MAXIMIZING BUFFER AND DISK UTILIZATIONS FOR NEWS ON-DEMAND

By

Jinhai Yang

B. Sc. (Computer Science) University of Science and Technology of China

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1994

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date: Sept. 28. 1994

Abstract

In this thesis, we study the problem of how to maximize the throughput of a multimedia system, given a fixed amount of buffer space and disk bandwidth, both pre-determined at design-time. Our approach is to maximize the utilization of disk and buffers. We propose two methods. First, we analyze a scheme that allows multiple streams to share buffers. Our analysis and simulation results indicate that buffer sharing could lead to as much as a 50% reduction in total buffer requirements. Second, we develop three prefetching strategies: straight forward prefetching (SP), and two types of intelligent prefetchings (IP1 and IP2). All these strategies try to read a certain amount of data into memory before a query is actually activated. We demonstrate that SP is not effective at all, but both IP1 and IP2 can maximize the effective use of buffers and disk, leading to as much as a 40% improvement in system throughput. We also extend the two intelligent prefetching schemes to a multiple disk environment.

Table of Contents

List of Figures

Acknowledgments

First of all, I would like to express my sincere thanks to Dr. Raymond T. Ng, my supervisor, for his guidance, commitment and understanding throughout my research work.

I would also like to thank my co-supervisor Prof. Sameul T. Chanson. He introduced UBC to me, gave me many helps in my earlier year in UBC, and gave me many wonderful advises on how to do research.

Special thanks are also to Prof. Kellogg Booth for being the second reader of this thesis, and for all the valuable suggestions he gave me.

The financial support from the Department of Computer Science at the University of British Columbia, in the form of a research assistantship under NSERC strategic grants is gratefully acknowledged.

Chapter 1

Introduction

1.1 Distributed multiple-user multimedia systems

Multimedia systems are computer systems that can support multiple media. Here, "media" refers to text, static graphics, audio data, video data, etc. For the last several decades, the computational power of modern computers has increased dramatically. However, the user interfaces and software technology have not improved at the same pace. Usually, conventional computer systems only support text and static graphics. Thus their application domains are limited. On the other hand, a multimedia computer system can support not only text and graphics, but also audio and video data. In addition, we will expect that all the media work in a consistent and cooperative manner. This gives users a better opportunity to perform many kinds of tasks that are otherwise not possible. The application of multimedia computer systems has been spreading very quickly in areas such as distance education, teleconference, news-on-demand, video-on-demand, computer games, etc.

Multimedia systems on single-machine, single-user platforms have been studied and used for several years. However, the research and development in a multiple-user, distributed environment is just getting started. With the advancement of techniques in storage systems, computer networks, compressing methods and the promised "Information Super-highway", distributed multiple-user multimedia systems will become a reality

in the near future.

Multimedia systems present tremendous challenges to current technologies, especially when a distributed, multiple-user environment is being considered. A multimedia file server is the core of such an environment, and is focus of this thesis. In a distributed environment, a multimedia file server should have the ability to support multiple users simultaneously. The users can be local users who submit requests directly to the server, or remote users who submit requests through the network. The response time for both the local users and the remote users should be guaranteed. In addition to response time (latency), multimedia requests have a further requirement called continuity requirement. The continuity requirement requires once a stream (e.g. video) gets started, it should not be interrupted until it finishes.

Closer study shows that disk bandwidth and memory are the two most important system constraints in a multimedia file server. How to make efficient use of these two resources is a key performance issue when designing and implementing such servers. Consider a simple example. A digital video stream with acceptable quality requires 30 frames per second. For each frame, suppose there are 400×200 pixel. For each pixel, we use 9 bits to represent color. So for each second of this video stream, the data requirement is $30 \times 400 \times 200 \times 9 = 21,600,000$bits. This is about 2.7Mbytes. Even if we use compression (and assume a compression ratio 10:1), it still requires 270Kbytes of data. If we have a video stream of one hour long, the data requirement will be 972Mbytes! As for the disk bandwidth requirement, if the server needs to support a maximum of 50 users, the disk bandwidth requirement would be at least 13.5Mbytes/s.

1.2 Motivation and contribution of this thesis

Many excellent studies regarding multimedia file servers have been conducted [Y+89], [RV93], [LS93], [Gem93], [RW93], [CKY93], [RV91], [GC92], [RVR92], [AOG92], [TB93] and [Pol91]. With respect to the topic of this thesis, these works can be grouped into three major categories. The first group [Y+89][RV93][LS93] is primarily concerned with data placement, i.e. how to organize data on disk more efficiently so that the data retrieval speed can be improved. The second group [Gem93][RW93] [CKY93] studies the issue of intelligent disk scheduling. The basic idea of these approaches is to optimize disk arm movement so that the disk seek time can be reduced. The third group [RV91] [GC92][RVR92] [AOG92][TB93][Pol91] discusses how to build a multimedia file server. The key problems studied by these papers are admission control and on-line scheduling. To a large extent, most of these proposals aim to minimize disk seek latency so as to satisfy the continuity requirements of multimedia streams. Most of the proposed soultions are designed from a static point of view.

Complementary to the problems addressed in the studies mentioned above, in this thesis, we study the multimedia file server in a more dynamic environment with randomly arriving queries. A typical example of this kind of system is a multimedia file server for News-On-demand. More specifically, given a fixed amount of buffer space and disk bandwidth, both pre-determined at design time, we study how to maximize the throughput of a multimedia system, and minimize the response time of queries.

There are basically two key ideas we propose to improve disk and buffer utilizations and system throughput: *prefetching* and *buffer sharing*. The idea of buffer sharing is allowing multiple streams to dynamically share buffer. Our analysis and simulation results show that this can reduce the overall buffer requirement by up to 50%. The idea

of prefetching is to read some data into the memory even if the system has not enough resources to serve it at this moment. Because of the prefetched data, the request will have a better chance to get admitted at a later time. Simulation results show prefetching can improve system performance up to 40%.

This thesis is mainly concentrated on the discussion of these two key ideas. More specifically, the following contributions are made this thesis:

- analyzing buffer sharing in the simple case with all the streams having the same consumption rates;
- analyzing buffer sharing in the more general case in which streams can have different playback rates;
- using simulation to evaluate these two schemes;
- analyzing prefetching;
- proposing three prefetching strategy: SP, IP1 and IP2;
- using simulation to evaluate the proposed prefetching schemes;
- studying extensions of buffer sharing and prefetching to a multiple disk environment.

1.3 Outline of this thesis

The thesis is organized as follows. Chapter 2 defines the basic concepts of our framework and introduces some basic formulas used in our later analysis. Chapter 3 is a brief summary of related work. We list all the important work that other researchers have done in this area. Chapter 4 and Chapter 5 discuss buffer sharing and prefetching, respectively. The most important results of this thesis are given in these two chapters.

To achieve better performance, using multiple disks is the most obvious solution. Chapter 6 discusses how to extend the results in Chapters 4 and 5 to a multiple disk environment. Chapter 7 describes our simulation package and gives some important simulation results. The last chapter summarizes the results in the thesis.

Chapter 2

Preliminary

In this chapter, some preliminary knowledge (e.g. basic notations, formulas) of our work will be given. After a brief description of the general framework, the fixed order reading scheme, which is the basic framework that this thesis is built upon, is described. Followed, the basic formulas for calculating reading period length t and buffer requirement B is given. Finally, a scheme of how to approximate non-contiguous data placement using contiguous curve is presented.

2.1 General framework

Due to the real-time nature of multimedia data and memory constraints, most multimedia file servers work in a round robin manner. The server serves all the requests cyclically. In each cycle, the server will read a certain amount of data for each request. This amount of data must be large enough to last the period of the cycle. This requirement is called the continuity requirement for a multimedia stream. If the condition can not be satisfied, we say a starvation occurs.

In this thesis, we consider such a scenario: A multimedia file server serves multiple user requests concurrently. We call a user's request for multimedia data a query. The multimedia data that a query requests is called stream. The server maintains a working set and a waiting queue. A query in the working set is being served by the server. The queries that cannot be served at the current time will be put in the waiting queue, and

will be served at a later time.

The biggest difference between our model and most of the models discussed in the literature is that our model has more dynamic characteristics. Queries arrive dynamically, and the data streams that queries request may have different lengths and consumption rates. With the frequent arrival and termination of queries, the server state changes rapidly. There are basically two system states, we call them stable state and transient state. When there is no admission and/or termination of queries happening, the server is in the stable state. When the server is in the processing of admitting incoming queries and/or deleting terminated queries, it is in the transient state. The time period when the server is in the transient state is called the transient period. Handling the transient period in a multimedia file server is challenging work.

2.2 Fixed order reading scheme

In the cyclic reading scheme, the reading order within a period (cycle) can be fixed or variable. Variable reading order provides a better chance to optimize disk arm movement, so as to minimize total seek time and improve system throughput. On the other hand, the behavior of fixed reading order scheme is more predictable, so that we might be able to allocate buffer in a more efficient way. In addition, buffer sharing is also easier to handle in the case of fixed reading order.

In this thesis, only the fixed reading order scheme is used. In our initial algorithm, we assume that the data for each stream are continuously placed on the disk. In Section 3.4, we will discuss how to remove this constraint.

Figure 3.1 shows how fixed order cyclic reading works. In this example, there are three streams being served. In each cycle, these streams are served in the order: S_1, S_2, S_3. At

the end of the cycle, there might be certain amount of disk idle time left.

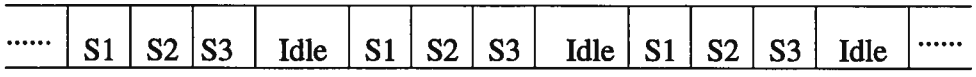

Figure 2.1: Fixed order cyclic reading

More generally, let there be n multimedia streams denoted by $S_1, S_2, ..., S_n$. Let the consumption rate of Stream S_i be P_i. Let t_i be the amount of time required to read S_i in each period. Let t be the total length of a period, and let $s_{i,j}$ be the seek time from stream S_i to stream S_j. Obviously, in each period:

$$t \geq t_1 + t_2 + ... + t_n + s_{1,2} + s_{2,3} + ... + s_{n,1} \tag{2.1}$$

To simplify notation, let $s = s_{1,2} + s_{2,3} + ... + s_{n,1}$, and $P = P_1 + P_2 + ... + P_n$.

Disk utilization is defined as:

$$\rho = \frac{t_1 + t_2 + ... + t_n + s}{t} \tag{2.2}$$

Our notation of the disk utilization only refers to the n multimedia streams being served. If the system also serves other users, those disk usages are not counted. The disk utilization ρ only represents how busy the disk is. Since the disk time includes both reading time and seek time, it does not represent how much *real* work the disk does.

The following table summarizes the meanings of the symbols to be used in this thesis.

Symbol	Meaning of symbol
B_{max}	maximum number of available buffers
B	total buffer consumption of n streams
B_{shar}	total buffer consumption of n streams with buffer sharing
B_i	buffer consumption of stream S_i
L	block size in non-contiguous placement
G	seek time between non-adjacent blocks in non-contiguous placement
P	total consumption rate of all the n streams
P_i	consumption rate of stream S_i
P_i^{pft}	adjusted consumption rate of stream S_i after prefetching
R	maximum disk reading rate
S_i	the i-th stream
s	total switching time within a period
$s_{i,j}$	switching time between stream S_i and stream S_j
t	length of a cycle
t_i	reading time for S_i within a cycle
T_i	length of stream S_i
ρ	disk utilization

In the fixed reading order scheme, in each period, the data consumed (played back) by stream S_i is $t \times P_i$, and the data produced (read) for stream S_i is $t_i \times R$. The *continuity requirement* can be expressed more precisely as follows:

$$t_i \times R \geq t \times P_i, \quad 1 \leq i \leq n \tag{2.3}$$

In order to avoid overflow and reduce buffer requirement, the ideal case is:

$$t_i \times R = t \times P_i, \quad 1 \leq i \leq n \tag{2.4}$$

From Equation 2.4, we derive:

$$\frac{t_i}{t_j} = \frac{P_i}{P_j}, \quad 1 \leq i, j \leq n \tag{2.5}$$

To minimize buffer consumption, the reading time for each stream should be proportional to its consumption rate. By combining Equation 2.2 and 2.5, t_i can be determined exactly as:

$$t_i = (\rho \times t - s) \times \frac{P_i}{P} \tag{2.6}$$

2.3 Determining the lower bound of t

Equation 2.6 can be used to calculate t_i in each cycle provided t is given. A lower bound of t could be established from Equations 2.3 and 2.6:

$$t \geq \frac{s \times R}{\rho \times R - P} \tag{2.7}$$

This equation leads to two interesting observations. First, the equation is valid only if $\rho \times R - P > 0$. Even if the disk utilization ρ is set to the maximum value 1, it is still necessary that $R > P$. This is the most obvious admission control criterion. That is, without violating their continuity requirements, a system cannot admit so many streams that their total consumption rate P exceeds the disk bandwidth R. In Chapter 5, this constraint can be relaxed by prefetching.

