2: An example of stream optimised caching

An example of stream optimised caching is given in figure 3.2. Here, the intervals between streams 2, 3 and 4 can be satisfied by the cache, but the interval between streams 1 and 2 is larger than the space remaining in the cache and cannot therefore be satisfied. Physical time is shown to be advancing from the top to the bottom of the diagram which has discrete stages (a), (b) and (c) displayed; each stream, and its cached interval (where

possible), can be seen to be advancing along the logical timeline as real time advances. At stable state, stream 3 reuses the data fetched for stream 2, and similarly, stream 4 reuses stream 3's data; only streams 1 and 2 need to access the disk.

3.4.2 Modulated Stream Optimisation

The stream optimisation given above does not use any residual part of the cache which is smaller than the smallest unserviced interval; an attempt to do so would result in exhaustion before re-use. However, use can be made of this space by alternating the behaviour between retaining and discarding frames over this interval, such that a static fraction of the modulated interval's data is maintained and is periodically available for re-use by the adjacent advancing stream. Consider the example in figure 3.3.

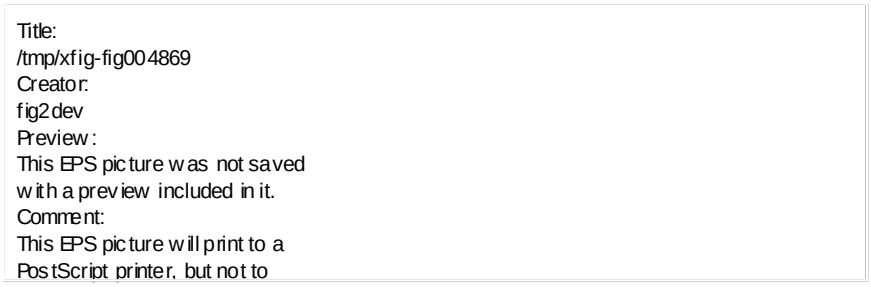


Figure 3.3: An example of modulated stream caching

At stage (a) there are two streams with an interval between them which is larger than the size of the cache; stream 1 is beginning to fill the interval between the two streams. At stage (b), both streams have advanced along the timeline and all of the data that stream 1 has accessed remains in the cache. Once the cache becomes full however, the data most recently used by stream 1 is discarded leaving the older data in the cache. By stage (c), stream 2 is beginning to use the data left behind by stream 1, and stream 1 can again leave data in the cache as stream 2 uses and discards the old data. A ‘gap’ in the cache interval is visible where stream 1 switches from retaining to discarding frames and back once stream 2 begins freeing up cache space. Clearly, if data left in the cache by stream 1 were discarded by the age of the data, stream 2 would not benefit from the cache; the first stream's data tail would reach a maximum length, namely the size of the cache, and would never extend to reach stream 2.

3.4.3 Results of Simulation Tests

In the simulation, an initial assumption is made that the clips and offsets are selected at random from a uniform distribution. In practice, there will be a significant correlation between students, with most students starting the scheduled activity at the beginning of the class and progressing at near the expected rate. However, the actual rate of progress is bound to vary, with late arrivals, backtracking and other variations. Thus, the

assumption of a uniform distribution of requests may be pessimistic, but gives a safe basis for the predicted behaviour of different algorithms. Figure 3.4 illustrates the performance of LRU, stream caching and modulated stream caching algorithms for a sample of 20 users of three clips. Linington further showed that the stream-based algorithms not only gave better efficiency than LRU, but also resulted in a more predictable performance, as shown in Figure 3.5.

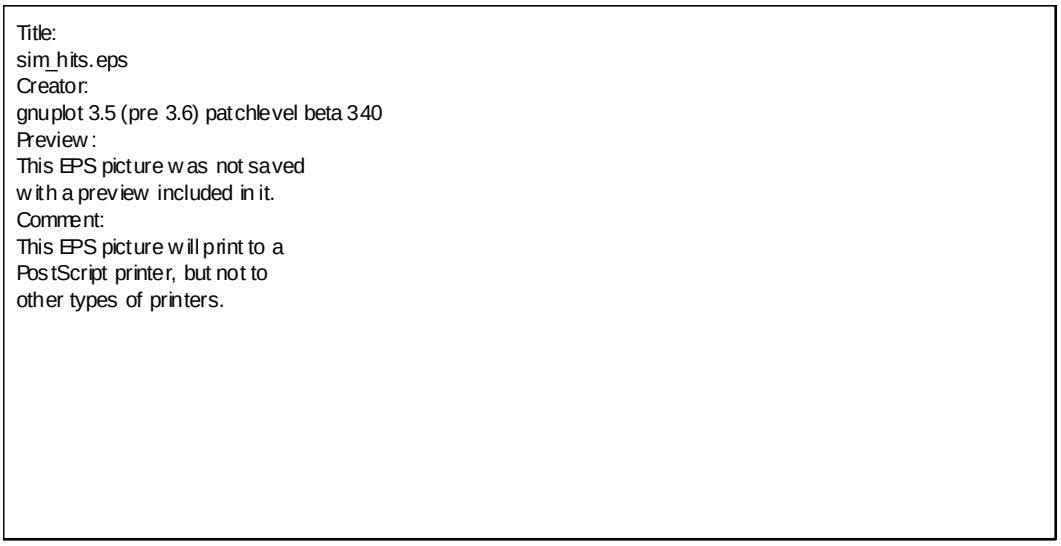


Figure 3.4: Comparison for Access to three Clips by 20 users

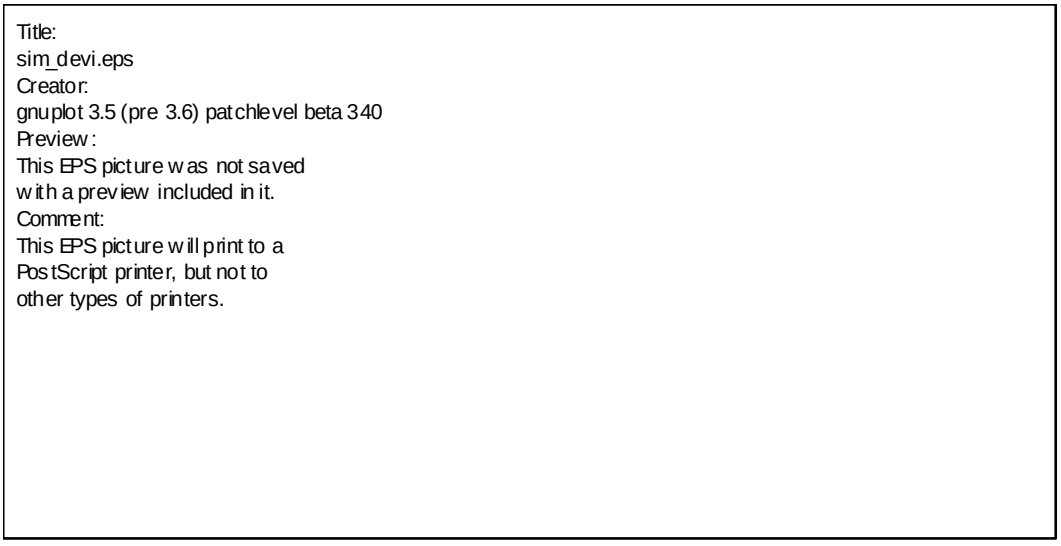


Figure 3.5: Standard Deviation of Disk Reads Corresponding to Figure 3.4

3.5 Requirements of Predicted Locality of Access

In order to allow the cache to be organised by predicted temporal locality of access, it is necessary for the server to have a knowledge of the playback parameters of all the streams that it manages. If the server maintains the playback model of each of its clients, then it must also be responsible for the timing of transmission of data frames. To do otherwise

would be to duplicate the stream model on both server *and* client, which could potentially lead to differences in client and server models, and consequent sub-optimal organisation of the cache where the models became out-of-step. A further consequence of the server maintaining the stream model is the style of control of the stream clients; a higher level of abstraction results, consisting of such basic primitives as ‘play’, ‘pause’, and playback rate and direction controls. These primitives were given in table 3.2 as *user* stream operators; they now become *system* stream operators also.

In fact, the above model and control requirements complement one another by reducing the latency of transmission when a frame is due. If the client were maintaining its own stream model and issuing requests for frame transmission, each frame, or group of frames, would require a ‘request to send’ message (such as `read(file, length);`). Such a message itself takes time to cross the network and be processed by both client and server protocol stacks, in addition to increasing the load on the network. By placing transmission-timing responsibilities with the server, this source of latency is eliminated. Furthermore, by placing the stream model and transmission responsibilities with the server, the clients' requirements are reduced and their implementations simplified.

3.6 The Cache Architecture

The stream controller needs to organise individual streams such that the data cache can be optimised for stream caching as discussed above. The monitoring of all streams on an individual basis could be potentially expensive, especially where a large number of streams were active; such a solution would not scale well. Similarly, the management of the data cache structures needs to be efficient with low maintenance overhead. In theory, each cache block and each stream could be considered separately and all management structures updated for each access of every stream. However, for multiple users the maintenance costs of such an approach would likely require immense processing power. Hence, cache maintenance costs are a consideration in the cache design.

For the above reasons and further to simplify the controller, a set of streams with contiguous intervals is arranged into a *stream group*. The stream groups are therefore either separated by intervals that are too large to service, or the groups belong to different media files. A modulated interval is marked separately since there can be at most one such interval, and its behaviour is different from the fully cached intervals as discussed in section 3.4.2.

As the streams are arranged into logical groups, where each group consists of a number of playback streams whose intervals lie adjacent to each other on the media's time-line, this is reflected in the structure managing the streams. At the top level, we deal with playback files, and at the very bottom, with individual streams. Thus, each file consists of one or more stream groups, and each group of one or more streams. This arrangement is illustrated in figure 3.6.

In order to simplify the operation of this structure, each group maintains the forward and reverse playing streams separately; calculating the edges of a section of file to be discarded is made quicker and easier if the forward and reverse streams are semi-sorted such that the head stream is first and the tail stream last in each list. This allows the edge of the discard area to be determined by a simple comparison between the head and tail members of the opposing directional groups. A single bi-directional list of streams would require a sort for the leading forward stream and tail reverse stream (if any), or the leading reverse and tail forward streams. This structural split is therefore a simplification; either implementation would work.

On group assessment, that is, the point in time at which the groups are formed from the streams, the previous state of the groups can be destroyed since they were only a prediction of cache usage, and we begin afresh. Each file's streams are sorted on their model's current times into increasing temporal order and a list of intervals between adjacent streams generated; each interval records the two streams which form it, and the streams keep a record of the file to which they belong. The interval list is then sorted by increasing size such that cache space can be allocated to those gaps, smallest first. Clearly, any file that has only a single user has no intervals and will therefore have no entry in the interval list – a cache ‘singleton’.

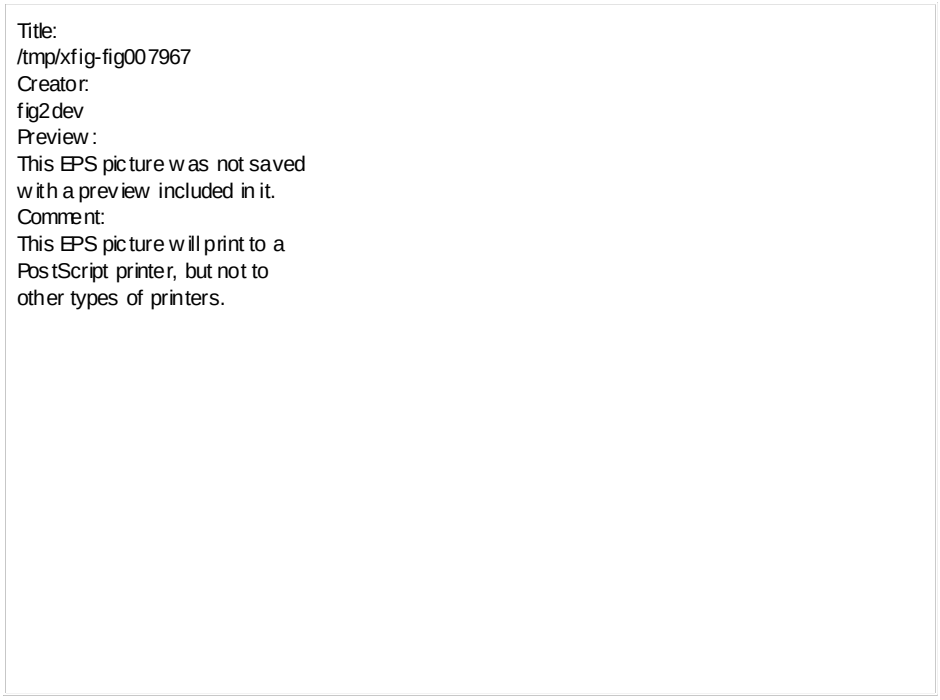


Figure 3.6: Stream Management Data Structure

3.6.1 Physical and Logical Cache Allocation

It is important to note that the formation of the stream groups allows only a *logical* allocation of cache space to those streams. No *physical* allocation is made since only a

prediction of cache usage is being made not a determination of that usage. The physical allocation is determined by the look-ahead and play-out buffer activities, which fill the group intervals over time. No immediate attempt is made to fetch the data that forms the groups into the cache, that is, to make the logical allocation physical. The pattern of usage can potentially change very rapidly, and consequently, filling the stream intervals may take far too long. Consider the example in figure 3.7. Here, the third stream is repositioned between discrete times (a) and (b). At instance (a), the cache is at stable state, with the data fetched by streams 1 and 3 having filled the two group intervals. Thus, at time (a), the physical allocation of cache space reflects the logical group allocation. However, after the reposition, stream 2 has yet to fill the interval to stream 3, so a physical gap in the logical allocation exists over this interval; the physical allocation for stream 3 will be that of its own look-ahead buffering activities.

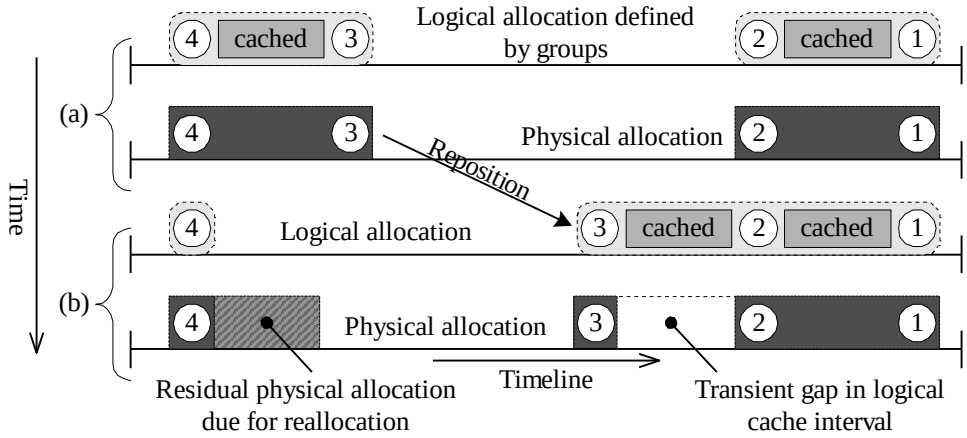


Figure 3.7: Illustration of physical cache versus logical cache content

It is possible as an optimisation that the intervals could be filled when cache hit rate is near 100% and the disks are idling. In this case, the play-out requirements of the streams are met directly from the cache-data. Such an optimisation is only possible however, where there are stream groups with only a part of their data set in the cache, that is, that the physical and logical allocations do not agree. This is a transient effect that can follow a group reassessment, since the newly formed groups may have changed significantly from their previous state and the physical contents of the cache will lie outside of the new logical groups as a result. If no further reassessments occur, then the groups can reach a stable state where each contains its full data set in the cache; at stable state the logical and physical cache allocations will be identical. Consequently, filling such groups while the disks are idle is obviated because all the groups are by definition full at stable state.

3.6.2 Cache Prefetch

Each stream has a prefetch region, which serves as a play-out buffer and can be said to be its length. The length of each stream must be taken into account before allocating cache

space to the stream intervals because a stream must always be able to place data into the cache for transmission to its client, regardless of whether that data has any reuse. Additionally, care must be taken not to allocate more than one cache block to any block of media data, which would be possible where the prefetch areas of several streams overlap. Hence, the prefetch lengths should be allocated once each file's streams have been sorted into order such that overlapping of allocations is avoided; the result is a set of partially contiguous but non-overlapping prefetch areas. How much cache space is given to each stream's look-ahead buffering is dependent upon the properties of the individual stream. For example, paused streams do not require a prefetch, and different streams consume varying amounts of cache per unit time and are further dependent upon the rate of playback.

An example of prefetch allocation is given in figure 3.8. In 3.8 (a), the three forward playing streams have been allocated prefetch from out of cache space *before* the intervals are formed. In 3.8 (b) a group is formed from the intervals between streams 1 & 2, and 2 & 3. The prefetch of streams 2 and 3 can clearly be seen to overlap with the group's intervals; cache space will have been allocated for both the group's intervals *and* for the prefetch. In 3.8 (c), the overlap is detected and the prefetch effectively reduced to zero to prevent the double allocation.

Consider the case where the amount of cache space remaining on consideration of the stream group in figure 3.8 (b) is slightly less than the size of the two intervals between streams 1 and 3. In this case, only the smaller of the two intervals could be satisfied if prefetch were not taken into account at the point that each group was allocated. It is therefore imperative that the group space requirement be reduced by any prefetch overlaps at the time that the group allocation is attempted. If it were attempted as a post allocation adjustment, the group allocation could unnecessarily fail as demonstrated by the above example. Note that the prefetch of leading stream 1 remains allocated after group formation since it lies beyond the edge of the stream group.

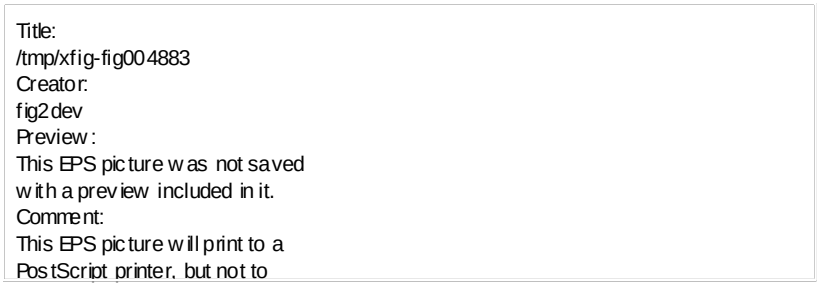


Figure 3.8: An example of prefetch overlapping with a stream group

Continuing the above discussion, the aim is to maintain a 1:1 mapping between file block and cache block during formation of the groups; each stream will have prefetch which can overlap with the intervals that make up the groups. This is clearer on

consideration of a stream which lies in between two others, such as stream 2 in figure 3.8; its prefetch will overlap with one of the intervals on either side, dependent upon its direction of play. Since the prefetch areas were generated not to be overlapping, each can at most span over a single interval. Thus, a simple test of overlap between the interval and the adjacent streams' prefetch areas allows a reduction in the size of cache allocation that the interval requires by the size of any such overlap.

It is then trivial to form the new stream groups by allocating the remaining space in the cache to each interval in turn, until either the interval list is exhausted, or there is insufficient space left to service the interval under consideration. Note however, that if cache space is exhausted by the intervals, each stream individually still has sufficient buffer space for its operation because it was allocated a prefetch segment in the cache.

3.6.3 Maintaining the Groups

Each time the cache is physically exhausted by the continual prefetching of the streams or other file system activity, a reassessment of the groups is made. The reassessment can determine which sections of the cache are not predicted to be used and can therefore be flushed to make way for new data. Any data within a group will remain since it either is in use or may potentially be reused. The act of reassessment however, can be put off for longer if those data at a tail end of a group and that are known to have no successor, are flushed after use, or otherwise marked for replacement. The reassessment cannot be permanently avoided however, as demonstrated in the following example.

In figure 3.9, the two streams 1 & 2 play at normal rate, whilst stream 3 plays at half speed. Thus, at time (b) the size of the interval between 1 & 2 remains temporally the same, though that between 2 & 3 lengthens. At stage (c), the space available in the cache is no longer sufficient to satisfy the both the interval between 1 & 2, and the ever growing interval between 2 & 3. Hence, divergence of streams within the single group can cause increasing cache demands, which must result in the division of the group.

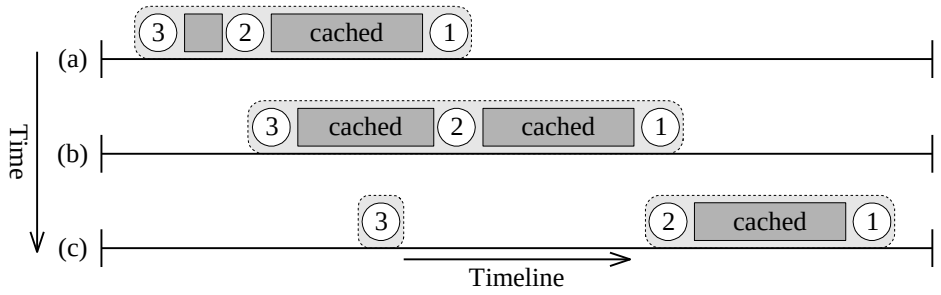


Figure 3.9: Divergence of stream within a single stream group

Since the streams within a single group can diverge because of play-rate differences, until collectively they are larger than the size of the cache, reassessment cannot be indefinitely postponed. Similarly, normal non-stream data usage can increase and require

a larger segment of cache space. This is why reassessment must be triggered when physical space is exhausted. Furthermore, some user actions must also cause a reassessment, for example, repositioning or closing of a stream. In this case, the logical allocations and group structures would not reflect the stream models accurately and would therefore give an inefficient use of cache space.

Consider the example in figure 3.10. Initially, there are two stream groups consisting of streams 1 & 2, and 3 & 4, respectively; the interval between streams 2 and 3 is larger than the remaining space left in the cache and hence the four streams cannot form a single group. In the example, stream 3 is progressing at a rate greater than the other three streams. At stage (b), the interval forming the lower group has extended, but is less than the interval between streams 2 and 3. By stage (c), this is no longer the case and the reassessment consequently groups streams 1, 2 and 3, while stream 4 becomes a singleton. For maximum caching efficiency, the cross-over from stage (b) to (c) should happen as soon as stream 3 is closer to stream 2 than 4. If this cross-over is delayed then stream 2 will not switch to retaining frames until after this optimum cross-point; similarly, stream 3 will continue to retain frames after the cross-point and the cache would then no longer be operating over the shortest set of intervals.

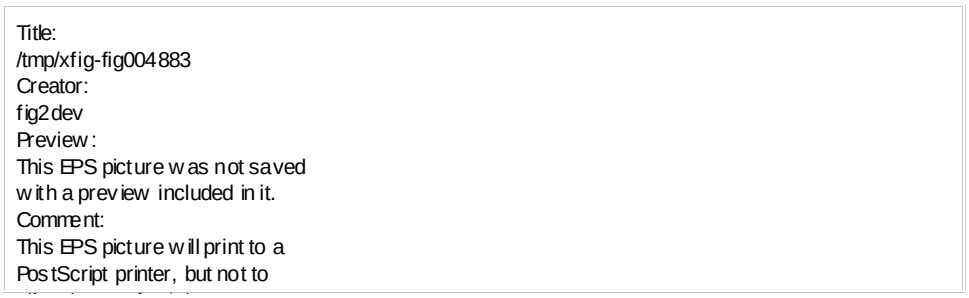


Figure 3.10: An example of group formation on continuous reassessment

There is a further point to consider when switching a stream between two adjacent groups once the switching stream reaches the cross-point. In the example above, stream 3 will have filled the interval to stream 4 with data. Once the cross-over in the groups is made, the data ahead of stream 4 ought not to be discarded until used by stream 4 if caching efficiency is to be optimal; otherwise, that same data is obtained from disk by the prefetch of stream 4 thus decreasing cache efficiency. At the same time, stream 2 must leave data for use by stream 3 and will do so at the expense of the 3 to 4 stream interval if cache space does not suffice. In fact, this behaviour is identical to modulating the smallest unserviced interval if the interval is the smallest unserviced interval in the set of all intervals. Modulated intervals are discussed further in section 3.6.4.

When performing an on-the-fly discard of frames that have no predicted successor, any re-use if the user intervenes is automatically lost. For example, a single user of a clip

repositioning the video to review an earlier section; the required data will have been discarded because it was predicted to have no successor. Provided then, that the cache has more free space than is being used, any discarding of data should not be performed.

Once the cache is operating at capacity, discarding frames for which there is no predicted re-use becomes essential, but only viable if the CPU cost is sufficiently small; the simplest implementation would drive reassessment and cache discard purely by reacting to physical cache exhaustion and user events. Similarly, using a system organisation where blocks are *marked* for re-allocation such that they are available for re-use until actually replaced is only possible if it is not disruptive to cache operation. The greater the accuracy of the groups in respect of being formed from the smallest set of intervals at any instance, the better the caching efficiency will be. Furthermore, the efficiency is best if the physical contents of the cache exactly reflect the logical groups of which it is made at every instance. Nevertheless, the practical realisation of a stream optimised cache has real-time calculation constraints, which unless met, render the cache useless in a real-time environment. At the same time, minimising the need to both reassess the streams and maintain the cache contents leads to a less optimal stream cache; clearly, there is a balance to be struck.

3.6.4 Modulated Stream Caching

To achieve the modulated behaviour in the cache, the modulated interval must neither be converging nor diverging. Converging intervals are caused by head-on groups which prefetch towards each other; diverging groups are those playing in opposite directions with an interval which increases with time and within which there will be no reuse. Figure 3.11 illustrates two streams that are diverging; stream 1 advances forward along its logical timeline and stream 2 plays in reverse towards the start of the media clip. Note that each stream will never use the data left in the cache by the other; it is therefore worthless modulating the interval in this example. Indeed, in this example the interval need not be cached at all.

Figure 3.11: An example of a diverging stream interval

Note that in the above definitions, a stream at ½-speed is not considered ‘diverging’ from its leading group since although the interval grows with time, re-use of this interval

is possible. Similarly, a stream at double-speed is not ‘converging’ on its leading group if that group is moving in the same direction; re-use is again possible.

On considering the movement of a group's data on the timeline of some arbitrary media, it can be seen to ‘drift’ either up or down the timeline, provided that the group is not converging on or diverging from the adjacent group. In figure 3.12, two groups move along a timeline in the same direction. Relative to the leading group 1, used data leaving the tail end of the group drifts down to the leading stream of the following group 2. The ‘drift’ here is relative, since the data are actually static in the cache. The drift is clearer if the figure is reorganised such that the timelines are displayed relative to the current position of stream 2, as shown in figure 3.13.

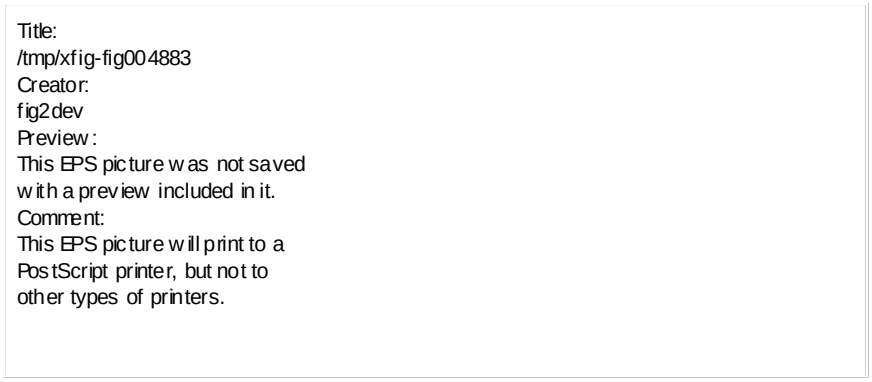


Figure 3.12: An example of modulated stream caching showing drifting data

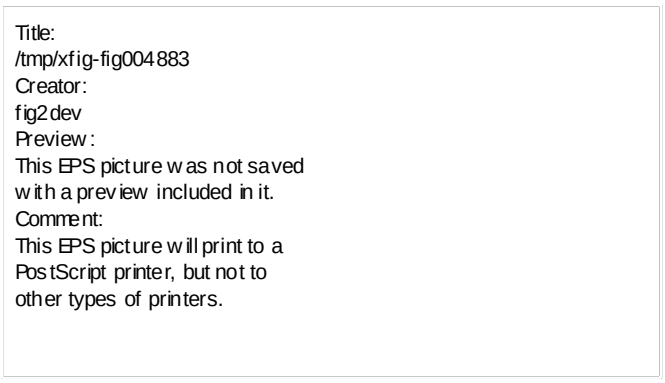


Figure 3.13: Modulated stream caching with a timeline relative to stream 2

Thus, the direction of flow of the data between the leading and trailing group over the modulated interval may additionally be identified. If the drift is ‘upwards’ then data is discarded at the lower end of the interval and retained at the higher end, as the following group will move backwards into it. If the drift is ‘downwards’ then data is discarded at the higher end and retained the lower end, as the following group will play forwards into it. In the example in figure 3.12, once group 2 reaches the drifting data and begins to free it from the cache, group 1 will again start leaving data in the cache until cache space is

exhausted once again; this was seen back in figure 3.3, and is not repeated here. During a reassessment and flush, the modulated interval is always last to be considered since its discard is partial; the cache will try to leave as much data as the logical allocations allow.

3.6.5 Sharing the Stream Cache with Non-Stream Data

It is possible to accommodate non-streamed data into the cache by keeping a separate count of cache space occupied by that data, and being able to identify which files have streams and which have not. When the cache contains streamed data, a limit should be placed upon how much space normal files may occupy, and for what period of time it may remain there if not used. This way, normal files cannot consume cache space in preference to streamed files, and idle data from normal files cannot consume cache resources indefinitely. Furthermore, when there are no streams active, the cache may be filled by normal file data. Since the consumption of space by normal files is tracked, it can be deducted from the total cache space in calculating the total logical space available for allocation to stream groups. During a cache reassessment, normal file data can be discarded if its usage timeout has expired and then on a basis of least-recently-used, for example, until the normal file usage falls within the maximum allowed by the cache.

The upper limit of normal file usage and its residency timeout may be fixed values, but could be made dynamically depending upon the loading conditions of the server. If the caching of normal file data is not effective when normal file usage is heavy, then the efficiency of the streams will be affected because the bandwidth available from the disks will be reduced by normal file activities. The reverse is also true in that a poorly managed stream cache will have a heavier demand of the disks and will interfere with the operation of the normal file cache.

The practical realisation of such a mechanism could be made by separately monitoring disk activity and cache hit ratio for both streamed and normal file data-traffic. Both monitoring activities would need to be taken into account, since a poor hit ratio alone does not necessarily equate with heavy disk access. However, disk activity and hit ratio together would be enough to recognise a poorly performing cache. When the ratio of cache hit rate to disk activity of one of the usage types becomes suitably out-of-step with the other, the cache-space usage limit could be moved to favour the traffic type with worse ratio. The dynamic adjustment of the cache usage boundary and the residency timeout remain a subject for future research.

3.7 Summary

In this chapter, the concept of stream optimised caching has been introduced, together with a basic outline of how it could be implemented and the basic issues involved. Simulation results were presented which suggest that much could be gained from a caching algorithm that takes account of the nature of stream traffic.

An argument was put forward that a media server employing such a caching algorithm would require detailed knowledge of the streams it managed, and that this complemented both the requirement of the streams' models residing with the server, and the style of user control over those streams that must result.

The chapter concluded with a suggestion for cache coexistence between normal file data using known file-caching techniques for that data, and stream optimised caching for streamed files.

4. Architecture and Implementation

> "For every problem there is one solution, which is simple, neat, and wrong"
>
> – H. L. Mencken

In order to examine the effectiveness of a stream-caching algorithm, it is first necessary to construct a continuous media server using that algorithm. Although detailed simulations of stream caching algorithms are conceivable, they are never a substitute for observing the behaviour of a real-world system. In this chapter, the prototype server architecture that was built for experimenting with caching algorithms is presented.

The chapter begins with a presentation of the physical architectural components. The performance of these components is examined and the theoretical performance of the server architecture derived. The following sections then cover the individual parts of the server piece by piece, namely: the file system; the continuous media file format; the network interface, its driver and associated network protocol structures (from link layer to physical layer); and ultimately, the cache and streams management controllers, whose operations tie the previously covered items together to form the complete network server.

Each section discusses, where appropriate, the underlying hardware construction and the issues involved in arriving at a solution. Where several design options are described in a section, the section states the option that was taken. The intention in all sections is to give the philosophy in the design and state the reasons why a particular implementation was taken. Each section concludes with an analysis of that component to show that its performance is sufficient given the requirements of the server.

4.1 A Base Architecture – Foundations

To successfully transmit video data to a client with sufficient temporal accuracy requires a processor or operating system with known highly predictable timing characteristics. In particular, it *must* respond to real-time events, most notably the transmission of a frame of video, within a sub-millisecond time frame. One such processor is the INMOS Transputer; while traditional Unix implementations give a good base on which to build, their real-time response is far too coarse for the successful implementation of a multi-user video file server.

The Transputer has been, up until very recently, unique in its employment of four high-speed serial links to allow the networking of multiple processors. In terms of the video server, this allows a breaking up of process responsibility, a dedication of function and therefore the avoidance of overloading a single processor with all the many tasks required of the server.

The prototype architecture is arranged as illustrated by figure 4.1. At the right of the diamond topology is a dedicated filesystem TRAnsputer Module (TRAM), and at the left, an ATM network interface TRAM. The four TRAMs across the middle of the topology are responsible for the main data cache. The Root TRAM is used for booting the server and for logging of system data and events as required by the server. It also provides a console interface to the server.

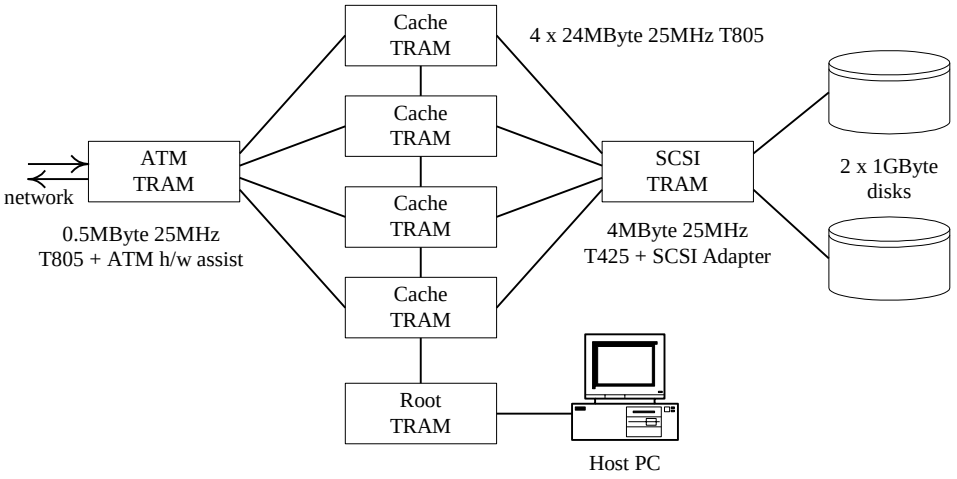


Figure 4.1: The Prototype Server Hardware Architecture

4.1.1 Theoretical Requirements

The primary objectives of the server are firstly to stream data from the disk subsystem into the cache, and secondly, to stream data from the cache to the ATM TRAM for transmission. These two data paths are illustrated in figure 4.2 and are labelled as paths X and Y, respectively. These paths are the result of the server being multiprocessor; the data transmission must be staged between the processors, and furthermore, the data must be prepared on the ATM TRAM for transmission to the network. Figure 4.2 shows the ATM TRAM as receiving a block for transmission into a “Service Data Unit” (SDU) buffer, which is then segmented into ATM cells for subsequent transmission.

The data cache affords a higher outgoing transmission rate to the network than the disks alone would allow. Hence, in figure 4.2, the SCSI TRAM does not receive all data requests from the clients, since some of these will be satisfied by cache data. For the purposes of the following feasibility study, a cache-hit ratio of 80% was assumed.

Given the physical characteristics of the Transputer, the theoretical requirements of the architecture outlined above can be examined. The following calculation was a ‘back of the envelope’ estimate performed prior to construction of the server; it is not intended as an accurate mathematical model of the prototype server.

Let:

d_m = Maximum disk transfer rate (bytes/s)

e_d = disk event rate (Requests/s)
H = cache hit rate
M = cache miss rate = 1 - H
n = number of video streams
s_v = average size of video frame
f_r = video frame rate (frames/s)
e_f = cumulative frame request rate(frames/s)
r_{hw} = hardware link transfer rate (bytes/s)
t_{hw} = time of transfer of frame across a hardware link
n_{hw} = number of hardware links used for transfer
t_a = time of memory allocation for frame

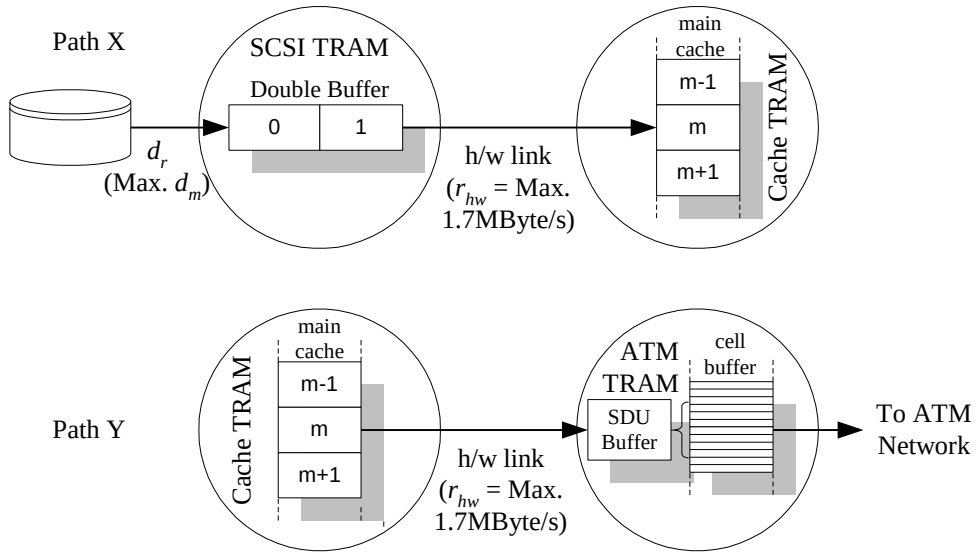


Figure 4.2: Data flows within the server – Disk to Cache and Cache to Network

Firstly, are the disks fast enough to support the required *n* users given the assumed hit ratio, and further, how fast do the disks need to be?