Second, t is inversely proportional to ρ. In other words, the longer the length of the period, the less utilized the disk becomes (which means the disk has more free time).

This is because as t increases, the proportion of time wasted in switching (i.e. $\frac{s}{t}$) within every period becomes smaller. In other words, a longer period corresponds to a higher percentage of useful work (reading) done by the disk, and the disk becomes more effective. Hence, the proportion of the idle disk time becomes higher. In Chapter 5, we will show how to make use of this relationship between t and ρ to maximize prefetching.

Figure 2.2 helps us to understand the relationship between disk rate R and cycle length t better. In this figure, the R and t relation curve is drawn under the condition that both P and ρ are fixed. It is easy to see that the requirement for disk rate R will decrease with the increase of period length t. As expected, the curve corresponding to $\rho = 0.5$ is above the curve for $\rho = 1$.

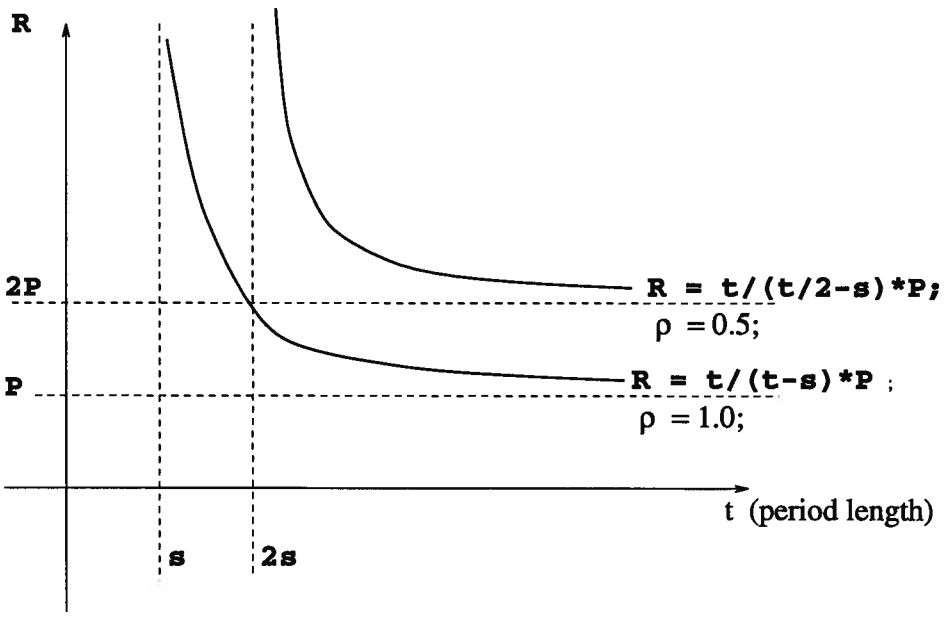

Figure 2.2: The curve of disk rate R and cycle length t relation

2.4 Buffer allocation: determining an upper bound for t

Thus far, we have analyzed the handling of multiple streams primarily from the viewpoint of disk bandwidth allocation, and we derived a formula to calculate a lower bound for t. There is, however, another very important issue: the allocation of buffers.

In this section, we will show how buffers should be allocated under the condition that no buffers will be shared among the streams. Sharing buffers among all the streams will be discussed in Chapter 4.

As stated earlier, in this thesis, the reading order of all the streams within a cycle is assumed to be fixed. Without losing generality, we assume that $S_1, S_2, ...S_n$ are being served in that order. If stream S_i is the ith stream being served in period 1, it will still be the ith stream being served in all of the following periods. Under this assumption, the next read for a stream is exactly t time units after the start time of the current reading of B_i in the stream. So, in each period, each stream must get enough data to last the time of t.[1]

As can be observed from Figure 2.3, for stream S_i, the maximum number of buffers is needed right after S_i has just finished reading. Thus, the number of buffers required by S_i is:

$$B_i = t_i \times R - t_i \times P_i \tag{2.8}$$

By substituting Equation 2.6 into the above, we get:

$$B_i = P_i \times (R - P_i) \times \frac{\rho \times t - s}{P} \tag{2.9}$$

[1]In a variable reading order scheme, this value might be as large as $2t$, since a stream can be served as the first one in the current cycle, but the last one in the next cycle.

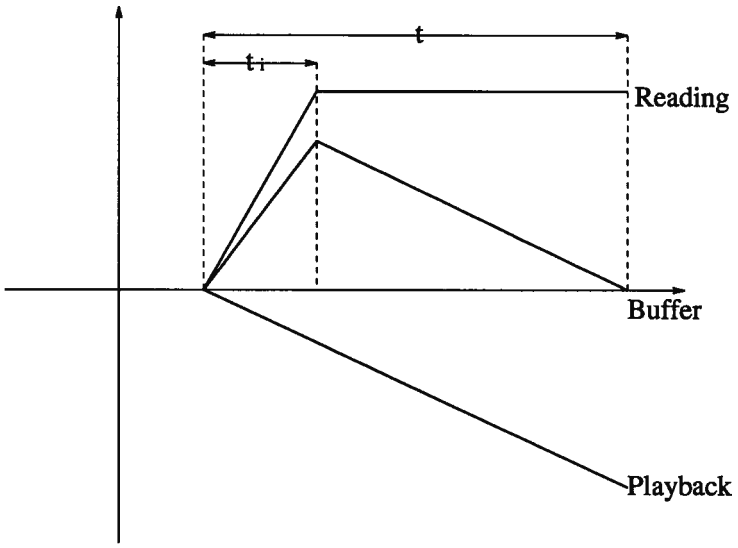

Figure 2.3: The buffer requirement curve

Thus, the total buffer requirement for the n streams is:

$$B = \sum_{i=1}^{n} B_i = \frac{\rho \times t - s}{P} \sum_{i=1}^{n} P_i \times (R - P_i) \tag{2.10}$$

Two important observations can be drawn from Equation 2.10. First, the exact number of buffers each stream needs can be calculated. If the buffer requirement cannot be satisfied, the stream set should not be accepted. On the other hand, giving more buffers than the minimum necessary to a specific stream does not help at all. All the extra buffers will be wasted.

Second, it is obvious that the longer the period length t, the higher the value of B will be. Since B must be bounded by the maximum available buffer B_{max}(i.e. $B \leq B_{max}$), an upper bound for t can be derived by substituting Equation 2.10:

$$t \leq \frac{B_{max} \times P}{\rho \times \sum_{i=1}^{n} P_i \times (R - P_i)} + \frac{s}{\rho} \tag{2.11}$$

Combining Equations 2.7 and 2.11, a simple admission control policy can be given.

Admission Control 1

Let $S_1, ..., S_n$ be all the streams in the current cycle, and let S_n be the stream to be decided whether admission is possible.

1. *Compute the lower bound for t using Equation 2.7 and the upper bound for t using Equation 2.11.*
2. *If the lower bound is strictly greater than the upper bound, then it is not possible to add S_n without violating the continuity requirements.*
3. *Otherwise, S_n can be admitted to form a new cycle, and any value between the lower and upper bound can be chosen as the length of the new cycle.*

When we discuss prefetching in Chapter 5, we will return to the issue of picking a value for t in this range.

2.5 Approximation of non-contiguously placed data

So far, we have assumed that for each stream, the data is contiguously placed on the disk, this means that there is no seek throughout t_i when Stream S_i is being read. This data placement policy is possible when using a spiral disk. However, it might not be practical in many other situations. In this section, we will show how to use the equations established so far to handle the case when data are not placed so perfectly.

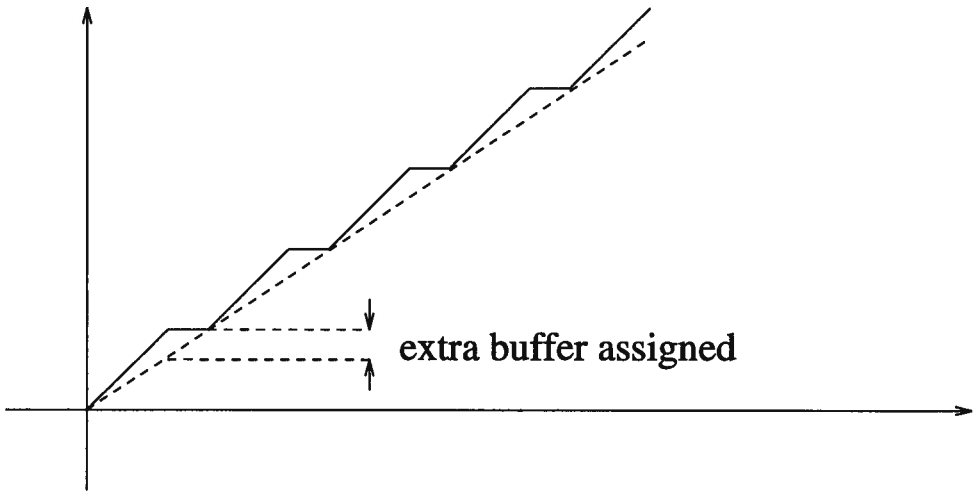

Figure 2.4: Using a continuous curve to approximate block reading

---

Consider a simple case first. Assume that data are stored in blocks, the block size is L, and the seek time between adjacent blocks is G. As shown in Figure 2.4, the solid line represents the reading curve of a stream of this kind. To use the equations that we have developed so far, we can approximate the given reading curve by one that assumes contiguous placement. This approximate curve is a straight line that is always below the original reading curve, but with a slope as large as possible. The dotted line in Figure 2.4 represents the approximate curve. By a simple coordinate-geometry analysis, we can calculate that the dotted line has a slope R_{approx} given by:

$$R_{approx} = \frac{L}{G + L/R} \tag{2.12}$$

This value can be used to replace R in all the equations that we have encountered so far.

Note that since the approximate reading curve is always below the actual reading curve, at various points in time, the disk may read faster than approximated. This leads to two observations. First, the continuity requirement will not be violated by the approximation, because the disk is never slower than approximated. Thus, there is no need to worry about starvation.

Second, extra buffers are needed to store the data that are retrieved faster than expected. See Figure 2.4 for an illustration. By another simple coordinate-geometry analysis, the number of extra buffers we need is given by:

$$B_{extra} = L - \frac{L^2}{L + G \times R} \tag{2.13}$$

B_{extra} is less than the block size L. In many cases, this is not a big burden to the system.

Similar approximations can be applied to other non-contiguous data placement situations. We omit those analyses here.

The above method is not the only method for approximation. There are many other ways to do it, too. For example, if short time overload (dropping one or two frames) is allowed, the approximation curve need not to be always below the real reading curve. It can pass right through the middle of the real reading curve. This method is, in some sense, more accurate than the above one.

2.6 Summary

In this chapter, the fixed reading order scheme, which is the basic framework this thesis is built upon, was described. First, after describing the general framework this thesis based on, we introduced the basic notations and formulas. Then, we analyzed disk scheduling

and buffer allocation, and gave the formulas for calculating cycle length t and buffer requirement B. Based on these analyse, we gave a simple admission control policy based on the upper bound and lower bound of t. Finally, in Section 2.5, we proposed a scheme for approximating non-contiguously placed data by a contiguous curve.

Chapter 3

Related Work

Much excellent work has been done in the area of multimedia file servers. With respect to the topics that will be discussed in this thesis, this work can be grouped into the following categories:

- Data placement
- Intelligent disk scheduling
- On-line scheduling and admission control

This chapter will give a brief summary of the work in these areas.

3.1 Data placement

The speed of retrieving multimedia data from a disk depends to a large extent on how the data are organized on the disk. Many researchers have studied how to wisely organize data on disk so that the retrieval can be performed under the constraint of the continuity requirement.

If there is only one stream on the disk, the solution is quite straightforward. Based on the consumption rate requirement, the block size (length of contiguously stored media) and scattering parameter (length between two adjacent blocks) can be calculated. Then all the data can be stored accordingly.

Several papers have discussed how to store two or more streams on the same disk.

When the streams are played back, the continuity requirement should be satisfied for all the streams.

Yu and his colleagues[Y+89] discussed how to merge two or more media streams on one disk in the strictest manner. The key idea in their method is trying to fit each stream into the other one's gaps, and the continuity requirement must be strictly satisfied at any time point for any stream. In their scheme, a logical block is not allowed to be broken into small pieces. This method does not need buffering and read-ahead. However, usually multiple media streams do not fit so well as to satisfy this merging condition.

Rangan and Vin[RV93] studied the same problem, but their approach is quite different. In their model, buffering and read-ahead are used to streamline the video stream so that the continuity requirement is relaxed. Their requirement is, in a certain time period constrained by the buffers, that the actual data read from the disk should be greater than the sum of the playback rates of all the streams. In addition, a logical block can be broken into small pieces to increase disk utilization. This method can achieve better disk utilization than Yu's approach (theoretically, it can approach 100%). However, the computing and buffering overhead should not be underestimated.

There is a serious drawback in both of the above methods: They both assume the stream accessing patterns to be fixed. For example, if two streams A and B are merged together, they are supposed to be played back at the same time. This is not practical in an interactive environment. If stream A is played back now and a request for stream B comes ten minutes later, the system may not be able to handle this situation properly and the continuity requirement could be violated.

Louhger and Shepherd[LS93] studied how to use a disk array (also called striped disks) to increase disk bandwidth. In their experimental implementation, they have one

dictionary disk and two data disks. The dictionary disk is used to store metadata. The real multimedia data are stored in the data disks. Editing is performed at the metadata level. The real multimedia data in data disks will never be changed after being placed.

3.2 Intelligent disk scheduling

Intelligent disk scheduling schemes like SCAN, C-SCAN, SSTF and EDF have been used in conventional file servers for a long time. Recently, several researchers have studied the possibility of using intelligent disk scheduling techniques in the multimedia file server environment [Gem93] [RW93] [CKY93].

Schemes like SCAN and SSTF can optimize disk arm movement, reduce seek time and improve system throughput. However, these schemes do not consider the special real-time requirements of multimedia requests.

Reddy and Wyllie[RW93] proposed an algorithm called SCAN-EDF. This algorithm combines the features of SCAN type of seek optimizing algorithms with Earliest Deadline First (EDF) type of real-time scheduling algorithms. The SCAN-EDF scheme guarantees that the real-time requirement of a user request will not be violated. Under this constraint, it tries to optimize disk arm movement.

The other two approaches [CKY93] [Gem93] are quite similar to the SCAN-EDF scheme. Those two schemes are called sort-set algorithm (SSA) [CKY93] and group sweeping scheduling (GSS) [Gem93]. The basic idea is: In a pure SCAN type scheme, the serving order of all the requests in the task set is rearranged in order to optimize the disk arm movement. If the task set is large, the serving time for a particular request may vary greatly in the current and the next cycle. Due to the uncertainty of serving time, sometimes the continuity requirement of the request may not be satisfied. In the

SSA and GSS schemes, the whole task set is divided into several groups. The groups are served in a fixed order. However, within each group, SCAN or some other scheme is used to optimize disk arm movement.

Although buffer requirements are analyzed in almost all these schemes, optimizing disk arm movement is the major goal of these schemes, not buffer usage.

3.3 Scheduling and on-line admission control

There are many papers that study the problem of scheduling and admission control in a multimedia file server environment. As first observed in [RV91] and [GC92], the best way to do scheduling in a multimedia file server is doing it in a round-robin (cyclic) manner. The key issue is how to decide the cycle-length and how to serve all the requests within each cycle. [RV91] and [GC92] only deal with a simplified scenario which assumes that all the users' requests have the same playback rates. In addition, they assume that all the data blocks have the same size. Under these conditions, a simple round-robin scheme is sufficient.

Rangan's group[RVR92] proposed a scheduling scheme called Quality Proportional Multi-subscriber Servicing (QPMS). According to this scheme, during each reading cycle, the data read for each user is proportional to its playback rate. However, QPMS statically test if a task set is schedulable or not. The problem of transient period is also introduced in this paper. The transient period is the length of time during which a new request is added to the server or a finished request is deleted from the server. In this period, the continuity requirements of all the already serving requests should not be violated. A simple scheme is given in this paper to deal with this problem. However, this simple scheme will usually result in a long transient period.

Andersion et al.[AOG92] described the design of a multimedia file system called CMFS. First, an interface is designed for the user. When the user submits a request, the system creates a "session". A session has such information as: file name, required data rate, buffer requirement, the amount of read-ahead allowed, etc. The CMFS uses this information to do an "acceptance test". If it passes the test, the system will accept the request and re-schedule all the "sessions" in the system. The key part of CMFS is the "acceptance test", which is done statically, and based on the worst-case situation. However, the actual scheduling algorithm uses some dynamic methods to improve performance.

Polimensis[Pol91] described a file system that supports multimedia data. Their approach of admission control is similar to the "acceptance test" in [AOG92]. A schedulable tasks set is constrained by two criteria: the total consumption rate should be less than the disk bandwidth, and the total read-ahead should be bounded by the available buffer space. These two factors are related to each other. A faster disk will require a smaller buffer space. On the other hand, more buffer space will usually reduce the number of seeks and increase the actual disk transfer rate. This paper gives a detailed analysis of these problems. The problem of transient period is also analyzed more thoroughly, and a good solution is given. However, as will be shown later, the approach of this paper is different from ours in two different ways. First, the buffer management policy is static and is performed per task, not dynamically and globally. Second, the author did not consider looking ahead in the request queue (prefetching).

3.4 Summary

As we noticed in Section 3.1, a static placement scheme does not work well if the accessing pattern is dynamic. For all the techniques described in Section 3.2, optimizing disk arm movement (seeking time) is the big concern, minimizing buffer usage does not get much attention. Section 3.3 describes several scheduling and admission control algorithms. Those algorithms all work in more or less the same manner, however, none of them studies sharing buffer among all the streams and looking ahead in the request queue (prefetching). In the following chapters, we will address the above limitations.

Chapter 4

Buffer Sharing

4.1 Benefits of buffer sharing

In the previous chapter, a scheme to do admission control and buffer allocation was established. Multimedia systems require a large amount of buffers. Given a fixed amount of buffer space, predetermined at design time, maximizing buffer utilization is a very important design goal. In this chapter, we will study how to share buffers among the served streams so that buffer utilization can be improved.

As defined in Equation 2.9, the total buffer requirement of n streams is based on the assumption that each stream S_i occupies B_i buffers in each cycle. However, S_i may not need all the B_i buffers at all that time during the cycle. B_i is just the peak buffer requirement.

Further study shows, that when one of the streams (say S_i) requires its maximum number of buffers, all the other streams do not need their maximum number of buffers at that moment. Thus, a simple way to minimize total buffer consumption and maximize buffer utilization is to allow the n streams to share buffers.

In the following sections, we will first show how buffer sharing works in a simplified situation in which all the streams have the same consumption rates. Then we generalize to the more general situation of buffer sharing among streams with different consumption rates. Some implementation considerations will be discussed at the end of this chapter.

4.2 A simple case: all the streams have the identical playback rates

First, we study a simple case in which all the streams have the same consumption rates. Figure 4.1 shows three streams S_1, S_2, S_3 in the cycle, all of which have the same consumption rate. Thus, by Equation 2.9, each stream has an equal amount of reading time, i.e. the same t_i. Since the cycle length t is normally much larger than the total switching time s, here we consider the simplified situation that $t_i = t/3$. Let us consider the total buffer requirement at time $4t/3$, at which point S_1 has just finished reading and requires b buffers, the maximum number of buffers that it ever needs. S_2, which is about to start reading, has run out of data. Thus, the buffer requirement of S_2 is 0. As for S_3, there were b buffers at time t, but at time $5t/3$, all the data in those buffers will be consumed. Thus, at the current time $4t/3$, S_3 needs $b/2$ buffers. Hence, the total number of buffers required by all the three streams is $b + 0 + b/2 = 3b/2$. Note that if all the streams have identical consumption rates (and we assume ρ equals 1), their total buffer requirement does not change with time. Thus, $3b/2$ buffers are all the three streams need. However, without buffer sharing, $3b$ buffers are required. Thus, buffer sharing gives a 50% reduction in total buffer consumption.

In the following, we will analyze the situation when there are n streams with identical consumption rates and the disk utilization is 1, which means each stream will get t/n time for reading and there is no disk idle time.

Since the consumption rates are the same for all the streams, the reading time t_i for each stream is also the same, say equal to t_0. Similarly, the buffer requirement B_i is the same, which is equal to b say. Now, consider the time when S_n has just finished reading. The buffer requirement for each stream is listed in the following table:

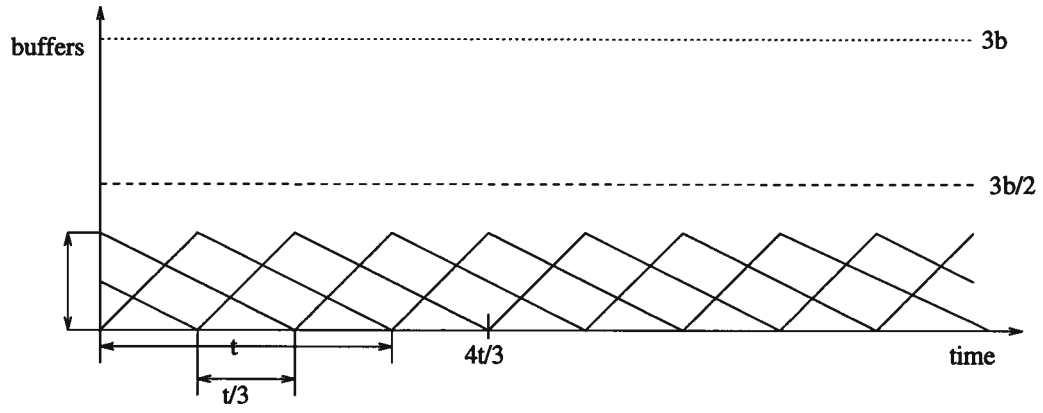

Figure 4.1: Buffer sharing for three streams with identical consumption rates

Stream	Buffer needed
S_1	0
S_2	$\frac{1}{n-1}b$
S_3	$\frac{2}{n-1}b$
...	...
S_n	$\frac{n-1}{n-1}b$

First, S_n has just finished reading, thus it requires all b buffers. S_1 is about to start reading, so it has 0 buffered data at this time. S_2, at an earlier point of time, had b buffers of data which are supposed to cover the consumption of S_2 for a period of $(n-1) \times t_0$. At the point when S_n has just finished reading, $(n-2) \times t_0$ has elapsed, or alternatively, S_2 will run out of data t_0 seconds later. Thus, the current level of buffered data for S_2 is $\frac{t_0}{(n-1)\times t_0}b = \frac{1}{n-1}b$. Similarly, it is not difficult to see that the current level of buffered data for S_3 is $\frac{2}{n-1}b$. Hence, the total buffer for all the n streams is:

$$B_{shar} = \sum_{i=1}^{n} \frac{i-1}{n-1} b = \frac{n}{2} b \tag{4.1}$$

Since the buffer requirement will not change with the time [1], the above number will be the maximum buffer requirement of the n streams. Remember in the no sharing case, the buffer requirement is nb, thus buffer sharing achieves a 50% savings in buffer space when $\rho = 1$.