The cumulative frame rate is the product of all video streams:

$$e_f = n \times f_r$$

The cache will catch a percentage of these requests, so the event rate seen by the disk is lower:

$$e_d = e_f \times M$$
$$e_d = n.f_r.(1-H) \qquad (4.1)$$

The actual disk transfer rate required, d_r, may thus be calculated:

$$d_r = e_d \times s_v$$

$$d_r = n.f_r.(1-H).s_v \qquad (4.2)$$

where

$$d_r \le d_m$$

For each block transferred into memory from the disk, the block will be transmitted out on one of the hardware links at the rate of r_{hw} bytes per second (Path X in figure 4.2). However, in order for the block to be placed in main cache memory, it must first have been allocated a buffer. The maximum rate of the disk and hardware link are fixed, giving an amount of time remaining per second from which buffers can be allocated. This allocation time will also include any necessary freeing of other buffers where there is not enough memory on the initial buffer request. In the following, the effect of the double buffer in the SCSI TRAM is ignored, since the worst case is the transfer of single blocks of data scattered over the disks. In this case, the double buffering is defeated because moving the disk head breaks the transfer pipeline. Furthermore, consideration of the worst case gives a safe basis for prediction of system performance.

$$t_a = \frac{1}{e_d} - \left(\frac{s_v}{d_m} + \frac{s_v}{r_{hw}.n_{hw}} \right)$$

$$t_a = \frac{1}{n.f_r.(1-H)} - \left(\frac{s_v}{d_m} + \frac{s_v}{r_{hw}.n_{hw}} \right) \qquad \text{from equation 4.1}$$

Each Transputer link is rated at 20Mbps. For each byte of data transmitted over this link, there is 1 start bit and 2 stop bits, that is, an overhead of 3 bits for each 8 bits transmitted. Consequently, there is an effective throughput on each link of $20 \times 8 \div (1+8+2)$ = 14.55Mbps or 1.73MBytes/s.

Let:

$$s_v = 20\,\text{KBytes}$$
$$f_r = 25\,\text{frames/s}$$
$$d_m = 2\,\text{Mbytes/s} = 2097152\,\text{Bytes/s}$$
$$r_{hw} = 1.73\,\text{Mbytes/s} = 1814000\,\text{Bytes/s}$$
$$n = 10$$
$$H = 0.80$$

It is assumed that an 80% hit ratio H on the cache can be achieved. Thus:

$$t_a = 7.4\text{ms}$$

This proves that the proposed architecture is feasible; it should be possible to build a memory allocator at least an order of magnitude faster than this.

The above is, however, pessimistic: for any of the four cache TRAMs, we may assume

that a frame request can be made, the buffer allocation made for it, and the data then placed into the main cache buffer. It will take far more time for the SCSI TRAM to fetch the data and transfer it to a buffer, than to make the allocation of that buffer. Hence, for the cache TRAMs, only t_{hw} is effectively incurred where

$$t_{hw} = \frac{s_v}{r_{hw}}$$

For a cache hit, data is output over the hardware link to the ATM TRAM; for a miss, the data must firstly be fetched from the SCSI TRAM. Thus, a cache miss is at least twice the cost of a cache hit. For a miss, the cache will be kept waiting while the frame is fetched from disk. However, since the server is dealing with video streams, the frames can be obtained in advance of transmission through the ATM TRAM; to the ATM TRAM, the cache success (i.e. delivery of data) rate could potentially be 100 percent.

Given the time taken by the cache in transmitting data to the ATM TRAM per second, sufficient idle time must exist to allow the server to perform allocations, disk fetches and other cache maintenance operations. Thus, does the server have sufficient time outside of cache to ATM transmissions for other tasks?

Let:

$$n_t = \text{the number of transmissions the cache deals with per second}$$
$$n_c = \text{the number of TRAMs making up the cache}$$

then

$$n_t = \frac{e_f}{n_c}$$

If n_c = 4, and e_f = 250 (10 users at 25 fps):

$$\Rightarrow \qquad n_t = \frac{250}{4}$$
$$n_t = 63 \text{ link transfers per second per TRAM}$$

Taking these transfers as occurring per second, we define

$$c_a = \text{Cache activity}$$
$$c_a = \frac{n_t . s_v}{r_{hw}} \text{ s or \%(since per second)}$$

In the above case,

$$c_a = \frac{63 \times 20}{1.73 \times 1024}$$
$$c_a = 71.1\% \text{ busy}$$

In the above, we have ≈ 290 ms remaining per second. This time is available for waiting on the SCSI TRAM or for any allocations that may be required. Here too, the architecture looks feasible.

For the ATM TRAM, the rate at which data can be transferred over the hardware links is limited (Path Y in figure 4.2). Given this maximum link rate and the video bit-rate, how many users can be supported by the server? If each hardware link can run at 1.73 Mb/s, then for all four links:

$$\begin{aligned} R_{hw} &= \text{total data rate of all available links} \\ &= r_{hw}.n_{hw} \\ &= 1.73\,\text{MBytes} \times 4 \\ R_{hw} &= 6.92\,\text{Mbytes/s} = 58\,\text{Mbps} \end{aligned}$$

The maximum number of streams that can be handled can be calculated thus:

$$n = \frac{R_{hw}.E_{hw}}{r_v}$$

where

$$\begin{aligned} E_{hw} &= \text{Efficiency of hardware links achieved} \\ r_v &= \text{Rate of video per second} = s_v.f_r \end{aligned}$$

It is unlikely that the links will be kept fully busy; it is assumed that all four links can be kept partially busy and that they are serviced simultaneously – this may not be possible in practice. Note that the following calculation assumes a video bandwidth of 2Mbps ≈ 250 KBytes/s (MPEG streams used in practice are 1.5 Mbps; we consider a worse case here). If the four link engines are only 80 percent efficient:

$$\begin{aligned} n &= \frac{7256\,\text{KBytes} \times 0.80}{250 \times 1024} \\ n &= 23 \text{ streams} \end{aligned}$$

Thus, the cache to ATM TRAM transfer paths are capable of the requirements that will be made of them. From equation 4.2, it is possible to check that the disks are fast enough for the calculated maximum number of users:

$$\begin{aligned} d_r &= n.(1-H).r_v \\ d_r &= 23.(1-0.8).250 \times 1024 \\ d_r &= 1150\,\text{KBytes/s} = 1.1\,\text{Mbytes/s} \end{aligned}$$

Theoretically therefore, the architecture should be able to support the desired twenty simultaneous users. Furthermore, the above calculation suggests that the disk speed required by the server is within the performance envelope of any standard disk, provided

that the cache hit rate can be maintained. With respect to the required cache hit rate, equation 4.2 can be arranged thus:

$$H = \frac{n.r_v - d_r}{n.r_v}$$

$$H = \frac{20.250 \times 1024 - 2 \times 1024 \times 1024}{20.250 \times 1024}$$

$$H = 0.59 = 59\ \%$$

The above calculation takes the maximum disk rate as 2Mbytes/s. Hence, the architecture requires a cache hit rate of 59% in order to succeed as a video server.

4.1.2 Practical Realities – Building an Infrastructure

In order to test the achievable rate of data transfer from the network and file system TRAMs to the data cache and vice versa, an initial infrastructure was created. The test-bed transferred blocks of 'fake video' data from the file system to the cache to the network adapter, and separately, back in the other direction. Unless the theoretical requirements could be satisfied, there would have been little point in continuing with the construction of the video server using the given architecture.

It was clear that in order to succeed with the Transputer based architecture, the links would need to be driven at near 100 per cent. Furthermore, if the time between packets of data outbound on a link were to be minimised, the $(n+1)^{th}$ packet would need to be prepared and ready for transmission before the n^{th} packet completed transmission; that is, double buffering would be necessary on any outbound data packets.

With the Transputer link limited to a user bandwidth of approximately 15Mbps (1.73 MBytes/s), it was clear that all four links would need to be engaged simultaneously when fetching data from the file system to the central data cache. This realisation was crucial in the design of the layout of the contents of the data cache; each successive data block would need to be transferred to a different cache TRAM than the previous three blocks if all four links were to be utilised. Consequently, the cache addressing is arranged such that the least significant two bits of a file's block offset (relative to that file's own start block), are the selector for which cache TRAM should contain that data.

4.1.3 Basic Performance Requirements in the Server

As the data transfer time was critical to the success of the base architecture, memory copying of file data between processes on a single TRAM needed to be avoided. Thus, blocks of data are passed between processes by reference to their block offset within the block array from which they were allocated; this is effectively the block's address, although Occam explicitly forbids pointers – hence the alternative approach.

To add to this complexity, each physical Transputer link allows the placement of only one Occam channel in each direction of flow. Thus, a multiplexor/demultiplexor process

pair is needed at each end of each bi-directional link. In addition, each outgoing compound data structure has an associated scheduling overhead for each protocol data unit within it. For example, the protocol '`BOOL ; INT ; [5]INT`' contains three protocol data units, namely, one boolean, one integer, and a fixed size integer array of five; this requires three CPU scheduler cycles to transmit or receive across a link. A separate transfer for each protocol component is required because each part may have independent (that is, non-contiguous) memory addresses from the other parts; a '`memcpy()`' operation is required for each part.

The larger each transfer packet, the fewer packets per unit time; similarly, the fewer the number of protocol units per packet, the lower the CPU overheads per packet. Collectively, the lower the scheduling overhead, the greater the resulting bandwidth and CPU time available. Consequently, multiple protocol data units are packed into a single outgoing buffer, including the multiplexor's identifying tag used by the demultiplexor to determine which process should be forwarded any received data packet. Of course, the media data is the exception here – the multiplexor sends the tag separately from the data such that a large memory copy from the cache into the multiplexor is avoided.

4.1.4 Analysis of the Base Architecture

At this early stage in the server's development, only the infrastructure had been written. Consequently, the only data paths that it was possible to exercise were between the file system and the cache, the network TRAM and the cache, and vice versa in both these cases. The purpose of this evaluation step was to check that the data paths were fast enough to satisfy the theoretical requirements.

A double buffered transfer over any single unidirectional link was found to produce either 11Mbps (1.4MBytes/s) or 14Mbps (1.7MBytes/s). The pattern of slower links was found to correspond exactly with those hardware links that had been connected up via a C004 link routing processor; the others were all direct link connections. Having noted this pattern, it was decided that for a 'release' version of the server, all the links would need to be directly connected such that maximum performance were achieved. For the purposes of experimentation however, it was felt that this was not necessary.

When engaging all four hardware links in a sustained transfer of data blocks via the link multiplexor/demultiplexor process pair, a sustained transfer rate of over 40Mbps (4.8MBytes/s) was observed. These transfers used a block size of 4096 Bytes such that CPU scheduling overheads were minimised, this number being chosen on its convenience in respect of both file system logical block size and ATM network packet size.

Given the theoretical requirement that all the links be near 100% utilised, and that the four links together could only be driven as fast as the slowest link, the infrastructure was seen to be more than satisfactory and development continued.

4.2 A File System For Continuous Media

A file system for the storage and retrieval of multimedia data, in particular video files, was needed that gave optimal bandwidth as needed for multimedia applications, and that gave long-term reliability and crash resistance.

The recovery operation of the file system needed to be quick as the initial project prototype that would be using it would likely crash frequently (it did). It was important that the file system be recovered quickly after a crash if the project was not to be held up by frequent repairs.

The underlying operational details of any file system are hidden from the user to whom each file appears as a stream of bytes and the physical disk devices appear as a single storage facility. The file system therefore presents an interface to the user allowing byte level manipulation of an arbitrary collection of named files.

Each file has attributes describing its symbolic filename, owner and permission information for security, plus timestamps and size information as per the Unix BSD4.3 file system [MJL84].

In order to work out how many files were likely to be stored, it was assumed that the smallest clips were likely to be in the order of 5 seconds. It had been found previously that frame size averaged approximately 20KBytes per frame for MJPEG footage. Hence, assuming an average frame rate of 23 (25 is desired, but we consider a worse case) then:

$$t_s = 5\text{s} \times 23\text{fps} \times 20\text{Kbytes}$$

$$t_s = 2300\text{Kbytes}$$

Thus, over one of the 900 Megabyte devices used in the prototype server, there was a requirement of 400 large files, in contrast to the Unix assumption of thousands of mainly very small files.

4.2.1 Block Allocation – The Principle of Locality

The raw disk had been found capable of transfer rates in the region of 2.2 MBytes/s, but with the introduction of a file system, the effective transfer rate would inevitably be degraded. The file system needed to layout its data in such a way that the resulting transfer rate was close to the raw rate of the device in question; the speed of data access was a priority in the design. The approach taken was that of data locality, that is, storing related data within the same area of the disk. In this way, the seek time incurred in reading these data is minimised.

For all files, each has a header that describes the attributes of the file and the location of its data on the disk. Thus, access to the file necessitates access to the header. To maximise transfer rate, the seek-time between the acquisition of the header and the file data must be minimised; the file header must be located on the disk local to the file that it

describes. The file header is called an f-node [Dun89], similar to the Unix i-node. The contents of the nodes however, are not the same.

The maximum transfer rate is achieved by accessing the disk sequentially. The files should therefore be laid out sequentially on the disk. A contiguous file system however is very inflexible, so each file is made up of several extents[2]. The f-node therefore records the starting block number and length of each extent, the extents being sorted in the order of increasing logical file address. To access a specific location in a file requires a search of the extent descriptors in the f-node. This is a fast operation provided that the file is not made up of very large numbers of extents, ie. is highly fragmented. It is the responsibility of the block allocation strategy to prevent such excessive fragmentation, so making the cost of traversal insignificant. The benefit of the approach is that a large file may be described by a few extents, as opposed to a Unix like i-node which would have one entry for each block that the file occupied; for a large file this becomes significantly larger than the f-node equivalent, even when highly fragmented. Twinning this strategy with a good allocation mechanism and a defragmentation utility should give better performance than the i-node alternative.

Seek times can be lower if more file header areas, or bands (See figure 4.3), are available since the distance to the nearest f-node band will be lower on average. The disadvantage of many node bands however, is more wasted disk space due to unused f-nodes and smaller contiguous areas forcing large files to be made of up more extents. Node banding therefore gives much less benefit than in the Unix environment for which it was designed and instead becomes a source of fragmentation in data files; consequently, the approach was ruled out.

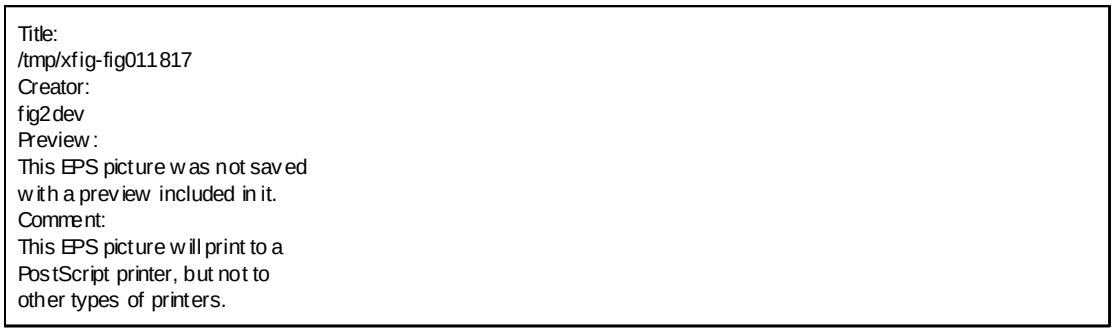


Figure 4.3: Demonstration of node banding to achieve data locality

When considering the disk as a whole, many files need to be described, all of which use the principle of locality. Not all files can be located near the single central f-node band, so the files are spread over the disk into several regions. The cost of fetching a file header therefore increases as the location of a file moves away from the single f-node

[2] An extent is a contiguous arbitrarily sized fragment of a file.

band. Due to the size of the files concerned and the duration for which each is held open, the increase in access time should be insignificant relative to the amount of data accessed before the file is closed. In fact, this cost is obviated by permanently caching the entire f-node band.

4.2.2 Variable Track Disk Format

In large disks, the number of sectors per track decreases from the outer to the inner cylinders. This has the effect that the raw transfer rate is higher on the outside cylinders and decreases toward the inside cylinders. In order to reduce the average seek-distance (and hence seek time), the allocation of files in Unix BSD4.3 occurs from the central cylinder outwards, i.e. from half radius, in both directions. Thus, the average seek distance is only ¼ (½ of half) of the disk radius. Disk optimisation can place the most used files around the central cylinder such that disk layout mirrors the distribution of access – the disk seeks are therefore minimised.

Unfortunately, the central cylinder does not have the greatest transfer rate, so the faster outside tracks tend to be largely unused. Thus, the server's file system locates its f-node band at and allocated around the first third of the disk (from the outer cylinder) in order to take advantage of the greater transfer rate. Disk optimisation utilities could likewise use this strategy so that the most used files are located around the f-node band on the fastest area of the disk.

4.2.3 Reliability and Crash Resilience

In addition to ordering the writes to the disk to ensure that directory and f-node data cannot become inconsistent, the node band is used for all file and directory f-nodes. This allows a fast recovery from failures since the location of the file and directory headers is known, and thus a lengthy scan across the entire disk is avoided. A valid f-node block is identified via a unique signature, so the probability of attempting to recover invalid or corrupted files is minimised.

The location and size of the node band are stored in the superblock of the device, so that on recovery, only the blocks indicated need be examined. Should all superblock copies be destroyed however, then the entire disk would need to be scanned for the signatures and other signs of structure data. This expensive procedure is eliminated by locating the band after a known interval from the first sector. The interval is calculated as the total number of blocks on the device divided by three (rounded down).

To prevent loss of data due to overwriting in an inconsistent filestore, a flag in the superblock indicates whether the file system was cleanly shutdown or not. If the flag shows the filestore to be ‘dirty’ then the recovery procedures, a tailored version of ‘*fsck*’, are invoked automatically. In practice, this proved to be a valuable debugging aid while perfecting the file system code.

4.2.4 F-Node Contents

The contents of an f-node are shown in table 4.1 below. The reference count refers to the number of directory entries that point to a file. The mode field indicates if the f-node describes a directory or a file, its permissions, and further, whether or not the last index entry is a direct extent pointer, or an indirect pointer to another node.

Each extent is described by the initial sector and the length of the extent. If the logical block size is assumed to be at least 4KBytes, with a 2-byte length field, the maximum extent length is at least $2^{16} - 1 = 65535$ blocks = 262140KBytes or 256MBytes.

A single 1-Gigabyte device has 218910 sectors of 4096 bytes each. It is not enough to use 2-bytes as a sector number, and three byte fields are difficult to manipulate; the physical sector address range requires 4 bytes for a 1 Gigabyte device. Thus, each extent requires 6 bytes. In general, six extents should be enough to describe a file, giving an f-node size of $6 \times 6 + 26 \leq 64$ bytes (word aligned). Increasing the size of a logical block to 16K would mean that the necessary addressing range decreased from 0..218909 to 0..54727 and further that only 2 bytes are required per block address, allowing 9 extent entries. However, this approach limits the maximum device and volume set[3] capacity supported to 1 Gigabyte. In the general case however, 6 entries should be more than enough for the average sized file, especially as each file will ideally be stored as one, or at most two extents.

Table 4.1: Contents of F-node

Field	Size/Bytes
Generation	2
Owner uid	2
Owner gid	2
Permissions	2
File size	4
Time created	4
Time last accessed	4
Time last modified	4
Mode	1
Reference count	1
Total	26

If the number of direct extents are insufficient to describe a file, then the last entry becomes a pointer to another f-node. Thus, the new f-node contains 6 further extents,

[3] See section 4.2.9 for a discussion of Volume Sets.

with the mode byte indicating that the f-node is an indirect block and an f-node pointer to a further indirect block if required. The f-node chaining can continue indefinitely such that highly fragmented files may be described, though this is clearly undesirable. After an f-node becomes an indirect block, the majority of the fields in the f-node become redundant and are used instead as pointers to the parent f-node to assist the recovery operation in the event of a crash.

The addressing of f-nodes is not done by either physical sector numbering or logical block numbering, since several f-nodes exist within one sector. The f-nodes are addressed as an array and require only 16-bits giving a maximum of 65535 f-nodes in total. Disregarding directories and indirect blocks, this gives over 65000 files addressable by the file system per device.

Considering a 512 byte sector, each of the 8×62 byte f-nodes leaves 2 bytes unused. This area is used to store a unique f-node signature such that during recovery, valid f-nodes can be identified immediately. The signature occupies the first 2 bytes of each f-node and thus increases the size of an f-node to 64 bytes; this results in no unused space within an f-node block.

4.2.5 Directories

The root directory is defined by the superblock. The superblock points to an f-node which is itself marked as a directory f-node via the mode byte. The operation of the extent entries is exactly as for data files, but the data contained are information on the file structure. The directory entries are structured as a list of arbitrary length file names and f-node pointer pairs; the indicated f-node contains the header information for the named file. Each entry also contains a ‘space used’ field so that it is known how much space is unused between two adjacent entries, and new entries can be allocated based on this information. Thus, in an empty directory, the size used field is the logical block size. No directory entries may span over a logical block boundary.

When the file system is first initialised, the first f-node entry is reserved for the root directory. On recovery, the first f-node is therefore known to mark the root directory of the file system. Similarly to f-node blocks, the first 4 bytes of a directory block contain a unique signature indicating the block to be a valid directory.

4.2.6 Cache Management for Optimisation

In order to access data within a file, the f-node of the file concerned needs to be fetched. This occurs for every data access, so there are two accesses to the disk for each data access. In order to improve performance, the f-node access is eliminated simply by caching the f-nodes. Thus, for all such files, there is only one access made by the disk device for each read request made.

A similar argument applies to directories; their f-nodes are cached with the data file f-nodes. In addition, an entire directory (consisting of one or more data blocks) is read with a single disk access, and is likewise cached so that frequent operations on such a directory are optimal. If all f-nodes are already in the cache, then directory access is limited only by the time it takes to read the directory block(s) from disk. Once the blocks are cached however, subsequent directory operations will hit the local metadata-cache and require no disk access.

4.2.7 Free Block Mapping

The allocation of disk blocks is controlled by two bitmaps located directly below the superblock in the lower track of the inner cylinder. Since the bitmaps are rarely read/written, and are not critical to the integrity of the disk data, they are placed on the slowest area of the disk so that faster regions remain available to disk data where transfer rate is more valuable.

The two maps are for f-nodes and data zones (logical blocks). For the zone map, each zone on the disk is represented by a single bit, such that every eight zones require one byte of data. For the f-node map, each f-node is also given one bit, so that each 64 f-nodes require 8 bytes. For a disk with 8 (4096 byte) f-node blocks this gives $8 \times 8 = 64$ bytes. In fact, the last block of the zone map has a large number of bytes unused. Thus, the f-node map does not have a block reserved for it alone; the f-node map follows the zone map and thus shares its last block. This has the advantage of probably saving one disk block access when flushing the maps. In terms of disk space, this is an insignificant gain, though in terms of meta-data caching an extra block of f-nodes or a directory block is significant. The cost of flushing the maps is also reduced.

Since both maps are very small, they are made memory resident to achieve maximum performance. Both maps are flushed to the disk during periods of relative inactivity, and at system shutdown. In addition, they are also flushed after a time-out period during heavy disk access, such that the flush is not indefinitely postponed. The current file-system flush timeout period is set at thirty seconds.

When the disk is first initialised, the free-maps will be largely zeros. The zone map marks where the superblock(s), free-maps, and the f-node bands are located. The f-node map marks the root directory. Hereafter, the map is used as the basis for allocating f-nodes or zones during write operations. Should the file system become corrupted, then the free-maps are rebuilt from the f-node and superblock data; they are not used again until the file system has been recovered since illegal f-node or block reusage is inevitable – this is particularly so if the free-map has not been flushed to disk before a crash and after a long series of write operations.

4.2.8 The File System

It is the responsibility of the superblock to describe to the file system manager how the disk is divided up. Generally, it must locate its root directory f-node, specify the zone size for the device, point to the free-map, and so on. The superblock is thus critical to the successful operation of the file system and is replicated on a second location on the disk. The primary copy is located on the first physical sector, which is on the upper surface and the outermost track. This area is the fastest on the disk[4]; the superblock is placed here so that marking and unmarking of the 'dirty' bit can occur as quickly as possible prior to any data writes onto the disk. The secondary block is located on the last sector of the disk, which is on the lower surface and innermost track. Since it is rarely referenced besides start-up and shutdown, this leaves more valuable disk areas for data. Thus, if the primary superblock becomes irretrievable, the secondary should remain available. In addition, if other physical devices are available, the superblock of each can be replicated ahead of the free-maps of every other device. Thus, if both primary and secondary superblocks are destroyed, one of the other devices should contain a valid copy. A diagram of the server's file system is shown in figure 4.4.

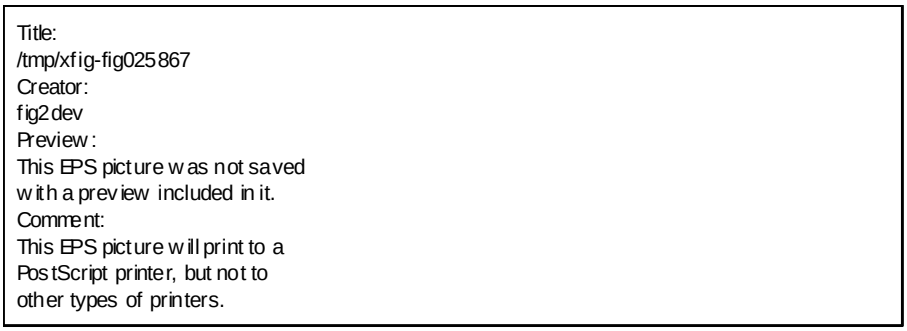


Figure 4.4: General Structure of the File System Volume

On start-up, the superblock of each device is read into memory where it remains until shutdown when each is flushed back onto its respective disk. Both the primary and secondary blocks are updated on shutdown, plus the copies of any other devices' superblocks. After the initial superblock read, any write that occurs will set a flag in the superblock to indicate that the superblock was read in, but has not been shut down. As this operation occurs prior to any writes, if a session occurs with no writes, the superblock will still indicate that the filestore contents are consistent and time is saved after a crash as no recovery operation need be invoked. When the system is shut down, or the cache contents are flushed, this flag is again altered to indicate that the file system is in a consistent state. Thus, if the file system should crash after any series of writes, the file system manager will find the 'dirty' flag on boot-up, will know that the file system is

[4] Although the time of one revolution of the disk is the same for any track on the disk, the outer tracks have more sectors than the inner tracks. Hence, the quantity of data transferred per revolution is higher on the outer tracks; the data transfer rate is therefore greater.

possibly inconsistent, and may invoke appropriate measures. The date and time of the last ‘file system check’ are recorded as part of the superblock.

4.2.9 Volumes and Volume Sets

From the users point of view the file system appears as a single logical device. To achieve this illusion, each device has a unique volume label and serial number. Within the superblock of the root device is a list of volume labels and serial numbers identifying which devices make up a particular volume set, plus a structure name, which identifies the set as a whole. This is a principle used by the VMS Version 5.2 file system [McK90]. The root device has the relative device ID of 0. The remaining volumes in the set are identified by increasing ID in the order in which they are declared. The resulting logical block numbers contained by the f-nodes therefore contain the device ID within the top byte of the 4 bytes. Thus, each device is limited to 24-bit block descriptors – a range of 0..16777215, which is not a problem (this is 8192 Gigabytes on a device with 512 byte sectors and blocks). The superblocks of mounted devices indicate that they are part of a volume set, and specify the volume label and serial ID of the root device. The f-node numbering of a volume set is linear, the address space of the $(n+1)^{th}$ device occurring after the n^{th}.

It is emphasised that a volume set is a tight coupling – once the devices are formed into a set, removal of any one of the devices will render the file system useless without a tool to make the remaining volumes in the set consistent. By a similar token, corruption of one of the devices affects the whole set, and the in-built *fsck* utility will attempt to clean up the set as a unit.

The allocation of blocks for files can thus occur across any of the volumes in the set. However, the allocation strategy tries to prevent files from spanning across multiple disks, the exception occurring when a device cannot hold a particularly large file. This should improve reliability since if one disk is destroyed, not all files are affected; only those on the device concerned are lost. A further barrier to disk striping is the way the files are stored in f-nodes, namely, by extents. Striping would result in each file f-node containing many extents, each extent on a different device to the next. This problem could be overcome by having a logical sector numbering scheme where each x sector group is placed on the next device to the previous group and so on. Thus, a single large extent can exist over all volumes within the set. This would however, further constrain the set in that the removal or destruction of any one device will ruin all files on the set.

In using a fixed distance to the f-node band, the criticality of the superblock is reduced since it does not need to record the band's location. The root directory can be located by indexing the first f-node of the f-node band, the device's format can be obtained by sending SCSI commands, and backups of other device superblocks identify themselves. The file system is therefore not dependent on the availability of the

superblock and is more reliable as a result. All of the above information is however, recorded in the superblock for completeness. The only information that cannot be recovered directly is the number of blocks per data zone. Without this, the f-node indexing information is useless because it indexes zones, not physical sectors. In fact, all of the above locations need not be fixed; however, until the file system becomes stable beyond doubt, the default locations are being used to assist recovery.

4.2.10 Analysis of File System Performance

A discussion of the testing of the file system is deferred until the section on the implementation of NFS. The sections between this and that of NFS cover the necessary components to complete the prototype as an NFS server.

4.3 Implementing AAL5 on the UKC ATM Network TRAM

Using the in-house ATM network TRAM designed by Tripp[Tri95] and the cell driver designed and build by Prof P. Linington, it was possible to build upon the cell I/O routines in order to create higher level protocols, and hence enable basic communication with other network entities. In particular, the ATM Adaptation Layer (AAL), equivalent to layer two of the OSI protocol stack, was a prerequisite to the development of protocols such as IP and UDP. The next section outlines the ATM hardware structure and requirements, briefly describes the structure of AAL and goes on to discuss its efficient implementation on the Transputer.

4.3.1 Hardware and Hardware Driver Issues

The ATM hardware board developed by Tripp[Tri95] comprises a T805 Transputer coupled with three Xilinx FPGAs[5], which drive the underlying network adapter hardware. Each of the FPGA designs provides a set of memory mapped hardware registers that allow the T805 to operate the adapter hardware. In addition, the Transputer memory space, which is located at address 0x8000000, is shadowed from the address 0xC000000. The shadow memory space allows the T805 to exchange data in the main program memory-space with data in the ATM hardware FIFOs.

Each ATM cell consists of 53 bytes, of which 48 are the transport data and 5 bytes are network cell header. The network header contains the VPI/VCI address pair, a cell loss priority bit, a payload type field, flow control information and a checksum. In addition to the network header, there is also a set-up header, which contains behavioural directions and flags for the ATM hardware and a cell timestamp.

4.3.1.1 Cell Transmission Modes

In the transmission of cells onto the network, the Xilinx firmware supports three cell timing modes: immediate, absolute and relative cell timing. The ATM hardware

[5] Field Programmable Gate Array

maintains a 16-bit clock using a tick rate of 80ns, giving a clock wrap time of 5.2 milliseconds. The firmware treats the ½ clock period before the current time as in the past and the ½ period after the current time as in the future. Thus, a cell in absolute mode with a transmission timestamp in the past is transmitted immediately. Furthermore, a series of absolute timestamps that are in the past will cause back-to-back cell transmission at maximum cell rate.

In relative mode, a cell is timed relative to the previous cell submitted and the timestamp in this case is specified as zero-relative from the transmission time of the previous cell. In firmware terms, the absolute timestamp of the previous transmission is recorded by the firmware and the relative transmit-timestamp calculated from this. Hence, relative timing builds on top of the absolute timing mode and each relatively timed cell uses absolute timing mode with a calculated absolute timestamp internally within the firmware.

Care must be taken in both cell-timing modes that the absolute timestamp is presented to the hardware within ½ a clock period of the previously submitted cell, if that cell was transmitted. If the timestamp, once submitted to the firmware, falls within this ½ clock period, there are two possibilities. Firstly, if the timestamp indicates a time in the near future, the firmware waits for the necessary period. Here, the cell is correctly timed relative to the previous cell (early submission). Secondly, if submission to the firmware is late relative to the timestamp, it will appear to be due for transmission in the past, so causing an immediate cell transmission. Alternatively, if the CPU timing is such that the absolute transmission timestamp falls further outside of the 'past' ½ clock period, then it instead appears to be due for transmission (relatively) far into the future. In such a case, the ATM adapter stops transmitting for up to 2½ milliseconds.

At the time of writing, immediate cell transmit mode is not implemented. Its purpose was as the first cell in a block of relatively timed cells, such that a new transmission after more than half a clock period does not risk stalling the transmitter. However, the absolute cell-timing mode must instead be used by using the ATM clock's value for the first cell to be transmitted.

4.3.1.2 Cell Structures

The two byte transmit timestamp in the set-up header are coupled with a further two bytes, which are used to extend the received timestamp on cells. A dummy-cell indicator flag in the set-up header indicates that the ATM hardware clock has wrapped. Thus, the software driver is able to detect the wrap and track the upper word of the ATM clock, and furthermore, to generate the full 32-bit received timestamp of all cells.

Together with the 4 byte timestamp, cell-timing mode byte, 'wrap' byte and network header, the ATM cell length in main memory is 53+6 = 59 bytes. However, long-word alignment is maintained in both header and data sections in order to facilitate set-up and

access to the cell headers by the Transputer, thus rounding the cell length up to 64 bytes. Each cell therefore has a cell-header of 16 bytes, followed by 48 bytes of received, or to-transmit data. Consequently, the received or to-transmit data are segmented into sections of 48 bytes, each separated by 16 bytes. This is illustrated as follows:

16	48 bytes	16	48 bytes	16	48 bytes	
hdr	data	hdr	data	hdr	data	

The transfer of data into or out of the hardware FIFOs is achieved by reading the shadow address corresponding to the header of the first cell to be transferred. The number of cells to be transferred is specified in the length register of the Xilinx that owns the FIFO. The hardware uses an access page size of 1024 bytes, so at most 16 cells may be transferred per access. The software driver employs an ISR[6] that services the Transputer event pin; the ATM hardware uses this pin to indicate state changes within the adapter. In addition to the event pin, the ISR uses the hardware registers to determine the reason for the interrupt and the actions that it needs to perform.

The software driver written by Linington is briefly described in the following sections. The cell driver is responsible only for the driving of the raw ATM hardware from the Transputer's main memory, and to build the 32-bit precision timestamp in received cells. However, it is not responsible for the set-up header, network header (5 byte cell header), or the contents of the data segment (48 bytes) in each cell.

4.3.2 Implementation Issues

The initial input and output mechanisms developed by Linington allowed ATM cell level transfers to occur; the construction and interpretation of the cells transferred are entirely the user's responsibility, most notably the cell header contents. Figure 4.5 shows the basic design of the I/O system using Occam Model notation.

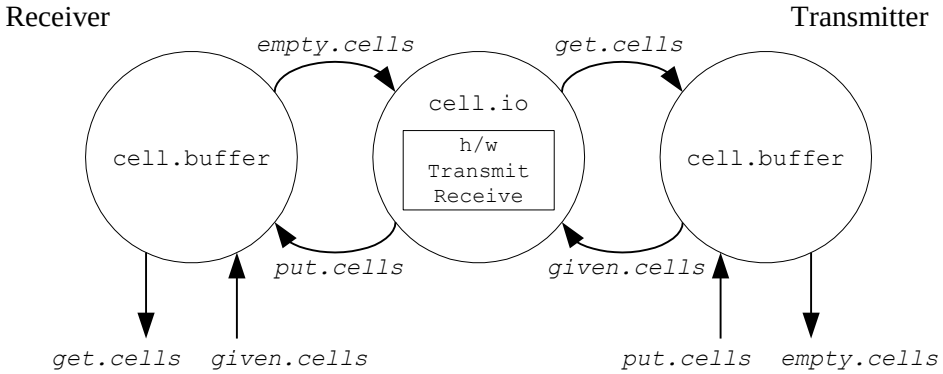


Figure 4.5: ATM Cell I/O Process Diagram

[6] Interrupt Service Routine

The **cell.io** driver process is responsible for the transmission and receipt of ATM cells to or from the network. For efficiency reasons, cell transfers are batched together so that the ATM hardware sends or receives multiple cells while the Transputer continues with other processing. Memory space for cell I/O is obtained from the two **cell.buffer** routines which maintain a cyclic cell array with queuing behaviour. The purpose of each call to the **cell.buffer** is twofold: i) to return processed cells, which will have been read from or written into (if any); and ii) to request a number of cells for reading from or writing into as appropriate. As said above, the reader from or writer to the cell buffers is allowed to request more than one cell buffer for use such that generation or processing of cell traffic may occur faster due to the decreased **cell.buffer** process interaction and consequent reduction in CPU scheduling overheads required.