Now, we relax the assumption that $\rho = 1$, and study buffer sharing in a more general situation. We still calculate the buffer required for each stream at the time S_n has just finished reading. The above table can be generalized to:

Stream	Buffer needed
S_1	cb
S_2	$(c + \frac{\rho}{n-\rho})b$
S_3	$(c + \frac{2\rho}{n-\rho})b$
...	...
S_n	$(c + \frac{(n-1)\rho}{n-\rho})b$

In this table, $c = \frac{1-\rho}{1-\rho/n}$. As we will prove later, if $\rho \neq 1$, the maximum buffer requirement will not be a constant throughout the whole period. However, it will happen at the moment S_n has just finished reading. Sum the numbers in the table together, we calculate the total buffer requirement as follows:

$$B_{shar} = \sum_{i=1}^{n} (cb + \frac{(i-1)\rho}{n-\rho} b) = \frac{2n - n\rho - \rho}{2(n-\rho)} \times nb \tag{4.2}$$

[1]Under the condition $\rho = 1$. We will prove this later.

Notice, if $\rho = 1$, B_{shar} will be $nb/2$, which means 50% savings of buffer space happens when the disk utilization is 1, agreeing with Equation 4.1. Moreover, we can see, as ρ decreases, B_{shar} will increase, which means 50% is the maximum saving we can achieve using buffer sharing.

Using Equation 2.9, we can substitute b into Equation 4.2. The total buffer requirement can then be expressed as:

$$B_{shar} = (R - \frac{P}{n}) \times (t \times \rho - s) \times \frac{2n - n\rho - \rho}{2(n - \rho)} \tag{4.3}$$

This equation can replace Equation 2.10 (and thus Equation 2.11) in the admission control test shown in Section 2.4.

This completes the analysis of the case in which all the streams have identical consumption rates. Simulation result of this buffer sharing scheme will be discussed in Chapter 7.

4.3 General case

In this section, we will discuss buffer sharing in the most general case. The constraint that all the streams have the identical playback rates will no longer be required. First, we introduce some additional notation that will be used in this section.

B_i is the buffer requirement of stream S_i in no sharing case BS_i is the buffer requirement of stream S_i in the sharing case, and BA_i is the total buffer requirement for all the streams at the time S_i has just finished reading.

Recall the formulas in Chapter 2, without buffer sharing, the maximum buffer requirement for stream S_i happens at the end of reading of S_i, and that amount of data will last $t - t_i$ time units.

Let P_i be variable and let disk utilization ρ be less than 1, but still assume in each period the reading order of all the streams is

$$S_1 \mid S_2 \mid S_3 \mid ... \mid S_n \mid idletime$$

At the beginning of the cycle (which is the time when S_1 just starts reading), the buffer requirement for all the streams are:

Stream	Buffer needed
BS_1	0
BS_2	$\frac{t_1}{t-t_2}B_2$
BS_3	$\frac{t_1+t_2}{t-t_3}B_3$
...	...
BS_n	$\frac{t_1+t_2+...+t_{n-1}}{t-t_n}B_n$

The above formulas are easy to derive. S_1 is about to start reading, it has no data left, thus it requires no buffer. The buffered data for S_2 will last $t-t_2$ time units, and its reading is still t_1 time units away. Right then the amount of buffered data is $t_1/(t-t_2)$. The buffer requirements for all the other streams can be calculated in a similar way. Suming the buffer requirements for all the streams together, the total buffer requirement at the time S_0 is about to start reading will be:

$$BA_0 = \sum_{i=1}^{n} BS_i = \sum_{i=2}^{n} \frac{\sum_{j=1}^{i-1} t_j}{t-t_i} B_i \tag{4.4}$$

After deriving BA_0, it is not difficult to derive BA_1, BA_2, ... To calculate these values, first we can compare the buffer requirement at the time S_i finishes reading and S_{i+1} finishes reading, and then we compute the difference.

First, at the time S_1 finishes reading, the buffers required by S_1 will be increased

from 0 to B_1. However, for all the other streams, since t_1 time was elapsed, their buffer requirement will decrease by $B_j \times t_1/(t - t_j)$. Thus,

$$BA_1 = BA_0 + B_1 - \sum_{j=2}^{n} \frac{t_1}{t - t_j} B_j \tag{4.5}$$

More generally,

$$BA_{i+1} = BA_i + B_{i+1} - \sum_{j=1, j\neq i}^{n} \frac{t_{i+1}}{t - t_j} B_j \quad 1 \leq i \leq n \tag{4.6}$$

Using Equations 4.4 and 4.6, $BA_1, BA_2, ..., BA_n$ can be calculated easily.

The maximum of these values will be the total buffer requirement (we do not need to count BA_0, it is always equal to or less than BA_n). That is to say,

$$B_{shar} = Max(BA_i) \qquad (i \in [1, n]) \tag{4.7}$$

After deriving this more general formula, we can verify the result presented in Section 4.2. If all P_i are the same, it is easy to see that BA_{i+1} is always greater than or equal to BA_i. That is to say, the maximum buffer requirement always happens at the end of the reading of S_n (when ρ is 1, the buffer requirement is a constant). A detailed derivation of these is as follos.

Assuming $B_i = b$ for all i, then Equation 4.4 becomes:

$$BA_0 = \frac{n(n-1)\rho}{2(n-\rho)} b \tag{4.8}$$

and Equation 4.6 becomes:

$$BA_{i+1} = BA_i + \frac{n - n\rho}{n - \rho} b \tag{4.9}$$

If $\rho = 1$, then $BA_{i+1} = BA_i$, which means the buffer requirement remains a constant through the whole cycle.

If $\rho \neq 1$, then $BA_{i+1} > BA_i$. So the maximum buffer requirement happens at the end of S_n finishes reading, i.e.

$$B_{max} = BA_n = BA_0 + n \times \frac{n - n\rho}{n - \rho} b = \frac{2n - n\rho - \rho}{2(n - \rho)} \times nb \tag{4.10}$$

This is exactly the same result that we derived in Section 4.2 (Equation 4.3).

This completes the analysis of buffer sharing in the general case. These formulas are somewhat complicated. In order to calculate B_{shar}, all of the BA_i must be calculated first, then the maximum is selected. In a real system, computing overhead may affect performance slightly. It is interesting observe that if the streams are served in ascending order according to their playback rates, the maximum value will happen at the end of S_n finishing reading. If this is the case, BA_n will be B_{shar}, and no comparison is needed.

If the buffer requirement is the only concern, we may be lead to believe that reordering the streams in a cycle is a good thing to do. However, as we will show later, that this will be too expensive if we also need to consider prefetching and handling transient period at the same time [2]. In most cases, it is not worth doing.

The implementation of a buffer sharing scheme is not a trivial task. The difficulty lies in the fact that the buffers used by all the streams are changing from time to time. In other words, each stream does not have a dedicated buffer set. A study of the implementation issues is under way, but is beyond the scope of this thesis.

[2]Prefetching and transient period will be discussed in more detail in Chapter 5 and 6, respectively.

4.4 Summary

In this chapter, buffer sharing among all the served streams in the fixed reading order scheme was analyzed. Buffer sharing in the simplified scenario in which all the streams have the same playback rates was first studied. After that, the analysis was extended to buffer sharing in the more general case in which playback rates can be different. We showed that by sharing buffers, the total buffer requirement can be reduced by up to 50%.

Chapter 5

Prefetching

In this chapter, prefetching as a scheme to improve disk and buffer utilizations is introduced. First, the benefits of prefetching are discussed in Section 5.1. Then, a simple straightforward prefetching scheme SP is described in Section 5.2. As our analysis and simulation results show, SP will not give satisfactory performance. An intelligent prefetching scheme IP1 to overcome the shortcomings in SP is described in Section 5.3. Finally, IP2 is discussed in Section 5.4 as a further refinement of IP1. Our analysis shows that intelligent prefetching strategies IP1 and IP2 can improve the performance of a multimedia file server dramatically.

5.1 Benefits of prefetching

The admission control and scheduling schemes we have discussed so far do not consider prefetching of data. On receiving a new request for a stream (referred to as a new query from now on), the admission controller that we have discussed so far simply checks if there is enough disk bandwidth and buffer space to accept the new query, using Equations 2.7 and 2.11. If there are enough resources, the query is activated. Otherwise, the query will be left idle in the waiting queue. Consequently, there are resources – buffers and disk bandwidth – that are not utilized at all.

We first consider a simplified example. Suppose there is a server whose disk's maximum reading speed is 1000KB per second. Furthermore, suppose that all the requests

have the same consumption rate equal to 260KB per second. If we do not consider switching time and there are always enough buffers, the system can support 3 streams simultaneously. That will use $260 \times 3 = 780$KB/s disk bandwidth. Consequently, 220KB/s disk bandwidth will be wasted. If there are enough buffers, we can make use of this wasted disk bandwidth to do some prefetching for the next waiting request. By doing this, the prefetched request will have a better chance of being admitted (since its actual consumption rate will be reduced after prefetching). For the same reason, the server will have more free disk bandwidth to accept more new requests.

Usually, the system performance of a multimedia file server is measured by throughput and query response time. For a system with pre-determined (at design time) disk bandwidth and amount of buffer space, the response time and throughput are primarily determined by the utilization of disks and buffers. Our goal here is to try to use these resources as much as possible. More specifically, we will explore how data prefetching can maximize resource utilization, and thus lead to an increase in system throughput.

Intuitively, there are several ways that prefetching can help a query:

- First, if a query has a consumption rate P_i that is larger than R, then the play back of that query cannot be started immediately after its reading begins. In this case, prefetching some data ahead of time is a must.
- Second, prefetching portions of a query before activation has the effect of reducing the effective consumption rate of the query after activation. This is illustrated in Figure 5.1. The solid line represent the original consumption rate P_i. If an amount pf is prefetched, then the new, prefetched consumption rate is given by the slope of the dotted line. A simple analysis shows that if T_i is the length of the query, the consumption rate after prefetching is given by:

$$P_i^{pft} = P_i - \frac{pf}{T_i} \tag{5.1}$$

Since the new rate is less than the original rate, there is a possibility that the new rate may pass the admission control test, while the old one cannot. Whenever this happens, the response time of the query is substantially reduced.

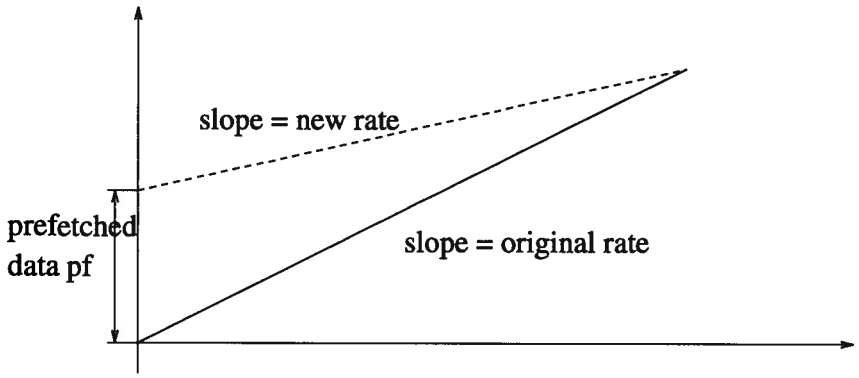

Figure 5.1: Reducing the consumption rate by prefetching

- Even if the above conditions cannot be applied, for a normal query, prefetching can still help the query reduce its response time by one cycle. To see this, assume that $S_1, S_2, ..., S_n$ are the active queries. At some time point, S_1 has finished and S_{n+1} is activated. For the reason of *transient period* (which will be discussed in Chapter 7), the reading order for the new set should be $S_2, ..., S_{n+1}$. If no data has been prefetched for S_{n+1}, then its playback cannot be started until it gets a chance to start reading, which is at the end of the cycle. However, if there is a sufficient amount of prefetched data for S_{n+1}, its playback can be started immediately at the

beginning of the cycle. Thus, there is a difference in response time which may be as large as one cycle length.

The above claims will be shown more formally in the following sections. Our analysis will mainly concentrate on the second situation. First, we will discuss how to decide cycle length t in the prefetching situation, and we will give a straightforward prefetching strategy SP. Then after observing that the effectiveness of SP may be hindered by several shortcomings, we will develop another prefetching strategy IP1 which tries to maximize overall system throughput. However, there are too many intuitive and heuristic factors in the strategy IP1. We will further analyze thoroughly what is happening in the system and give a more formal IP strategy called IP2. Interestingly, as will be shown in Chapter 7, IP2 does not significantly outperform IP1, which means the heuristic used in IP1 is good enough in most cases.

5.2 Straightforward prefetching strategy (SP)

Just like normal data retrieval from disk, prefetching requires both disk bandwidth and buffers. One obvious way to allow prefetching is to dedicate a certain level of disk bandwidth and buffers to prefetching. But this is not a good solution as it will reduce the disk bandwidth and buffers available to activated queries. We should make sure that prefetching is not done at the expense of activated queries. To this end, recall that the cycle length t for the activated queries (streams) $S_1, ..., S_n$ are bounded below and above, respectively, by Equations 2.7 and 2.11. If the system does not support prefetching at all, any value between this range can be picked as the value of t. However, to support prefetching, an immediate question to answer is how to pick t so as to maximize

prefetching, but not at the expense of the activated queries.

Setting t to any value between the upper and lower bounds does not have any influence whatsoever on the completion times of the activated queries, as the completion time of a query is determined by its consumption rate and length. However, the value of t will affect prefetching.

First, consider setting t to its lower bound. As discussed in Chapter 2, this corresponds to set disk utilization ρ to 1. In other words, all the disk bandwidth is used up for the activated queries, and there will be nothing left for prefetching. On the other hand, consider setting t to its upper bound. From the point of view of disk bandwidth allocation, this time there is ample room for prefetching. As discussed in Chapter 2, a longer cycle length corresponds to a lower disk utilization ρ. However, the trouble is that all the buffers will be used up for the activated queries. Thus, at the end there will be no prefetching done either, not because of lack of disk bandwidth, but because of lack of buffer space. Hence, the problem to address is which value of t in between the upper and lower bounds maximizes prefetching. Since the disk utilization to cycle length(ρ *vs* t) and buffer utilization to cycle length(B *vs* t) functions are unlikely to be linear, just selecting the average of upper bound and lower bound may not be a good decision.

Figure 5.2 can help us to understand this better. In Figure 5.2, we define disk utilization U_{disk}(actually ρ) and buffer utilization U_{buffer} as:

$$\begin{aligned} U_{disk} &= \frac{s}{t} + \frac{P}{R} \\ U_{buffer} &= \frac{t}{R \times B_{max}} \sum_{i=1}^{n} P_i \times (R - P_i) \end{aligned} \tag{5.2}$$

We plot how they change with cycle length t. For both U_{disk} and U_{buffer}, we draw the curves when there are n requests and $n+1$ requests being served. First we can see

from the graph that if U_{disk} approaches 1, even though at this time U_{buffer} is low, which means there are many buffers left for prefetching. Since the disk is too busy, we cannot do very much prefetching. A similar situation happens when U_{buffer} approaches 1. Second, if the system load is increased (more requests are being served at the same time), the choices for t will be narrowed, which also means that the room left for prefetching will be reduced.

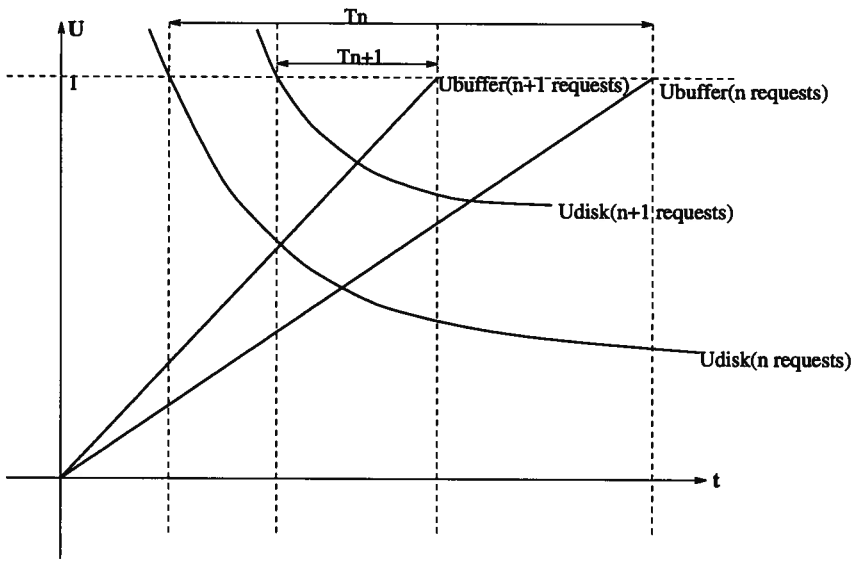

Figure 5.2: How to decide the reading period length t for prefetching

We can see from the above discussion that there is no obvious way to get a good value of t for prefetching. The main reason for this is that the information we considered so far is not sufficient. Actually, there is another factor that affects the amount of prefetching that can be done. All the above analysis is based on the assumption that the cycle for the current set of activated queries keeps on going. Let T_{finish} denote the time the earliest

next activated query will terminate. We should notice that the range bounding t is only valid before T_{finish}. After that time the system state will be changed, and a new cycle will be recalculated. If we select prefetching as much as possible before T_{finish} as the goal, this leads to a straightforward prefetching strategy SP. In addition, it suggests a way to calculate t. This is formalized as follows.

First, if we consider just the free disk bandwidth, it is obvious that the amount of data that can be prefetched in time T_{finish} is:

$$D_{prefetch} = T_{finish} \times R \times (1 - \rho) \tag{5.3}$$

This means we can only use the free disk bandwidth $(1 - \rho)$ to do prefetching. By substituting ρ to Equation 5.3, there is:

$$D_{prefetch} = T_{finish} \times R \times (1 - \frac{s}{t} - \frac{P}{R}) \tag{5.4}$$

On the other hand, if we consider just the available free buffers, the buffers available for prefetching should be:

$$B_{prefetch} = B_{max} - B = B_{max} - \frac{t}{R} \sum_{i=1}^{n} P_i \times (R - P_i) \tag{5.5}$$

$D_{prefetch}$ increases with t, and $B_{prefetch}$ decreases with t. Furthermore, it is necessary that $D_{prefetch} \leq B_{prefetch}$. Thus, to maximize prefetching, we should let $D_{prefetch} = B_{prefetch}$, which is equivalent to:

$$T_{finish} \times R \times (1 - \frac{s}{t} - \frac{P}{R}) = B_{max} - \frac{t}{R} \sum_{i=1}^{n} P_i \times (R - P_i) \tag{5.6}$$

This is a quadratic equation which is equivalent to

$$a \times t^2 + b \times t + c = 0 \tag{5.7}$$

where the coefficients are:

$$\begin{aligned} a &= \frac{\sum_{i=1}^{n} P_i \times (R - P_i)}{R}, \\ b &= TR - TP - B_{max}, \\ c &= -TRS. \end{aligned}$$

Solving this quadratic equation will give a positive solution v_0 (and a negative solution that we ignore). If v_0 falls within the lower and upper bounds of t, which occurs more often than not, v_0 is the value of t. Otherwise, if v_0 is strictly less than the lower bound, t will be set to the lower bound. And if v_0 is strictly greater than the upper bound, the upper bound becomes the value of t.

The equations presented above do not assume buffer sharing; the calculation of buffers is based on Equation 2.10. Since prefetching is orthogonal to buffer sharing, a similar set of equations can be derived for the buffer sharing case based on Equation 4.7.

We summarize the straightforward prefetching strategy SP as follows.

Strategy SP

Let $S_1, ...S_n$ be all the activated queries, as allowed by the admission controller. Let S_{n+1} be the query at the head of the waiting queue.

- *Use Equation 5.7 to determine the length t of the cycle for $S_1, ...S_n$.*
- *Use the remaining disk bandwidth and buffers to prefetch for S_{n+1} at the end of each cycle.*

- *Prefetching stops when an activated query has finished, or the system has run out of buffers, or the entire query has been read into the memory.*

5.3 Intelligent prefetching strategy 1 (IP1)

The straightforward prefetching strategy SP can maximize prefetching for the query at the head of the waiting queue. However, as a result, S_{n+1} may use up too many system resources (particularly free buffers), for its own good, but not necessarily for the overall performance of the system. More specifically, SP just lets S_{n+1} prefetch as much as possible, but does not consider whether S_{n+1} really needs that much data or not. This may have two bad effects. First, since accepting a new query always needs some working buffers, this may affect the possibility of accepting a new query. Second, it also reduces the possibility of prefetching for another query. Because of this, the data prefetched for a query should be just enough for its activation.

The above discussion indicates that while maximizing prefetching, SP's approach of prefetching as much as possible for the query at the head of the waiting queue may not be sufficient. Naturally, the next question that needs to be answered is: Considering all the queries in the waiting queue, if there is a chance, which query should be selected on which to perform prefetching? Using the margin gain analysis technique used in [NFS91], we can show that if First-Come-First-Serve for admission control still applies, then given the same amount of buffer space, prefetching for a shorter query can give greater benefit than for a long query, which means prefetching for the head of the waiting queue may not be a good decision. This is illustrated by the following example.

Consider a situation where there are four activated queries (S_1, S_2, S_3, S_4) being served, each with a consumption rate 240KB/s. S_1 will finish in 10 seconds, but the

other three will last longer. The disk has a maximum reading rate of $R = 1150KB/s$. There is 1MB free buffer space left at the current moment. S_5 and S_6 are two queries in the waiting queue, both having a consumption rate 240KB/s. S_5 is 30 seconds long, and S_6 is 15 seconds long. We will show what will happen if we prefetch either S_5 or S_6.

If we decide to prefetch data for S_5, the maximum amount that can be prefetched is 1MBs. By Equation 5.1, the consumption rate of S_5 can be reduced to $240 - 1000/30 =$ 207KB/s. At the time S_1 finishes, the total consumption rate of all the other queries will be $3 \times 240 + 240 + 207 = 1167$KB/s, which is greater than the maximum disk reading rate. This means that by prefetching S_5, we cannot admit both S_5 and S_6 when S_1 finishes. So consider prefetching S_6 instead. By prefetching 1MB, the consumption rate of S_6 can be reduced to $240 - 1000/15 = 173$KB/s. So the total consumption rate will only be $3 \times 240 + 240 + 173 = 1133$KB/s. This is less then 1150, which means that when S_1 finishes, we can admit both S_5 and S_6 at the same time. □

To address the problems in SP, we propose the following new strategy. With this strategy, we will find the shortest query to prefetch, so as to maximize prefetching and the number of queries that can be activated once an active query has completed.

Strategy IP1

Let $S_1, ..., S_n$ be all the activated queries, as allowed by the admission controller. Among them, let S_j $(1 \leq j \leq n)$ be the query that will finish the earliest. Also let $S_{n+1}, S_{n+2}, ...$ be the queries in the waiting queue, and let B_{free} be the total number of buffers available for prefetching.

The algorithm works as the following:

1. **Calculate** t

   Use Equation 5.7 to determine the length t of the cycle for $S_1, ..., S_n$.