As with the cache TRAMs, the multiplexor/demultiplexor processes pass I/O blocks by reference so that memory copying is avoided. However, when placing data into or out of the cell buffers, segmentation or reassembly is required. Consequently, at this point, a memory copy is unavoidable. To this end, the Transputer instruction `MOVE2D` was employed to both segment and reassemble data. The `MOVE2D` instruction is intended for windowed graphics applications to copy bitmap data from non-contiguous screen memory (the lines of data in a window bitmap) into another contiguous (or non-contiguous) area. For example, to copy a picture from a back-buffer into a visible screen window, or move the window on the screen. Thus, in the segmentation of an SDU buffer, the SDU is equivalent to a bitmap back-buffer, and the cells lines of 64 picture elements across; the SDU data is copied into the rightmost 48 bytes of each line – equivalent to a window.

4.3.3 AAL Type 5 Structure Overview

The requirement was for a layer on top of the **cell.io** mechanism which allowed for variable sized data transfers to a given destination or from an arbitrary source, without the need to know the layout of the ATM cell header or be concerned with the Segmentation And Seassembly (SAR) of user data to and from ATM cells. The ATM Adaptation Layer (AAL) provides precisely this service, though the server is concerned only with Type 5 AAL. The structure of AAL Type 5 is as shown in figure 4.6.

As seen in figure 4.6, the AAL structure has AAL and ATM (cell level) Service Access Points (SAPs), and is subdivided into two main parts: the Convergence Sublayer (CS); and the Segmentation and Reassembly Sublayer (SAR). The Convergence Sublayer is further subdivided into the Service Specific Convergence Sublayer and the Common Part Convergence Sublayer. In the server's case, the SSCS is null and bridges the CPCS primitives directly onto the AAL primitives.

The CPCS sender accepts a payload from AAL-UNITDATA and appends a trailer which includes a CRC for the AAL-SDU. The SAR sender then has the responsibility of

segmenting the CPCS-PDU into ATM cells and sending them onto the network with the ATM-UNITDATA primitive.

The SAR receiver receives an ATM-UNITDATA indication when a segment of data has arrived. The SAR then signals the CPCS receiver with the cell data so that it can be appended onto the appropriate receive buffer or a new buffer allocated. Once the last segment of a block of user data has been received, the CPCS performs a CRC check and, if successful, delivers the data to the upper layer with an AAL-UNITDATA indication. The reassembly process is thus co-operative between the SAR and CPCS. For more detailed information on AAL, the reader is referred to [ITU93].

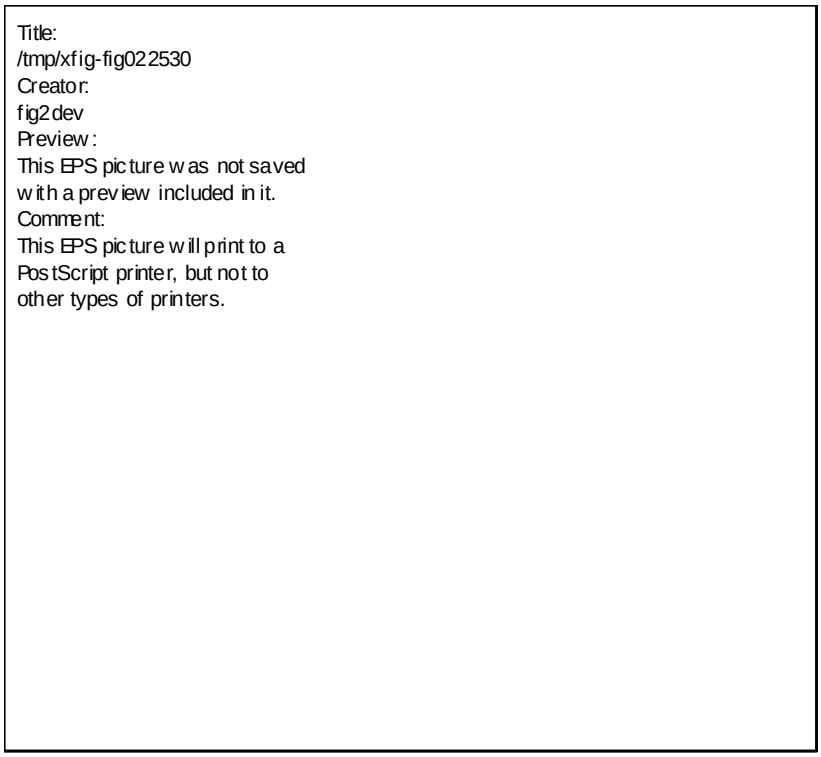


Figure 4.6: Structure Diagram of ATM Adaptation Layer 5[ITU93]

4.3.4 Design Objectives

The original objectives were to be able to build further protocols on top of the AAL5 SAP; in particular, IP, UDP and NFS protocols needed to be implemented. The ATM cell I/O system had been shown to be capable of its full 100 MBits/s transfer rate, which was far in excess of the bandwidth required for the intended application, and faster than the theoretical maximum rate of the 4 Transputer hardware links in any case. However, with the performance of the network connection being critical to the application, the implementation needed to allow for IP control flows of around 10 Mbits/s, and AAL5 video data flows across the Transputer hardware links at near full link bandwidth (approximately 50Mbits/s).

Due to the size and volume of the information being transferred over the network, it was vital that Transputer memory copies be minimised in the transfer of user data

between the layers of AAL. At a bare minimum however, memory copies *had* to occur between the SAR and CPCS layers due to the need to segment the CPCS-PDU into 48 bytes per 64 byte ATM cell and vice versa on reassembly. This process remains a notable critical section in the implementation of AAL.

Since the AAL5 video data transfers needed to be optimal, the IP and NFS paths not having such strict real-time constraints, only partial adherence to the AAL recommendation was considered. Namely, avoidance of generation and processing of AAL block CRCs are considered. It was for this reason that the CPCS trailers were included in the Continuous Media File Format; to generate the CRC for each trailer in real-time for every block of every frame is simply not feasible – it would be far too slow. In fact, this results in conformance to the AAL5 specification.

The complete functionality of AAL did not necessarily need to be implemented solely on the ATM TRAM; it was possible to use the four cache TRAMs for part of the data processing. Using the four cache TRAMs had the advantage that the ATM TRAM could dedicate its CPU time to the transmission and receipt of ATM cells and little besides, so that the efficiency of the ATM TRAM was optimal. In practice, this turned out to be essential, with much effort expended in optimising the AAL datapaths. The reader will note in the following sections that a single TRAM AAL5 implementation is possible with the AAL design presented.

4.3.5 Implementation Considerations

The sender and receiver sides of AAL are differentiated as each has different responsibilities from its counterpart. A simplified process diagram of AAL is given in figure 4.7; the actual implementation is given in more detail in the following sections.

The **SDU.Buffer** process manages a pool of Service Data Unit (SDU) buffers that other processes may request or return. On the receive side, the CPCS requests SDU buffers for reassembly of SAR-SDU's into a CPCS SDU, which once assembled, is passed up to the AAL receiver. The write controller returns the SDU buffer once the data are transferred across one of the Transputer hardware links. To transmit a data block, the read controller requests SDU buffers, fills them with data from over a link, and sends them on to the AAL sender and so on down the layers until SAR sender frees the SDU buffer once segmentation is complete. This approach minimises memory copies between the sublayers to only hardware link I/O (if required) and the segmentation and reassembly of data to or from ATM cells.

Since each **cell.buffer** may only have one reader and one writer, at most one SAR sender and receiver is allowed, though there is no restriction on the number of CPCS processes connected to SAR. In fact, this is taken advantage of by placing CPCS stages on each cache processor, which places the transfer control stages in the middle of CPCS and above the AAL-SAP.

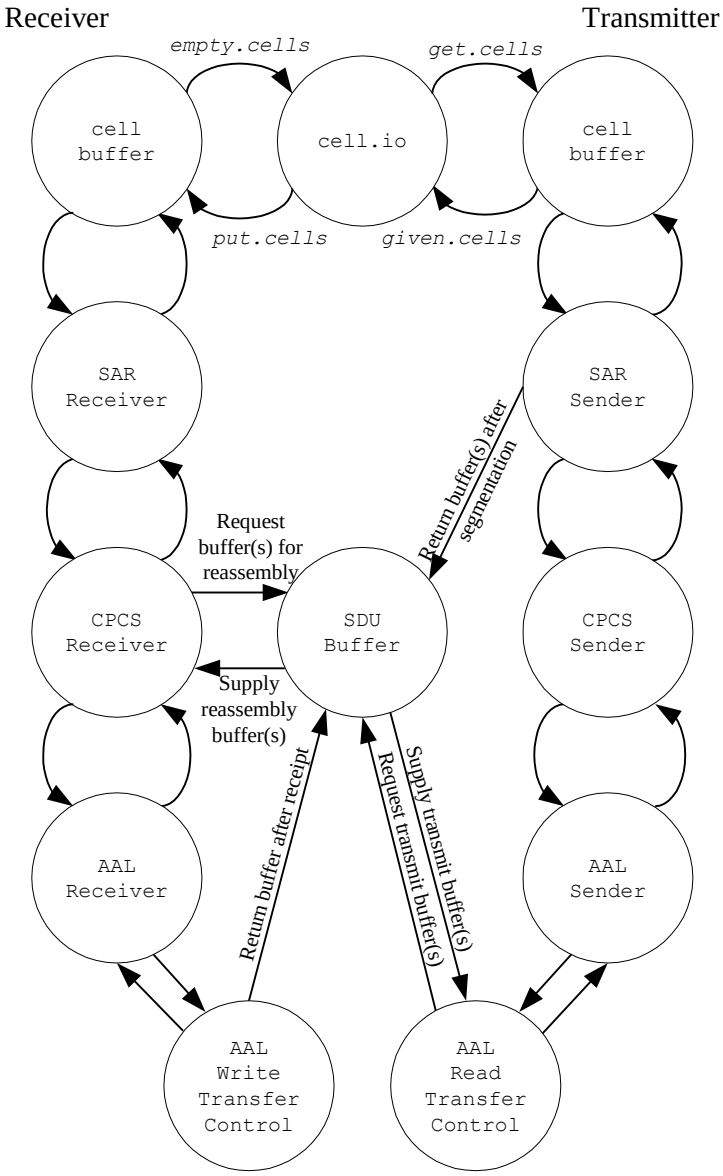


Figure 4.7: Process Diagram of AAL Implementation

4.3.6 AAL Receiver

The SAR receiver requests a number of cells from the incoming cell buffer and identifies contiguous sections of ATM cells belonging to the same transmission. Any non-user cells should be directed to a management layer, but in its absence, are currently discarded. Once this has been done, a pointer to these cells is passed to the CPCS receiver so that reassembly into an SDU buffer can occur. Since there may be several interleaved incoming transmissions, the CPCS is capable of reassembling into several SDU buffers, one for each incoming transmission. Once the end of a CPCS-PDU is detected, the SDU buffer pointer is passed on to the AAL receiver. The CRC check after reassembly may be

omitted for video data if the performance penalty is prohibitive, though video recording remains largely unimplemented at the time of writing.

The CPCS-Receiver maintains a table of active streams, and requests a single SDU buffer in advance of a new stream arriving, such that the overhead of fetching a new SDU buffer for a new CPCS-PDU is avoided. Any table entries that remain incomplete after a set interval of time are flushed from the table and their data discarded. This prevents receipt of partial AAL5 packets from permanently consuming buffer resources.

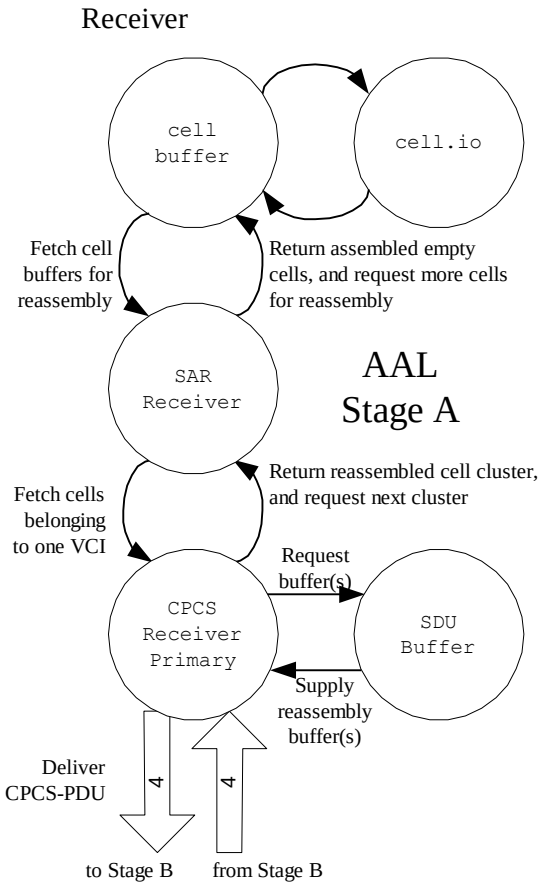


Figure 4.8: Process diagram of AAL Receiver Implementation for ATM TRAM

Since the cost of inter-Transputer communication is relatively high, each processor involved in AAL is given its own protocol layers, rather than arbitrarily choosing one processor to deal with the higher layers. If one processor were used for protocol processing, the recovered packets would then need transmitting to the target TRAM, which is expensive in link terms. Duplicating the protocol layers across all subsidiary TRAMs has the disadvantage of increased code size on each. However, it has the advantage that they all look and behave the same way, and further that all of the processors are used in the data processing, which produces a faster implementation. Each subsidiary TRAM therefore acts independently of all others, and communicates with other Transputers only where a central repository of information need be consulted, for

example, the file system data cache management structures. The above design is summarised in figures 4.8 and 4.9.

For any incoming data, it cannot be predicted when data will arrive, nor how much there will be. With any non-data packets, it is irrelevant which of the subsidiary TRAMs deals with the request; the **CPCS.rec.primary** (figure 4.8) may make an arbitrary decision. However, when dealing with data blocks destined for the cache, it becomes important which TRAM is given the data. In the case of the server's cache, a single cache block must be allocated to receive the data. For example, consider a data packet that has already been received into an SDU buffer on a cache TRAM for which it was not intended. Here, the packet needs either to be moved, or reallocated a cache slot on the TRAM concerned. Consequently, the decision as to which TRAM should accept the data must be made before the hardware link transfer occurs.

The CPCS primary receiver delegates the job of link transfer to a link transfer control process, which itself delegates the responsibility of communicating over the hardware link to a link server, such that neither process blocks. This arrangement is shown in figure 4.10. For normal non-data traffic, the link control and server processes make an arbitrary decision on destination. In this case, the transfer occurs into an SDU buffer on the chosen target TRAM and is subsequently processed by the relevant protocol stack, namely IP. For record data traffic however, more specifically video data, the link server must be given a cache address into which it can make the transfer. This address can only be given to it via the link transfer controller, which must obtain the addresses from the cache manager.

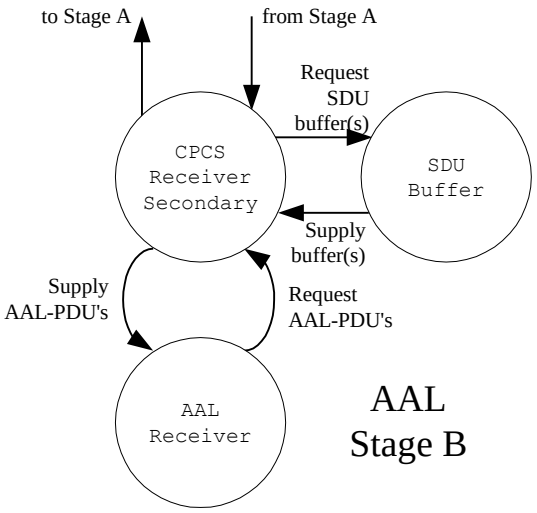


Figure 4.9: Process diagram of AAL Receiver Implementation for Cache TRAMs

4.3.7 AAL Sender

In the sender case, the CPCS process attaches an aligned trailer, which includes a CRC, and submits the generated PDU to SAR. The CPCS is therefore placed on the cache

TRAMs, leaving only SAR-sender on the ATM TRAM together with a link read-transfer control process. The transfer controller is attached to similar but independent link servers as the receiver's link write-transfer control process; this allows both unidirectional links to be engaged simultaneously. A diagram of the read controller is omitted, since it is almost identical to that given in figure 4.10. The sender's link transfer control process accepts the requests of multiple CPCS processes and those of the stream control processes. Hence, the ATM TRAM is responsible only for the receipt of full CPCS PDUs, their segmentation into ATM cells and their subsequent transmission. The cache TRAMs are in charge of the creation of the higher protocol layers from and above CPCS.

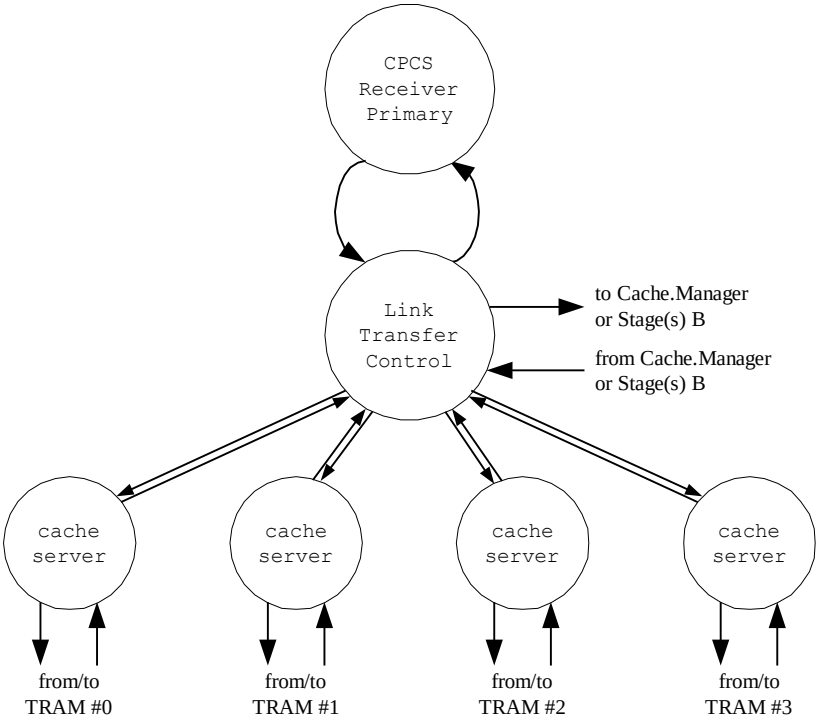


Figure 4.10: Hardware Link Transfer Engine of ATM TRAM AAL Implementation

A modification of the above scheme was considered where the cache TRAMs performed the segmentation into cells prior to transmission to the ATM TRAM. In this scenario, the ATM TRAM would then merely manage the sequencing of transmissions and should be capable of sending onto the network at the rate at which the Transputer links could provide that data. However, the effective data efficiency would then be 75% since both the cell headers and the cell data would be transmitted over the links. Given that the achieved aggregate link-bandwidth was around 40Mbps, such a scheme would at best achieve little over 30Mbps. In addition, transmission latency due to the segmentation time required before each link transfer would also be incurred. It was felt at the design stage that the performance of the server would be better whilst performing segmentation on the ATM TRAM.

4.3.7.1 SAR Sender

The SAR sender process obtains ATM cell buffers from the **cell.buffer** process, and CPCS-PDUs from the CPCS, or more accurately, the AAL read controller process. The PDUs are segmented into ATM cells and the appropriate headers generated for the cells. The SAR should theoretically be able to deal with up to *n* simultaneous streams, where the streams the SAR has obtained are cell interleaved for transmission. In this implementation, the value of *n* is effectively set to 1. The SAR-sender delays forwarding of transmit cells to **cell.IO** where more data is expected, for example, where a next block of frame is to follow; this reduces process talk with the **cell.buffer** thus decreasing CPU scheduling overheads and increasing over all data throughput.

4.3.7.2 The Sender's Link Transfer Controller

Requests to transmit include a cache address list or an SDU buffer address, and the destination via a VPI/VCI pair. The requests are placed onto a transmit queue inside the link transfer control process. The data blocks are obtained from the cache TRAMs using the same mechanism as the receiver, but this time reading from instead of writing into the cache or buffer space; these blocks are forwarded to the SAR process for segmentation. To implement the interleaved transmission mechanism required by SAR, the incoming requests are queued based on their time of submission; blocks are then sent from the first *n* frames on the queue. This can be seen in figure 4.11 where the frames in transmission are those in the active queue. Transmission ordering is achieved by ensuring that all members of the active queue have unique VPI/VCI destination identifiers.

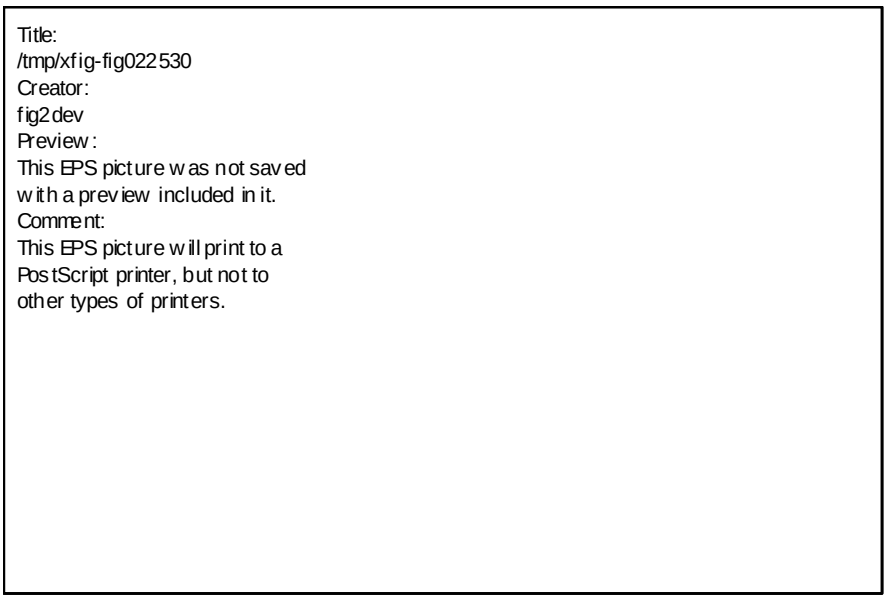


Figure 4.11: Process Diagram for Frame Transmission on ATM TRAM

The frame queue's block transmit requests feed into four link ‘block queues’, each corresponding to one of the cache TRAMs. The frame transmit requests come directly

from the streams manager process, which is discussed in section 4.9. Each of the block queues additionally takes input from the cache TRAMs directly for the input of AAL5 packets generated by the IP processes. The read control process can then take transmit jobs from the queues as the links become free from previous jobs. Each link can have at most two jobs active at any instance (i.e. is double buffered) such that each link always has work to do as each block link transfer finishes. As shown in figure 4.11, each hardware link has its own link server, multiplexor and demultiplexor process. The link servers are essentially buffers to allow the main read controller to continue to allocate jobs across the four links without blocking on the multiplexor or demultiplexor. In addition, the link server supplies the destination SDU buffer address to the demultiplexor before the remote cache TRAM transmits the block of data. Thus, the data block can be received as soon as it appears on the hardware link without the additional temporal overhead of fetching the destination address. A cache TRAM receiving a transfer job will simply transmit the indicated data block section onto the hardware link. Once a block has been successfully received from a hardware link, the demultiplexor acknowledges an internal process of the read controller, which then causes several events:

1) the block to be forwarded to the SAR-sender process;
2) another transfer job to be allocated; and an acknowledgement forwarded to the frame queue such that the next block of that frame may be appended on the appropriate block queue.

The read controller, in addition to allocating transfer jobs to the four hardware links, is responsible for obtaining SDU buffers into which to receive the data. In order to maximise efficiency, eight SDU buffers are fetched once the last buffer is exhausted; this minimises communication with the SDU buffer controller, reducing processor scheduling overheads and increasing process efficiency. In addition, to maximise the efficiency of usage of SDU buffers, up to two blocks of each frame are placed into a single SDU buffer, which again decreases process talk with the SDU process.

4.3.7.3 Block and Cell Buffers

Since the sender may block when the cell buffers become full, the majority of memory is given to the receiver's cell buffers, since a lack of buffer space would result in discarded cell data. However, enough transmit cell space must exist to allow the effective double buffering of cell transmission and cell preparation by SAR-sender. This is also true of the SDU buffers where the receiver should always have a buffer available for reassembling a new block, but the transmitter must have enough buffers for the double buffered receive of data from all four cache TRAMs.

As the sender and receiver share the same **SDU.Buffer** process, it is important that the sender does not starve the receiver of reassembly buffers by filling them with blocks

to transmit. Care is taken however, to prevent the receiver from consuming all buffers and consequently deadlocking the transmitter. This is achieved by limiting the number of buffers that are used for both transmission and receipt of data. A design alternative was to use separate transmit and receive buffer spaces. However, this approach was rejected because it makes less efficient usage of physical memory space, which is at a premium on a 512KByte TRAM. The limiting of buffer allocation is achieved by allowing only transmit side processes to allocate the last *n* buffers, which are reserved. Hence, the receiving processes cannot fill all system buffers with incoming data and cause deadlock in the system by preventing transmissions.

4.3.8 Analysis of AAL5 Performance

Despite a great deal of effort to improve the performance of the AAL5 implementation, it remains a bottleneck in the server. In particular, the need to segment and reassemble cells between the SDU buffers and cell buffers, and despite usage of the dedicated instruction MOVE2D, it is still a major consumer of CPU time.

Nonetheless, the maximum sustainable throughput currently lies at 28Mbps[7] for outgoing stream data. This is achievable with ideal conditions where transmitted frames are both large and each frame is block aligned such as with CBR data. This is because the overheads per byte of transmitted data in setting up the data transfers are then minimised. The typical maximum sustainable transfer rate, particularly with VBR data, is approximately 26Mbps.

Removing the segmentation/reassembly instruction allows the link utilisation to improve markedly to well over 40Mbps from a theoretical maximum of 50Mbps, part of which will be consumed in setting up the block data transfers. Regrettably however, the CPU is not fast enough on segmentation to maintain this level of throughput. Nevertheless, there is sufficient bandwidth available from the existing implementation to enable a small scale video file server, allowing for up to 18 of a desired 20 1.5Mbps streams, or more for lower bandwidth streams.

4.4 A File Format for Continuous Media

A format for the real-time data stored on the Video File Server's filesystem needed to be specified such that appropriate mechanisms could be built which manipulated them. However, as outlined here, there were many ways to achieve the same end, each with its own advantages and disadvantages.

As the server is based around an ATM network and a file system with a zone size of 4 Kbytes, it is most sensible to enforce a maximum CPCS-PDU of that same size. Thus, the unit of both transmission and storage is the PDU, greatly simplifying the management

[7] The first implementation had a maximum sustainable transmission rate of 17½Mbps

of the associated structures, not to mention having huge gains from not needing to generate the CPCS-PDU CRC's in real time on playback.

It is assumed for any data recorded that the same data and timings are reproduced exactly on playback. It is therefore necessary for the server to generate an index containing timing information together with length and offset information into the recorded data. Such information, however, cannot be embedded in the data PDUs because firstly the CRC's would be invalidated, and secondly, examining the meta-data in support of random access would require reading of the data blocks, which is very slow.

4.4.1 Visible Versus Hidden Format

The following approaches were considered during the design phase of the file format:

A. **Hidden Index:** One possibility was to modify the existing file system such that the metadata structures used by the file server for stream activities were hidden from the user. While simplifying the appearance of files to the user however, such an approach precluded their implementation or manipulation by any system other than the file server.

B. **Index Attribute:** A similar approach to A is to make the sequencing data a visible attribute of the video file as possible with NTFS [Cus94], in the same way that the filename and file length are attributes. However, this had the disadvantage of requiring major modifications to the existing filesystem structures. Furthermore, such files would be difficult to store on *standard* file systems, hence preventing general manipulation tools on some systems, i.e. those that cannot read or otherwise obtain the sequencing attribute.

C. **Appended Index:** An existing solution by Henshaw[Hen95] places the sequencing information at the end of the main data file and stores a pointer to it in the file header. While this solution is practical, it suffers during recording because the header and index cannot be written to the file until the recording is stopped.

D. **Visible Index File:** A better solution than C is to keep the header and index independent of the recorded data, giving each the same name stem, but the index a different extension. A convention was adopted that file header and sequencing data have the extension "*.tim*". The file data itself is not constrained to have any particular extension so that families of differing media may coexist in the same directory, for example, *aeroplane.vid*, *aeroplane.vid.tim*, *aeroplane.audio*, and *aeroplane.audio.tim.*

In taking the above approach D, the format is fully visible to the user, but has the big advantage of being portable across *any* file system and hence enables manipulation by any workstation with the appropriate tools. This was essential here, where the first

working prototype to be tested was a play only single stream server, which required the preparation of material for playback on a workstation. Further, it is feasible that this format be used as a general ATM media format, not necessarily tied to the file server concerned within this thesis.

4.4.2 Storage of Real-Time Data

With respect to the media data, each unit of transmission is stored encapsulated within a CPCS-PDU. This is done such that playback of the file does not require the generation of a CPCS-PDU CRC and hence reduces the CPU load and the transmission latency as a consequence. The MTU (Maximum Transfer Unit) for media transmission was chosen to be the size giving the largest throughput into the client workstation and being efficiently stored in a disk zone, in this case 4096 bytes. As all AAL5 transmissions are formed from 48 byte cells, this effectively gives an MTU of 4080 bytes and a payload size of 4072 bytes. Consequently, 16 bytes are unused in each 4096 byte zone giving a 0.4% space efficiency loss.

The following table outlines the options that were considered for the storage of media within the file-system data zones:

P. **Zone Aligned Storage:** As the file server's cache stores disk zones, each ‘start of frame’ must be zone aligned; the CPCS-PDU cannot lie over a zone boundary and one zone must belong to one frame. In this approach, two frames never share a single file system zone.

Q. **Shared Zone Storage:** For a typical frame of compressed video, the size[8] of the frame is almost never a multiple of the PDU payload, so the last PDU is mostly *partial*, i.e. doesn't have a 4072 byte payload. Consequently, an average of 50% of the final PDU of each frame is wasted in return for a transmission mechanism with very low latency. It is possible however, to reorganise the file structure so that the first PDU of a frame need not be of the maximum payload size. This would allow it to share a disk zone with the last PDU of the previous frame. In this way, the space efficiency is increased, but one more PDU per frame may be transmitted thus increasing the latency of transmission per frame. On the other hand, disk I/O requirements are potentially reduced, resulting in better cache performance because the file system must always transfer data in the unit of zones.

The simpler approach P above was taken to trade off space efficiency for time efficiency in the first instance. For the initial implementation, the simpler zone aligned approach allowed a working system in the shortest time. Whatever choice made here,

[8] The length in bytes and the *byte*-offset of the PDU *from the start of the file* are stored in the *.tim* index.

there are impacts upon the structure of the index of the recorded data, as discussed in the following section.

4.4.3 Generation of Index

In storing the temporal information about data received, the two options for organising the index are either by PDU block or by frame. During a recording session, it is difficult to arbitrarily identify frame boundaries without prior knowledge of the stream behaviour. For transmission, guarantees can be made about ordering of the data PDUs in either case, but there are other trade-offs, which are covered below.

4.4.3.1 Index by Block (Option R.)

This simplest approach assumes no association between data blocks. The index is organised on block offset and stores in each entry the *timestamp* and *total.PDU.length* in bytes, as illustrated below. Each 32-bit field is little-Endian byte order such that the Transputer can access them as fast as possible. In each 4K index block then, 4096 ÷ (2 × 4) = 512 entries per index block. If a typical frame size of 5 × 4Kbyte blocks at 25 frames per second is assumed, 5 × 25 = 125 index entries per second is required. Essentially, this is ¼ index block per second of data – one index per 4 seconds of data.

32-bits	32-bits
Timestamp / μs	*Total PDU Length / Bytes*

If the more compressed shared-zone format of data is used, then an extra *offset* field is needed for each index entry. The offset allows the frame start position within a zone to be located; this latter arrangement is illustrated below. In each 4K then, there are 4096 ÷ (3 × 4) = 340 blocks per second. Taking the same video example as before, the packed version is 33% worse off, but far larger savings are made within the main data file.

32-bits	32-bits	32-bits
Timestamp / μs	*PDU Length / Bytes*	*Frame start offset/ Blocks*

Although this scheme appears to assume no relation between adjacent blocks, there is a need to calculate a relationship when streams are modified in either position or play direction; play must be restarted at the start of a frame and replay the blocks of a single frame in the correct order regardless of the direction of play. Doing this on playback is not ideal because the system response time is critical, which is a serious disadvantage to this scheme. Frame boundary calculation is discussed below.

4.4.3.2 Index by Frame (Option S.)

In observing the time distribution of blocks arriving it is possible to identify clusters of blocks which logically form single ‘frames’. The exact characteristics of such distributions could be calculated either during the recording process, or as a post-process. To be able to form frames during the recording would require some knowledge of the behaviour of the media, i.e. some parameterisation by the user. This is difficult to do because different media streams have wildly different properties; it can be difficult to predict what these parameters are precisely enough. Relying on the position of partial PDU's as a way to find frame boundaries does not work for streams where they do not occur, e.g. audio streams, or when the final frame PDU is perfectly aligned. Nevertheless, the technique may be used as a hint coupled with timestamp based mechanisms.

A better approach is to record using the block index option `R` and reduce it to a frame index once recording has stopped. Thus, a global view of the data distribution on the time line is achieved, which may be reduced by intelligently calculating the frame boundaries based on timestamp distributions and the position of partial PDU's in the data stream. Using the block index during recording for later analysis, however, requires more memory than a pure frame index and the cache memory is consumed much faster. Nonetheless, it is feasible that a recording block index be flushed onto the disk and retrieved later for the post analysis, in which case memory consumption during recording is not a problem.

The main advantage of organising the index block by frame is that on playback, no time need be spent working out where the start of a frame is and ensuring that all blocks of the frame are transmitted; the information is implicit in a frame index.

For each entry in the index block, the required fields are *timestamp*, *total.length* and *block.offset*. If using the packed version of the data format, three words are still required, since *block.offset* becomes *byte.offset* from which the block offset can be inferred. In each 4K index block then, $4096 \div (3 \times 4) = 340$ entries per index block. Consequently, $340 \div 25 = 13.6$ seconds of video per index block are achieved. This is a clear improvement over a block index, especially for files with large frames. In the degenerate case where 1 frame = 1 block, frame indexing is 66% space efficient compared to non-packed data with block index, or as efficient in the packed case.

4.4.4 Manipulation of Index in Memory

Because the stream manager process uses the same mechanisms to read the index blocks as it does the data blocks, the index is broken into a number of zone sized blocks. Once these blocks are made memory resident by the cache manager, they need to be linked

together such that the index blocks may be ‘walked’. There are two main approaches to do this, as outlined in the next sections.

4.4.4.1 Index Block Chaining (Option Y.)

Two words of each index block are reserved as pointers to the previous and next blocks in sequence such that a doubly linked-list of index blocks is formed in memory. The stream controller need only store the index block offset of the first memory-resident block and a pointer to that block such that a contiguous extent of the index is formed in memory. Each index block therefore allows 511 blocks or 340 frame entries, and the above examples remain largely unaltered.

While the approach is easy to implement, for *very* large files the index tends to become relatively large as demonstrated above, although that problem is alleviated where some of the index may be paged out. Unfortunately, sudden changes in stream properties may trigger a need for the entire index block, which will adversely affect system performance, i.e. in the case where the media file is being accessed at both of its temporal extremes. In the ideal case, the index should be permanently resident until there are no more clients of the media file concerned.

A big advantage could be gained from having the stream server store the memory address of the last index block searched so that subsequent timestamp searches may occur from that index block. However, there is no guarantee that the index block will be in the same place twice, i.e. where it was swapped out. Such an approach therefore demands that the index be made fully resident and never paged out. In some implementations, this may not be practical, forcing a fresh search of the index chain.

4.4.4.2 Index Page Table (Option Z.)

For media files with large indices, it is desirable that only those sections of the index that are being used are memory resident. In this case, it is possible to indirect via a page table that has as each entry a pointer to a memory cache block, and the last timestamp of that index block. This scheme allows searching of the complete index *without* needing to page parts of it in. This approach is better than block chaining in the case where a stream's position is altered to a point where the required index block is not resident. Here, the chaining method requires reading the entire index up to the point that is required. In that case, the delay for a continuation of the stream will be less for the page table, but it is difficult to predict by how much in advance of construction. The above hour clip example would suggest a possible delay of up to a half a second with a 2 Megabyte/s disk. In any case, it is preferable to make resident the entire index, if possible, such that this problem is not an issue. Indirection via a page table would also allow slightly faster search times on the index block for a timestamp because there is no chain to negotiate.

Although the same timestamp comparisons would be made, there is scope for a smarter search algorithm on the page table.

The last word of each page table is reserved as a pointer to another table. If each page table entry takes 2 32-bit words, and each block index contains 512 block entries, $512 \times 4092 \div (2 \times 4) = 261632$ block entries are achieved, which covers a 1 Gigabyte file. Similarly, for a frame based index we achieve $340 \times 4092 / (2 \times 4) = 173740$ frames, or 115 minutes of video at 25 frames per second.

4.4.5 A Comparison of Indexing Methods

From the above discussion, it seems clear that the frame-based index is better on the grounds that it has no overhead in forming frames on playback. The block to frame reduction penalty is paid only on recording, and the resultant index blocks are much smaller by a factor dependent on the size of frame. As the framing process has to occur regardless of the method adopted, this cannot be used as an argument for elimination. Thus, the server uses the frame-based index (option `S`) for playback files.

In respect of packing the media data, it made sense to include the option in the file index such that this alteration became a trivial step later. In the first instance, the block byte offsets could be set to zero. Hence, option `Q` was taken in principle, with initial tests using zone aligned frames.

Taking the example of a 1½ hour clip, the index required is $1.5 \times 60 \times 60 \times 25 \div 340 = 400$ frame index blocks, a meagre 1.6 Megabytes. Considering that the requirements are for far smaller clips, paging of the index block to disk is not necessary and the index block can be maintained in memory while a media file is busy. In this case, the only argument between choosing index chaining or a page table is the time to access. Here, the page table wins because there is no chain to follow prior to searching on the timestamps; the search may start at the point in the table that was last referenced. In addition, the exact index block required can be located through the page table without needing to read in any of the index blocks. However, these gains would only be apparent for clips of significant size. Hence, option `Z` was taken for the server file format.

For the current requirements, the advantages of a page table will not be particularly significant, but it does allow for expansion later. At the very least, it would not be *worse* than block chaining and previous experience would suggest planning for the future.

4.4.6 ATM Media Header Content

Enough information must be placed into the header to allow:

i) Identification of the file type. A signature is placed into the header such that an ‘alien’ file cannot be mistaken for a media file. The text string “ATMMedia” followed by the arbitrary hexadecimal number AE9422BF shall be used.

ii) Version number of file format (1) and index type identification (block. vs. frame)

iii) Information about the data blocks. Storage of the total number of blocks, total number of frames, and the total duration of the media.

iv) Information on the data format. It is a requirement that the compression format used on the data be independent of the ATMMedia format. Therefore, an arbitrary piece of information is placed into the header that is meaningful only to the client application. For example, a JPEG format identifier and the JPEG compression parameters.

The above is summarised in figure 4.12.

A total of 4KBytes is reserved for use by the header. This is followed immediately by the index page table(s) and index blocks, respectively. The number of index page table blocks present can be deduced from the *Block.Size* and *Total.Frames* or *Total.Blocks*.

In previous calculations it was assumed that one video file would occupy one filesystem f-node. After the design of the file format was completed, it transpired that each ATMM file would consume two f-nodes, so a recalculation of required f-nodes in the filesystem was performed to maintain the maximum required file capacity.

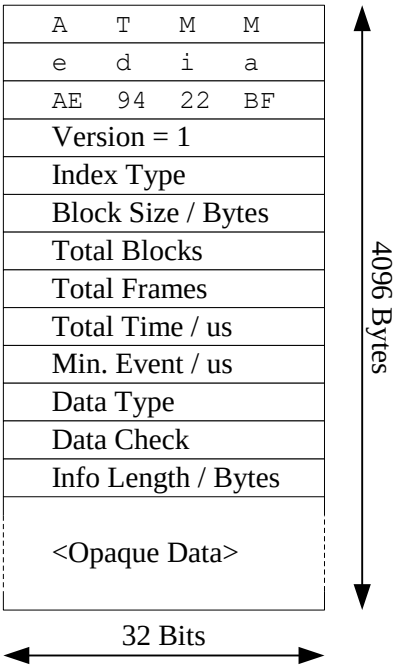


Figure 4.12: Format of the file header.

4.4.7 A File Format for Continuous Media – Summary

To allow maximum versatility and future expansion, the Continuous Media File Format uses a frame based index with page tables. The real-time recording of data should use a block based index which is reduced to a frame index as a post-process, though no stream recording has, at the time of writing, been implemented in the server. The header and

index use the "*.tim*" convention with the main data taking the same name stem. The main data may occur in either compressed or non-compressed versions with the frame index allowing either version via using the unit of Bytes for its *data offset* field. The server can, in theory, record in either compressed or non-compressed formats where two consecutive PDU transmissions are no larger than a cache block. Conversion between these two formats should be possible with an appropriate tool.

For the purposes of future expansion, the server file format was modified such that the 5 least-significant bits of the frame transmission timestamp were reserved. The purpose of the 5-bit field was as an inter-frame priority for use with, for example, MPEG clips. A mechanism to clip to a 0-priority frame after a user-reposition, or during fast playback, has not been attempted with the prototype server.

4.5 Management of the Data Cache

The cache is divided into blocks of a fixed size, equal to the file system block size. The cache addresses are arranged such that the two least significant bits of a 16-bit wide address are the cache offset number (0 to 3). The remaining 14 bits form the block row index, logically shifted left by two. Thus, the block ordering is from cache 0 through cache 3 at row address 0, then again at row address 1, and so on. This addressing arrangement is illustrated by figure 4.13.

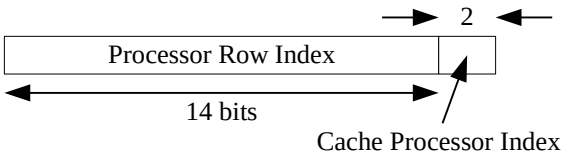


Figure 4.13: Diagram of 16-bit Cache Address

As the cache blocks are distributed across the four TRAMs, it is not possible to add information to each of them to enable their management; such a structure must be centralised for reasons of speed of access. Additionally, to do so would break the contiguity of blocks within the cache TRAM; that would restrict implementation possibilities with respect to the maximum size of data transfers with SCSI and ATM devices. Thus, the management structures are a separate entity.

The primary index of the management structure is a quantized cache block address, in this case, the row address. This minimises the total size of the management structure and improves search times when looking for file blocks; each row of the cache is given only one management structure element. The size of the management structure for a 90Mbyte cache is approximately 112Kbytes, which would be 450Kbytes were row addressing not used. However, this reduction gives a cache size gain of 1.3Mbytes since all four TRAMs must use identical cache memory sizes when striping data in transfers. Thus, the primary index of the management structure is the 14 most significant bits of a cache

address. The implications of this arrangement are that the file system must then retrieve a minimum of four blocks into the cache. In fact, this is not a problem, since such a large transfer size (16Kbytes minimum) is required to maintain good file-system performance. The management structure is shown in figure 4.14; the contents of each element are shown in figure 4.15.

Each file can be uniquely identified by a file handle, which can be used as a key when looking for blocks belonging to a particular file in the cache. The structure elements allow the blocks of a single file to be chained together in sequence. Similarly, the unallocated cache rows are chained into a single free-list. The files active in the cache are stored in a most recently used order, such that eviction of files from cache space can be done on a least-recently-used basis. The 'down' pointer of the management element is used only by the first block of each file, and is used to chain together the first block of all files.

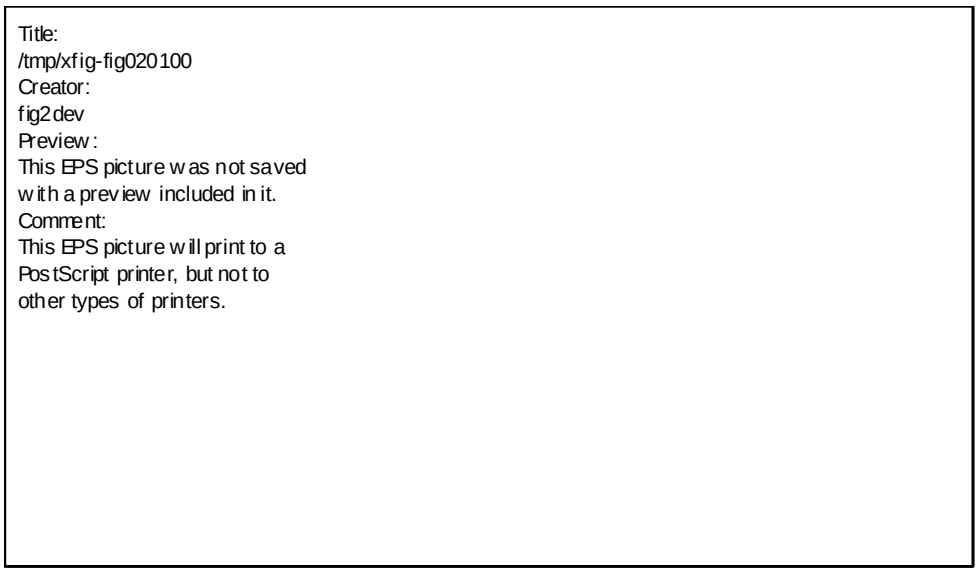


Figure 4.14: Diagram of the Cache Block Data Management Structure

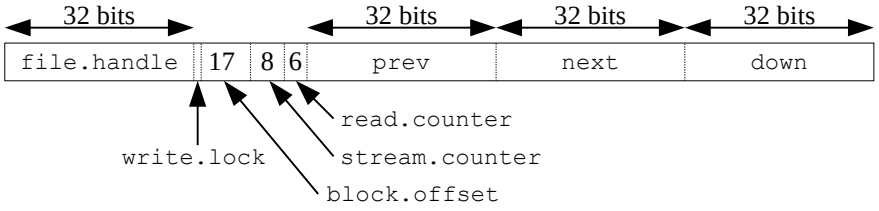


Figure 4.15: Cache Block Management Structure Element Contents

4.6 Communications – TCP/IP and RPC

Once the file-system, network interface, and cache management were all in place, Andrew Smith's UDP/IP implementation, from another project run in the laboratory [LIP95], was modified and used to give UDP and ICMP (including 'ping') support to the server. With this complete, an RPC [RPC88] implementation was built on top of Smith's

UDP engine, and on that the Network File System (NFS) protocol [NFS89, NFS93]. The implementations of RPC (XDR [XDR87]) and NFS were based on specifications available on the Internet. The completion of these steps enabled the connection of the prototype server with other network end-systems. However, NFS support only provided file meta-data exchanges, and not file-data exchanges, which required further issues to be addressed. These issues are covered in the next section.

4.7 NFS I/O

This section outlines the possible approaches that could have been taken in transferring user data from the inbound network packets onto the disks, and from the disks into outbound packets. Based upon the arguments presented in the main text, it concludes with the most appropriate approach to the given problem.

4.7.1 NFS I/O Implementation Issues

The file system transfers data via the four Transputer hardware links to or from the connected cache TRAMs; these data are taken directly from or directly to the disk with no caching structures in between. For each zone of the file system, there is a requirement of one address corresponding to a block on one of the cache TRAMs, where the block is the same size as the disk zone.

When the file server receives an NFS WRITE call, the SDU buffer containing the CPCS-PDU is arbitrarily transferred to one of the cache TRAMs for IP processing. It is at this point that the NFS call is identified and herein lies the problem: by the time the WRITE call is recognised, the data to be written is already on one of the cache TRAMs, and furthermore, the data to write is *not* in cache-block space. A mechanism needed to be determined by which the IP data could be manipulated into an appropriate form for writing onto the disk.

For an NFS READ call, the relevant disk blocks can be fetched into cache blocks, but these are distributed across several TRAMs, and it is possible that none of those TRAMs will correspond to that of the responding IP TRAM. The question that needed to be answered was ‘how can data be gathered from one or more Transputer address spaces into a single outgoing IP packet?’

As can be seen, the READ and WRITE cases are subtly different from one another, and are thus dealt with separately.

4.7.2 The NFS READ Call

On the occurrence of a READ call, the file blocks required may or may not be resident in the cache. In the latter case, the cache manager will fetch the required blocks before identifying them to the IP process dealing with the READ. It is assumed that the READ will always go via the cache, rather than reading data directly from the file system and

thus defeating the purpose of the main cache. In generating the response to the call, there are the following possible approaches:

A. The required sections of the cache blocks are requested from the appropriate cache TRAMs and transmitted to the IP process for assembly into a single outgoing packet. Therefore, the cost of transmitting data between the cache TRAMs will be incurred, which will occur ¾ of the time assuming a uniform distribution of data across the TRAMs. Of these transfers, one to three TRAM hops to the destination will be needed, the worst case with the IP responder at the end of the cache chain. It was anticipated that such an approach would be prohibitively slow and inappropriate to a real-time environment.

B. Each segment of data to transmit is identified by the IP process, and a command transmitted for each segment to generate an IP packet for each part, i.e. IP fragmentation is used. This has the advantage of avoiding inter-cache data transfers and will be faster than case A. Furthermore, as IP fragments have a sequence number, the order of transmission becomes irrelevant and the time of response therefore potentially quicker. The disadvantage of this scheme, however, is that a memory copy from the cache block into the SDU buffer that the returning IP packet is prepared in is still required. A further development of the scheme is thus to lock down the cache block until transmission is completed, and therefore to transmit the data in-situ and avoid the memory copy. This latter approach would complicate the CPCS-PDU generation as the CRC would need to be taken across two, not one data segment, and would require modification of the process handling I/O to the ATM TRAM for the same reason.

C. Similar to option B, but partial CPCS-PDUs are transmitted to the ATM TRAM rather than IP fragments – the streaming mode AAL service. As the CRC of each fragment is dependent upon all previous segments, a sequential processing of the data segments to transmit is enforced. Consequently, despite its lighter weight as compared to B, it may be slower if parallel processing cannot be taken to advantage; ideally, a way to calculate a single block CRC from segments of the block concerned should be done in parallel[9], and the part CRCs combined.

   A further limiting factor is the need to ensure sequential transmission of the parts as ordering now becomes important; there are no implicit sequence numbers. The inherent sequential ordering imposed by the CRC calculation goes some way to solving this, but it is possible that the ordering could be broken where some of the ATM TRAM hardware links were busy with non-related transfers. However, the

[9] The author became aware of a technique to recombine partial CRCs calculated in parallel at the end of the SERVICE project. However, no references to this work could be found.

default IP MTU for use over ATM is 9188 octets (8 Kbytes data payload). If it is assumed therefore that a single IP packet is at most 8K, then this can be spread over at most three cache blocks/TRAMs and the problem is finite; the order of arrival should be as intended without any ordering function for even very high loading conditions. The current implementation of the SDU buffer transmitter maintains four independent queues of requests and it services them as the links become free. Any raw AAL5, i.e. video data transmissions, will take precedence over the IP transmissions. Thus, where one to three of the hardware links are saturated with AAL5 data, the remaining 'free' links will allow the SDU based transmissions to get ahead on the respective TRAM. For this reason, ordering of SDU based transmissions *cannot* be guaranteed without *modification* of the transmission mechanisms. For example, one modification would be to use a single SDU queue and service it sequentially. Although this would solve the problem, the parallelism of the transmitter would be impeded and resultant throughput likely diminished.

4.7.3 The NFS WRITE Call

There are two approaches to copying the IP data payload onto the disk. In the following, it is assumed that NFS writes may be of arbitrary byte size and write offset:

A. The file system I/O mechanism could be modified such that a section of data within an SDU buffer can be transferred directly into the SCSI TRAM. As the file system itself is block based, a READ must first be executed so that the data can be transferred in at the correct point of the block, and then written to the disk. Such a read is not required where the write is both block-sized and block aligned. Consequently, each write operation always bypasses the main cache and is a read/modify/write where the data is not block-sized and aligned. In addition, as the data to write is not distributed across the cache TRAMs, there is a lower maximum throughput in writing the data to the disks; the file system is capable of engaging all four TRAMs where the data to transfer is striped across them.

B. To use the current file system I/O mechanisms, the IP data payload must be transferred inter-TRAM such that the data to write then exists at the correct point within the appropriate cache block. It is clear then, that a penalty similar to READ case A is incurred. However, by using cache blocks it is possible that the block of the file concerned is already in the cache, and there is no need to READ in the full block from the disk (read/modify/write). Secondly, by using the cache it is possible to defer the time of a write and hence increase throughput by reducing the disk load. The initial disk READ is obviated where the payload occupies the entire (non-resident) cache block. A further advantage is that the I/O mechanism of the file system would not require any modification to make this approach work.

4.7.4 NFS I/O Implementation

The efficiency of an NFS read was not a critical component of the server. Further, considering the either non-existent or unpredictable nature of some IP fragmentation implementations, the case C READ with CPCS-PDU fragmentation was seen to be the most appropriate solution. On the assumption that most random file accesses are for small amounts of data, then data fragmentation should not occur frequently, the worst case being two fragments where the requested data straddled a zone boundary. In practice, NFS read requests are always block aligned, and hence this latter situation does not in fact occur; the server can nonetheless cope with it. With respect to enforcing the ordering of transmission, a split transfer (that is, across two or more TRAMs) is marked as such, and the TRAM that will transmit the next segment of data is declared to the ATM TRAM. In addition, to avoid confusing two or more such split IP transfers from a single IP TRAM, the partial block CRC is used as a transfer identifier. The CRC is used to both identify and validate a transfer stream. Namely, where a mismatch occurs between the last-CRC given to the ATM TRAM and the last-CRC offered by the next cache TRAM to transmit, the transmission is aborted.

In the WRITE case, there is a potentially significant gain to be had in operating through the cache. However, on the assumption that NFS writes are a rare occurrence, such a gain would be largely insignificant. In any case, with the file server optimised for reading, the time taken for the write to occur is less important and the simpler solution to write-through the cache is the better one. Given also that actual NFS implementations transfer in block sized units, the performance to the server's writes should at least be reasonable. Thus, option A was selected for server NFS writes.

To improve the NFS throughput performance of the prototype, a *work avoidance* technique was applied as suggested by Juszczak [JUS89]. For example, if a client requests a block of data from a heavily loaded fileserver, the client may retry the operation because of the high server latency, perhaps several times. The server therefore has the original request and one or more duplicates within its processing queues. Thus, the server in fact replies more than once to the client for a single logical operation, which further slows the server and in turn causes more requests to be repeated. The technique suggested by Juszczak produces an improvement in server bandwidth by recognising and eliminating duplicate client requests that have already been processed, and hence allows the server to allocate resources elsewhere.

4.8 Analysis of IP and NFS Implementation

Initial testing was based upon trying simple file operations to exercise individual NFS and file system primitives, which enabled the server to become operational. To more

thoroughly test the correctness and efficiency of the server's NFS implementation, the NFS benchmark [SCW88] was later employed with great success. This allowed the validation of the file system under heavy loading conditions, with the benchmark checking that its operations were correctly carried out. In particular, no abortion of split IP (i.e. fragmented CPCS-PDU) transfers occurred at any time.

In table 4.1, the machine 'cobra' is a Hewlett Packard Apollo 725/50 running HP-UX version 9. The machine 'python' is an HP Series 700, also running HP-UX 9. Both of these machines have 64Mbytes of RAM. Thus, these test machines are high power workstations and consequently give excellent NFS performance. The machine named 'sequoia' is a Sun IPX SPARC station with 16Mbytes RAM, running Solaris 2.5.1. This latter machine differs in its NFS implementation in that it uses an 8Kbyte block size for data transfers. This has a clear effect on the prototype server's NFS performance. The control data is taken from the original 1988 NFSstone documentation and is simply given for reference.

The two prototype servers, vfs-atm and cimis-vfs, were configured in these tests to use the same hardware and cache size. However, cimis-vfs uses a single 7200rpm disk compared to the development prototype's two 2100rpm disks. The benchmark results are therefore a clear indicator that a faster disk will significantly improve server performance.

Table 4.1: NFS Benchmark Results for 45522 NFSstones

Hosts		**Benchmark**	
From	*To*	*Time / s*	*NFSstones / s*
Sun3/60 v3.4	control server	-	85
cobra	vfs-atm	362	126
cobra	cimis-vfs	313	145
python	vfs-atm	351	130
python	cimis-vfs	309	148
sequoia	vfs-atm	205	223
sequoia	cimis-vfs	181	252
cobra (Ethernet)	python	168	271
cobra (ATM)	python	182	250

The prototype server is at a disadvantage as compared to the workstations, since each network packet must be transferred internally across at least one Transputer link. This adds in extra receive and transmit latency, which the workstations do not incur. In addition, the Transputers are responsible for CRC generation in the AAL5 packets, both for incoming and outgoing packets. This is a feature performed in the workstation's much faster ATM hardware. The processors on the workstations are also a good deal

faster than the Transputers of the prototype server, on the HP's in particular. Not surprisingly then, the workstations give better performance figures. The 8Kbyte block-size of 'sequoia' effectively reduces the effects of receive/transmit latency, since half the number of I/O exchanges occur compared to the HP-UX boxes. In particular, the server's disks can achieve better write performance with larger block size. Consequently, the more modest Solaris workstation produces better benchmark results.

Notice that the figures for the Ethernet and ATM benchmarks between the HP-UX workstations are not significantly different. This would suggest that the benchmark is not network-bandwidth limited. Furthermore, the lower ATM performance between these machines would suggest an immature ATM driver. Regardless, the implementation of the prototype is shown as correct, and its performance is very reasonable.

4.8.1 Behaviour of the ATM hardware in the File-Server

An analysis of the ATM and AAL implementations could only usefully be performed with a server under heavy loading conditions. Thus, the NFS benchmarks also served as a good environment in which to examine the behaviour of the cells output by the ATM hardware. This analysis revealed that the ATM adapter was occasionally outputting two back-to-back cells, which is caused by a relative cell mode transmission being made long after half the hardware clock period since the last transmitted cell. This discovery further indicated that the ATM hardware could stall if the first cell to transmit, which underlyingly uses an absolute mode transmit timestamp, is delayed. Such delays can occur because of heavy CPU loading; the hardware will not transmit the cell for up to a *further* half clock period. Hence, the implementation of immediate-mode cell-transmission in the ATM firmware should improve server performance for both NFS and streams clients.

4.9 Implementing Stream optimised Caching

The diagram in figure 4.16 illustrates the process diagram for what is essentially the heart of the prototype server. The processes again use the Occam 'circle' notation, but in 4.16, some circles have attached rectangles; these represent data structures, which are referred to in the text below. Where a single structure occurs across more than one process, the processes involved share the structures, and do not have separate copies.

The use of a synchronisation mechanism occurs only once in figure 4.16, between the cache manager and the transmitter. The sharing of data among processes is carefully managed such that expensive synchronisation mechanisms are not needed – it was the concern of the author that heavy use of mutual exclusion, for example, would quickly make the host Transputer CPU bound.

The following sections concentrate on the different areas of figure 4.16, which together make up the cache manager and streams control suite. The operations of the

cache and stream managers are closely linked, and are therefore discussed together. Similarly, because of the close relationship between the NFS daemon and the cache manager, a discussion of how these components fit together has been deliberately deferred to the next subsection.

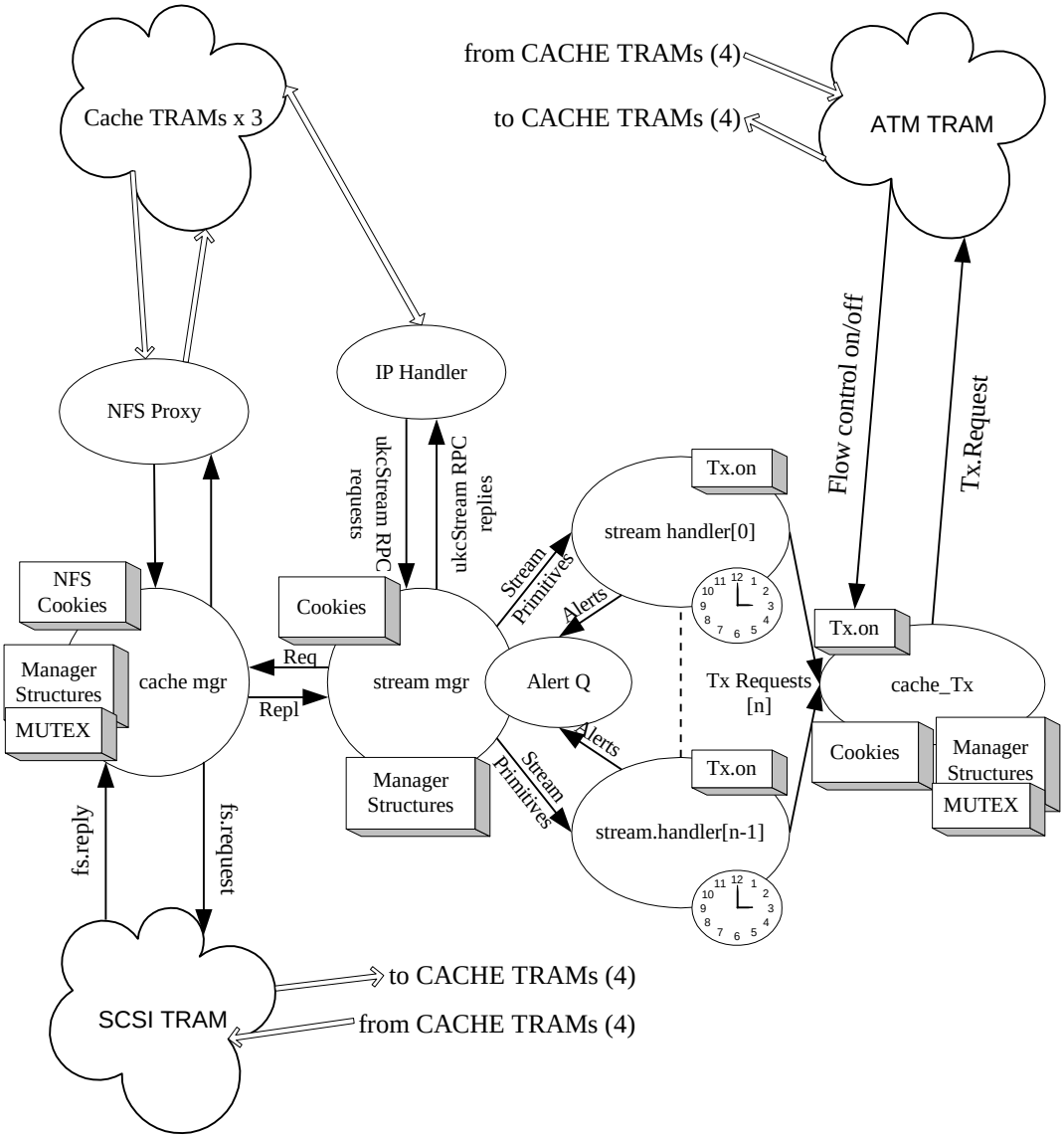


Figure 4.16: Cache and Streams Management Process Diagram

The edges of the diagram in figure 4.16 show the connections to the ATM and SCSI TRAMs. In addition, the IP process is shown as one process, though its responsibility is split over the four TRAMs in the backbone of the cache. For the prototype, the cache manager TRAM does not receive incoming IP traffic directly; the CPU is therefore dedicated to maintaining good real-time response during the operations of client streams. Hence, the ukcStreams requests are forwarded to the manager TRAM via an IPC[10]

[10] IPC – Inter Processor Communication

mechanism linking the four TRAMs. NFS data requests are handled *without* forwarding the entire RPC packet to the manager TRAM, as explained in the following subsection.

4.9.1 NFS and the Cache Manager

Each of the other cache TRAMs has an IP process which deals with traffic from the network. In general, the IP processes deal with NFS requests by talking directly with the file system on the SCSI TRAM. For data reads however, the cache needs to be used. In this case, the NFS daemon requests that the cache manager load data into the cache and return the address, or addresses, of that data. Of course, the data may already be present in the cache, in which case no data need be transferred from the filesystem TRAM.

The IPC mechanism connecting the processors on the cache backbone is used to make the data request to the cache manager. In figure 4.16, the NFS proxy receives this request and forwards it to the cache manager. Similarly, the list of addresses given by the cache manager is returned back through the IPC system to the calling NFS daemon. Once the addresses of the data are returned, the return RPC packet can be constructed using the case C READ in section 4.8.2.

Since the cache is allocated in 16Kbyte rows, data read from the file system is limited to multiples of the row size. In fact, this is a standard performance technique in file systems; the prototype uses a 32Kbyte basic fetch size. In addition, the cache manager will ensure that the next 16Kbytes of data after that requested by the caller is held in the cache, since normal file data are typically sequentially accessed.

To further improve NFS performance, ‘cookies’ are used to reduce the time spent searching file chains. Each file in the cache has its last accessed cache-row offset recorded. Thus, for each access to a file, the cookie is used as the start point of the file chain search. It is therefore important to maintain the file ownership of all lines in the cache management structures (figure 4.15), since the cookie might otherwise cause a search on the wrong file chain, or the free-list. Since the majority of file access is sequential, the search of the file chain is either not needed (same cache row), or is of length one.

Cache space is divided between NFS files and stream files by counting the number of stream users of each file in the file header chains; any files with zero counters are normal data files. NFS is allowed to occupy at most *n* Mbytes of space if the server has stream clients; in the prototype’s case, 4Mbytes is the maximum space given to normal files if the server has stream clients.

The inclusion of a stream counter within the head of each file in the file chain requires that streamed files never be flushed from the cache, since that would lose the client number state. However, streams typically always have data in the cache, so the constraint is not a limiting one. During group reassessment, the state counters make it easy for the manager to track resource allocation to normal and streamed data files.

The normal data files each have an expiry timer when streams are active. If no file activity to the normal file is made within a set timeout period, then the file concerned is flushed from the cache, hence making way for stream data. The prototype uses a timeout of five seconds.

If the cache becomes full, the cache will flush NFS data from the cache on a least recently used basis until enough space is available. A full cache will also trigger a reassessment of stream groups, which will also cause space to be freed in the cache.

4.9.2 The Cache and Stream Managers

The stream manager is responsible for maintaining the structures set out in Chapter 3, viz. figure 3.6. Namely, it keeps track of which files have stream clients, and organises the streams into groups. However, the stream manager does not make any data transfer requests to the file system directly; any transfer requests are made to the cache manager.

The details of how the streams are formed into groups was given back in Chapter 3, and is not repeated here. The prototype implementation is specifically a pure streams algorithm, and does not make use of the residual space left after allocating the cache to the smallest set of intervals. The discard of material from the end of each stream group is passive, that is, is not removed from the cache until the streams are reassessed. Reassessment occurs either when a client stream issues a reposition, a new stream arrives or another leaves, or the cache runs out of space.

The streams manager is driven by the incoming ukcStreams protocol requests, which are forwarded from the IP processes on the other three TRAMs on the cache backbone. The stream manager is therefore the ‘ukcStreams daemon’, and is responsible for replying to the protocol requests. Hence, the manager TRAM always responds directly onto the network to ukcStreams requests, and does not reply to the forwarding IP process.

The IPC mechanisms are used by the stream manager when setting up a connection with a client. The file that a client specifies has its metadata checksum checked against the main data file. Since the first file block is located in the first cache TRAM, an IPC request is made to it. Once the checksum reply is made, and the checksum is found correct, the connection procedure is completed.

4.9.3 Organisation of the Streams

For each CONNECT primitive received, the stream manager assigns a stream handler process to the maintenance of the client stream. Each handler process maintains the logical clock of its stream, and is responsible for the transmission requests of the frames of that stream. Thus, each handler has access to the meta data index of its charge. The index is fetched by the manager as part of the client set-up call.

Once the manager has assigned a handler to the operation of a client stream, it will alter or interrogate the stream as appropriate, according to the ukcStreams primitives that

arrive from the client. The handler processes communicate with the manager process only through the 'Alert' queue, since to do otherwise would cause deadlock.

The alert message is either a prefetch request, or a cancel notification. Each stream makes a prefetch request when the previous prefetch's data has been half consumed. The prototype tries to fetch a second of data ahead, and therefore makes requests each half second, even if the playback rate is altered. If the rate of play is altered, this affects the amount of data fetched from the disk. The prototype imposes a minimum transfer size of 32Kbytes and a maximum of 256Kbytes to keep performance high, while preventing one stream from consuming all of the disk bandwidth. If the alert queue has a previous request from the same stream, the previous request is overwritten, since the server is presumably too busy and has not yet serviced the last prefetch request.

The cancel notifications are caused by timeouts in the handlers. If there is no client activity for a paused stream for a specified period, then the handler tells the manager to free up the stream. The timeout period is specified as part of the CONNECT call.

The stream manager services the prefetch requests in First Come First Served (FCFS) order. At start-up therefore, all active streams will access the disk; the logical allocations of the stream manager made during assessment will slowly be turned into physical allocations. Once stable state is reached, only the leading streams of each group will access the disk, with the prefetch requests of the others being absorbed by the cache.

The stream handler processes request frame transmissions to the cache transmitter process. The transmitter, on receiving a request, searches the cache for the blocks requested. If the data are found, then the request is forwarded to the ATM TRAM for transmission to the client. As the cache manager can be adding or destroying file chains in parallel with the searches of the transmitter, mutual exclusion (a 'mutex') is needed to protect the transmitter from having the cache structures pulled from under it.

Similarly to the use of cookies with normal data files, the CPU time spent searching for file blocks to transmit is minimised by recording the cache location last accessed for each stream. The search for a file block continues from the point indicated by the cookie, and search complexity is generally reduced to one frame.

4.9.4 Flow Control and Deadlock Prevention

Should the ATM TRAM become congested, and the transmit queues back up enough to fill, the ATM adapter will switch off the flow of transmission requests. The variable '`transmit.on`' shared between the stream handlers and the transmitter, is checked by the handlers before any attempt to transmit is made. This approach prevents the handlers from blocking on a request to the transmitter, because the transmitter is itself blocked on a request to the ATM TRAM. The flow control is enabled once the ATM adapter is able to reduce its outgoing frame transmit queue to 25%. The ATM TRAM will throw away requests that it cannot cope with, though this results in the transmitter process making

requests that are promptly discarded. The flow control mechanism therefore switches off the requests at their source and saves link bandwidth and CPU time for both the manager and ATM TRAMs.

4.9.5 ukcStreams Protocol – Extensions to Palantir

During the development of the server, two other projects in the group made an attempt to play back synchronised audio and video. The results were not perfect, with lip synchronisation in particular being poor. Thus, the protocol designed for the prototype server included an 'ASSOCIATE' primitive, which slaves the clock of the client stream to that of another stream. Since the server is responsible for the transmission of data from two or more streams from a single logical clock, the synchronisation between the two should be far better. In addition, the slaving of the clock can be offset by a positive or negative amount, such that delays in processing different streams at the client can be accounted for. Furthermore, the size of the offset can be altered 'on the fly'. At this time, the ability to associate streams has only been demonstrated between multiple video streams; it has yet to be used for the synchronisation of audio and video with lip-sync.

4.10 Summary

In this chapter, the components of the video server were presented together with an explanation of the alternative approaches considered and the reasons why particular implementation paths were taken. On analysis, each of the system components was shown to satisfy the initial estimate of required performance, and was therefore sufficient to enable the successful construction of a networked video file server.

A number of notable performance related lessons were learned whilst constructing the server. These optimisations were necessary in order to achieve the required server performance. However, a discussion of these techniques is deferred until the closing Chapter 7.

5. Analysing the Principle of Stream Caching

The experiments given in this chapter were conducted on the prototype server described in the previous chapter. The cache algorithm used throughout all of the experiments was a pure stream cache, that is, without a modulated interval; assuming that Linington's cache simulation is sufficiently accurate, then the performance of pure stream versus modulated stream caching should not be very different. The simplest cache replacement strategy was used with the cache reassessment being set to occur on a user interaction, or when cache space became exhausted. Although this is not the most efficient algorithm, it does give a safe basis for prediction of the efficiency of stream optimised caching.

5.1 Method of Evaluation

For a classroom of users viewing a set of clips, the behaviour of those users is dependent upon a multitude of factors; no two sessions would ever produce identical traffic profiles to the server. Thus, to make such traffic reproducible for purposes of experimentation, the inputs would need to be recorded as a script of user requests for later playback. However, the server response would itself have an effect on the profile of such inputs. Hence, for the purposes of experimentation, the scripts played to the server ideally need to be independent of the server's response to those inputs, such that that response might be analysed in isolation.

A better approach than using real users is therefore to replay a script that, to the server, is indistinguishable from actual real-user input. Such inputs can be generated based upon predicted rather than actual user behaviours, to produce 'emulated' behaviours. Such generated behaviours have the advantage that they are quick to build and are easily modifiable. These inputs are then a known quantity and do not require extensive analysis to determine their effects on the server, as would real-user inputs.

To emulate a number of users a pattern generator can be used, which employs behavioural rules that predict that of real users. For example, each user has a probability per second that they will review the material they just watched, and another probability that they might alternatively skip forward over a section. Similarly, the user may choose to change the clip that they look at, which can again be governed by a probability. The pauses between the use of different clips can also be emulated by choosing a probability of such an intermission for each clip, and a further probability for how long they are not viewing any continuous media. Such a generator needs to output user 'events' at discrete logical times, namely the VCR primitives given in table 3.2. For each event generated, the appropriate argument(s) must also be created, for example, the new logical position of a skip operation or the relative index of a new clip to be viewed.

As a pattern generator also creates the logical times of the actions that it generates as it executes, i.e. the generator does not execute in real-time, such user emulation scripts can be generated within seconds. Thus, the expensive and repetitive step of recording user interaction data is avoided. Additionally, since the user behaviours are probability based, each run of the pattern generator will produce a different set of user actions. Further, the general behaviour of the output user-script can be quickly altered by adjusting the user behavioural descriptions that the pattern generator is using. For example, the frequency of backtracking can be reduced to very unlikely, since it is not expected, or made highly likely, for environments in which it would be expected. Hence, consideration of many different types of user behaviours can quickly and easily be generated, as opposed to collecting real-user data for each of the different target environments.

The user usage pattern generator, designed and written by Peter Linington as part of the SERVICE project, takes a description of each clip as given in Appendix A.1. Each clip has a specified length, and three properties that are specific to the clip. Each piece of footage will have an interest to the viewer, which if not very high, will result in a short viewing time before the user chooses another clip. In the descriptor, this transition has both a probability per second, and an upper bound. Similarly, depending upon the viewers interest in the clip, the user may pause, skip, continue or otherwise interact whilst watching the clip.

The variable NCLIP determines the number of movie clips that the emulated users will be accessing during the experiment. The experiments in this chapter used 10 clips, the parameterisations of which are also listed in Appendix A.1. The clips used were of varying length and size, and were taken from a related project; the “Co-ordination and control Interfaces in Multimedia Information Systems” (CIMIS) project used MPEG compressed clips to provide an interactive tourist information system. The clips used within these emulations are therefore ‘real’ data, not ‘almost MPEG’ mocked-up 1.5Mbps CBR data. The total amount of data over these ten clips was 177Mbytes, which is twice the size of the largest cache size for the experiments in this chapter. Thus, the prototype does need to access the disk; it cannot operate entirely from the cache. More details on the CIMIS project and its impact upon SERVICE are given later in the chapter.