2. **Initialization**

   Set target *to* S_{n+1}, candidiateSet *to* S_{n+1}, *and* finalAmt *to 0.*

3. **First chance**

   If the combined consumption rate of all the streams in candidiateSet *is not greater than the consumption rate of* $S_j (i.e. P_j \geq \sum_{S_k \epsilon candidiateSet} P_k)$, *go to Step 6.*

4. **Second chance**

   Otherwise,

   (a) *Calculate the necessary prefetched consumption rate* P^{pft}_{target} *of* target *so that all the streams in* candidiateSet *can possibly be activated when* S_j *has finished, i.e.* $P^{pft}_{target} + \sum_{S_k \neq target; S_k \epsilon candidiateSet} P_k \leq P_j + (1 - \rho) \times R$.

   (b) *Use Equation 5.1 to calculate the amount that needs to be prefetched in order to reduce the consumption rate of* target *to* P^{pft}_{target}, *i.e.* targetAmt $= (P_{target} - P^{pft}_{target}) \times T_{target}$.

   (c) *If* $targetAmt > B_{free}$, *then go to Step 5 to try the next condition.*

   (d) *Otherwise, use the admission control test given in Chapter 2 to determine if all streams in* candidiateSet, *including the prefetched one, can fit into a cycle with all the current activated queries except* S_j. *If the admission control test fails, go to Step 5.*

   (e) *Otherwise, set* finalTarget *to* target *and* finalAmt *to* targetAmt. *Go to Step 6.*

5. **Third and final chance**

(a) Set targetAmt *to* B_{free}.

(b) Use Equation 5.1 to calculate the prefetched consumption rate after prefetching: $P_{target}^{pft} = P_{target} - \frac{targetAmt}{T_{target}}$.

(c) Use the admission control test given in Chapter 2 to determine if all streams in candidiateSet, *including the prefetched one, can fit into a cycle with all the current activated queries except* S_j. *If the admission control test fails, go to Step 7.*

(d) Otherwise, set finalTarget *to* target *and* finalAmt *to* targetAmt. *Go to Step 6.*

6. **See if more queries can be activated**

Consider the next query S_{next} *in the waiting queue that is not in* candidiateSet. *Add* S_{next} *to* candidiateSet. *Compare the length of* S_{next} *with the length of the* target. *Set the* target *to be the stream with shorter length. Go back to Step 3.*

7. **No more queries can be activated**

If $finalAmt > 0$, *prefetch* finalTarget *for the amount of* finalAmt.

□

The above algorithm is fairly lengthy. We will explain it in more detail.

Step 1 is to calculate t. This calculation is separated from the remaining part of the algorithm, which means changing the admission control policy will not affect the calculation of t.

Step 2 is the initialization part. *Target* will be the query we perform prefetching on. *CandidateSet* is our working set. We consider all the queries in this set as a whole

when doing admission control. And $finalAmt$ is the amount of data we will prefetch for $target$.

The purpose of introducing *candidateSet* is to enforce FIFO in the activation of queries. If S_k is before S_{k+1} in the waiting queue, we should not admit S_k later than S_{k+1}. Even though it is highly possible we will prefetch S_{k+1} instead of S_k, if that is the case, S_{k+1} and S_k must be accepted together. Breaking FIFO will give better chances to improve performance, but it also introduces many other problems like fairness, deadlock prevention, etc. These are not discussed in this thesis.

Steps 3—6 are the main part of the algorithm. In each iteration of IP1, a new query is added to the *candidateSet*, and the shortest one in the *candidateSet* is selected as *target* for possible prefetching. If all the queries in the *candidateSet* can pass the admission test as a whole, a new iteration begins; otherwise, the algorithm stops. There are three possibilities for all the queries in the *candidateSet* to be activated once S_j has completed (steps 3, 4 and 5):

- First (Step 3), if the combined consumption rate of all the queries in *candidateSet* does not exceed the consumption rate of S_j, all the queries in *candidateSet* are guaranteed to be activated once S_j has completed. In addition, nothing needs to be prefetched in this case. Execution goes to Step 6 to see if there are more queries in the waiting queue that can be activated.
- Second, if the first condition fails, execution goes to Step 4. In this case, IP1 tests if a sufficient amount of *target* can be prefetched so that all the queries in *candidateSet* can be activated, provided that this amount of data does not exceed the number of buffers currently available for prefetching (Step 4c). If admission control in Step 4d verifies that all queries can be activated with the help of prefetching, both *target*

and the prefetching amount *targetAmt* are recorded in the variables *finalTarget* and *finalAmt*. Execution then goes to Step 6 to try to add another query from the waiting queue to *candidateSet*, and a new iteration begins.

- Third, if both of the above steps fail, IP tries the "last resort". It simply tests if using all free buffers to prefetch for *target* will be sufficient to activate all queries in *candidateSet*. If admission control returns a positive answer, all the necessary work will be done in Step 5d and Step 6, and a new iteration begins.

If all of the above steps fail, this means not all the queries in the *candidateSet* can be activated. More precisely, all but the last added query in *candidateSet* can be activated once S_j has completed. Step 7 prepares for this event by correctly setting *finalTarget* and *finalAmt*.

The control flow of SP1 is illustrated in Figure 5.3.

At the beginning of this chapter, we listed three possibilities for how prefetching can help a query. However, algorithm IP1 only handles Case 2: reduce the effective consumption rate. When this is impossible, prefetching data for one cycle could at least improve the response time to some degree. This can be considered as an improvement to algorithm IP1. However, this improvement is not as significant as reducing effective consumption rate.

In the above algorithm IP1, if everything fails, the scheduler will not do any prefetching. Actually, it might be a good idea to apply SP if this happens. Although this prefetching will not help the query to be admitted right away, there is a possibility it will save the system some disk bandwidth in the future, thus improving performance.

As we noticed earlier, buffer sharing is orthogonal to prefetching. It is not difficult to combine buffer sharing and prefetching together. We omit the result for simplicity.

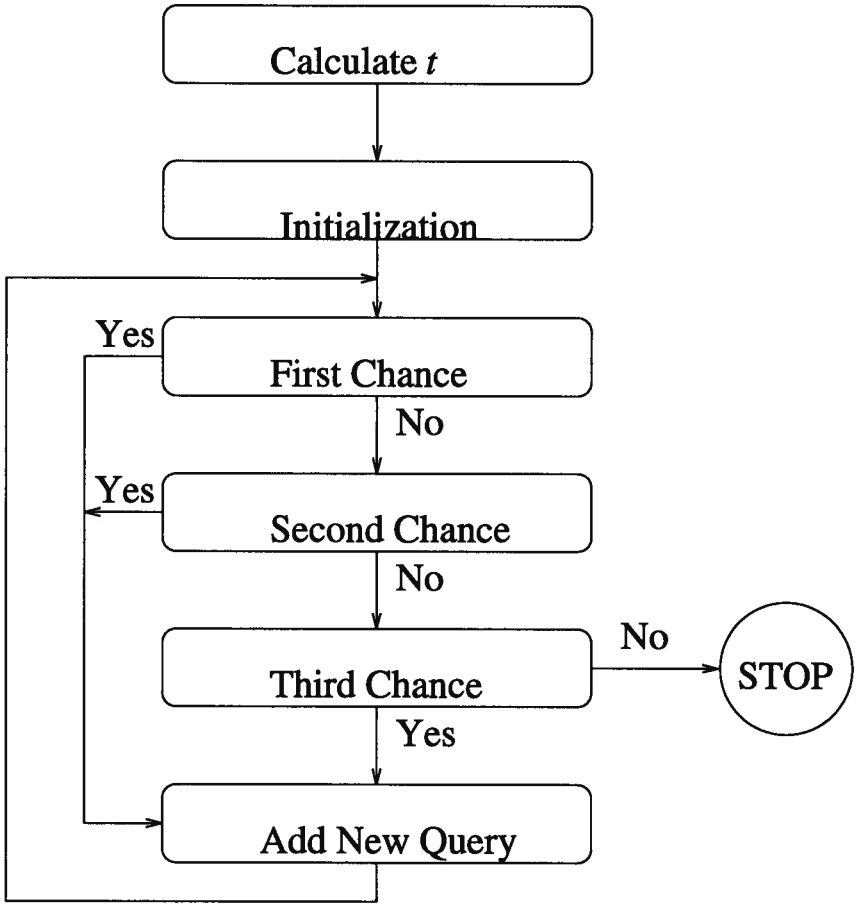

Figure 5.3: The control flow of IP1

5.4 Intelligent prefetching strategy 2 (IP2)

An intelligent prefetching strategy (IP1) was introduced in last section. In contrast to SP, which always prefetch the first query in the waiting queue, IP1 tries to prefetch the shortest possible query in the waiting queue, so that system performance can be improved. However, closer study of strategy IP1 reveals some problems. First, in Step 3, if we take switching time into consideration, the condition $P_j \geq \sum_{S_k \epsilon candidiateSet} P_k$ is not

sufficient to guarantee an admission. The reason is: if we let more than one query in, the switching time of these queries may be much greater than what S_j used. Similarly, in Step 4, condition $P^{pft}_{target} + \sum_{S_k \neq target; S_k \epsilon candidiateSet} P_k \leq P_j + (1-\rho) \times R$ cannot guarantee an admission either. This means the value calculated in Step 4(a)(b) might not pass the test in Step 4(d). A similar situation may occur in Step 5.

Thus, the next question is: How to decide exactly if a query can be acceptted or not at the time the prefetching amount is calculated? And what is the optimal amount to prefetch?

These are hard questions to answer, for reasons presented next.

First, we have to keep track of all the system buffers. In order to accept a new request, we need some working buffers for cyclic reading, and we may also need some buffers for prefetching. The buffers available now can be used for either prefetching or cyclic reading. The buffers released at T_{finish} by S_j can only be used for cyclic reading. The reason is that those buffers will not be released until T_{finish}, but prefetching is performed before that time. There might also be some buffers released between now and T_{finish} (it depends on if the prefetching buffers are released gradually or at the termination of the query). Keeping track of all these buffers is complicated.

Second, even if the above problem is solved, the story is still not finished. The remaining question is: What is the optimal amount of data to prefetch? On one hand, prefetching can reduce the consumption rate for a query, so the query will have a better chance to be admitted. On the other hand, if we use too many buffers for prefetching, we may not have enough buffers left for cyclic reading. Because the total number of buffers is fixed, the problem is how to balance between these two extremes.

To simplify the analysis, we assume that the prefetching buffers are released at the

termination of the query. Let the total system buffer space be B, the buffers used for cyclic reading be B_c, and the buffers occupied by prefetched data be B_p. The number of free buffers will be:

$$B_{free} = B - B_c - B_p \tag{5.8}$$

Assume that the number of prefetching buffers that will be released at T_{finish} by S_j is B_j.

Suppose the new request to be admitted is S_{new}, its playback rate is P_{new}, and its length (duration time) is l. Let D be the amount of data that will be prefetched for S_{new}. Now the question is how to decide on a value of D so that P_{new} is maximized.

First, the modified consumption rate of the query after prefetching is:

$$P'_{new} = P_{new} - D/l \tag{5.9}$$

The amount of buffers left for cyclic reading at T_{finish} is:

$$B'_c = B - B_p + B_j - D \tag{5.10}$$

In order to admit S_{new}, the first condition is that $D \leq B_{free}$. This is equivalent to

$$D \leq B - B_c - B_p. \tag{5.11}$$

In addition, the admission control criteria defined in Equations 2.7 and 2.11 must be satisfied, i.e.

$$t \geq \frac{s \times R}{R - P}$$

$$t \leq \frac{R \times B'_c}{\sum_{i=1}^{n-1} P_i(R - P_i) + P'_{new}(R - P'_{new})}. \tag{5.12}$$

In order to obtain a valid t, the upper bound should be greater than the lower bound, i.e.

$$\frac{s \times R}{R - P} \leq \frac{R \times B'_c}{\sum_{i=1}^{n-1} P_i(R - P_i) + P'_{new}(R - P'_{new})}. \tag{5.13}$$

The total consumption rates of all the queries is:

$$P = \sum_{i=1, i \neq j}^{n} P_i + P'_{new} \tag{5.14}$$

Substitue this value for P into Equation 5.13, we obtain:

$$B'_c - \frac{s}{R - \sum_{i=1}^{n-1} P_i - P'_{new}} (\sum_{i=1}^{n-1} P_i(R - P_i) + P'_{new}(R - P'_{new}) > 0 \tag{5.15}$$

which is equivalent to the following quadratic equation:

$$aD^2 + bD + c > 0 \tag{5.16}$$

where the coefficients are:

$$\begin{aligned} a &= \frac{s}{l^2} - \frac{1}{l}, \\ b &= -R + \sum_{i=1}^{n-1} P_i + P_{new} + \frac{B - B_p + B_j + sR - 2sP_{new}}{l}, \\ c &= (B - B_p + B_j)(R - \sum_{i=1}^{n-1} P_i - P_{new}) - s(P_{new}(R - P_{new}) + \sum_{i=1}^{n-1} P_i(R - P_i)). \end{aligned}$$

Solving this equation, and Equation 5.11, we can obtain a range for D that maximizes P_{new}.

Thus, we have finished the presentation of how to calculate D. This calculation can replace Steps 4 and 5 in IP1. More important, by using this equation, an admission control strategy can be guaranteed to be successful if a valid (positive) solution of D is obtained.

Strategy IP2

Replace Step 4 and 5 in IP1 by:

Calculate D using Equation 5.16. □

The above formula seems quite complicated. However, the computing overhead is actually much smaller than IP1. The reason is that we eliminate all the failed attempts when doing admission control. One interesting thing to notice is: although IP2 is more accurate in calculating D, and the computing overhead is greatly reduced, the actual system throughput is almost the same as IP1. This will be shown in Chapter 7 when we discuss simulation results.

5.5 Summary

Prefetching is a method to improve system performance of multimedia file servers. The basic idea of prefetching is to read some data for a new query before the query is admitted by using wasted disk bandwidth and buffer space. A simple straightforward prefetching strategy SP was described. SP prefetches as much as possible for the first query in the waiting queue, and usually has bad performance. An intelligent prefetching strategy IP1 is introduced as an improvement to SP. IP1 performs prefetching in a more intelligent way and usually gives much better performance result than SP. A further improvement to IP1, aimed at reducing the overhead of admission controll, is the intelligent prefetching strategy IP2. Preliminary analysis shows intelligent prefetchings greatly improve system

performance.

Chapter 6

Multiple Disk Environment

6.1 System configuration of multiple disk environment

To improve system throughput, using multiple disks is an obvious extension to the single disk solution. While there are many possible system configurations for a multiple disk environment, in this thesis, we assume that a system consists of n disks and all the disks can transfer data concurrently. In addition, we assume that the I/O bus can transfer data as fast as a disk can read/write.

In such an environment, there are two factors that affect system performance. The first factor is how the system buffers are distributed and used. The second factor is how the data is organized on the disks.

There are two possible ways to manage system buffers. All the disks can share one system buffer pool, or each disk can have its own buffer set that cannot be shared with other disks. The latter case is trivial. But system performance is not as good as in the first case. So in our discussion, we will concentrate on the case where there is a single shared buffer pool.

There are several methods for data placement in a multiple disk environment:

1. The simplest one is that all the data are replicated on all the disks. Because all the disks keep the same copy of data, real concurrency can be supported. However, this scenario only has limited application, it is not practical.

2. All the data are striped perfectly on all the disks. In this case, the data unit for striping is a fixed size, and the data are placed on the disks in strict order. If implementation details are not considered, then we can treat this as a single disk with a higher bandwidth.

3. The third case is that any single stream is placed on only one disk, so the whole system can be viewed as a collection of disks.

This list is more or less just a theoretical discussion. Practically, we can use RAID disk array [PGK88] as a study example. Basically, data in a RAID disk array can be read/written concurrently. Depending on the different stripping data units (from bits to blocks, to whole files), the reading behavior of a RAID disk array is similar to the Cases 2 or 3 above.

6.2 Extension from a single disk environment

In this section, we discuss how to extend our original algorithms so that they can work well in a multiple disk environment.

If the data is replicated or perfectly striped(cases 1 and 2), there will be no problems at all. We can logically treat this as a single disk, and use whatever policies we used in the single disk case. If $R_{all} = \sum R_i$ is the sum of the disk rates of all the disks in the system, the only modification to our original algorithm is to replace R with R_{all} in all the formulas.

However, if the data is placed according to Case 3 (a stream is placed on only one disk), we have to perform some modifications on our algorithm to make it work.

Our goals here are to achieve high throughput and high concurrency, but at the same time we want the disks to work as independently as possible so that the overhead in

coordinating the actions of different disks can be minimized.

In the single disk case, there is only one request queue. In the multiple disk case, there is one system request queue, and each disk has its own request queue. When a request is submitted to the system queue, it will be re-submitted to its corresponding disk queue depending on where the data are stored. For each disk queue, the admission control policy should still be First-Come-First-Serve. We treat each disk as a *subsystem*, and we can use whatever policy we like within the subsystem. As a first step, no synchronization will be considered among user requests, so the admission control and buffer management are simplified. There are only some minor modifications that need to be performed on the original single disk algorithms.

Specifically, if each disk has its own buffer space and no sharing is allowed, it will work independently of all the other disks. However, if a shared system buffer pool is used, then all the per-disk subsystems will compete for buffers. Keeping track of all the system buffers is a nontrivial task. We also need to enforce some fairness rules to prohibit any subsystem from occupying all the buffers at some time point. These issues will be discussed in the reminder of this section.

6.2.1 Handling buffer sharing and prefetching

There are two important issues that we need to deal with in a multiple disk environment, namely, buffer sharing and prefetching.

Buffer allocation and buffer sharing

In each per-disk subsystem, we will perform buffer sharing using the same scheme as in the single disk case. Since each subsystem is independent of all the other subsystems and their admission controls are not synchronized, performing buffer sharing among all

the serving requests in the whole system level will be too complicated.

However, since all the subsystems are competing for a fixed number of system buffers, in order to improve the utilization of buffers, the number of buffers assigned to each subsystem should not be fixed. This can be thought as higher level buffer sharing.

There are different ways to assign buffers to all the subsystems and enforce fairness among them. Studying the efficiency of these methods is a complicated issue and beyond the scope of this thesis. In the algorithm that we will give later in this section, a simple scheme is used.

As for the implementation concern, we have to keep track of all the buffers used by all the disks and the requests in the system at any one time. A simple way to do this is: whenever a subsystem wants some buffers (this subsystem may not need all these buffers at this moment, but will need them later), it sends a request to the system buffer manager. If the buffer manager's response is favorable, the subsystem will reserve the buffers so that no other subsystem can use them. When the subsystem finishes with the buffers, it will return them to the system buffer manager.

Prefetching

Since we are not considering the problem of synchronization at this moment, each disk works independently of all the other disks. So prefetching is also performed at the per-disk subsystem level.

As we already noticed in the single disk case, the most difficult problem in handling prefetching is to figure out exactly how many free buffers the system has now and will have when the new request is actually admitted. However, using the *request/reserve/release* scheme as we discussed earlier, this will not be a problem, although it may not be optimal in the global sense.

Just like the algorithms without prefetching, the problem of fairness also exists with prefetching. Since prefetching usually consumes much more buffers than regular reading, the situation gets even worse here. Buffers occupied by prefetched data in one subsystem will affect the performance of all the other subsystems. How to coordinate the prefetchings in all the subsystems is a tough and interesting topic, but again this problem is beyond the scope of this thesis. In our algorithm, we will just use a simple policy: At any time, only one subsystem is allowed to do prefetching.

6.2.2 Handling synchronization

Synchronization is the most difficult issue when considering multiple stream retrieval. We will show how this is handled in our multiple disk environment. We use a very simple scheme which may not be optimal. However, it shows synchronized requests can be handled correctly within our framework.

We use the simplest synchronization scenario as an example: *multiple streams are to be started at the same time.*

In the single disk case, because there is only one admission controller, the problem is relatively easy. We can bundle all the requests into one, and always admit them or reject them together.

In a multiple disk environment, we have multiple admission controllers, and the requests may belong to different subsystems. We need to design a scheme to communicate among all the subsystems (on different disks). When one subsystem is ready and the other is busy, the ready one should wait until the busy one is ready too. During the time it is waiting, the ready subsystem should not admit other requests. When all requests can be admitted, all the subsystems should admit them at the same time.

As we said earlier, this is not an optimal solution. When one disk is ready and the other is busy, the ready disk will be idle for some time, so the disk utilization is not quite good. However, this simple solution guarantees no deadlock will happen and it is relatively fair. For a counter-example, suppose we have two disks to serve a synchronized request which requires two streams, one from each disk. At time t, disk D1 is ready and D2 is busy. If we do not want to waste any disk time of D1, and let other requests waiting on D1 get in now, when D2 is ready, D1 may be busy serving the new request. Thus D2 will have to wait for D1. If this situation is not handled carefully, we may easily get into circular waiting, and the synchronization requests will never be served.

It is possible to design a scheme to achieve both high disk utilization and fairness, but that needs more research, so we leave it as a future improvement. Instead, we outline an simple algorithm for the multiple disk environment as follows.

Algorithm MD:

1. *All the requests submitted to the system waiting queue will be re-submitted to the corresponding subsystem waiting queues.*
2. *A system buffer manager will handle all the buffer requests from the subsystems.*
   (a) *In order to use buffers, a subsystem should first send request to the system buffer manager.*
   (b) *The system buffer manager will assign buffers to subsystems according to a predefined policy.*
   (c) *Subsystems will reserve the buffers after they get them from the system buffer manager, and will return them back to the system buffer manager when finished.*

3. *Each subsystem has its own admission controller and scheduler. Each subsystem works independently of the others.*

4. *Synchronization will be handled using the simple stop-wait scheme described above.*

□

In the above algorithm, the buffer manager can use different buffer assignment policies. These policies should guarantee that no subsystem will grab all the system buffers at any time. We use a simple heuristic that no subsystem is allowed to get more than 50% of the total buffers, and simulation results are shown in Chapter 7.

6.3 Summary

This chapter describes a preliminary study on how to extend the work described in Chapters 4 and 5 to a multiple disk environment. This preliminary study showed the framework we established in the last two chapters will work in the new multiple disk environment. Specifically, we discussed how to handle buffer sharing, prefetching, and synchronized requests.

Some simulation results will be given in Chapter 7, and directions for further study will be discussed in the last chapter of this thesis.

Chapter 7

Performance Evaluation by Simulation

7.1 Simulation methodology and program design

We have implemented a simulation package to evaluate the algorithms discussed in this thesis. The package runs under Unix on Sun Sparc workstations. It consists of about 5,000 lines of C source code.

The methodology we used in the simulation package is discrete event-driven simulation. Everything that can cause a change in the system state, such as request arrivals, request completion, cyclic read completion, etc. are defined as events. The main purpose of our program is the generation and processing of these events.

The simulation package can have two different kinds of output. The first requires all the requests being submitted to the request queue at the beginning of the simulation. In this case, we can report the total finishing time, peak disk utilization, peak buffer utilization, average disk utilization, average buffer utilization, etc. The second kind of output is to simulate an interactive system. In this case, in addition to the above results, we can also report query response time. For simplicity, most of the simulation results described in this chapter are reported using the first kind of output.

Our simulations are based on realistic figures. This makes the simulation results more meaningful. The following table lists the ranges of key values we selected.

min seek time(per track)	5 msec
max seek time	25 msec
query playback rate	240 kbytes/s
max disk rate	2000 kbytes/s
block size	20 kbytes
buffer space	100k – 8 M

More details of the simulation package are given in the following sections.

7.2 Implementation concerns

To implement/simulate our ideas, there are lots of issues that need to be considered very carefully. In this section, we discuss several issues we encountered in the design of the simulation package.