An NCLIP by NCLIP matrix named UCLIPSEQ contains transition probabilities where each entry gives the likelihood of transition from clip *i* to clip *j*. This arrangement allows a most probable route for each user, whilst including the possibility of a deviation in the user’s path. The clip transition matrix is given in Appendix A.2.

Each user is characterised such that each user has a clip interest tolerance – some users are more patient that others. A delay before entry to the emulation simply accounts for the fact that users do not arrive to class at the same time, nor do they typically start at

precisely the same time. The user parameters used for the experiments in this chapter are given in Appendix A.3. Where less than 25 users were active during an experiment, only the first user descriptors listed in Appendix A.3 were used up to the number of users.

The maximum number of users was chosen as 25 so as to intentionally overload the server and examine its behaviour under a breaking strain; the practical maximum originally aimed for was 20 users.

Several different user interaction styles can be described by further a table of values. Hence, for a single set of users, a number of different user behaviours can co-exist. For example, frequently backtracking users can be mixed with others that never backtrack, but tend to skip over small sections of every clip. The table entry format is given in Appendix A.4. These experiments use only one user style; the emulated users are organised such that they are more likely to skip through than pause in each clip, but if they pause, they will continue reasonably quickly. The source clips range in length from 40 seconds to 4 minutes, so the skip behaviour is set to an average of 15 seconds backwards. The effect of the standard deviation is that the change of position will generally be between 45 seconds ahead, or 1 minute 15 seconds backwards. All user timing-distributions are exponential in the generator.

Although all NUSER users begin at the first clip, this clip can be set to length zero, in which case a distribution of starting point probabilities can be specified in the clip transition matrix. In allowing each user to pick their start clip and jump between clips, this produces a worst case for reuse, since ordinarily if users were progressing through some learning task, their access to the media clips would tend to be far more linear and predictable. Again, this would give a safe basis for the predicted performance of the caching algorithm being examined. The experiments of this chapter, however, assume a more orderly starting distribution, as this is expected in the target classroom environment.

The output of the usage emulator is a sequence of action records where each record consists of an event time in milliseconds, an operation identifier, the user's identity and an optional operation argument. For example, a SKIP action for user offset 0 at 1234 milliseconds needs a clip position argument to skip to, say 120000 milliseconds. A playback utility reads the action script and spawns NUSER client processes that convert the actions into the appropriate ukcStreams protocol requests and therefore emulate real users; the server itself does not know a real from an emulated user. The action script is further augmented with monitoring actions, which the playback utility issues to the server as the time fields of the operations dictate; a separate RPC protocol (vfsBench) is used for monitoring operations and is given in Appendix B.

5.2 Capture of Data at the Server

To collect data at the server, the server sources were modified to add in counting and summation of the most notable information. This was done without requiring a major reworking of the server code.

The vfsBench protocol needed to have near-zero impact upon the server's performance, so the items collected on each 'collect' call were chosen in such a way as to cause minimal additional CPU load. Since the data is collected from the manager TRAM, this processor was chosen as the only source of data; there is no instrumentation in any other server processor. The server collects data about each stream process and each cache block, and maintains totals across all such streams and blocks. The data on individual streams was originally designed to allow comparisons of behaviour between the streams of a single experiment, though it is not considered here for that purpose.

5.2.1 Cache Block Data Records

The server maintains the following data about cache blocks for return via vfsBench protocol :

`blocks.used`	Total number of blocks in cache that are either allocated but are so far unused, or allocated and used once only
`blocks.reused`	Total number of blocks allocated and used more than once, i.e. have been reused

The use/reuse counters are achieved in the server by the addition of an element to the cache management structure (figure 4.15). For each cache line entry, a one-byte state element is given to each cache block. Thus, for the prototype with four blocks in each cache line, this gives an additional four bytes overhead to each cache line structure.

The state entry is one of `ALLOCD`, `USED` or `REUSED`. The state flags are modified by the cache routines for allocation, de-allocation and searching (for cache blocks to transmit). The totalling counters `blocks.used` and `blocks.reused` are increased and decreased based on the block states as these cache operations are called, thus distributing their computation evenly over time. These values can be used to calculate the spatial efficiency of the cache content. The block state elements further enable the measurement of temporal cache efficiency as outlined in the following section.

5.2.2 Per Stream-Process Data Records

The per stream process (not per active-stream) recorded information is as follows:

`fps.counter`	Number of entries in the `fps.table` (listed below) limited to the table size and reset by new connections and continuations.
`fps.index`	Next index to be used in `fps.table`

`fps.table`	The indexed element records the processor timestamp of the next frame to be transmitted. The actual frame rate can therefore be calculated by consideration of the last `fps.counter` elements.
`transmitted`	Total number of successful transmissions
`overdue`	Total transmit latency from frame transmit timer-interrupt event time to cache search success and subsequent transmission
`successes`	Total number of successful cache frame searches including those enabling reuse (*all* transmissions)
`reuse.hits`	Total number of successful cache frame searches on frames that had been used before, i.e. enabled reuse of data
`failures`	Total number of unsuccessful cache frame searches
`dropped`	Total number of frames dropped due to flow control actions between the cache TRAMs and the ATM board.

In the calculation of the actual frame rate of each stream, paused streams are not considered, so that they are not confused with poor quality play. Similarly, a resuming stream has its `fps.counter` reset to zero, so that the period of the pause is not taken into account when calculating the stream's frame rate. The frame rate calculation is:

$$1000000\ \mu s \times \frac{\text{fps.counter} - 1}{\text{fps.table[most.recent.time]} - \text{fps.table[oldest.time]}}$$

Each stream process firstly checks that the ATM adapter is not overloaded with traffic before it requests a frame transmission. Thus, if the ATM TRAM has switched on the flow control mechanism, the frame is dropped before any request to transmit is made. Consequently, the transmitted count does not count those frames that were dropped because of flow control actions.

When a stream handler process does request a frame to send, a separate transmission process then searches the cache to check that the data is available for transmission. If all the data is present, then a search success is recorded for each attempted transmission. If only some or none of the data is found, then a cache failure is recorded. If a full cache success is made, then the frame transmission descriptor is forwarded to the ATM TRAM for processing. Thus, for the prototype server, the number of cache successes and successful transmissions are in fact identical. This would not be the case however, if transmissions could fail after a cache search success. For the server, frame transmissions submitted to the ATM adapter always succeed.

The searching of the cache in the transmission process is also responsible for the alteration of each block's state from `ALLOCD` to `USED`, or `USED` to `REUSED` for each cache block pending transmission. For a cache search success, it is then possible to additionally

increase the `reuse.hits` counter for the stream process if the first block in the frame indicates that it was already `USED` or `REUSED`. The temporal cache reuse efficiency can thus be calculated based upon total successes, total reuse hits and total failures.

Since each stream handler is requesting to the stream manager to fetch the data it will require, a cache failure will occur where the prefetch request could not be serviced before the stream reached the data it requested. Hence, the cache success- and failure-counts include those streams that are relying on their prefetch area as a playout buffer. That is, the definition of cache success here includes the first use of any cache block for each transmission attempted. Therefore, the 'cache success' given here is not an indication of cache re-use; this measurement of a 'hit' is given by the term 'reuse'.

The overdue counter records the cumulative time taken to respond to each frame transmission, though only successful transmissions are counted. The measurement begins within the stream handler process at the wake-up time that the frame was due. It ends within the transmission process after the cache search success has been established and before the request is forwarded to the ATM TRAM. Hence, the transmission response latency is measured only within the manager TRAM. However, this latency will be affected by CPU loading, since higher CPU loads will delay outbound requests.

Note that individual stream sessions cannot be told apart, since two sessions may use the same underlying stream process. Indeed, in a single sample interval, many such successive stream sessions may be handled by the same stream handler process. For this reason, no attempt has been made to analyse the difference of behaviour between the streams of each single experiment. In any case, this would massively increase the size of the chapter, which concerns itself only with average performance across all streams and hence allows a comparison between different experimental runs.

5.2.3 Global Stream Data Records

For all streams on the server, the following totals are recorded:

`cache.success.count`	Number of cache frame successes/transmissions
`cache.reuse.count`	Number of successes that enabled reuse in the cache
`cache.failure.count`	Number of unsuccessful cache frame searches
`disk.accesses`	Number of accesses made to the file system
`disk.bytes`	Number of bytes transferred from the file system
`disk.time`	Amount of time spent in file system data reads

The totals for both cache successes, reuse hits and failures are performed at the same point as for the individual stream-process data within the frame transmission process. Again, any frames dropped due to ATM flow control do not have any impact upon cache successes and failures, since no frame transmission request is attempted in this case.

The disk activity values are accumulated within the stream manager process, which fetches data on behalf of the stream handler processes. The fetches are typically of a fixed prefetch size, but the cache manager can break these down and possibly eliminate them where the data already resides in the cache and does not need to be fetched from the file system. Thus, a disk hit is counted only as one request to the cache manager, even if the cache contents caused several separate file system transfers to be made. However, where no bytes were transferred, no hit on the file system is counted. The total time taken by each transfer request to the cache manager is accumulated, even when there is no access to the file system. In the latter case, only the overheads of checking that the cache already had the data are timed.

The amount of data transferred from the file system is actually measured in units of blocks and multiplied by the logical file-system block size. Therefore, the raw transfer rate of the file-system is being measured. The effective transfer rate depends upon the storage efficiency of the clip concerned; there are always at least 16 Bytes unused at the end of each block for the server's AAL5 encapsulated file format. The actual storage efficiency of a clip is dependent upon the encoding of the media file.

5.3 Results of Emulated Multiple Users

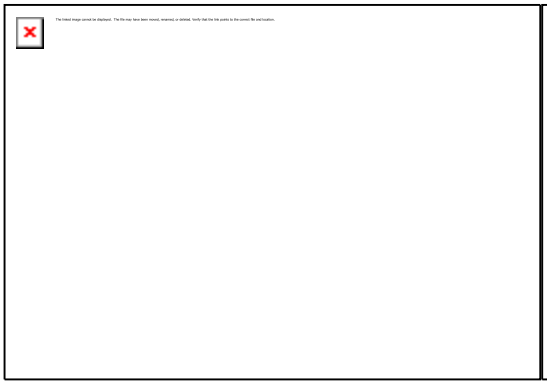
All of the following results were taken during the busiest part of the generated emulation such that all the users in the emulation cause simultaneous loading on the server at some point in the trace. Note that the time axis is taken relative to the start of the full emulation period, not the start of the monitor's readings. In the case of the experiments here, the start of the readings is after only 100 seconds of emulation, and therefore the traces begin near the start of each graph. The reason for the early start was simply that the traces quickly ramp up to maximum user load. A snapshot of server performance was taken using the vfsBench RPC protocol with a sample interval of 10 seconds over a period of 2400 seconds (40 minutes) for all experiments.

In the following sections, each starts with a definition of the item being analysed. In most cases, this is the same as one of the totalling counters given in section 5.2. However, some of the graphs display data derived from the basic vfsBench data. At the end of each subsection, the data are discussed in isolation with some referencing to previous subsections, where appropriate. Section 5.4 discusses all of the results collectively and draws some conclusions.

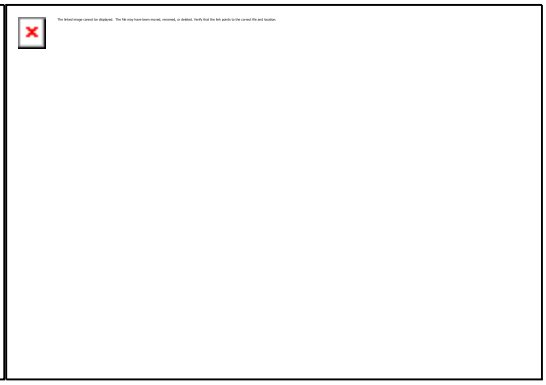
5.3.1 Number of Users Active

The number of users active is the number of users that are actively playing at the sample instance, i.e. non-paused users. Thus, each sample value does not account for the number of users in previous samples.

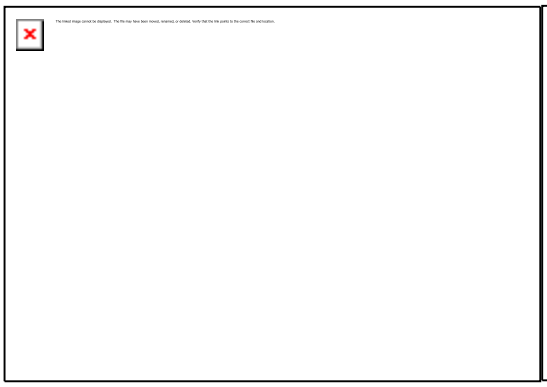
The graphs 5.1 to 5.4 trace the number of users connected to and using the server per unit time during the progression of the experiments. Since the trace records the number of users actively playing back a file, and does not include those streams that have paused, the traces should record the number of users that are having an impact upon the performance of the video server; paused users do not use system resources. These data sets are not of great interest in themselves, but do provide a basis for interpretation of the results which follow in later sections.



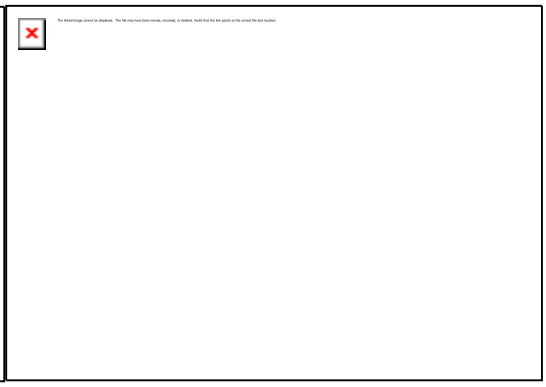
Graph 5.1: Number of Users with time (10 users)



Graph 5.2: Number of Users with time (15 users)



Graph 5.3: Number of Users with time (20 users)



Graph 5.4: Number of Users with time (25 users)

As the emulations progress, the number of users active on the server can be seen to be continually changing as the emulated users are constantly altering their behaviour. This includes both pause-resume operations and changing the viewed clip. For all of the experimental data presented in this and following sections, the time axis records only the first 2500 seconds (42 minutes) of data. It is this section of the emulation trace that is most interesting and contains the rapid build-up to peak usage and the slow decay in the number of active users. Thus, the traces presented contain the busiest part of the emulations and record the server behaviour under the heaviest loading conditions for each emulation. The drop off in the number of users with time is a result of the hastiest users in the emulation finishing more quickly than the more leisurely users.

On examination of the resultant traces in graphs 5.1 to 5.4, the number of users for each set of users (that is, 5 (graph omitted), 10, 15, 20 and 25 users) is very similar regardless of any variation in cache size. This breaks down for larger user sets, viz. 20 and 25, where the traces for the different cache sizes do slightly differ. This is most likely due to greater loading of the server causing longer delays to the servicing of user requests. Consequently, since the experimental traces are sampled at 10 second intervals from a Unix host clock, the observing monitor process will catch these variations in control response. For example, for a lightly loaded server, a continue request may be serviced before a sample point, whereas a busy system may take the sample before the continue request is serviced. The result is a difference of one user in the sample. The vfsBench protocol request is also subject to server delays and will therefore be temporally affected by heavy server load.

Note that not all experimental data is presented in this chapter. Where two or more graphs do not provide additional insight over one graph, the latter graphs are intentionally omitted. In this subsection, the 5-user set traces are similar to the 10-user traces, but with half the vertical scale; the graph is omitted here for the sake of brevity.

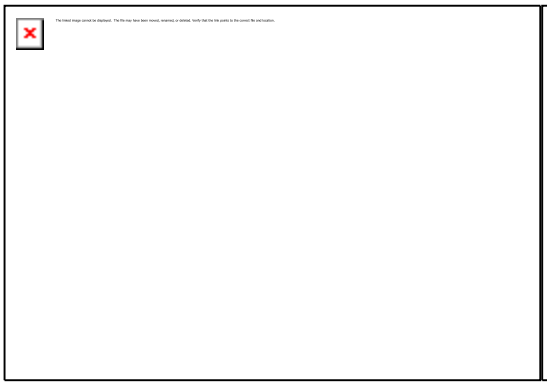
5.3.2 Number of User Groups Active

A stream group is defined as a set of contiguous stream intervals. Two stream groups will therefore either belong to different files, or will be separated by an uncached interval. Similarly with the sampling of the number of users, the number of groups recorded in each sample is precisely that number at the instance that the sample was taken, and does not consider the number of groups active prior to that sample.

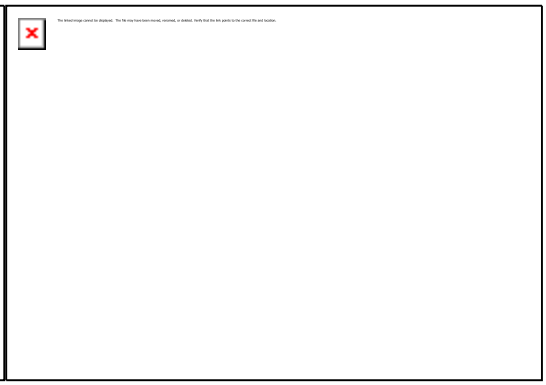
Stream optimised caching is based around the concept of grouping the set of streams with the shortest inter-user intervals. A good indication of algorithmic success should therefore be given by tracking the number of groups for a set of users, that is, the group to user ratio. Essentially, the fewer the number of groups, the more efficient the cache algorithm; efficiency is determined by group followers since these enable cache reuse. Of course, the efficiency will be limited by the number of users involved; mathematically, high performance is only possible for a large number of users. This is because reuse is more likely with a large set of users. Algorithm efficiency is also determined by the number of target clips on the video server. For example, a user population that is larger than the number of clips available will always lead to reuse. However, the practical efficiency is determined more directly by the temporal locality of accesses among users. Therefore, the worst case behaviour will be that where each user accesses a different clip, since no temporal locality of access is then possible.

With a few users, for example in graph 5.5, the size of the cache is less critical, with a small cache giving equal performance to a larger cache. Here, the number of users most frequently mirrors the number of groups. In this case, the extra memory used by the

cache is of no extra benefit. This is due to the few users being unlikely to view the same material at around the same time, with typically the emulated users viewing different clips. In those cases where two users do view the same clip, the interval is often too large to be cached. An example of this can be seen at around 800 and 2000 seconds into the 5 user set trace, where the larger cache size enables non-singular groups to form with large intervals. The smaller cache sizes are unable to service such intervals as they exceed the total available cache.



Graph 5.5: Number of Groups with Time (5 users)



Graph 5.6: Number of Groups with Time (20 users)

Increasing the size of the user set gives a clearer separation of the effect of cache size on the grouping algorithm efficiency, for example in graph 5.6. For the 20 and 25-user sets, the large cache size enables a few large groups to form, whereas the smallest cache size is unable to employ cache space efficiently. These results are clear evidence of the requirement of a large user set in order to achieve a high ratio of users to groups, and therefore algorithm efficiency.

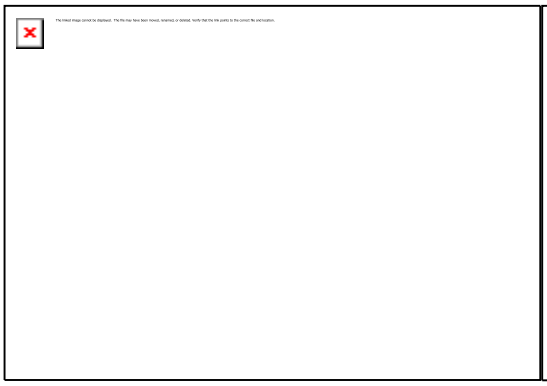
5.3.3 Cache Success to Failure Ratio

The success to failure ratio is derived from the 20 most recent total successes (successful transmissions) and total failures samples (unsuccessful transmissions). The samples that are 20 sample periods in the past for both cache successes and failures are subtracted from the most recent samples. The success ratio is therefore taken from the successes and failures in the cache over the duration of the last 20 sample periods. Hence, a recent success ratio is calculated, and not a ratio from the start of the experiment. The latter would be more heavily biased by historical data the further into the resulting trace. The success ratio formula may be summarised thus:

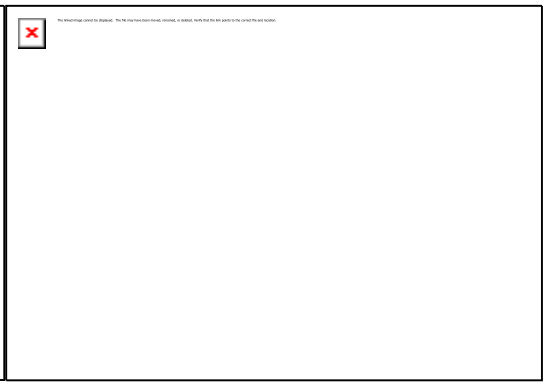
$$success.ratio_n = \frac{successes_n - successes_{n-19}}{(successes_n - successes_{n-19}) + (failures_n - failures_{n-19})}$$

Although the group to user ratio (sections 5.3.1 and 5.3.2) can give a good idea of cache algorithm efficiency, it goes not give a good guide to practical server performance.

In the ideal, the media server will always have available in the cache any frame that is requested for transmission. That is, the ideal server has 100% cache success rate.



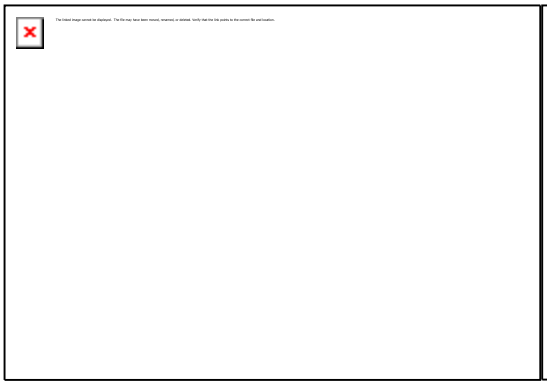
Graph 5.7: Cache Success Ratio (10 users)



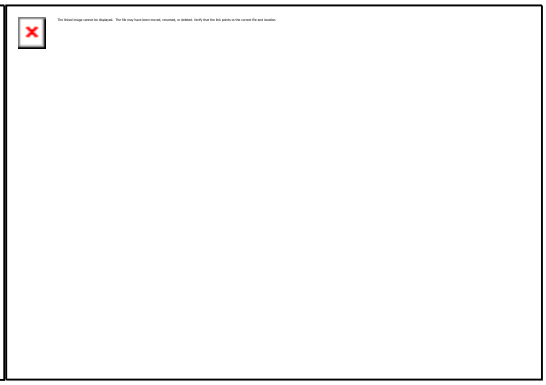
Graph 5.8: Cache Success Ratio (15 users)

Perhaps unsurprisingly, the five-user set trace produces a near 100% success rate, with only a minor degradation for smaller cache size. The reason for the less than 1% cache failure can be explained by user reposition requests, which will cause cache failure until the file system can service the prefetch for the stream concerned. Although the file system can fetch a large chunk of data for a stream client, this can take more time than a single inter-frame interval. Consequently, frame requests are made before the file-system has completed the transfer of data into the cache; these give rise to failures.

For a few users, once the streams diverge to the point that stream caching does not occur, the disk is able to satisfy the demand for data. The more serious cache failures for any cache size and user set size are caused by the inability of the file system to satisfy those streams that cannot be satisfied from the cache.



Graph 5.9: Cache Success Ratio (20 users)



Graph 5.10: Cache Success Ratio (25 users)

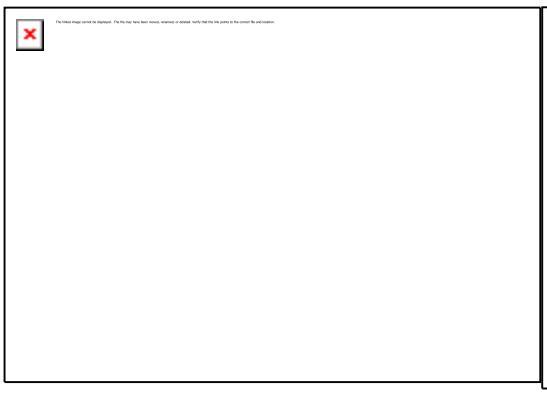
When increasing the number of users in the set from graph 5.7 to 5.10, the cache success rate is made worse by decreasing the size of the data cache. Even so, for the 20-user set of graph 5.9, a near perfect success rate is maintained for the largest cache size, bar some brief spikes at a few instances. The reader will notice that for each user set trace

of the larger user sets, the different cache size traces are amplifications of one another, with the smaller sizes giving worse performance. For the 25-user trace of graph 5.10, the largest cache size cannot provide perfect success rate, though not enough data has yet been presented to explain *why* the cache success rate falls. However, the later sections on disk performance do provide an answer. The reader will further note that the sections of cache failure correspond with a high number of groups for the given user set size and cache size – this is not obvious from the graphs given in this section.

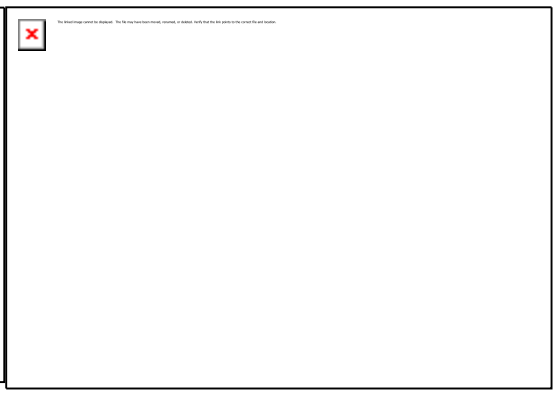
5.3.4 Actual Frame Rate

The frame rates reported by the server for each stream consider only the most recently transmitted frames of each stream at the instance the sample was taken. Thus, the average frame rate is the average of all such values, but does not consider the zero frame-rate streams, which are paused streams. Consequently, the value gives an idea of the quality of output of all the streams. In practice, if the average frame rate is not 25 in such a trace then a few of the streams might be dropping frames whilst the others are at 25 frames per second, for example. A more accurate picture of the frame rate of each active stream would require examination of the individual stream process data of each experiment, which is not considered in these experiments.

As stated in the cache success/failure ratio section, the users' primary concern is with the quality of service. It has already been seen that frames can be dropped, though this may be single frames dropped, or a burst of dropped frames. Regardless, the effective frame rate produced by the server gives some indication of practical performance.



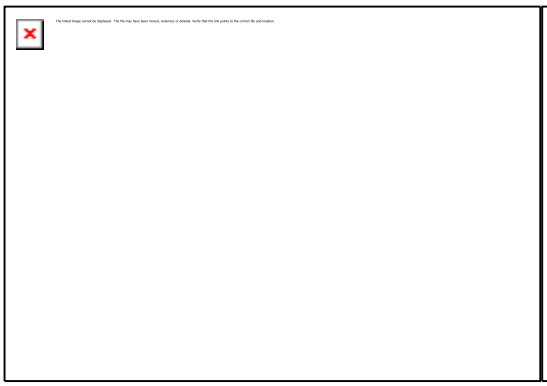
Graph 5.11: Average Server Frame Rate (10 users)



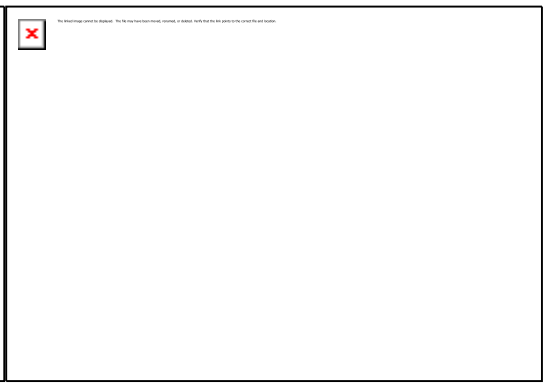
Graph 5.12: Average Server Frame Rate (15 users)

It is of no surprise to find that the shape of the frame rate traces of graphs 5.11 to 5.14 are almost identical to those of the success ratio in the previous section, despite the different unit of measurement. As before then, the frame rate is degraded more by smaller cache sizes; the effect is amplified by the larger user sets. For the largest set of 25 users in graph 5.14, serious degradation of outgoing frame rate does not occur until all 25 users are simultaneously active. This however, is not surprising since their aggregate

bandwidth then exceeds the current capacity of the ATM TRAM. The flow control mechanism between the cache backbone and the network interface switches off the transmit requests of the stream handler processes and therefore has a direct impact upon both the measured and perceived frame rate. This degradation also occurs in the largest 20-user trace to small degree for the same reasons.



Graph 5.13: Average Server Frame Rate (20 users)



Graph 5.14: Average Server Frame Rate (25 users)

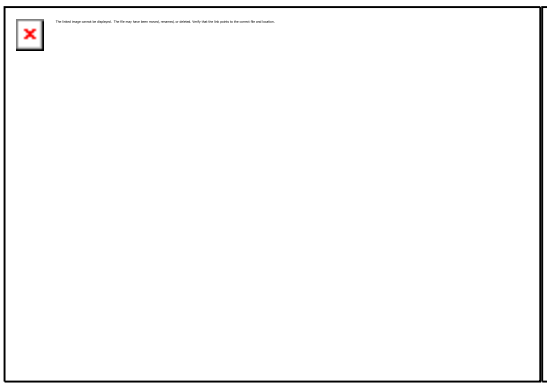
As mentioned in the previous section, a discussion of why the frame rate can drop is deferred until the disk performance is examined.

5.3.5 Frames Dropped on Transmission

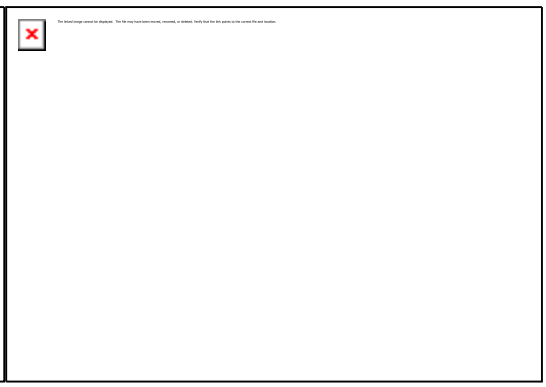
If the network interface buffer pipeline becomes full, a flow control mechanism throttles back transmission requests from the main cache controller, such that the network interface can recover. Only a subset of the 25 user set traces display any significant frame dropping behaviour, namely, for the two largest cache sizes. The 20 user set trace in graph 5.15 shows a series of spikes for the largest cache size. The smaller user sets below 20 users do not show any frames dropped for these experiments, and thus their graphs are omitted here. This behaviour is not surprising, given that the aggregate bandwidth of less than 20 streams cannot saturate the CPU-bound 28Mbps network-interface. The reader will note that the ATM hardware is rated at 100Mbps, though data segmentation demands on the CPU limit the effective throughput for this device.

For the 20 user set trace in graph 5.15, the ramp in the trace is concentrated from around 600 to 900 seconds into the largest cache experiment. Notice that the cache success ratio and average frame rate are slightly affected at these points also. For the 25-user trace in graph 5.16, the server overload during the duration of maximum user activity can clearly be seen, between 250 and 1000 seconds. This corresponds well with the cache success ratio and frame rate graphs over the same period. Simply, these sections of the experiments are where the server has become limited at least by available network bandwidth. This further suggests that the stream optimised cache is capable of greater outgoing aggregate transfer rates. Notice that 20 1.4Mbps streams produces a total of

28Mbps, which exactly matches the observed maximum throughput of the ATM network interface under ideal conditions. In addition, the 90Mbyte experiments output in excess of 700,000 frames, so these losses represent a total of around 0.1% and 1%, respectively.



Graph 5.15: Frames Dropped with time (20 users)



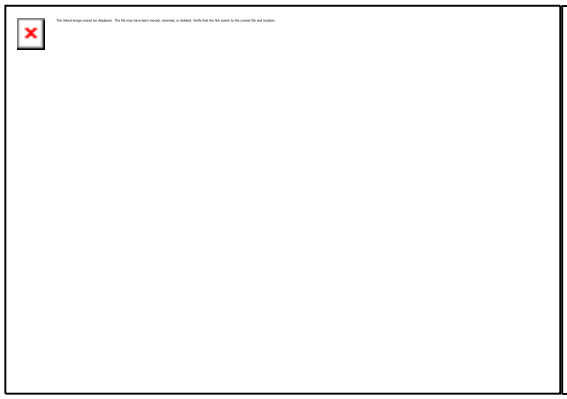
Graph 5.16: Frames Dropped with time (25 users)

As noted above, the 20 and 25 user sets do not all produce dropped frames; the larger the cache size, the more frames are dropped. The reason for this is that the smaller cache sizes are not efficient enough with their data cache, that is, there is a poor cache-success ratio and therefore a poor effective frame rate output. Consequently, the smaller cache sizes do not produce enough outgoing aggregate bandwidth to saturate the network interface; the flow control mechanism will never be triggered.

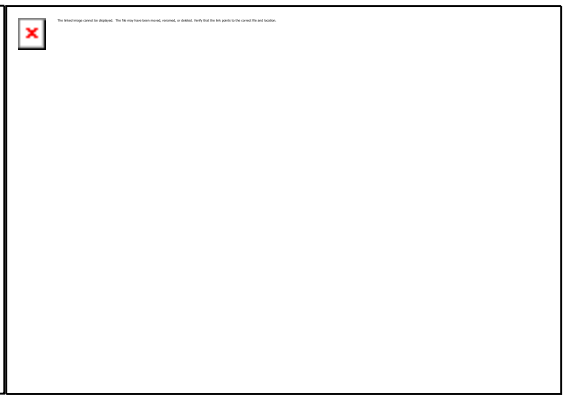
5.3.6 Temporal Cache Reuse Efficiency

Temporal cache reuse efficiency is calculated based upon the number of cache successes, cache reuse-hits and cache failures in the duration of one sample period:

$$reuse.efficiency_n = \frac{reuse.hits_n - reuse.hits_{n-1}}{(successes_n - successes_{n-1}) + (failures_n - failures_{n-1})}$$



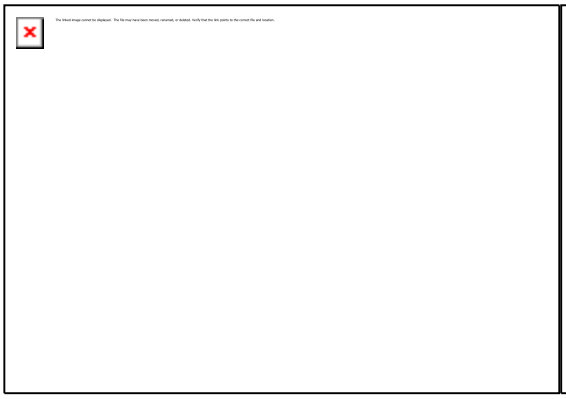
Graph 5.17: Temporal Cache Reuse Efficiency (5 users)



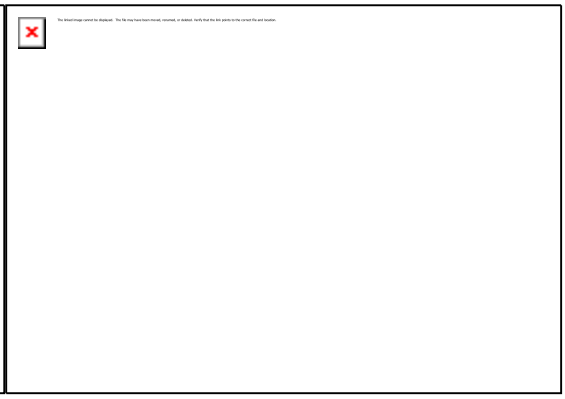
Graph 5.18: Temporal Cache Reuse Efficiency (15 users)

As with previous graphs, the different cache traces tend to be amplifications of each other, showing that a larger cache enables better use of the available space in stream

caching. In increasing the number of users, the trace becomes more stable up to the optimal number of users.



Graph 5.19: Temporal Cache Reuse Efficiency (20 users)



Graph 5.20: Temporal Cache Reuse Efficiency (25 users)

Generally, the efficiency of cache reuse is inversely proportional to the number of groups, as would be expected. That is, formation of a few dense groups gives better reuse. For the 5-user trace, the groups tend to be singletons such that reuse is not possible – thus, the graph tends to suggest poor, or at least widely varying performance. Performance is also generally lower at the tail of each experiment, where the few users remaining diverge, making reuse difficult. In particular, the 25-user trace is poor at around 1750s. Here, the users apparently form independent groups, so no reuse is possible; notice that success ratio and average frame-rate are both unaffected over this period. In addition, the 20-user trace also shows a poor efficiency of around 40%, but does better than the 25-user case. In the 25-user case, the users form singletons, whilst the file system is still able to cope with this number of simultaneous users. For the 20-user trace, the cache is most likely taking advantage of residual cache data.

Residual data is that which is outside of any interval and is therefore predicted not to be reused. If the cache is not running at capacity, then such non-interval data can remain in the cache until flushed (or recalled) at a later time, i.e. the data is residual.

5.3.7 Temporal Cache Reuse Efficiency and Mean Group Size

For each group, the leader fetches data from the disk, whilst all other members of the group reuse data that the leader has fetched. Hence, for a single group *j*, reuse efficiency can be expressed thus:

$$group.eff_j = \frac{u_j - 1}{u_j} = 1 - \frac{1}{u_j}$$

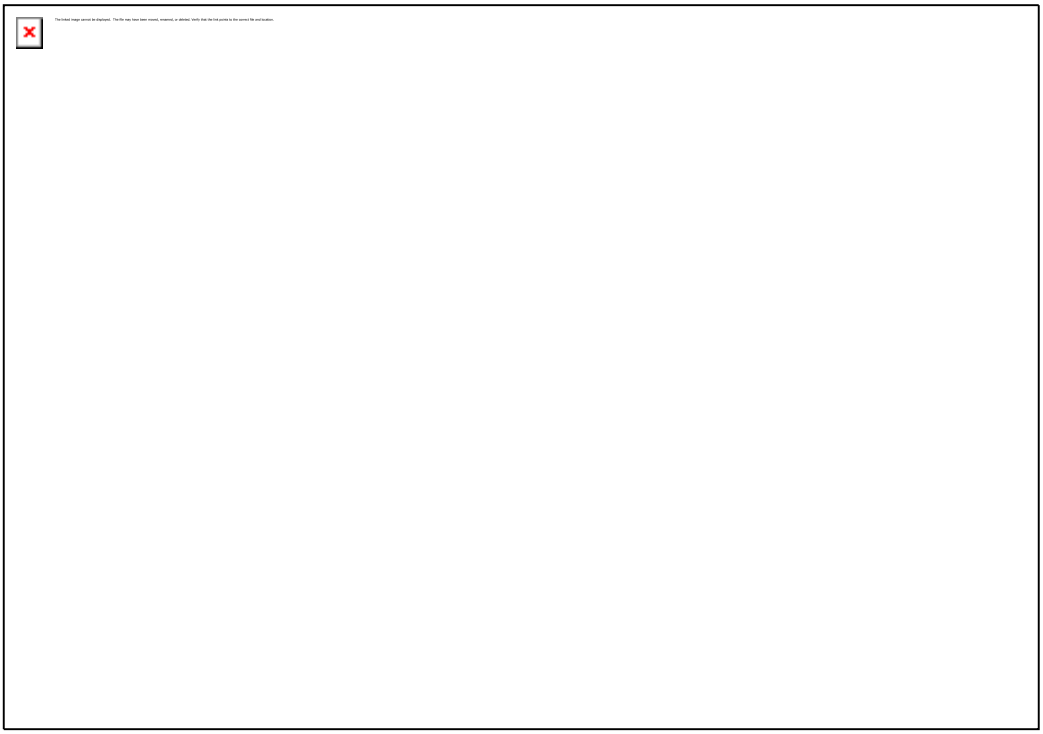
where u_j is the number of users in group *j* and *group.eff* ranges from 0 to 1, i.e. 0 to 100%. Further, each user in each group *j* causes *k* events per second. For *G* groups then:

$$total.eff = \frac{\sum_{j=1}^{G} k.(u_j - 1)}{\sum_{j=1}^{G} k.u_j}$$

$$total.eff = 1 - \frac{1}{u_{mean}}$$

Hence, the ideal implementation of stream caching will follow the above-derived rule. Therefore, the mean group size in graph 5.21 is transformed through taking the reciprocal, such that the y-axis of the graphs is ($1 \div u_{mean}$) and ranges from 'perfect reuse' (0) to 'no reuse' (1). As with the graphs in the previous section, the points in the graphs consider only the sample period at the instance of the sample. The value for u_{mean} is derived from the *total* number of users (not *active* users) and total groups at the sample instance. In graph 5.21, all four cache size experiments for 20 users are plotted in the same trace, since this produces an efficiency plot ranging from 0 through 100% and also shows that the individual plots agree well with one another. Only the 20-user experiment is given here, since the traces of all user populations give approximately the same shape.

Graph 5.21 confirms the relationship $eff_t = 1 - (1 \div meanGroupSize)$ where $0 \leq eff_t < 1$, which is a straight line from 0% reuse for 1 user per group, to 100% reuse at ∞ users ($y = 0$). In fact, none of the experimental traces follow this 'ideal' profile, but take the same essential path as graph 5.21. This suggests that, on average, below around 70% reuse, the cache is getting worse reuse than the ideal model. However, it does better at over 70%. For example, with an average group size of 2 ($y = 0.5$), reuse should be 50%. For the prototype, reuse ranges from 10 to 50%.



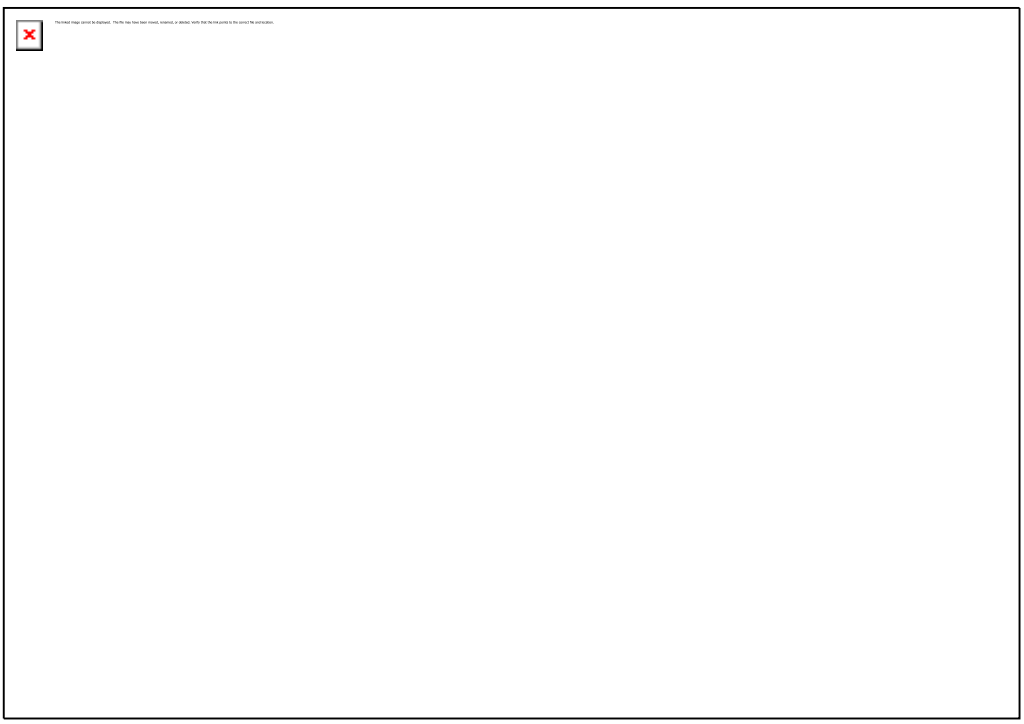
Graph 5.21: Temporal Reuse against 1 / Mean Group Size (20 users contour plot)

The above phenomenon can be explained through prefetch and use of residual cache data. Prefetch for the MPEG clips used here is roughly constant, regardless of the cache

size. Hence, for smaller cache sizes, relatively less cache space is available for interval caching, so reuse is less likely. For large cache sizes, the relative portion of cache space used for prefetch is much less compared to that available for interval caching. Furthermore, the larger the number of groups, the more prefetch is required; the denser and fewer the groups, the less the requirement for prefetch. The higher cost of prefetch for smaller cache sizes is therefore compounded by the lower grouping efficiency.

Residual data in the cache can result in better than expected performance, since an attempt to use data that is not predicted to be reused in fact makes a reuse hit on the cache. This phenomenon is more likely when grouping density is high, since residual data then becomes available; the sum of the intervals is less than the size of the cache.

The above is clearer when examining the four cache size traces separately, as shown in graph 5.22. The reader will note that the 90Mbyte trace on average falls above the 'ideal', whereas the lowest cache size mostly falls below it. As remarked above, the relative cost of precache is higher for a smaller cache size. In addition, large cache sizes have greater potential for residual data caching.

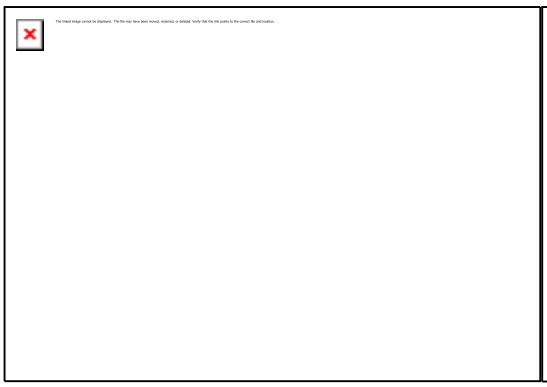


Graph 5.22: Temporal Reuse against 1 / Mean Group Size (20 users 2-D plot)

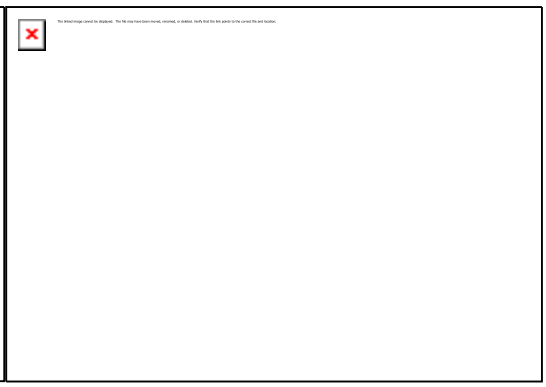
5.3.8 Internal Latency to Frame Transmission

The overdue time, or latency, reported by the server is a cumulative counter since the start of monitoring the experiment. In a similar approach to the calculation of the success to failure ratio, only the most recent 10 samples of latency are considered when deriving the internal latency. Each stream has a separate latency counter such that different stream handler processes might be compared, though the facility is not employed here; only the

sum of latencies for each sample are used. The latency samples 10 periods ago are subtracted from the most recent figures, and divided by the number of cache successes over that sample period. The latency is divided among the cache successes over the same period, since only each successful transmission has its latency added to the total for that stream.



Graph 5.23: Server Transmission Latency with time (15 users)



Graph 5.24: Server Transmission Latency with time (25 users)

In chapter 2, the importance of a constant and very low frame transmission-delay was covered. Failure to keep a constant time between frames results in a jitter within the client display. For the prototype server, if the transmission latency were too high, then the consequent control lag would be irritating to the user. Similarly, if that latency response is highly variable, it may produce jitter. Additionally, a scaling up of the number of users active on the system must not produce significant additional latency or greater latency deviation.

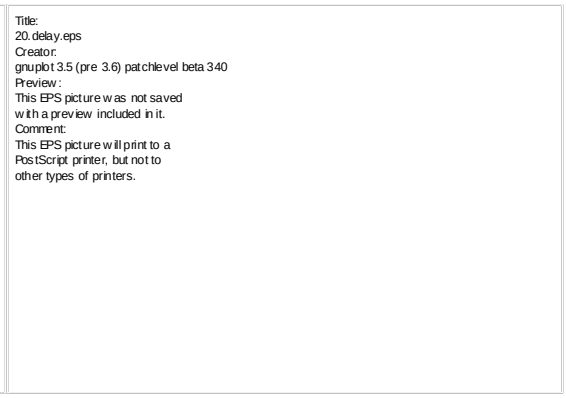
Accurate timing across two or more TRAM systems is not possible, because each has a different clock. Thus, the latency measured in these graphs covers only the time between when a stream handler process wakes in order to transmit, to the time the transmission block descriptors leave for the ATM TRAM. That is, only variations within the management TRAM are measured. However, the ATM TRAM will generally provide uniform transmit response times unless the interface is becoming overloaded. In this case, the buffer pipeline will back up, and hence an additional delay equal to the time it takes to reach the head of the pipeline is incurred. However, this delay may or may not occur regardless of the underlying file system, cache control and stream management. Thus, the effects of the stream management and stream optimised caching alone may legitimately be measured in order to examine the effects of scaling this approach to the delivery of continuous media.

In any operational situation where cache space is at a premium, frequent group reassessments are needed in order to achieve the maximum benefit of the available memory. In such cases, the response latency can be affected because the amount of free CPU time available for processing transmission requests is reduced. Consequently,

transmit latency is generally lower for larger cache sizes where reassessment is not needed as frequently. However, where poor success ratio occurs, no frame transmit request is made, and therefore no additional latency from such failures can be recorded. Hence, those user-set experiments that experience poor success ratios do not *appear* to suffer from greater transmission latency. Furthermore, any comparison between cache sizes is not meaningful where their cache success ratios differ. Nonetheless, latency is seen to be around 400μs for all traces, and hence does not cause any significant problems with either control lag or playback jitter. Indeed, the time taken to move the data from the cache into the network interface TRAM and transmit all the parts is significantly greater than this, at around 2ms per 4Kbyte block. All of the user-set traces are very similar, hence only the 15 and 25 user sets are shown in graphs 5.23 and 5.24, respectively.

The spread of the latencies in the experiment was examined by repeating the experiment with much shorter sample interval times. The 20-user experiment was repeated for the largest cache size with 1 second, 500ms and 100ms sample intervals. The 100ms and 1 second experiments are shown in figures 5.25 and 5.26, respectively. The graphs of all three intervals show very similar traces, with a general worst case latency of 2ms. The different sample intervals therefore suggest that the spread of latencies over the experiment samples is not large, and that the maximum worst case latency is no more than 4ms. The increase in the sampling rate *will* produce additional server loading that will itself affect the transmission latency of the server. However, the sampling intervals used here should not have a very significant effect on latency.

Graph 5.25: Server Transmission Latency with time using 100ms sample interval (20 users)



Graph 5.26: Server Transmission Latency with time using 1 second sample interval (20 users)

With respect to the buffering on the ATM TRAM, a worst case jitter arises where all active streams but one stop. The remaining stream would suffer a jitter equal to the length of the suddenly empty buffers; the next frame after the stops arrives early at the target relative to the previous frame. Similarly, if one stream is active and many streams suddenly all begin simultaneously, then buffers can fill rapidly and the next frame of the original stream arrives late at the target. The prototype's buffering takes at most 50ms to

empty in the worst case from 100% full. In either case, the single stream considered in the examples suffers at most 50ms worst-case jitter. In general, however, the delay imposed by the length of ATM buffering would rise and fall gradually, thus not adding significantly to transmission jitter. Regardless, the 4ms worst case jitter caused by the manager TRAM in submitting transmissions does not add significantly to the worst case jitter caused by the ATM board's buffering. In any case, this buffering delay is an artefact caused by the CPU limitation on the ATM board; a faster implementation of this device would not suffer the buffer back-up phenomenon. Such an implementation would therefore not be a source of potentially significant jitter.

It is clear from the 10 second interval graphs that the latency on the manager TRAM does increase with a larger number of active users, reaching a maximum of 1ms whilst overloaded with either 20 or 25 users. Given however, the lack-lustre performance of the T805 30MHz Transputer as compared to today's CPUs, the very low latency response of the prototype should be maintainable for a far greater number of users. Considering that each client display must be buffering received frames by at least one frame, a deviation in the order of a few milliseconds can be tolerated without introducing jitter into the output. Thus, these experiments suggest that the prototype meets the requirement of low latency frame transmission with very low timing deviation.

5.3.9 Amount of Data Transferred from Disk

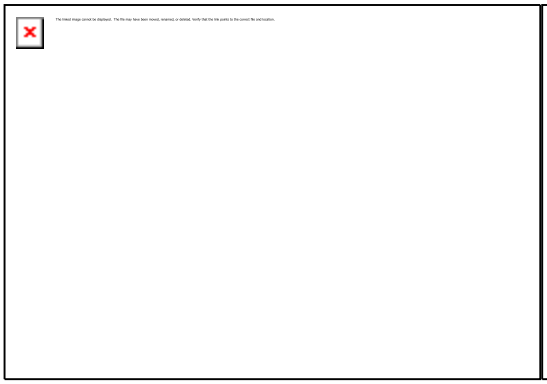
The amount of data transferred for each point in these graphs is the difference between the current and previous server samples at that point, since the server reported values are cumulative totals.

The purpose of streams caching is to reduce the I/O demands on the file system and consequently enable a larger number of users to be supported. This section and the next two sections therefore look at the efficiency of the prototype's file system.

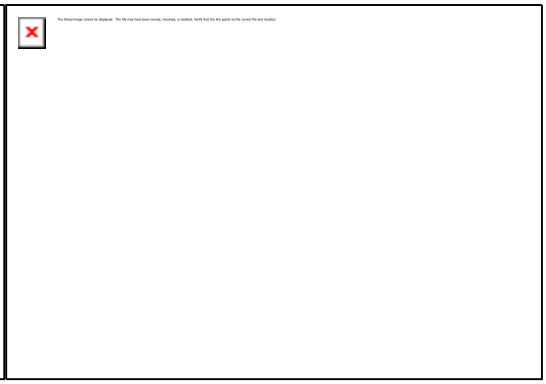
Examination of the effective disk transfer-rate graphs reveals the maximum transfer rate for the disks used within the prototype. Most of the graphs show distinct maxima of approximately 15Mbytes/sample, for example in graph 5.28. Given that each sample represents 10 seconds of activity, this represents an average sustainable transfer rate of 1.5Mbytes/s. The prototype's disks' maximum sustainable transfer rate is 2.2Mbytes/s under best case conditions. Thus, the actual raw transfer rate achieved by the file system with its First Come First Served servicing order is reasonably efficient.

For the five-user set traces, the disks are not saturated by any cache size and therefore excellent server performance is achieved. In this case, even where cache reuse cannot occur between the users, the combination of disk bandwidth and play-out buffer can satisfy all active users. For the 10 users of graph 5.27 however, the smallest cache size can be seen to saturate the file system during the peak of user activity. This peak corresponds precisely with a drop in cache success to failure ratio and frame rate for the

11Mbyte 10 user-set graph traces. Taking the maximum disk throughput as 1.5Mbytes/s with 1.4Mb/s (0.175Mbytes/s) streams, at most eight users could be satisfied with play-out buffers alone.



Graph 5.27: Relative Disk Data Transfer over time (10 users)



Graph 5.28: Relative Disk Data Transfer over time (20 users)

For smaller cache sizes with large numbers of users, the file system is 100% busy for almost all sample intervals. In these cases, the disk cache is failing to satisfy a majority of transmit requests and consequently is placing a heavy load on the file system. Notice however, for the largest cache size that the level of disk I/O is almost consistently below the point of saturation. This suggests, in this data set, that the cache is being efficient at satisfying data transmit requests. Comparison with the corresponding reuse efficiency graph confirms this to be the case.

Were the transfer rate ceiling not enough, faster disks could be installed, which would allow a greater amount of data to be routed into the cache per unit time. This would also enable more than 20 users to be supported, which in these emulations, appears to be the maximum number of users that could be supported; the 25 user trace does show periods of saturation of the file system interface.

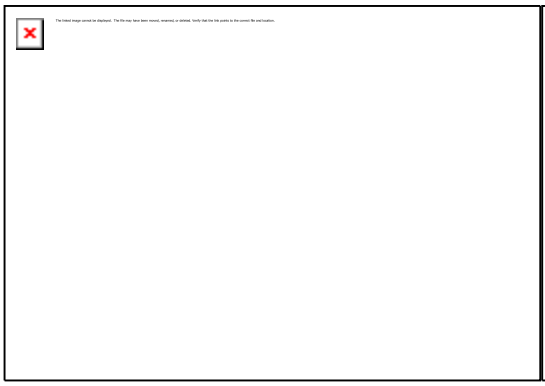
5.3.10 Time Used in Waiting for Disk Transfer

Similarly to the amount of bytes transferred from the file system, the time spent in disk transfer is the difference between the two most recent server samples at that point.

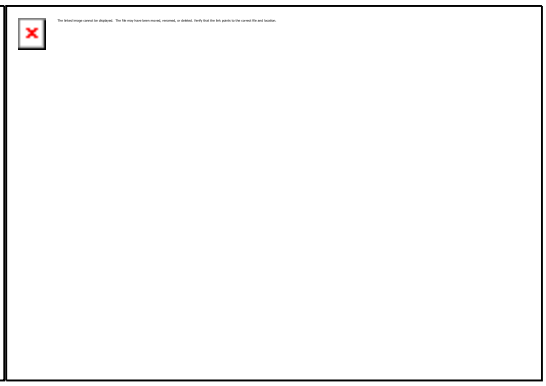
The time spent in disk I/O transfer shows a similar tale to those of the amounts of data transferred. Again, an obvious ceiling of almost 10 seconds can be seen, which corresponds with the sample interval time. Those emulation traces that follow this ceiling over a series of samples are saturating the file system with I/O requests, namely those that have too many users for too little cache space. The ceiling therefore corresponds with a 100% disk busy time.

As before, the five-user traces are all consistently below the 10 second ceiling and never saturate the file system with transfer requests. Similarly, the 10-user trace of graph 5.29 shows some saturation for the smallest cache size. Examining the 20-user set largest

cache trace in graph 5.30 however, a period of I/O saturation which is *after* the peak of user I/O activity is visible. This is most likely due to the divergence of the streams and therefore an increased number of stream groups. This will inevitably cause higher disk activity, as the cache will not be as efficient under these divergent conditions. Indeed, graph 5.19 shows temporal caching efficiency to be poorest over this period of I/O saturation. For 25 users, a similar pattern emerges, though the file system is saturated in the build up to the busiest part of the emulation. However, the largest cache size trace is not permanently saturating the file system with I/O. The graphs would suggest that the larger the cache size the less time is spent in disk I/O for that cache size. This is to be expected, since a larger cache size will produce, and is shown to produce, a better cache success to failure ratio and therefore a reduced requirement of the physical disk.



Graph 5.29: Relative Disk Access Time over time (10 users)



Graph 5.30: Relative Disk Access Time over time (20 users)

A comparison between the amount of data transferred per sample and time spent in I/O per sample for each user-set trace shows a remarkable resemblance in shape. This relationship is not surprising however, since the amount of time spent in I/O will be directly proportional to the amount transferred. The traces for the largest cache size of even 20 users is also remarkable however, when one considers that 20 1.4Mbps streams is a total of 3.5Mbytes/s – this is far beyond the capability of the raw disks. Hence, for this large number of users, the larger cache size does appear to enable excellent performance in the provision of the continuous media.

It is interesting to note that the cache efficiency graphs follow a trace that is almost the exact inverse of the disk-time used graphs. For lower cache sizes, the data access traces tend to saturate, thus not following this observation. This behaviour confirms that the demand for data that the cache cannot meet must be met by the file system.

5.3.11 Number of Disk Hits

The number of hits on the file system, similarly to the time spent in and amount of bytes transferred in I/O, is the difference between the two most recent samples. Each hit on the file system however, is a single prefetch request that needs to access the file system.

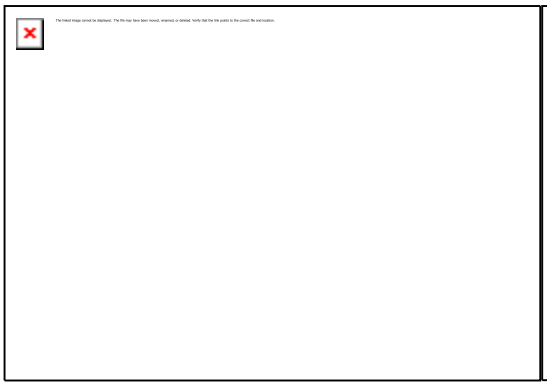
Such a request may be split into several shorter disk transfers where the cache already contained some of the data over the requested interval. In this case, the number of hits to the disk is still counted as one for this experiment.

Generally, the number of disk hits rises and falls with the number of stream groups, such as in graph 5.31. However, for the large user-sets of smaller cache sizes, such as in graph 5.32, the number of transfers reaches a ceiling, which is around 80 accesses per sample, or 8 per second. The larger cache sizes are likely to have fewer disk accesses, since the prefetch requests of the streams will be more frequently satisfied by the cache. In particular, the resemblance of the disk hits traces with the number of groups traces should be expected, and suggests that the cache algorithm is successfully reducing the disk requirements to the leading stream of each group. Each disk hits trace does not match up with the corresponding groups trace where the disk is saturated with requests. In this case, the disk hits trace follows the graph ceiling until the number of groups drops sufficiently. The threshold at which this occurs is eight groups for the largest cache size. For the smaller cache sizes, the threshold is much lower and suggests that the disk is being consumed by streams that are not group leaders. This itself suggests that the cache is not large enough for stable state conditions to arise within the prototype's cache; this would cause following streams to access the disk and break the relationship between the number of groups and number of hits to the disk. This hypothesis is supported by the temporal efficiency graphs, which all indicate significantly lower cache performance under such conditions.

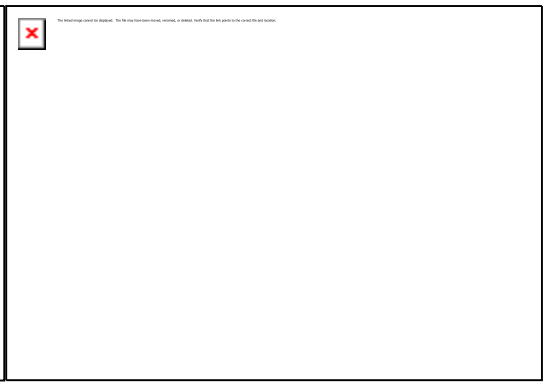
The ceiling of around eight accesses per second should correspond with the maximum transfer rate of 1.5Mbytes/s. In fact, dividing the amount of data transferred by the number of disk hits for any sample gives 0.175Mbytes/s (1.4Mbps), or less. That is, the average amount of data transferred per request. Where the cache is not of any benefit, each stream will access the disk and will require a full 1.4Mbps bandwidth. Taking the 1.5Mbytes/s maximum disk throughput, this would allow 8 streams per second, or 80 1 second prefetches over the sample period of the experiments. This is indeed the ceiling observed, e.g. figure 5.32 at 800 seconds. Where the trace goes above this ceiling, the size of the disk transfers is smaller than at the ceiling, thus allowing more of them.

Each stream handler makes two prefetch calls per second. The amount of data fetched from the disk will generally be half a second ahead. Thus, one would expect two accesses per stream per second. However, where the disk is having to service more users than it has bandwidth for, the service time for each stream will cause its play-out buffer to empty. In this case, each prefetch serviced will be the full 1 second amount because the earlier prefetch was not serviced. This explains why only 8 prefetches are serviced at the 'ceiling' of the graph. For all traces reaching this ceiling however, it is first breached, and the trace settles back down at the ceiling. This crossover effect is happening as more of

the active streams are running out of play-out buffer before the disk next services their prefetch requests. That is, as more of the streams request a full second of prefetch instead of half a second. Under such conditions, the frame rate of the client streams is likely to be 25 for part of the time, and zero for the remainder.



Graph 5.31: Relative Disk Hits over time (10 users)



Graph 5.32: Relative Disk Hits over time (20 users)

When the cache is not providing any benefit from re-use, or the file system is otherwise becoming overloaded, the cache space may better be put to use by increasing the amount of prefetch per stream. By increasing the size of the transfers, file system overheads would be reduced and the effective transfer rate improved towards the maximum rate of the disks. The approach would however, tie the monitoring activity to the prefetch mechanism as a trigger, which is not presently possible. However, these results do suggest that a cache reactive to monitoring output could improve the server performance under overloaded file system conditions.

5.4 Discussion of Experiments

5.4.1 Relationship Between Cache Performance and Disk Performance

When comparing the file system graphs to cache success ratio and effective frame rate, there is a clear correspondence between frame dropout and saturated disk I/O. During such periods, the number of requests serviced by the file system tends to exactly that number of 1-second prefetches of which it is capable. This indicates that a gap in some of the play-out buffers has appeared, possibly following a connection to an uncached clip, or a reposition to an uncached area of a clip. Otherwise, it is because the file system does not have enough bandwidth for the number of users it is trying to support; stable state conditions cannot be reached within the assigned groups. Where the cache size is not large enough for the number of active users, a less significant reuse is possible. Consequently, the cache fails regularly and more frequently saturates the file system.

For the 20-user largest cache trace, an area of increasing disk activity and worsening cache performance can be seen, despite a dropping number of active users. Examination

of the group trace reveals however, that the number of groups during the user peak is relatively low, and for a period thereafter that the group numbers increase with a dropping user population. This behaviour underlines the direct relationship between the number of groups and the consequent cache reuse efficiency and amount of disk activity. Indeed, after the 1400 second peak, the group numbers decrease and the disk activity drops below the point of saturation. The grouping technique is therefore shown to be a successful approach in satisfying data requests that would otherwise fall to the file system. A major finding of these experiments is that the disk activity follows the number of groups that can be formed, not the number of users. Hence, the fewer the number of groups that can be formed, the smaller the disk bandwidth requirements and the greater the cache efficiency.

As noted above, the number of groups will not necessarily be directly proportional to the number of users. The emulations have divergent user behaviour; group numbers are low initially, even despite a high number of active users. However, later divergence causes an increase in the group numbers. This in turn has a knock-on effect on the file system, as each group's leader will be making continuous requests on the file system; this was demonstrated by the disk performance traces. In practice, the similarity between users is, perhaps, likely to be considerable and less divergent than the emulations, since users tend to affect one another during the progress of a learning activity.

The cache success to failure ratio was seen to diminish where the file system was unable to satisfy the transfer requests made of it. Clearly, a less than 100% cache success rate for data transmissions will have a direct effect on outgoing frame rate. A direct relationship between saturated disk I/O and cache failure was demonstrated.

5.4.2 Prefetch and Cache Efficiency

For the largest cache sizes, the data prefetch mechanism was found satisfactory to produce good cache reuse efficiency. However, this somewhat simplistic scheduler proved ill effective for the smallest cache size experiments. This suggests a need for a more reactive scheduler to minimise prefetch costs in terms of cache space. The experiments show that good cache efficiency leads to a better cache success-rate generally. Hence, it is better to sacrifice disk I/O throughput in order to improve caching efficiency provided that no stream is starved during the scheduler cycle. In reducing prefetch spatial requirements, more space is available for interval (or residual) caching and thus the potential for reuse is increased.

The present disk scheduler attempts to maintain a second of data in prefetch for each stream by requesting a second of data ahead of the stream for each half second used. Thus, only half a second of data is fetched from the disk; under ideal conditions, all streams have at least half a second in prefetch present in the cache at all times. As the server becomes busier, the amount of prefetch remaining when a stream is serviced is

diminished towards system overload, where it is exhausted prior to servicing. Since the scheduler attempts to maintain a full second of prefetch for each stream, the less prefetch remaining at the instance of servicing, the more data is fetched from the disk. The use of a fixed large prefetch for each stream is a poor approach, as less cache space is then available for interval caching, which itself has a direct impact on cache efficiency. In addition, the usage efficiency of a large play-out buffer beyond the optimum for disk scheduling is worse the larger the buffer.

Aside from the prefetch issue, the experiments showed that a non-saturated cache, that is, a cache that can satisfy all intervals, achieves better than expected performance according to the observed rule $eff_t = 1 - (1 \div meanGroupSize)$. This latter rule can be used as an effectiveness measure for an implementation of stream caching, and indicates that the performance could be markedly improved for smaller cache sizes. Better than expected performance is enabled through the unpredicted reuse of residual cache data.

5.4.3 Use of Monitoring Feedback for Disk Scheduling

In the results of the disk hits section, it was suggested that the output of monitoring could be tied to the policy used by the prefetch mechanism. Where a poor cache success ratio is detected, together with a heavy file system loading and emptying play-out buffers, the prefetch of each stream could be lengthened to improve the effective data throughput into the cache towards the theoretical maximum. The present process responsibilities in the prototype however, would need substantial rearrangement to enable such a strategy.

The current scheduler does increase the amount of prefetch per stream as the server becomes busier, though this behaviour is incidental and not intentional in the original design. Consequently, the scheduler is not sensitive to how much prefetch each stream has or will have remaining in prefetch at the point of servicing. However, it does suggest that the strategy of increasing the amount of prefetch per stream may succeed. This observation is important since the strategy carries with it a risk. Namely, using more space for prefetch leads to less space for interval caching, which can lead to more demand on the disks, which can lead to greater prefetch requirements, and so on. Such a “run away” effect is clearly to be avoided, though it is difficult to predict if this would occur in practice without attempting to implement such a strategy.

The current scheduler fetches a maximum of one second ahead for each stream, which for MPEG-1 without audio, is approximately 170Kbytes each fetch. To achieve the maximum throughput rate of the disks, this unit fetch size is not enough. According to the ceiling observed in the data transferred graphs, the prototype’s 2.2Mbyte/s disks achieved only 1.5Mbytes/s in practice – 256Kbytes per fetch or greater is required. In increasing the transfer rate into the cache, stable state conditions within the groups might be restored. Once achieved, prefetch may then be reduced and more space made available for interval caching. For a small cache, it is more difficult to effectively utilise

stream caching. In this case, it is perhaps better to sacrifice more cache space for prefetch in order to increase disk throughput and devolve the disk scheduler to a standard continuous media scheduler without cache reuse.

Regardless, using the monitor output could improve server performance as the size of prefetch could then be appropriately minimised for the prevailing conditions, thus giving the maximum amount of cache space for interval caching.

5.4.4 Achieving Maximisation of Group-Size

In the results traces, a good average group-size could only be achieved where there were a large number of users in an experiment. This proves, at least for these experiments, that a reasonable user population is required before stream optimised caching becomes an efficient technique by which to provide continuous media. It was also found that for very few users, less significant reuse took place. However, the cache effectiveness for few users is dependent upon the similarity of accesses between those users, *and* on the behaviour of those users. For example, for a cache under light load, and users that are frequently backtracking and reviewing different clips from each other, the cache will return better success to failure ratio. In this scenario, the lightly loaded cache could maintain data not predicted for reuse (residual data), which would be reused by the reviewing users. This phenomenon was demonstrated in the traces for cache efficiency against mean group size.

5.4.5 Maximum Server Transmission Rate

The maximum outgoing transmission rate of the prototype was reached with the 25-user set traces, the rate being ‘clipped’ to the maximum of the CPU-bound ATM board. However, the disk performance into the cache does suggest that the system might support more than 20 users, though perhaps only with faster disks. The present disks are 2100-rpm with a maximum sustained transfer rate of 2.2Mbytes/s. The prototype’s file system and transfer mechanisms achieved a maximum of 68% disk efficiency. The experiments suggest that the prototype is able to support at least five users without any significant reuse or data cache. However, the aggregate bandwidth of this few users is within the achieved performance of the file system. This is not true of 10 or more users, where the performance of the over all system is dependent upon the success of the cache.

5.4.6 Frame Timing and Average Transmission Latency

The frame transmission and control latency of the server was found to be very low at around 400μs, with little deviation. Peaks of 1ms latency during points of frequent group reassessment were found with the 10 second sample interval experiments. Further investigation suggested an upper bound on transmit latency of around 4ms. Given however, that the server generates each transmission time from the manager TRAM’s clock relative to the clip’s calculated first frame time, only a slight deviation in

transmission timing will be experienced; no drifting of the inter-frame time will occur. That is, the server transmit timing will not slowly drift out of phase with the ideal frame transmit times, although a per-frame deviation from this ideal may occur. For example, a 25fps MPEG clip should transmit its frames at times, relative to the first frame, that are divisible by 40ms. However, if the server drifted out-of-phase with this ideal event timeline, the decoder buffers would either underflow or overflow. Given further that a client end-system will buffer at least a single frame (usually 33 or 40 ms), the latency to transmission (including the delay to the data leaving the server) are sufficiently low so as not to produce jitter. Additionally, the number of users on the server does not significantly affect the transmit latency, except where the ATM adapter becomes highly overloaded.

5.4.7 Conclusion

At the outset of the project to build the prototype server, a target of 20 users was aimed for in the provision of 1.5Mbps continuous media. The figure of 1.5Mbps was derived from the width of 'text book' MPEG streams, though in fact the media files used within the experiments had no audio track. Consequently, the clips used were 1.4Mbps. However, this has proved important in the experiments, with the targeted population of 20 users lightly touching on the ATM interface limitation of 28Mbps. This, of course, corresponds to the total aggregate bandwidth of 20 1.4Mbps streams. In conclusion then, the original aims of the SERVICE project have been reached. Further considering that the theoretical maximum transfer rate of the raw disks is around 18Mbps, reaching an outgoing bandwidth of 28Mbps gives another indicator of the success of stream optimised caching.

5.5 Deployment of SERVICE – success of implementation

As noted in section 5.1, the prototype server was used by the CIMIS project to provide real-time interactive video for a tourist information system. On two occasions, the CIMIS prototype was deployed for five working days in Royal Victoria Place Shopping Centre, Tunbridge Wells, in Kent. System reliability was of utmost importance and much effort was expended in ensuring that the video server was bug free and gave the necessary stability. In practice, the video server failed exactly once during the two trials. The author strongly believes that the system is now highly reliable with the last reliability problems having now been resolved. It is furthermore very likely that the prototype video server will be employed within future projects at the University of Kent involving continuous media data-types. Modifications and additions to the server itself are also possible, though the reader is referred to chapter 6, which discusses the possible future for the server and its underlying technology. In terms of results as considered in this chapter

however, the usage of the prototype has been as more than only for taking experimental measurements, and is robust and very reliable.

A discovery of the CIMIS project trials was that users do not expect an immediate response from a VCR control request. It was found that a delay of between one and two seconds was tolerable. In this case, the delay was caused by a large pipeline within the PC end system MPEG hardware. However, this delay to response may be put to use within the server to further optimise stream grouping organisation. Further discussion of this phenomenon can be found in chapter 6.

One of the first student projects in the department to use the video server was the multimedia ropes player of John Salter [JS97]. The ropes editor allows segments of various media clips to be embedded into the logical timeline of the rope. An example rope is given in figure 5.1, where the rope contains two clips each of video and audio. A user playing such a rope views the rope as a single multimedia clip, the playback application handling the interfacing to the underlying media server.

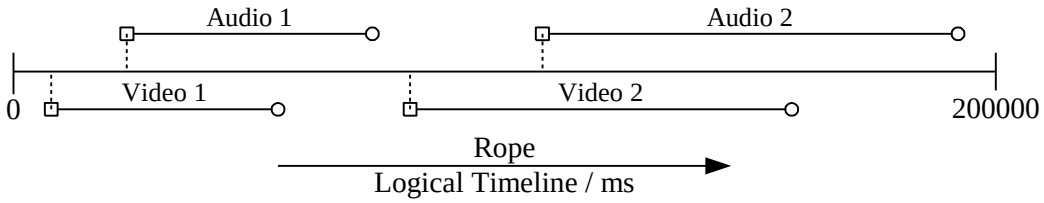


Figure 5.1: Example of a multimedia rope with two video and two audio clips embedded into the logical timeline of the rope

The ropes project was the first application to play audio clips from the prototype server simultaneously with video into the same client PC. Salter's analysis of the PC client display suggested acceptable performance, though suffered the same MPEG hardware/driver decode delays as the CIMIS project. Nevertheless, the prototype server was a necessary part in the success of this 'client' project. Other student projects are known to be ongoing and others under consideration within the Distributed Systems research group.

Salter's project required the construction of a CORBA protocol proxy with the server, as the PC was using COBRA as part of the ropes project. The CORBA proxy enabled the PC to communicate with the server; PCs do not support Sun Microsystems' RPC protocol. The COBRA proxy was ported by Salter from a previous proxy by Ian Buckner, which used the ANSAware communications library. Buckner's ANSA proxy was the first to enable the use of the prototype server within a distributed system.

Practical experience with the server within CIMIS and with other Motion JPEG and uncompressed video clients suggests that there is no discernible control lag to the user. That is, for example, that the client display changes immediately after a user reposition

and even despite an intermediate protocol proxy. Additionally, the viewed material does not suffer from jitter or dropout. This supports the conclusion that server latency is both consistent and low. At the time of writing, the largest real-user system loading has been two motion JPEG clients (8Mbps per stream) and one MPEG client. System response remains crisp under these conditions.

Observed output quality during execution of the experiments reinforces the above conclusion. During the largest cache size, 20 users experiment, the observed output was as when only one user was using the server. However, as predicted by the experiments in this chapter, once the server begins to breakdown, the output of the streams is periodical. That is, for each one second of video played, a one second 'gap' follows it. This was explained as an artefact of the disk scheduling policy.

As noted above, the server has successfully been used with varying media formats: motion JPEG, 256-greyscale raw video, 256-colour palette raw video, RIFF (wav) audio, and MPEG. This proves the success of the format-neutral container-format designed for the video server.

5.6 Summary

The chapter began with a description of how user emulation can be performed by a pattern generator, and how its output can be altered. It then described the scientific approach to how the stream optimised caching architecture was assessed. The details of how data was recorded during the experiments and how this data relates to the graphs presented were covered. Several configurations of user loading and cache size were used.

The results of the experiments were presented with a discussion of each aspect of the data taken. This concluded with a discussion of all those aspects, which was able to consider the system as a whole. The stream optimised caching technique was found to be a valuable approach to the provision of continuous media. In particular, it was shown that the throughput demands of the file system are directly proportional to the number of stream groups that can be formed. Additionally, it was shown that stream optimised caching efficiency follows the rule $eff_t = 1 - (1 \div meanGroupSize)$. It was suggested that the performance of the caching could be improved by minimising prefetch spatial cost under normal conditions and using the monitoring output to determine the data prefetch policy. The effectiveness of streams caching was found better for larger numbers of users. Transmission latency was found to be low with little spread, a conclusion that is supported by the observed jitter-free output. Server performance was found to be restricted by hardware limitations, however the goals of the SERVICE development project were reached.

The chapter ended with a justification of why the prototype server is more than a simple experimentation box, with a brief discussion of other areas of work in which the server has been involved.

6. Future Directions for Research and Development

"Nothing endures but change" — Proverb

With a limited project lifetime, only the most basic of system implementations has been achieved. Nonetheless, the ideas behind stream caching have, it is hoped, been proved both correct and appropriate for the intended environment. There is however, still much potential to be tapped from the prototype itself and the ideas on which it is based. This chapter considers these many areas of potential, first by exploring aspects of the prototype, and later by discussing options for server implementation using alternative technologies and system architectures.

6.1 Improving the Prototype Server

As it stands, the existing prototype server does allow for further experimentation with different caching algorithms, and has the advantage of the already existing telemetry and processing tools for analysis. However, the prototype server does have some physical limitations, most notably the 28Mbps bottleneck onto the ATM network; the cache was seen in the results to saturate the current ATM hardware. The first half of this chapter discusses the many possible optimisations to the prototype's software and hardware components.

6.1.1 The ATM Network Interface TRAM

Perhaps the most necessary and notable area for improvement in the prototype is the ATM network interface. By improving the other components in the system, although the quality of service provided would be improved, the maximum number of users that the server could support would be unchanged. Hence, the discussion for improvements begins here.

6.1.1.1 Reasons for CPU Limitation

The prototype's link transfer engine is responsible for data transfers from the SCSI to cache TRAMs, cache to ATM TRAM, and vice versa. It was optimised early on in the project to check that the necessary aggregate bandwidth could be achieved. For the ATM interface however, in addition to the simultaneous link transfers, the CPU is responsible for the PDU segmentation into cells and cell set-up. A consequence of this arrangement is that data transmission soon becomes CPU bound. An analysis of the ATM TRAM revealed that the MOVE2D segmentation and reassembly instructions were simply taking too long. An optimisation of the process models, their data exchanges and some algorithms found more CPU time and hence allowed more cells to be formatted for

transmission. The segmentation itself however, could not be optimised beyond the single MOVE2D instruction. Thus, despite the achieved parallel link transfer rate of over 40Mbps, CPU overheads within the ATM TRAM prevented this transfer rate being achieved in practice. In hindsight, the TRAM needed DMA segmentation/reassembly hardware assists to relieve the CPU of the task, and allow it to do other useful work.

6.1.1.2 Eliminating the Segmentation/Reassembly CPU Overhead

A possible alternative method to improve the ATM hardware's throughput without modification to the board may be to redesign the Xilinx firmware. The existing firmware co-locates the cell header with the data to be transmitted. However, if the cell headers could be specified separately from the data then segmentation/reassembly by the host CPU would then no longer be necessary.

One implementation may be to utilise an additional register for each of the transmit and receive Xilinx chips; the register specifies the location of a cell header data array. In the transmitter's case, the firmware would then take the cell headers from the register specified location; the receiver transfers received data headers into the specified location. As with the current implementation, transmit and receive would be initiated by reading from the shadow memory location. In this case, the shadow address read would correspond with the location of the actual data, not the cell header/data pair. Such an implementation therefore replaces the MOVE2D instruction with a write into the cell-header address register, these operations being per cell and per page (1024 bytes) respectively, regardless of the transfer direction. However, by eliminating the memory copy operation the performance of the board should be significantly improved. The interaction with the cell header address register may be minimised if the firmware auto-incremented the address register for each cell transferred, thus reducing interaction to the 'wrap' of the cell header memory array.

The current firmware implementation is capable of the full 100Mbps of the ATM hardware. However, the four hardware links of the Transputer are each capable of at most 14Mbps, giving a maximum server transmit rate of 56Mbps. Thus, it would be acceptable if the reorganisation of the firmware reduced the maximum performance of the raw interface by up to half, since then the server as a whole would still be faster than the existing prototype. Such a loss of efficiency might result from using separate data arrays for the cell headers and cell data.

The current firmware implementation achieves the maximum board performance by reading or writing 1Kbyte pages from the Video-RAM, that is, 16 cell-header/cell-data pairs. With separate arrays, the number of cells in one page is 21, giving a maximum transfer size of 1008 Bytes, with a residual 16 Bytes forming the start of the next cell's data and the base of the next page. In this case then, VRAM performance should not be significantly impacted with the transfer size being 1008 Bytes instead of the full 1K page.

However, for each 21 cell-data elements transferred, 21 cell-headers are also needed, which is a transfer size of only 336 Bytes. In this case, VRAM performance would be only a third of the current firmware implementation. Nonetheless, considering both cell-data and cell-headers, memory throughput performance would be 65% of the current firmware implementation. Hence, the necessary 50Mbps throughput from the interface should be maintainable.

One solution to maintain the maximum VRAM throughput performance may be to allow 63 cell-headers to be transferred ahead of each three 21 cell-data elements. However, this is only possible if sufficient buffer memory exists within the physical ATM hardware and firmware. Furthermore, this approach requires that at least 63 cells be queued for transmit or receive. However, this prerequisite is reasonable from a practical standpoint, though only in the transmitter case. For the receiver side, data receipt per cell must be possible, so the existing solution is perhaps optimal here. However, an asymmetrical solution to transmit/receive is not ruled out.

A second implementation may be to partially interleave the cell header and cell data. That is, for each 1Kbyte page, there are 16 cell headers and 16 48-Byte data pairs. However, the header/data pairs are not interleaved; the 16 headers are contiguous in memory and are followed by 768 Bytes of contiguous data corresponding to those headers. Thus, in this arrangement, the VRAM performance is maintained and therefore the full 100Mbps adapter throughput.

To avoid the MOVE2D instruction requires that the data to transfer place 768 Bytes of data in each 1024 Byte block. This may be achieved by modifying the link transfer engine to transfer in 768 Byte maximum units instead of the current 4096. The link transfer size will therefore vary from 16 to 768 Bytes because of cache block misalignment, which will increase CPU scheduler overheads. However, the technique should be faster than the current implementation with MOVE2D – the prototype already has such an ‘auto-segmentation’ link-engine placing each 48 bytes of link data into a 64-byte block. The approach was no faster than using 4096 Byte transfers together with the MOVE2D instruction, and was consequently disabled in the code. Therefore, in using a larger link transfer size, i.e. 16 to 768 Bytes, overall server performance should be increased. Modification of the existing auto-segmenting link-transfer-engine to transfer the data into 768 Byte sized 1K aligned blocks should be trivial and would provide a good indicator for expected performance from this scheme.

In the worst case, were the approach not faster than the current prototype implementation, the MOVE2D instruction could be reinstated. In this case, the segmentation would be to 768 Bytes per 1K page, instead of the current 48 per 64 Bytes. However, this may itself improve server performance because of increased memory throughput from the MOVE2D instruction itself, which is dependent upon the per-line

memory-copy size. In fact, a normal MOVE becomes necessary where less than 768 Bytes need to be segmented or reassembled.

One of the original aims of the ATM board design was to allow the interleaving of several outgoing VCI transmissions and allow traffic shaping within the individual cell streams. For the designs outlined above, this is not easily achievable, with the original design intending to use MOVE2D in order to achieve the outgoing interleave. In practice, VCI-interleaving was never implemented and would likely have resulted in poor aggregate performance. The performance of any scheme is dependent upon maintaining the 1K-page VRAM access size, that is, that the interleave must be achieved by the Transputer prior to submission to the firmware. Therefore, for the above suggestions, it might be possible to use the link transfer size as a means to coarsely interleave the outgoing transmissions into pages. Thus, a single outgoing cell-stream would then consist of bursts of cells at the maximum rate of the interface separated by one or more bursts of cells from other VCI-streams. The interleaving of several outgoing streams at the cell level is a general weakness within the design of the ATM TRAM.

In conclusion, any of the above suggestions should lead to improved throughput in the ATM adapter. An additional advantage to all of them over the existing firmware driver is that the Transputer's buffer requirements are reduced from both PDUs and cell header/data pairs to PDUs and cell headers. Thus, board performance may be further improved because of the larger buffer space available. However, the implementation of any scheme is dependent upon sufficient Xilinx storage space; the current implementation is itself at the limits of the Xilinx. However, it is hoped that a rewrite of the firmware would result in a similarly sized code image. Failing this, a newer larger FPGA might be used, though this would require a change in the hardware design.

6.1.1.3 Limitations of the Link Technology

Ultimately, faster hardware is needed to exploit the potential of the prototype's cache transfer rate. Even if the ATM interface were not a bottleneck, the prototype would then be limited to the rate of the inter-processor links, i.e. 56Mbps at best. The realistic transfer rate, however, could only be increased to little more than 40Mbps, unless all the Transputer links were hard-wired to bypass the C004 link-router. Ideally, the next generation of the server should be able to achieve 125Mbps sustained network throughput. SGS-Thompson offer a 50Mhz T450 part, which may improve throughput on the ATM network TRAM, though it is impossible to say if it will remove the segmentation/reassembly bottleneck.

If the use of Transputers is to continue then, a DSLink based component should be employed, such as in the T9000. The DSLink has a raw data throughput rated at

100Mbps per link, and in practice achieves a transfer rate of 6.6Mbytes/s (52Mbps)[11]. Hence, a potential parallel transfer rate of 200Mbps to the ATM TRAM is possible, resulting in aggregate network bandwidths far in excess of the 100Mbps-capable first generation technology.

The T9000 however, is likely to suffer from insufficient CPU power as did the prototype's T805 based TRAM if segmentation/reassembly remains performed by the MOVE2D instruction. Current T9000 parts are rated at 25MHz maximum, so the Transputer will likely not be quick enough; compare this to the 450MHz Pentium II, or 500MHz DEC Alpha! To add to the problem, SGS-Thompson has withdrawn support for the T9000 series.

6.1.1.4 Replacing the Transputer Board

The next generation of ATM hardware developed at the University of Kent is likely to employ Texas Instrument's TMS320C40 DSP chip, which is ideal for pipelined activities such as data transmission. The new board will also employ a DMA system to perform the segmentation and reassembly, thus relieving the CPU of the task.

The new board originally considered the use of a Transputer, though only for its link transfer engine, effectively rendering it a coprocessor to the DSP. The author believes the selection of a DSLink based Transputer essential, since a new video server based on the prototype would otherwise quickly reach the limits of the 20Mbps older type link. Interfacing with the older links is not a problem, with DSLinks being compatible with those of the T2, T4 and T8 series Transputers. The T9's recent status as 'obsolete' does make this unlikely, however.

The TI C4x series DSP processors have 4 to 6 bi-directional communications ports, which each give an application visible transfer rate of around 20Mbytes/s (80Mbps). The C4x comports therefore render the addition of a Transputer unnecessary, though its inclusion would have provided an upgrade path for the existing file server prototype.

Of course, the practicalities may not match with the theory; a series of experiments should first establish whether a DSLink based Transputer, such as the T9000, could simultaneously engage all four links and achieve the required aggregate bandwidth. This is especially so if the CPU is responsible for such intensive tasks as segmentation or reassembly for all data transferred over the links – such overheads must also be included in the experiments. If these simple tests cannot be satisfied then further development with the CPU under test would be rather pointless. In this case, a radical change in the system architecture is the only realistic solution. In fact, the second-generation adapter will not employ a Transputer part, thus closing this development option.

[11] Source: Michael Poole from the Parallel Systems Group of the University of Kent, formerly of INMOS, Bristol, U.K.

6.1.2 TCP/IP, RPC and Protocols

Most closely related to the raw network interface, is the protocol stack engine to drive communications with other network end systems. The TCP/IP module used in the prototype was an optimisation of an engine written by Andrew Smith for the ANSA-RPC project [LIP95]. This shortened the development time for the server, however the code was a 'single shot' approach that was intended only to provide UDP support; TCP is not available in this implementation. Remote Procedure Call (RPC) support was added atop of the prototype's optimised UDP support. However, similar to the UDP engine, the RPC code needed only to support specific protocols, viz. mount, NFS, ukcStreams, and vfsBench. As such, this support is directly coded and the RPC engine is not easily extendable. For this reason, an external proxy approach was used when needing to communicate with the server via other network protocols, such as ANSA and CORBA.

Thus, the TCP/IP and RPC engines might be considered an area for replacement with an existing or specifically developed code module. Such replacement should enable direct support of other protocols. However, given that the server is a bespoke system, then underlying support of any additional protocols would still need to be written. Given the necessary development effort to add any additional protocol support therefore, modifying the existing prototype engine is likely to be easier than replacement.

6.1.3 The Continuous Media File System

Some additional performance could be gained by further optimisation of the file system from which the cache reads its data. Although the prototype file system was fundamentally designed for continuous real-time media, a number of more obvious file system optimisations remain to be implemented. By increasing the throughput efficiency of the file system, the stream optimised cache will benefit, since it will be better able to cope with periods of poor hit ratio. As was shown in the results, a higher sustainable data throughput rate leads to a better cache success-rate. Consequently, a larger number of users may be supported.

The file-system maintains a queue of transfer requests, though the queue is not optimised for a sequential head scan (Scan scheduler) or any of its derivatives. Thus, the seeking costs of the First Come First Served policy may be high due to consequent random head movement. The high throughput efficiency of the prototype is achieved simply by using a large basic transfer request size of between 32 and 256Kbytes. The implementation of sequential head scanning would not immediately deliver any improvement to data throughput, however. The cache does not submit more than one prefetch request job to the file system at a time, regardless of the number of active streams. Therefore, only simultaneous NFS read traffic could benefit, since writes bypass the cache mechanism.

To benefit from Scan or a derivative, the prefetch servicing mechanism of the cache TRAM would need to be rewritten to allow all such prefetch requests to be submitted to the file system at one time. In doing so however, the issues of real-time servicing are likely to become more important, since I/O request starvation must not occur. In this sense, the server prototype would then share the problems of other continuous media servers, which use the shortest possible playout buffer, equivalent to the prototype's prefetch. The adoption of a variable size and variable length scan-based cycle mechanism for the prefetch requests may however be the route to take. It would produce a scheduler with real-time response to all the active streams and would vary disk throughput into the data cache as required. The stream optimised cache should then be more effective because of the higher disk throughput and increased interval space. This subject is discussed further in section 6.1.7.

Of course, the very easiest way to increase the disk performance is to simply use a faster disk. Exchanging the 2Mbyte/s disks of the prototype system for 10Mbyte/s devices should provide significant improvements in system performance. Such disk performance however, is far quicker than the performance of the ATM network interface. It is also faster than the aggregate throughput of all the Transputers' hardware links. This further implies a need for an improvement in the bandwidth between the file system and cache and between the cache and network interface. Consequently, any experiments of the kind in chapter 5 would not be very meaningful. For the same reason, the more difficult addition of a RAID based file-system would be pointless until the basic throughput issues have been addressed.

6.1.4 Human Computer Interface exploitation

Users of both the CIMIS prototype, and of other systems [SKS97], have been found not to expect an immediate response to a control request. A wait time of up to around two seconds can be effectively used within the server to optimise the organisation of the stream groups. Furthermore, users' requests for repositions are generally approximate; a user would not know the difference if the server played a section 5 seconds earlier, or later, than the time requested. This 'time shift' phenomenon could also be used to advantage in optimising stream groups. The reorganisation would be simply to increase the size of groups by 'piggy-backing' two or more streams, as described by Aggarwal et al [AWY96]. In this case, the interval across a group is zero, and it might be argued that the stream optimised cache is therefore not required. However, the HCI optimisation can only be applied to a point – a user will notice an optimisation if it is too harsh. However, as Aggarwal says, a slight increase in the trailing stream's, or decrease in the leading stream's rate of play-out is required in order to allow piggy-backing to occur. In this case, the stream optimised cache plays an important role since it allows the catch-up to occur without having any impact on the disks once the stream interval allows caching.

This will itself enable a greater number of simultaneous users to be supported, especially where VCR requests are the exception and not the rule.

To minimise the response time seen by a user for each interaction with the server, the passive/active accumulation scheme suggested by [KS97] might additionally be employed. In this case, the service to the user could continue uninterrupted over the period of the transition. The current prototype server honours reposition primitives immediately, so consequently causing cache failure until the stream's new prefetch request is serviced by the file-system unless the data is already cached. Therefore, with passive/active accumulation, the server would achieve a 100% cache success rate under ideal conditions whilst additionally servicing user control primitives.

Of course, the above approaches are only appropriate when the server's clients are real users. For 'robot' users such as the emulation scripts or multimedia ropes, or for editing applications, a precise and/or timely response from the server may be a requirement. In this case, the server may possibly be given a parameter during connection to indicate that such temporal optimisations must not be performed by the server.

Broadcasters might influence client behaviour by advertising, for example, a new film as 20% price reduced, if the movie is watched within a specified period. The majority of users are therefore likely to watch this movie at the prescribed time, and again, the probability of re-use and size of groups are much increased. A further technique to improve group density might be to insert an additional advert, such that a leading group client is delayed and the resulting group density increased. However, this latter approach may be a rather contentious issue, and would not be possible in an 'advert-free' system in any case.

6.1.5 Predicted File Usage Optimisation

For each class, the clips viewed are prescribed and in a known order. If the server were given 'hint scripts', or some expected access programme, the server may speculatively cache sections of media files, even though they have no users. In so doing, the user throughput can be increased by decreasing the start-up latency of each new clip session. This technique is employed by Wang et al [WCLW97] in their use of a memory buffer as a means to improve the level of service provided by a Video-on-Demand server – the generalised relay mechanism. For Wang's system, the clients are karaoke machines, with expected clip usage patterns. The server contains 5000 3-minute long MPEG-1 clips, around 100 of these on 4Gb of disk, and the remainder on a tape jukebox. The server memory buffer sizes examined range up to almost 1Gb. Thus, the scale of Wang's model is much larger than the developed prototype described in this thesis. Wang's simulation results show a large improvement upon their previous system [WCL97], which did not take advantage of expected clip usage, and was similar to pure streams caching. These

results do however, suggest that streams caching does scale up to a much larger number of users, and that it would perform far better than a least recently used approach.

Of some benefit to the caching efficiency would be the implementation of hotspot reorganisation, that is, placing the files used by a class on the fastest areas of the disks, and in order of their temporal access as the class proceeds. Such an optimisation would decrease seeking costs and therefore improve data throughput to the data cache, hence benefiting the cache efficiency as a whole. The same hint scripts used by speculative caching could also be used here to perform hotspot reorganisation on the disks.

6.1.6 Continuous Media File Format

In terms of the Continuous Media File Format, the contained data is always encoded as AAL5. If an ATM interface were used that were capable of generating the AAL5 trailer, including the computationally expensive CRC while maintaining the necessary network throughput, then the file format may allow for either pure raw data frames or AAL5 encapsulated data. As processor technology advances, such a trade off of space for processor-time optimisation is inevitable since it would allow more media to be stored on a disk. A further advantage of increasing the information density on the storage device is an improved effective data throughput as a result of the extra compression.

As an intermediate step however, the server's file format supports a more compact block-sharing version (section 4.5.2 option Q). Here, each file block can contain the data from one or more consecutive frames. Thus, the end of each frame does not waste an average of half a file block, since the remainder of the block is occupied by the next one or more frames. The number of blocks making up each frame is consequently increased by at most one. This may have a detrimental effect on the maximum throughput rate of the ATM interface because of the increased complexity. This was the reason the more compact version of the file format was not used. Nevertheless, the option still exists, and may benefit cache performance because of the increased effective disk data-throughput.

At the very least, the generator utility to convert media into the container format might be modified to create a 'shadow' compact file-format version (that is, without writing it to the disk). This would allow the utility to generate statistics on the non-block sharing and block sharing file format versions, thus allowing a simple comparison of the approaches in terms of file sizes and storage efficiency. In addition, the statistics on data storage efficiency are alone of great interest since the server performance is tightly linked to the effective data transfer performance of the file system.

A further part of the design of the file format was the use of a page table in the index blocks. The prototype locks down all of the index blocks of a media file, and does not attempt to page them. The ability to page out parts of the index becomes important when dealing with very large media files – particularly where the server does not have an index block space large enough to hold the entire file index. The prototype could therefore be

used to develop an appropriate algorithm for memory management in index blocks, and further to analyse the impact of paging these blocks on server performance. Such a development would allow a slightly larger main cache size, since the memory for index blocks and that for data is currently partitioned, not shared. Thus, if an extremely large file is loaded, the prototype may report that it does not have enough memory for the index blocks, despite having an empty main cache. Ideally, the main cache should be the maximum possible for the main cache TRAMs, with an additional memory space for the indices on the manager TRAM. There are obvious performance implications for the server if data blocks must be discarded to make way for index blocks, hence the partition decision in the prototype.

6.1.7 Approaches to Streams Caching

A more efficient caching algorithm would enable the server to satisfy more users from the cached data, and would therefore reduce I/O demands on the file-system. For the same effective transfer rate file-system then, more users could be supported by the server. Of the possible cache algorithms to examine, the most obvious is an implementation of modulated stream caching, while building in the ability to switch between modulated and non-modulated modes in order to allow a practical comparison between the two related algorithms. As mentioned in the section above, the effects of the compact file format may also be examined for both caching approaches.

6.1.7.1 Active Discard of Data Predicted to Have No Re-Use

Beyond modulated stream optimisation, an implementation of active discard may be a development option. That is, any data not predicted for reuse is marked for discard in an attempt to increase the accuracy of the physical cache content against the logical model. Such a scheme may be computationally expensive, which is largely the reason it was not attempted as part of the original development project; Transputers are not computationally powerful in comparison to current microprocessor technology. Nonetheless, the prototype does provide the necessary system support to enable practical experimentation with different algorithms.

After a frame has been transmitted for which there is no predicted successor, the data might be placed into a separate structure, though not freed. In this case, when looking for a block in the cache, both the main and marked structures would need to be searched. The marked-list might be a perpendicular structure however, so a search for a block in the cache would require a search of only the main cache structure. The marked-list could be sorted in a FIFO order, such that the data first marked for re-use is the first to be re-used. However, with a modulated interval active, which data is re-used becomes academic, since the modulated interval will consume all free-space *and* marked blocks. Where there

is no such interval, all stream intervals will be satisfied and the potential for re-use cannot be predicted. In this case, an oldest first re-use policy would be sufficient.

With a marked-list, when an allocation is made, the free-list is used first, and if exhausted, then the marked-list is used. However, if allocations are taken from the marked-list, those blocks must be unlinked from their current file chains. Once removed, they are then re-linked into their new owner file. Thus, allocation becomes a more complicated and expensive operation. However, the alternative is frequent group reassessment, so the extra cost of allocation may trade off against reassessment costs by making reassessment less frequent. A further complication to the scheme comes from the deferral of cache data access by the ATM adapter until the ATM TRAM can deal with the transmission request. Until the data has been successfully transmitted, it cannot be reallocated to other data blocks or another file chain. Synchronisation mechanisms therefore make this scheme more expensive than the existing implementation. However, with careful design to avoid heavy CPU loading peaks and balance processing load over time, a successful implementation is feasible.

A further possibility with the marked list is to discard from the data cache based upon the probability of data re-use, rather than on a FIFO basis. In so doing, a popular clip is more likely to remain in the cache whilst it has no clients. Such an approach would require a popularity distribution for the files on the server and is similar to the 'generalised relay mechanism' of Wen et al. [WCL97]. However, whereas Wen use the distribution to pull data into the cache that is predicted to be used, the scheme suggested here merely leaves data in the cache if it is likely to be reused without having immediate re-use. Such probability distributions may use both day-of-week and time-of-day as parameters in determining clip popularity at any instance.

6.1.7.2 Optimisation of Prefetch

Prefetch is allocated to each client stream, which may be absorbed into a stream group during the assessment process. The amount of prefetch allocated is based on an index look-up a small period into the future – the amount reserved is the amount of data to be consumed in this time. However, a better cache space usage could be achieved by improving the management of prefetch allocation.

All of the Video-on-Demand techniques covered in Chapter 2 try to minimise their equivalent of prefetch. For each stream, only enough prefetch is allocated such that the buffer runs empty just as the stream is next serviced by the disk. In the prototype server, the next prefetch request is made when half of the look-ahead window has been consumed. Consequently, there will generally always be some data still unused when data is next pulled from the disk. As noted by Ng and Yang [NY94], this results in poor utilisation of the memory buffer because space is occupied and not used during a cycle; this space could have been allocated elsewhere. Ideally, only enough prefetch should be

allocated such that this prefetch is running dry just as the stream is serviced by the file system. This would require a change of process responsibility, since the stream handler processes currently request data from the file system. To minimise the amount of cache space used by prefetching, the stream management process would need to additionally manage the file system demands of its stream handler clients. In so doing, the order of servicing of requests could be optimised using some variation of the Scan scheduler.

A further advantage to such a tuning of the prefetch for each stream is that very large fetches are possible in order to extract the maximum transfer rate of the file system. During user sessions with little reuse, the number of users that can be supported will therefore be higher than with the prototype. Care must be taken, however, since greater demands for prefetch space mean less space for interval caching. Hence, a user session exploiting reuse *and* requiring high disk throughput may cause the cache controller to become unstable. More specifically, higher prefetch space demands reduce available interval space. Reduced interval space will lead to poorer cache efficiency, which the results indicate causes higher load on the file system. Thus, to obtain higher data throughput, more space is given to lengthening the scheduler cycle (less to intervals), leading again to poorer cache efficiency and yet greater disk demands. Consequently, it is important to experiment with such a real-time scheduler to examine if such a "run away" effect can occur in practice, and if so, to limit its effects.

Stream caching causes an additional complication, in that the length of the scheduler cycle would vary, since some stream accesses might hit the cache and consequently have zero or very small prefetch. Were an optimised disk scheduler built to yield an equivalent technique to that used by cacheless Video-on-Demand servers, more interest in stream caching might result. Indeed, in the limiting case of no re-use, streams caching would then degenerate to a standard Video-on-Demand server bounded by the maximum disk bandwidth. Furthermore, in the case where two clients accessing the same clip are separated by an interval larger than the cache size, a modulated interval results. Hence, in this case, some benefit is gained from the memory buffer. The results in Chapter 5 have already shown the value of streams caching, though access to the disk is clearly an area for much optimisation.

In the current implementation, a disk transfer request must complete and an acknowledgement be received by the cache manager before the data is available in the cache. A consequence of this approach is that a stream can empty its prefetch while the data that the file system is fetching for it may already be partially available in cache memory. However, the stream cannot use that data until the full file transfer is complete, which for a 256Kbyte-transfer unit will take at least 130ms for the prototype.

Cache success to failure ratio should therefore be improved during heavy loading conditions by marking each cache line as resident in the cache once it is known to have

been received. A possible implementation is to use the '`write.lock`' bit of the management structure as a residency bit. An allocation of cache space will automatically set the lock bit to indicate that there is no data in the cache line. The link-transfer engine may then clear the write-bit of the cache line that was received in the previous transfer cycle. The final file transfer acknowledgement to the cache manager will cause the lock bit of the last line to be cleared. The clearing of the lock-bit must be deferred by one cycle, since the four Transputer links into the cache may complete their transfers at different times. Consequently, the receipt of a block at the manager TRAM does not guarantee the receipt of all blocks, i.e. the cache line, in all TRAMs.

6.1.7.3 NFS/Streams Cache Co-existence

The concept of sharing streamed data with normal file data, in this case NFS traffic, was only briefly explored in Chapter 3. Experimentation of the dynamic behaviour of cache sharing could prove invaluable, since the implementation of stream caching in general may then become a more attractive proposition. The benefits of stream caching over conventional approaches is undeniable – an acceptable coexistence with current caching techniques should lead to a wider acceptance of the principle. Such a development assumes sufficient real-time response from the server hosting the media.

6.1.8 Further Experiments with Caching

Although the emulated loading of the server can provide a good indication of expected real performance, there is no substitute for the predictably unpredictable human user. A possible realisation of real traffic may be derived from playback of multiple pre-recorded user sessions of some actual learning material containing video and audio. However, such an approach does not take account of the user to user interaction taking place within a live classroom. Such an interaction is, perhaps, likely to produce a 'near synchronisation' across the users. Ultimately, the most accurate test can only be gained from the involvement of a real classroom of users.

Stream optimised caching was originally devised for environments with predicted reuse, such as in a teaching room as adopted by SERVICE. However, the approach might also be successful in other environments. In particular, the results of the previous chapter showed that the caching approach works best with a larger number of users. In this case, the application of a memory buffer to a video subscriber service might be applicable, though provided that the temporal locality of access property could be demonstrated.

6.1.9 Quality of Service and Admission Control

The prototype server does not implement any kind of service quality guarantee for its clients, nor does it attempt to deny connections where the quality of service of any active streams would be jeopardised. To implement such a mechanism requires an accurate understanding of how the system behaves for a given set of users. For example, that the

disk bandwidth requirements of those users is understood, and that the maximum disk performance is not exceeded for the pattern of access. For a server based around a stream optimised cache, the outgoing bandwidth of the server can legitimately be in excess of the disk bandwidth. The policies of 'cacheless' Video-on-Demand servers are therefore inappropriate. Whether the disks can cope with a pattern of users while taking account of the stream cache is still not fully understood. For this reason, an admissions control policy cannot be applied without further study. At this stage, the only control policy possible is to deny connections once the potential outgoing bandwidth of the accepted streams exceeds the abilities of the network interface. Clearly, such a simplistic policy is not sufficient.

If a quality of service level is to be maintained, the HCI issues in section 6.1.4 are important from the QoS viewpoint also. The need to maintain the level of service might force a deferral of user requests for a few seconds until the request can be serviced without impacting upon the quality of other clients.

6.1.10 Recording Continuous Media Streams

One feature that the original server design considered was the recording of multimedia streams. To this end, the ATM TRAM adds a timestamp to each received AAL5 block with its own clock, thus avoiding the problem of clock differences across the cache TRAMs. Besides the designs however, the receive timestamping remains the only recording feature implemented in the prototype server. Because of the large available cache and consequent deferred write to the disk, multi-session recording is feasible. Furthermore, the prototype provides a ready platform on which to implement a real-time continuous media recorder.

6.2 A New Architecture

Regardless of the possible improvements to the Transputer prototype, it still suffers from a lack of development system support. For example, the UDP/IP and RPC support was hand written; TCP support is missing, and the extensibility of the system is poor. A more thorough implementation could only realistically be built on a platform with operating system support, *but provided only that the timing characteristics of that system were sufficient for the task*; the Transputer was chosen for the prototype because of its known timing characteristics. Unix had previously been rejected because of its far too heavyweight real-time response, which is in the order of 10ms granularity. Nevertheless, for the prototype, any attempt to add direct protocol support for CORBA, say, would be a large project in its own right. A re-engineering of the media server is therefore inevitable, though it is stressed that a feasibility study examining response latency for multiple users must first be carried out.

For general Video-on-Demand servers, RAID storage systems are now essential, since otherwise the necessary disk bandwidth cannot be achieved. The development of stream optimised caching should next be attempted with a RAID back-end, in order to examine the scalability of the solution. This again points to the need to use a hardware platform with operating system support, including RAID devices.

In the discussion of the ATM network interface in section 6.1.1, the TI C4x DSP was mentioned as a replacement for the T805 Transputer. Additionally, it was noted that the C4x devices contain communications ports, each with a performance of around 20 Mbytes/s. With the six ports of the C40, or the four ports of the C44, a number of parallel processing topologies are possible. Furthermore, with SGS-Thompson's announcement to discontinue support for the T9000 series, some other parallel or distributed computing platform must be found. If it is desirable to maintain the multiprocessor parallel architecture in the prototype video server, high throughput low latency inter-processor communications links are essential. Alternative technologies for such links are the C4x comport, Fibre Channel, FireWire and IEEE1355. These are now summarised in the following subsections.

6.2.1 Texas Instruments C4x

The Texas Instruments TMS320C4x [TI95] devices employ communications links, or comports, to communicate with other devices. Each comport is a bi-directional half-duplex asynchronous 8-bit wide parallel port running at 28Mbytes/s, of which an application will see around 20Mbytes/s (160Mbps). The multiple comports would allow a number of processor topologies; an architecture very similar to, though possibly not the same as the prototype server, are possible. Furthermore, the bandwidth available on a single comport is beyond the 58Mbps capability of all four Transputer hardware links. The construction of an ATM adapter based on the C4x should therefore be able to support a 155Mbps adapter to saturation by use of two or four comports in parallel. The half-duplex nature of the comport device may however, be an obstacle. The turn-around latency for each comport is relatively high, so ideally, two comports are required per device for a full-duplex communications link. Consequently, only two (320Mbps) or three (480Mbps) full-duplex links are possible, depending upon whether the C40, or C44 with all its comports, are used.

6.2.2 Analog Devices SHARC

An alternative to Texas Instruments DSP devices might be Analog Devices' SHARC DSP series [AD98]. Each SHARC features six 40Mbytes/s (320Mbps) bi-directional half-duplex synchronous point-to-point communications links. Thus, arranging these as three bi-directional links in the same topology as suggested for the TI DSPs, gives an aggregate bandwidth of 120Mbytes/s (960Mbps) in each direction, or 80Mbytes/s (640Mbps) if

only two bi-directional links are used. Since each link port transfer can be set as either transmit or receive, however, other topologies are possible which consider each of the six links as true bi-directional links. The SHARC does not suffer the link turn-around latency of the TI equivalent. In this case, the 40Mbytes/s would be shared between the up-link and down-link directions.

6.2.3 FireWire (IEEE 1394)

FireWire [And98], otherwise known as IEEE1394, is a joint development between Texas Instruments and Sony, originally having been developed by Apple Computer Corporation. The technology is a serial communication bus with a scalable bandwidth specification. At present, 100 and 200Mbps FireWire channels are available, with 400 and 800Mbps in development.

Unlike the C4x technology, which is a point-to-point link, FireWire is a shared bandwidth bus, similar to ThinLAN Ethernet. The technology was originally devised for the connection of video devices, such as camcorders, and other peripherals. Each device can be added and removed from the bus while in operation, without disturbing other devices. Digital video devices are already appearing on the market with IEEE 1394 sockets. In the context of a video server, the higher bandwidth serial bus provides enough raw bandwidth to cope with accessing an array of disks and moving data out onto the ATM adapter for transmission. A bus bandwidth of at least 200Mbps would be needed however, and it is not clear whether the bus would give sufficiently low inter-processor transfer latency.

6.2.4 Fibre Channel (FC)

Fibre Channel [Sta95] is a specification for the interconnection of storage devices and is another successor of the SCSI specification, besides UltraWide SCSI and Ultra2Wide SCSI. The Fibre Channel specification is purely a physical layer – a number of protocols are supported on top of FC as follows: SCSI, TCP/IP, AAL5, Link Encapsulation (FC-LE) and IEEE 802.2. FC-devices may be attached via either copper or fibre interfaces, so the name ‘fibre channel’ is somewhat misleading. Each FC node (a storage device or host, for example) uses a female DB-9 serial connector. The nodes may be connected together in one of three topologies.

In point-to-point topology, two nodes are connected directly together. This arrangement gives the lowest transfer latency since there are no intervening devices. The second topology connects the devices into an Arbitrated Loop (FC-AL). This arrangement is most similar to a token ring LAN, with the loop bandwidth being fairly shared between the nodes that form it. FC-AL allows up to 126 devices within a single loop. The third topology is a switched fabric, which allows any number of devices to be

switched with a single node network. Using a switched fabric is more expensive than FC-AL, since it requires a switch to be used to connect the FC nodes.

Each FC device link is rated at 100Mbytes per second. Thus, since each node may be full duplex, such as when connected point-to-point or via a switch, 100Mbytes/s can be achieved in both transmit and receive directions. Consequently, FC is as fast as the fastest and as yet unavailable FireWire link (800Mbps), and furthermore, allows this speed in both directions. Given further that this bandwidth is not necessarily shared between devices, that is, when using a switch or is point-to-point, it is a more appropriate technology for use within a video file server.

Consequently, Fibre Channel is likely to be the *ideal* technology for the connection of one or more storage devices to a cache host, since it is specifically intended for this purpose. FC allows a lot of flexibility in device interconnection, with an inter-node distance of up to 10km with single-mode fibre. FC also gives the most possibilities with respect to connecting a RAID storage system to a video server based around this technology, which is essential for any large-scale server.

With respect to the use of FC as a technology for the interconnection of multi-processors, there remains a question mark over whether the link latency would be sufficiently low. However, given that FC can be used in point-to-point mode, it would be a far more appropriate technology than FireWire in this respect.

6.2.5 DSLink/HSLink (IEEE1355)

IEEE1355, also known as ISO/IEC 14575, is a development of the T9000's 100Mbps DSLink originally developed by INMOS. The primary goal of IEEE1355 is as a low-latency high-throughput systems interconnect. A minimisation of cost and simple design were also factors in the specification of IEEE1355, such that each device can support many such links.

Similarly to the C4x and SHARC, the technology is a point-to-point inter-device communications link. The standard specifies interconnects from 300Mbps, the DSLink, to 1280Mbps, the HSLink. Efforts to design and manufacture devices using the technology are currently ongoing. Link routers and interface ICs are available on the market already, though the T9000 is presently the only *processor* that employs the link technology. Consequently, unless the standard is taken up by other silicon manufacturers, an exploitation of this technology is difficult. It would appear that the development of IEEE1394 has overtaken that of 1355; 1394 has substantial industrial backing. Indeed, consumer products are already available on the marketplace that contain IEEE 1394 communication links. An announcement from the PPRAM consortium dated 20th January 1996 does lend support to the 1355 standard, however. PPRAM includes as members, NEC, Texas Instruments, Matsushita and Sony corporations.

The PPRAM consortium press release announces an integrated device containing CPU, DRAM and communications links on a single chip, much like the Transputer. The device is rumoured to be of a CSP/Occam discipline and is the expected successor to the T9. No news of this device has surfaced since the press release, however, possibly due to 1394's dominance of the marketplace. It must be emphasised that the FireWire serial link technology is intended for peripheral interconnection, and not for high-speed parallel-computer communications; FireWire is not an appropriate technology for this application. Fibre Channel is not well suited to this purpose either, since it is overly complex for the task and it is unlikely that FC could be cost-effectively integrated into a single chip device. Consequently, IEEE 1355 may yet surface as the winning technology for inter-processor communications, though this looks unlikely with the current state of play.

6.2.6 Standard Workstation Technologies

Of course, the above solutions come from out of the original architecture used in the prototype server. A diamond topology was chosen to achieve the necessary aggregate bandwidth and large enough cache size. Consequently, modern day workstation technologies with multiple CPUs, dedicated ATM and Fibre Channel SCSI adapters may contain enough power to reengineer the prototype without resorting to more complicated architectures. However, it is not clear that such a solution would scale well, nor that the necessary timing characteristics of the prototype be maintained. It may nonetheless, provide a low-cost solution to the provision of Video-on-Demand, since such components are readily available on the marketplace and would not require any hardware development. As noted in the introduction to this section, a study of the timing characteristics of such commonplace technology is essential in order to determine the feasibility of this and the above approaches.

6.3 Summary

In this chapter, the major subsystems of the prototype server were explored for possibilities in improving the performance of the system – both pure software extensions, and simple to more complicated hardware solutions were covered. Further cache algorithms and experiments on the server were suggested for both the current cache algorithm and its extensions. Accepting the realistic practical limitations of the prototype, a new architecture for a caching media server was considered which builds upon the existing media file format, streams control protocol and a ported cache architecture and algorithm. A warning was given for a need of a feasibility study to ensure that the timing requirements of a media server could be met.

7. Conclusions

> "Theory without practice is sterile, but practice without theory is blind"
>
> — Lenin

This thesis has described the development of a small scale Video-on-Demand file server for a classroom environment. The idea of stream optimised caching has been refined and a simple implementation of it achieved. This has enabled the realisation of both working client systems and emulation experiments with the prototype server. This last chapter examines the contributions of the author, explores the successes of the work presented, and discusses how it has furthered the field of Video-on-Demand.

7.1 The Author's Contribution

In the construction of the server, all of the system components with two exceptions were designed and implemented by the author, including the neutral file format and RPC protocols. The first exception is that the server was build on top of Peter Linington's ATM cell driver, which is equivalent to the physical layer (layer 1) of the ISO network protocol stack. The second exception was in the use of Andrew Smith's UDP/IP implementation, which was much improved upon and optimised by the author. The ukcStreams protocol was derived from the ANSAware rate interface [Lin90]; all similarities with this protocol are deliberate.

To achieve the necessary throughput performance, the author optimised the ATM TRAM process layout and communications structure. A technique originally suggested by Linington was used to profile CPU usage. Essentially, the technique is to examine the Transputer's ready process-queues at regular intervals and count the quantized execution addresses. The author's implementation of this strategy enabled the location of those processes using the CPU most frequently. This analysis hence directed subsequent optimisation efforts. Further remarks about the above code profiling and other techniques used to optimise the server are discussed in section 7.3.

The HP-UX and PC client end-systems' display utilities were all developed by the author, including the PC's MPEG-hardware intermediate driver. The latter driver allowed the PC's hardware to display MPEG received from the network – a task it was not originally intended to do. The RPC ukcStreams protocol clients for HP-UX and Solaris were also the author's responsibility. The ANSAware and ORBIX proxy API's were developed from the author's RPC API by Ian Buckner and John Salter, respectively. The server content generation utilities and an MPEG clip-editing suite were entirely efforts of the author.

The concept of Stream optimised Caching and Modulated Stream Caching was originally devised by Peter Linington. The author was responsible for the refinement of the idea, and for producing an implementation of the concept in the prototype server. In the experiments with the prototype's caching performance, the user-behaviour script-generator and user behaviours were created by Peter Linington. The script driven emulator, the vfsBench protocol and the server monitor were all written by the author. Similarly, the server variables to monitor and the MPEG clip files and user-sets for the experiments were chosen by the author.

7.2 Degrees of Success

At the start of the project to build the video server, it was unclear how far the attempt would go, given the time constraints[12]. With time, it became clear that the amount of work involved was not insignificant, and that a great deal of work would be involved in reaching the project objective. Despite this worrying outlook, and through long term planning and efficient hard work, a fully functional and stable Video-on-Demand server has resulted. The use of the server within the CIMIS project, which was itself deployed to its intended clients, is some indication of the degree of success of the implementation.

In respect of stream optimised caching, the simulated multi-user sessions showed that 20 users could be supported with a cache of 90Mbytes with a very good level of service. Hence, the project goal to support 20 simultaneous users has been reached. This maximum of 20 is imposed more by the throughput limitation on the ATM network adapter, than any limitation of stream optimised caching. The prototype server now enables further work to be carried out in the investigation of stream optimised caching algorithms and other Video-on-Demand techniques.

The objective to produce a neutral file format, which can support multiple types of continuous media, has also been successful. The prototype has been used to provide both audio and video media types, and of various encodings in the latter case. The initial assumption that the low latency of ATM networks would obviate data resynchronisation protocols at the client has also been proved in practice.

The limitations in the solution very much derive from the prototype's hardware capabilities. The Transputer CPU and link technologies are now over ten years old, and are ripe for replacement. Given however, that the limits of this technology were reached in the prototype implementation, it should follow that a more powerful hardware solution will produce better results. In this sense also, the concept of streams caching and its implementation have been successful.

[12] The original SERVICE project lifetime was two years. CIMIS enabled further work on the prototype server for a further year. However, no more than 6 months of CIMIS' development time was used in additional server development.

7.3 Lessons Learned in Embedded Systems Development

Today's embedded systems development environments include tools for profiling code and locating those processes most responsible for consuming CPU time. However, in the absence of such tools, the ATM TRAM's CPU profiler demonstrates that a programmer can implement the equivalent functionality. For the prototype server, this analysis was essential in the success of the project, since without it, the maximum outgoing transfer rate would have been much lower. Thus, in a system where performance is a critical issue, code profiling is highly recommended as a means to locate bottlenecks and improve overall performance.

Operations such as link transfers, cell segmentation and Occam-channel protocol exchanges between processes all have associated overheads. Such overheads can be reduced, however, by maximising the size of unit transfers, thus minimising CPU scheduler intervention. In particular, each individual Occam protocol unit on a channel will cause rescheduling of the sending and receiving processes. Hence, in the prototype, the majority of channels use simple data arrays with the processes taking responsibility for marshalling and unmarshalling arguments, where necessary. In this respect, to avoid memory copies in processes, the arguments are simply 'aliased' within their data block and accessed by reference. Large memory copies are avoided by passing such data by reference, which is very important for systems with very high data throughput.

Communication overheads must always be considered and effort given to reducing interaction between processes on a critical line. The programmer should always be aware of the knock-on effect between processes sharing a CPU. For example, the prototype's IP implementation was optimised although it is not needed during streams playback. However, the IP process will draw on CPU resource, which will impact upon server performance if it has poor efficiency, especially if it reschedules frequently. Furthermore, overuse of high priority tasks should be avoided, since it can impact negatively on performance; each low priority task pre-emption has associated context-switching overheads. In essence, no process can be considered as isolated with respect to its system demands because of this knock-on effect.

In any new prototype system, one should never assume that the chosen components will be sufficient for the task. The feasibility (a.k.a. 'back of the envelope') calculation at the beginning of chapter four is an example of this. In performing such estimation, caution must be emphasised with respect to the real-world performance of the components, since these rarely match with the theoretical performances given in the manufacturers' brochures. Both the raw computational *and* the real-time response of such parts should be considered, in addition to any other system critical aspects.

A detailed knowledge of the components to be used is always necessary – this is particularly true of today's processors. For example, the Transputer instruction set uses

8-bit instructions, of which 4-bits are argument. Thus, when unrolling loops to improve code performance, 16 sequential operations with relative indices 0 through 15 are most efficient for this processor. Similarly, the Transputer has a variety of dedicated instructions such as the CRC instructions, which were used in the calculation of the AAL5 trailer CRC. Modern RISC CPUs must be noted particularly in this respect, with emphasis being placed on their instruction pipelines. For example, a sequence of instructions with two different orders may take differing amounts of CPU cycles to execute because of the effects of pipeline optimisations.

It will of course be noted that such code optimisation can only be achieved if the programmer writes the opcodes by hand in assembler. However, where performance is critical, the compiler output should be examined to check that it is indeed optimal. This is particularly true in the case of old compilers such as INMOS's Occam compiler; a programmer can often out-code it. An example of this in the prototype's source code is the routine used to reverse the order of bytes in each 32-bit longword – the Transputer is natively little-endian, whilst network protocols normally use big-endian byte order. Careful consideration of the number of registers and instruction cycle times produced a handcrafted routine far faster than the compiler could produce.

7.4 Contributions to the Field

Some of the efforts in the Video-on-Demand field have been concerned with re-clocking received data for jitter-free display on the client. Such an approach to smoothing the display output requires a buffer, which introduces delay in the display pipeline. Experience with the prototype server would suggest however, that such a re-clocking is not required for ATM network end-systems – at least between a few switches. The implications of this discovery are twofold. Firstly, the lower latency to the client display gives a little more time for the server to perform HCI type optimisations to achieve better server performance. Secondly, in combination with the VCR-like control protocol, the implementation of the client end-systems is greatly simplified, since the server is responsible for the client playback model and handling of frame transmissions.

The use of knowledge about users' relative positions and of their predicted behaviour has been shown useful in improving the level of service provided by a Video-on-Demand server. The use of a stream-optimised memory-buffer can enable a server to provide service to a number of users greater than it could with its disk(s) alone. This suggests that the relationship between a memory buffer and a file system is similar to that between a file system and a tape jukebox. That is, the memory, disk and tape are levels of a hierarchy, and each layer improves the level of service capability of those layers below it.

In the introduction to the Video-on-Demand chapter, the work of Wen, Wang, Chen et al [WCL97, WCLC97] was covered as most similar to that of this thesis. In fact, no other literature could be found that attempted to use knowledge of user activities together

with a memory buffer to improve the level of service. Wen's ideas on a relay mechanism are identical to those of group formation by stream interval, that is, stream optimised caching. In [WCL97] however, the *physical* server implementation, at the time of writing, is an earlier developed two disk striping approach without a memory buffer [WCL96]. The physical prototype of [WCL96] was used to derive the mathematical model of disk behaviour used in [WCL97, WCLC97]. Thus, their work on the relay mechanism is theoretical, and their conclusions are based upon a simulated server and simulated client load.

In contrast, the results of this thesis are based upon a working server implementation containing real movie clips with only the larger user loads being 'simulated'. These two efforts are further separated by the type of users using the system. For Wen et al, the assumption is that each playback request is on-demand, and plays to completion without user intervention once it is accepted. Such requests for service can be rejected if the system cannot start them within a finite period. For the streams caching prototype on the other hand, each request for service is assumed always to succeed. Furthermore, the users are expected to perform VCR-like operations, though this being exceptional behaviour, rather than very frequent. The scale of the prototype server is also much smaller in comparison to the simulations of Wen at al. They are therefore applying a pure streams-optimised-caching algorithm to a different target audience. They do not give an equivalent of modulated stream caching, which is a concept first introduced in this thesis.

A development in [WCLC97] from [WCL97] is a 'generalised relay mechanism', which forms speculative chains. That is, cache space is allocated to stream data that is not predicted to have any immediate reuse. However, such allocations are given to those data that are predicted to be most likely to be used by new users. As such, the generalised mechanism requires a popularity distribution for the clips that it serves, such that the algorithm may choose the most popular clips for speculative caching. Consequently, since idle cache space is being used for such speculative caching, modulating the smallest uncached interval is an *alternative* technique. The two techniques differ in that the interval modulation is based on a reuse predicted from current user behaviour, whereas the generalised mechanism uses predetermined (or possibility calculated) clip popularities. However, where the cache is not saturated, that is, where the sum of intervals is less than the total cache space, streams caching may retain data for which no reuse is predicted. In this case, speculative caching based on clip popularity is possible. However, in practice, the size of cache is likely to be considerably smaller than the size of disk it supports. Consequently, the cache will normally run under saturated conditions such that only one of speculative *or* modulated caching is possible for the remaining cache space.

The work on stream caching in this thesis has not concerned itself with situations in which disk bandwidth becomes limited; it does not try to prevent the disk becoming overloaded. Indeed, part of the exercise was to examine precisely those conditions under which the streams caching failed and find out why – it was not a study of a disk scheduling mechanism. Thus, the lack of the disk-scheduler issues is not an oversight on the part of the author. In relation to Wen's work, the prototype is further separated, since Wen does address the issue of limited disk resource, with the play-out quality of each stream being guaranteed. In this respect, the results of the simulation study in Chapter 5 give an insight into precisely those situations under which streams caching will perform well, or will break down.

In the discussions on the future of the prototype, a tuning of the disk scheduling mechanism was suggested, which is much the same as used within current 'cacheless' Video-on-Demand servers. Thus, it can be concluded that stream optimised caching is a complementary technique to these approaches, and *not* an alternative. This conclusion is further reinforced by the results of Wen et al. Video-on-Demand servers require memory in any case. Thus, commercial servers will inevitably contain significant amounts of memory, driven largely by the recent dramatic drop in the price of semiconductor memory – at the time of writing, less than $1 per MByte. Furthermore, in chapter 3, it was said that all caching algorithms are organised on predicted minimum time to reuse, which is precisely how stream optimised caching is defined. Therefore, stream optimised caching *must* be the optimal solution for caching continuous media data types. Stream optimised caching should therefore play an important role in how best to use the physical memory resource.

7.5 Future Work

Many potential avenues could be pursued in the light of this manuscript. However, the concentration suggested here is on stream optimised caching, this being the main thrust of the thesis. Furthermore, future work should be possible using the existing prototype server and without resorting to more exotic architectures in order to hasten the performance; raw power can always be used to improve performance, but that does not improve the technique itself.

Perhaps the most notable immediate performance improvement is the implementation of marking cache lines as resident in the cache as they are transferred from the file system. At present, it is only once the full transfer request is completed that the data is marked resident and available for use by cache clients. This optimisation will give much improved performance when the server loading is such that each stream's prefetch is starting to become exhausted before being serviced by the disk. The technique is similar to the *k*-buffer technique of [LDS95] with the buffer unit being one cache line. On the next scheduler cycle the stream will fetch up to a full second ahead and cause a

comparatively large transfer, since the stream prefetch is running dry. Thus, server behaviour is further degraded since the time to the data being marked cache resident is longer than when all prefetch requests can be serviced, that is, are all half a second. In using the *k*-buffer like approach, the time to the data being available is minimised.

Since the server is heavily dependent upon the performance of the effective file-system throughput, an implementation of block sharing in the container file format may show significant performance improvement. Further improvement may also be gained from an implementation of modulated stream caching. As suggested in chapter 6, the ability of the server to use a modulated interval should be switchable such that the difference in performance can be directly compared. Similarly, the packed file format should be compared against the non-block-sharing version for both caching algorithms. In this respect, a utility to convert a server file between the two formats would be very useful and could additionally generate storage efficiency statistics for the two cases.

As with the current prototype server, it is suggested that no attempt be made to exercise any admissions control algorithm. The point of this approach is to find and understand those conditions that cause service failure within the server, such as in the disk scheduling or cache algorithms. In particular, the server offers the possibility of experimentation in the slipping of user requests, both in real-time and on the logical timeline, in an attempt to prevent such server overload. In any case, an admissions control policy is complicated by the presence of the cache, which reduces load on the disks and improves outgoing bandwidth to above that of the disks. Nevertheless, an admissions control policy may be formulated based upon the located areas of breakdown in a stream optimised server.

The most promising development area for stream optimised caching is without doubt integration with a real-time disk scheduler. That is, to use one (or several) of the scheduling techniques documented in the literature. Achieving this would prove the assertion that the family of cache management techniques is complementary to the family of real-time disk scheduling techniques. In particular, the server's first-come-first-served (FCFS) approach has the worst performance in terms of incurred disk-head seeking and is a clear area for potential performance improvement. Due to the very likely tightly coupled cache and disk scheduling algorithms, this might give rise to the family of real-time caching disk scheduler algorithms, for example, SOC-Scan and SOC-BSCAN.

The use of cache space by stream prefetch in relation to space used in interval caching is an area that should be examined; it was shown in the results that maximising space available to intervals leads to improved cache performance. The prototype's approach is very simplistic and purely functional, however focus should be placed on minimising use of cache space for prefetch. At the same time, it is also necessary to increase prefetch lengths to achieve maximum disk throughput, when needed, whilst

reducing such lengthened prefetch once bandwidth requirements diminish. In so doing, the amount of cache space available for re-use data is maximised whilst achieving the necessary disk throughput. It is in this area that the performance monitor data may be used to influence the scheduler cycle length and size of stream prefetch. In fact, the replacement of the current disk scheduler with a real-time scheduler would attempt minimise prefetch lengths for the prevailing conditions by design.

7.6 Summary

In this chapter, the contributions of the author to the work contained in this thesis were declared. The chapter included an overview of the practices and techniques used in the construction of the server, and that are recommended in the development of other such systems. The impact of the work, both within the university department, and within the field of Video-on-Demand in general, was discussed. In conclusion, the development options for the prototype server and the principle of stream optimised caching were considered.

8. Bibliography

[AD98] SHARC DSP Microcomputer Preliminary Technical Data Sheet (ADSP-21160), *Analog Devices Inc.*, http://www.analog.com, One Technology Way, P.O. Box 9106, Norwood MA 02062-9106, USA, 1998

[And98] Don Anderson, Inc. MindShare, "Firewire System Architecture: IEEE 1394", *Addison-Wesley Pub Co.*, ISBN 0-20169-470-0, March 1998

[AOG92] David P. Anderson, Yoshitomo Osawa, Ramesh Govindan, "A File System for Continuous Media", *ACM Transactions in Computer Systems*, Vol. 10, No. 4, pp311-337, Nov. 1992

[AWY96] Charu Aggarwal, Joel Wolf, Philip S Yu, "On Optimal Piggyback Merging Policies for Video-On-Demand Systems", *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modelling of Computer Systems*, pp200-209, May 1996

[BP97] Milind M Buddhikot, Gurudatta M Parulkar, "Efficient data layout, scheduling and playout control in MARS", *Multimedia Systems*, Vol. 5, No. 3, pp199-212, 1997

[BR96] David W Brubeck, Lawrence A Rowe, "Hierarchical Storage Management in a Distributed Video-On-Demand System", *University of California, Berkeley CA*, Internal Report, 1996

[CKY93] MS Chen, DD Kandlur, PS Yu, "Optimisation of the Group Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams", *Proceedings of the 1st ACM International Conference on Multimedia*, Anaheim, CA, pp235-241, 1993

[CLG94] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, David A Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, Vol. 26, No. 2, pp145-185, June 1994

[Cus94] Helen Custer, "Inside the Windows NT File System", *Microsoft Press*, ISBN 1-155615-660-X, 1994

[CVV97] Tzi-Cker Chireh, Chitra Venkatramani, Michael Vernick, "Design and Implementation of the Stony Brook Video Server", *Software-Practice and Experience*, Vol. 27, No. 2, pp139-154, Feb. 1997

[CWH95] Alan J Chaney, Ian D Wilson, Andrew Hopper, "The Design and Implementation of a RAID-3 Multimedia File Server", *The 5th Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 1995)*, Durham NH, April 1995.

[DCP95] Z Dittia, J Cox, G Parulkar, "Design of the APIC: a high-performance ATM host-network interface chip", *Proceedings of the IEEE INFOCOM '95*, pp179-187, Boston, 1995

[Dun89] Ray Duncan, "Design Goals and Implementation of the New High Performance File System", *Microsoft Systems Journal*, pp1-13, September 1989

[FCP97] Fujitsu Enterprise 10K Series specification sheet, *Fujitsu Computer Products of America, Inc.*, 2904 Orchard Parkway, San Jose, CA 95134, http://www.fcpa.fujitsu.com, 1997

[GLM95] L Golubchik, J Lui, R Muntz, "Reducing I/O Demand in Video-on-Demand Storage Servers", Proceedings of the *ACM SIGMETRICS Conference on Measurement & Modelling*, pp25-36, Ottawa, Canada, 1995

[GVK95] D. James Gemmel, Harrick M. Vin, Dilip D Kandlur, et al., "Multimedia Storage Servers – A Tutorial", *Computer*, Vol. 28, No. 5, pp40-49, 1995

[Hen95] Paul Henshaw, "ATM Video FileStore", *University of Kent at Canterbury*, Internal Report, 1995

[HLL96] Jenwei Hsieh, Mengjou Lin, Jonathan CL Liu et al., "Performance of A Mass Storage System for Video-On-Demand", Proceedings of the *IEEE INFOCOM '96*, pp771-778, 1996

[HPN96] Barry G Haskell, Atul Puri, Arun N Netravali, "Digital Video: An Introduction to MPEG-2", *Chapman and Hall*, ISBN 0-412-08411-2, 1997

[HS96] Roger L Haskin, Frank B Schmuck, "The Tiger Shark File System", *Proceedings of the IEEE COMPCON '96,* pp226-231, San Jose CA, Feb. 1996

[IBM97] IBM Ultrastar 9ZX specification sheet, *IBM Storage Systems Division*, 5600 Cottle Road, San Jose, CA 95193, http://www.storage.ibm.com, 1997

[ITU93] "B-ISDN ATM Adaptation Layer (AAL) Specification", ITU-T I.363, *International Telecommunications Union*, March 1993

[JS97] John Salter, "A Multimedia Ropes Player", *University of Kent at Canterbury*, Distributed Systems MSc Dissertation, July 1997

[JUS89] Chet Juszczak, "Improving the Performance and Correctness of an NFS Server", *USENIX Winter Conference Proceedings*, pp53-63, 1989

[KCL97] Taeck-Geun Kwon, Yanghee Choi, Sukho Lee, "Disk Placement for Arbitrary-Rate Playback in an Interactive video server", *Multimedia Systems*, Vol.5, No.4, pp271-281, 1997

[KS97] Deepak R Kenchammana-Hosekote, Jaideep Srivastava, "I/O Scheduling for Digital Continuous Media", *Multimedia Systems*, Vol. 5, No. 4, pp213-237, 1997

[Kya95] Othmar Kyas, "ATM Networks", *International Thomson Publishing*, 1995, ISBN 1-850-32128-0

[LDS95] Jonathan CL Liu, David HC Du, James A Schnepf, "Supporting random access real-time retrieval of digital continuous media", *Computer Communications*, Vol. 18, No.3, pp145-159, March 1995

[Lin90] Prof. Peter Linington, Gerald Tripp, "Palantir Project", *University of Kent at Canterbury*, no references available, Early 90s.

[Lin95] Prof. Peter Linington, "The caching of video streams", *University of Kent at Canterbury*, SERVICE internal report 3/95, April 1995

[LIP95] Peter Linington, Alan Ibbetson, Ian A Penny, Andrew A Smith, Gerald EW Tripp, "A Parallel Implementation of the ANSA REX Protocol", *World Transputer Congress '95*, Harrogate, UK, September 1995

[LL95] S W Lau, John C S Lui, "A Novel Video-On-Demand Storage Architecture for Supporting Constant Frame Rate with Variable Bit Rate Retrieval", *The 5th Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 1995)*, Durham NH, April 1995

[LPS94] Phillip Lougher, David Pegler, Doug Shepherd, "Scalable Storage Servers for Digital Audio and Video", Presented at *the IEE International Conference on Storage and Recording Systems 1994*, Keele UK, April 1994

[LS93] Phillip Lougher, Doug Shepherd, "The Design of a Storage Server for Continuous Media", *The Computer Journal*, pp32-42, Vol. 36, No. 1, Feb. 1993

[LSP94] Phillip Lougher, Doug Shepherd, David Pegler, "The Impact of Digital Audio and Video on High Speed Storage", *13th IEEE Symp. on Mass Storage Systems*, Annecy, France, Jun. 1994

[McC95] Prof. John M McCann, Internet document titled, "Education on Demand", *Fuqua School of Business*, Duke University, July 1993 (Revised May 1995)

[McK90] Kirby McKoy, "VMS File System Internals", *Digital Press/Prentice Hall*, ISBN 1-55558-071-8, 1990

[MJL84] Marshall K McKusick, William N Joy, Samuel J Leffler, Robert S Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp181-197, August 1984

[MPF96] Joan L Mitchell, William B Pennebaker, Chad E Fogg, Didier LeGall, "MPEG Video Compression Standard", *Chapman and Hall*, ISBN 0-412-08771-5, 1996

[NFS89] "NFS: Network File System Protocol Specification", *Sun Microsystems*, RFC 1094, March 1989

[NFS93] "NFS: Network File System Version 3 Protocol Specification", *Sun Microsystems*, 1993

[NMH95] Gerald Neufeld, Dwight Makaroff, Norm Hutchinson, "The Design of a Variable Bit Rate Continuous Media Server", *The 5th Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 1995)*, Durham NH, April 1995

[NY94] Raymond T Ng, Jinhai Yang, "Maximising Buffer and Disk Utilisation for News-On-Demand", *Proceedings of the 20th VLDB Conference*, pp451-462, Santiago, Chile, 1994

[OB94] Banu Özden, Alexandros Biliris et al., "A Low-Cost Storage Server for Movie on Demand Databases", *Proceedings of the 20th VLDB Conference*, Sep. 1994

[OWC95] Yen-Jen Oyang, Chun-Hung Wen, Chih-Yuan Cheng et al, "A Multimedia Storage System for On-Demand Playback", *IEEE Transactions on Consumer Electronics*, Vol. 41, No. 1, pp53-63, February 1995

[PM92] William B Pennebaker, Joan C Mitchell, "JPEG Still Image Data Compression Standard", *Van Nostrand Reinhold*, 1992, ISBN 0-44201-272-1

[PGK88] David A. Patterson, Garth Gibson, Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *ACM SIGMOD '88*, pp109-116, Chicago, June 1988

[QC98] Quantum Atlas III specification sheet, *Quantum Corporation*, 500 McCarthy Boulevard, Milpitas, CA 95035, http://www.quantum.com, 1998

[RBE94] Lawrence A Rowe, John S Boreczky, Charles A Eads, "Indexes for User Access to Large Video Databases", *Symp. On Elec. Imaging Sci. & Tech.*, San Jose CA, Feb. 1994

[RP94] Lawrence A Rowe, Ketan D Patel, et al., "MPEG Video in Software: Representation, Transmission, and Playback", *Symp. On Elec. Imaging Sci. & Tech.*, San Jose CA, Feb. 1994

[RPC88] "RPC: Remote Procedure Call Protocol Specification, Version 2", *Sun Microsystems*, RFC 1057, June 1988

[SC97] Seagate Cheetah specification sheet, *Seagate*, 62 bis, avenue André Morizet, 92643 Boulogne-Billancourt Cedex, France, http://www.seagate.com, 1997

[SCW88] Barry Shein (*Software Tool & Die*), Mike Callahan, Paul Woodbury (*Encore Computer Corporation*), "NFSStone: A Network File Server Performance Benchmark", 1988

[SD95] Scott D Stoller, John D DeTreville, "Storage Replication and Layout in Video-on-Demand Servers", *The 5th Workshop on Network and Operating*

System Support for Digital Audio and Video (NOSSDAV 1995), Durham NH, April 1995

[SKS97] Alok Srivastava, Anup Kumer, Aditi Singru, "Design and Analysis of a Video-On-Demand Server", *Multimedia Systems*, Vol. 5, No. 4, pp238-254, 1997

[Sta95] Jeffrey D Stai, "Fibre Channel Bench Reference (ENDL SCSI Series)", *ENDL Publications*, ISBN 1-87993-617-8, August 1995

[Ste95] Ralf Steinmetz, "Multimedia file system survey: approaches for continuous media disk scheduling", *Computer Communications*, Vol. 18, No. 3, pp133-144, March 1995

[TI95] TMS320C44 DSP Specification Sheet, *Texas Instruments*, P.O Box 1443, Houston, TE 77251–1443, USA, December 1995

[Tri95] Gerald EW Tripp, "ATM Interface Technical Report", *University of Kent at Canterbury*, Internal Report, 1995

[VVC96] Michael Vernick, Chitra Venkatramani, Tzi-cker Chiueh, "Adventures in Building the Stony Brook Video Server", *ACM Multimedia 96*, pp287-295, Boston MA, 1996

[WCL96] Chun-Hung Wen, Chih-Yuan Cheng, Meng-Huang Lee et al., "Effective Utilisation of Disk Bandwidth for Supporting Interactive Video-On-Demand", *IEEE Transactions on Consumer Electronics*, Vol. 42, No. 1, pp71-79, 1996

[WCL97] Chun-Hung Wen, Chih-Yuan Cheng, Meng-Huang Lee, Yen-Jen Oyang, "Upgrading the Service Capacity of Video-on-Demand Servers with a Memory Buffer", *Future Generation Computer Systems*, Vol. 12, No. 6 pp565-577, 1997

[WCLW97] Fu-Ching Wang, Chih-Yuan Cheng, Meng-Huang Lee, et al, "Memory and Disk Bandwidth Management in Video-on-Demand Servers", *Department of Computer Science and Information Engineering, National Taiwan University*, 1997

[WDC97] WD Enterprise WDE9100[AV] specification sheet, *Western Digital Corporation*, 8105 Irvine Center Drive, Irvine, California 92618, http://www.wdc.com, 1997

[XDR87] "XDR: External Data Representation Standard", *Sun Microsystems*, RFC 1014, June 1987

[ZK97] Hui Zhang, Edward W Knightly, "RED-VBR: a renegotiation-based approach to support delay-sensitive VBR video", *Multimedia Systems*, Vol. 5, No. 3, pp164-176, 1997

9. Appendix A: Pattern Generator User Behaviour Scripts

This section gives the details of the pattern generator input scripts as used for the experiments of Chapter 5. The field descriptor tables are given again here for reference.

9.1 Target Clip Descriptors

CLIPLEN	Length of clip when played at normal speed (milliseconds)
CLIPWAIT	Waiting time before making a transition to a successor clip (ms)
CLIPDETAIL	Probability per second of interaction during the clip (0 to 1)
CLIPTERM	Probability per second that the clip will be terminated before its end (0 to 1)

```
10 #nclip
#CLIPLEN CLIPWAIT CLIPDETAIL CLIPTERM  File Size / Bytes
103760   500      0.9        0.01        17338368
38360    3000     0.9        0.01         6582272
154280   10000    0.9        0.01        23810048
104240   5000     0.9        0.01        23183360
129560   10000    0.9        0.01        31076352
232760   80000    0.9        0.01        49930240
70160    15000    0.9        0.01        10252288
37240    5000     0.9        0.1          7057408
61240    15000    0.9        0.01         9224192
38160    2000     0.9        0.1          6680576
                                Total: 185135104 Bytes
```

9.2 Clip Transition Probability Matrix

The following is the NCLIP by NCLIP transition probability matrix from clip i to j:

```
# clip transition matrix
0     0.9   0.1   0     0     0     0     0     0     0
0     0     0.8   0     0.1   0     0     0     0     0.1
0.1   0     0.1   0.8   0     0     0     0     0     0
0     0     0     0     0.9   0     0     0     0.1   0
0     0     0     0     0.05  0.95  0     0     0     0
0     0     0     0.05  0     0.15  0.7   0.05  0     0.05
0     0.05  0.05  0     0     0.1   0.1   0.6   0.05  0.05
0.05  0     0     0     0     0.05  0     0     0.85  0.05
0     0     0     0     0     0     0.05  0     0.05  0.9
0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.1   0.5
```

9.3 Emulated User Behaviour Descriptors

USERHASTE	A multiplier for CLIPWAIT above. Allows each user to have different tolerances for watching each clip (real number)
USERLATE	A delay before the user becomes active in the simulation (ms)

USERVCI An outgoing virtual channel identifier such that the resulting video can be watched.

USERSTYLE An index into a user interaction behaviour table - see below

```
25 # users

#HASTE LATE       VCI STYLE
0.5    0          128   1
0.6    0          129   1
0.6    0          130   1
0.9    0          131   1
1.0    60000      132   1
1.0    60000      133   1
0.6    60000      134   1
0.8    60000      135   1
0.7    120000     136   1
0.6    120000     137   1
0.5    120000     138   1
0.9    120000     139   1
0.5    180000     140   1
0.6    180000     141   1
0.7    180000     142   1
0.7    180000     143   1
1.0    240000     144   1
0.7    240000     145   1
0.6    250000     146   1
0.8    250000     147   1
0.7    180000     148   1
0.7    180000     149   1
1.0    240000     150   1
0.7    240000     151   1
0.6    250000     152   1
```

9.4 User Behavioural Style Descriptors

PAUSE A multiplier for CLIPDETAIL which gives the probability per second that the user will pause the playback

CONTINUE The probability per second that a user will continue with the playback

SKIP Similarly to PAUSE, is a multiplier for CLIPDETAIL giving the probability per second of the user skipping over the playback

MEANSKIP The average length of a skip action in seconds

VARSKIP The standard deviation of the skip length in seconds

```
1     #act types
# PAUSE CONTINUE SKIP MEANSKIP VARSKIP
0.005     0.1    0.01   -15000   30000
```

10. Appendix B: vfsBench Protocol Definition

```
/*
 * vfs data collection protocol
 *
 * An accompanying protocol for ukcStream protocol.  Simply allows
 * a client to start up data collection on the video server and to
 * collect that data as required.  Assumes that only one server collects
 * data at any time, hence the lack of session key from RESET.
 *
 * Kneale J Rothwell, 2/12/96
 *
 * Returning data for the active streams is all very well, but it tends
 * to leave out part of the picture.  I've therefore added some total
 * counts to remove some inconsistencies in the data.  (kjr1 - 24/1/97)
 */

#ifndef VFSBENCH_H
#define VFSBENCH_H

struct RESETres {
  int errno;
};

struct COLLECT_resok {
  u_long  streams_active;
  u_long  total_hits;
  u_long  cache_hits<>;
  u_long  total_misses;
  u_long  cache_misses<>;
  u_long  total_dropped;
  u_long  dropped_Tx<>;
  u_long  total_Txd;
  u_long  frames_Txd<>;
  float   frame_rate<>;
  u_long  total_overdue;
  u_long  overdue_us<>;
  u_long  total_daccess;
  u_long  disk_accesses<>;
  u_long  total_dbytes;
  u_long  disk_bytes<>;
  u_long  total_dtime_ms;
  u_long  disk_time_ms<>;
  u_long  groups_active;
};

union COLLECTres switch (int errno) {
case 0:
  COLLECT_resok  resok;
default:
  void;
};

struct HALTres {
  int errno;
};

union REINITres switch (int errno) {
case 0:
  u_long  size_set;      /* Size of cache in Bytes set by call */
default:
  void;
};

program VFSBENCH {
  version VFSBVERS {
    RESETres
```

```
      VFSBPROC_RESET  (void)   = 1;
    COLLECTres
      VFSBPROC_COLLECT(void)   = 2;
    HALTres
      VFSBPROC_HALT   (void)   = 3;
    REINITres
      VFSBPROC_REINIT (u_long) = 4; /* Arg of 0 sets all available mem */
  } = 2;
} = 200022;

#endif
```