7.2.1 When to activate the admission controller?

In our algorithms, reading of streams is performed in an cyclic manner. When a request finishes, the admission controller can be activated either right after the reading for this request is finished; or, after the whole cycle finishes. When the cycle-length is large and disk utilization is low (which means a big chunk of free time will be left at the end of each cycle), the above two schemes have large differences in system performance. We select the first scheme in our simulation. Although it is more complicated and harder to implement, we verified that it does improve system performance.

7.2.2 Buffer releasing

If prefetching is used in our algorithm, the prefetched data will occupy a certain number of buffers. We can release these buffers in two ways: after the query finishes, release all the buffers; or, at each reading cycle, whenever the buffered data is consumed, the buffers will be released (gradually). Releasing buffer gradually will improve the performance to some degree, but it will significantly increase the complexity of our analysis (and thus the complexities of the admission controller and scheduler). In our simulation package, we use the simpler (first) scheme. By doing this, the computing overhead of the scheduler is greatly reduced.

7.2.3 Transient period

As discussed in literal, [RVR92] and [Pol91], in a cyclic-reading scheme, when the cycle-length is changed (e.g. increased), starvation may happen. For example, suppose there are n streams being served, and the current reading period length is t_n. When a new request comes in, the new reading period length becomes t_{n+1}. In our algorithm, t_{n+1} is always bigger than t_n. However, the data we read in in the current period can only last the length of t_n, so starvation will happen.

To solve this problem, we divide the transient period into two steps: First, we change the reading period for the n requests from t_n to t_{n+1}. Then, we do some prefetching for the new request so that it will be ready for startup. After these two steps, we can start the normal cyclic reading with the new query added.

Chapter 5 has already shown how to prefetch data for a request. We will discuss how to change the reading period length from t_n to t_{n+1} in this chapter.

The idea is to do it gradually. Basically, what the system need to do is to try to read

some extra data in each cycle so that the next cycle will be a little bit longer. These steps will be repeated until we reach t_{n+1}.

Suppose disk utilization is ρ. The actual time used to read data in each cycle will be $t_n \times \rho - s$. If we use all the disk idle time to read data, the actual read time will be $t_n - s$. This amount of data will support the consumption of the next reading cycle, so the reading period length for the next cycle can be changed to :

$$t_n(1) = t_n \frac{t_n - s}{t_n \times \rho - s} \tag{7.1}$$

We can repeat this procedure to obtain $t_n(2), t_n(3), t_n(4)...$, until $t_n(i) \geq t_{n+1}$, then the transient period is completed.

From Equation 7.1, we can see that the rate $t_n(i)$ increases depends on ρ. If $\rho = 1$, nothing can be done. So when performing scheduling and calculating t, we cannot push ρ to as high as 1. That will make accepting a new request without losing data impossible. In our simulation package, we set the maximum of ρ to 0.95.

In the process of changing t, we do not need to worry about buffer limitations. This is because we always increase buffer usage in this procedure, but we already know there are enough buffers for the last state t_{n+1}, as guaranteed by the admission control test.

7.3 Evaluation of buffer sharing with varying disk rates

In Chapter 4, we analyzed buffer sharing, and the result showed that it can lead to a tremendous reduction in total buffer requirements. In the ideal case (disk utilization ρ equals to 1), the savings can be 50%. Here we simulated a situation when ρ keeps changing and has an average value less than 1. In this series of simulation, we used 50 queries, each with a consumption rate of 240 KB/s. The lengths of the queries

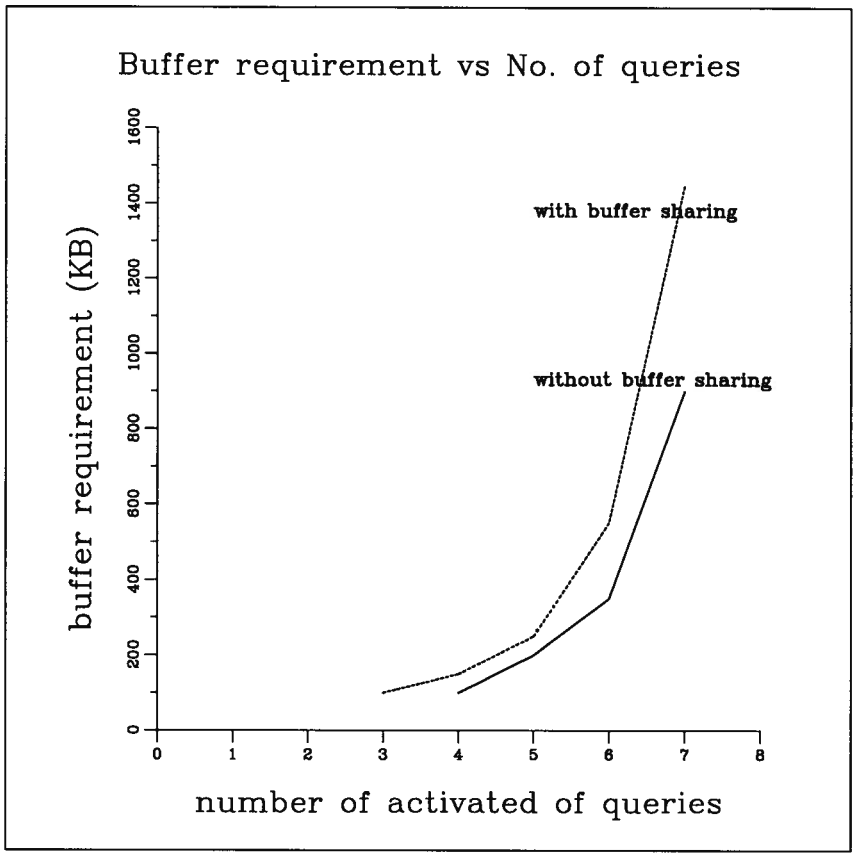

Figure 7.1: The benefit of buffer sharing

were from 20 to 120 seconds, with the average being 60 seconds. In order to support a significantly high number of concurrent queries, the maximum disk reading rate was set to R=2000KB/s. The graph in Figure 7.1 shows the minimum buffer space needed when the number of concurrent queries varies from 3 to 7—with and without buffer sharing. As expected, in all cases, buffer sharing requires less buffer space than without buffer sharing. The savings in buffer space were between 20% to 40%, depending on the average disk utilization.

7.4 Evaluation of buffer sharing with non-contiguous data placement

In Chapter 2, we studied how to use approximation to deal with the non-contiguous data placement case. By using approximation, a buffer sharing scheme can also be applied to non-contiguously placed data, so performance can be improved.

In this series of simulations, we attempted to show that non-contiguously placed streams can benefit from the approximation, which makes them amenable to buffer sharing (and the kind of prefetching we propose in this thesis). In particular, we repeated the simulation described in the previous section with two different queries. The first kind of queries request streams which were non-contiguously placed in blocks of size $Bl = 20$KB (i.e. roughly one disk track), with each block separated by a gap $G = 5$ms. The second kind use the streams which are the approximation of the above streams using Equation 2.12 and 2.13. Queries of the second type were allowed to share buffers (we cannot share buffer for the first type of queries by using our algorithm, because all our analysis is based on the contiguously-placed case.). Analogous to Figure 7.1, Figure 7.2 shows the minimum buffer space (in KB) needed for both kinds of queries, when the number of concurrent queries varies between three and five.

number of concurrent queries	3	4	5
non-contiguous streams	130	360	2520
approx. streams with buffer sharing	110	250	1420

Figure 7.2: Handling non-contiguous data placement using approximation

As can be seen clearly in the table, in all cases, it is beneficial to approximate non-contiguous streams with contiguous ones, and if allowed to share buffers, the approximating streams can lead to a reduction in total buffer requirements.

7.5 Evaluation of prefetching

In this series of simulations, we evaluated the effectiveness of our prefetching strategies. We again used 50 queries, each with a consumption rate 240 kB/s, and length 90 seconds. The maximum disk reading rate was set to 1000KB/s. The graphs in Figure 7.3 and 7.4 show the time taken to complete the 50 queries and the average disk utilization with varying amounts of buffer space. In both graphs, the x-axis is the amount of buffer space, varying form 5MB to 8.5MB. In figure 7.3, the y-axis is the total time taken to complete 50 queries using SP, IP1 and IP2, normalized by the time taken without prefetching. Thus, the horizontal line at 1.0 in Figure 7.3 represents the situation without prefetching. With small amount of space available to prefetching, intelligent prefetchings (both IP1 and IP2) do not lead to any gain in performance. However, as more and more space becomes available to prefetching, IP1 and IP2 are able to activate more and more queries faster than if no prefetching is allowed. Consequently, the total time taken becomes smaller. As shown in Figure 7.3, intelligent prefetching (both IP1 and IP2) could lead to a 30% saving in total time.

Figure 7.4 provides more insight on the comparisons. If no prefetching takes place, the average disk utilization is around 0.8. But as more buffer space becomes available to prefetching, IP1 and IP2 are able to better utilize the disk by prefetching, and the average disk utilization gradually climbs up to 1.0. Moreover, the utilization of buffers follows a similar trend.

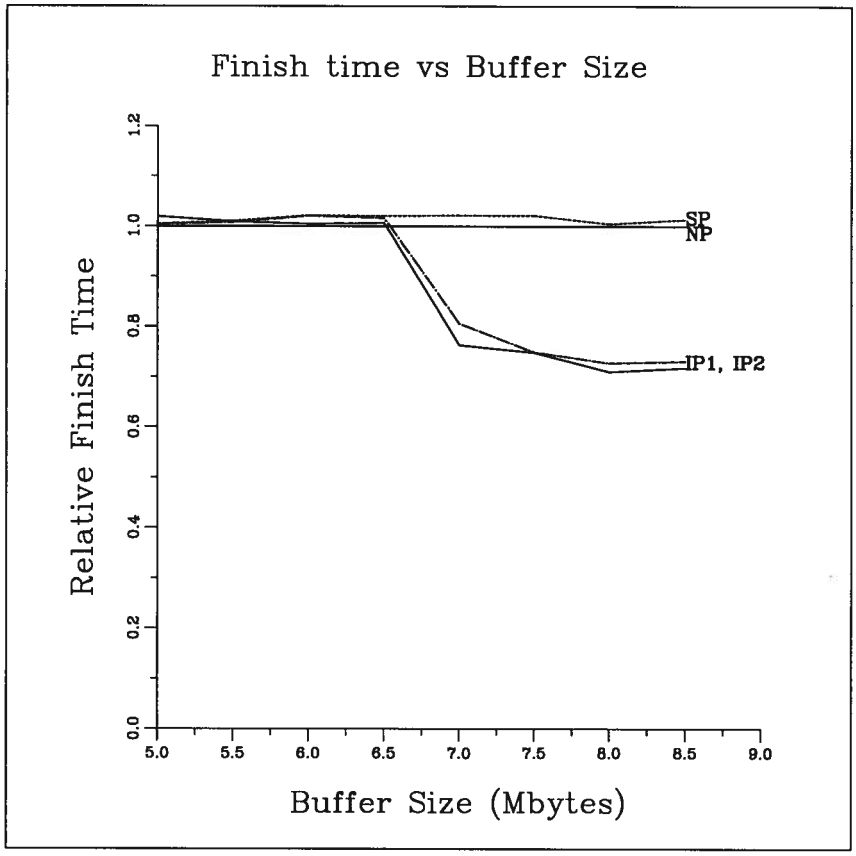

Figure 7.3: SP vs IP1 vs IP2: relative finish time

Another interesting thing shown in Figure 7.4 is that the average disk utilization for SP is still higher than if no prefetching is allowed. This indicates that while the disk is kept busy by prefetching, the way that SP conducts prefetching is problematic and totally ineffective.

Also we can observe from the graph, that IP1 and IP2 do not have significant differences in performance. As we pointed out in Chapter 5, this shows that the heuristics we used in IP1 are good enough in most of the cases. However, we should point out that the computing time used for admission control and scheduling is not showed in the

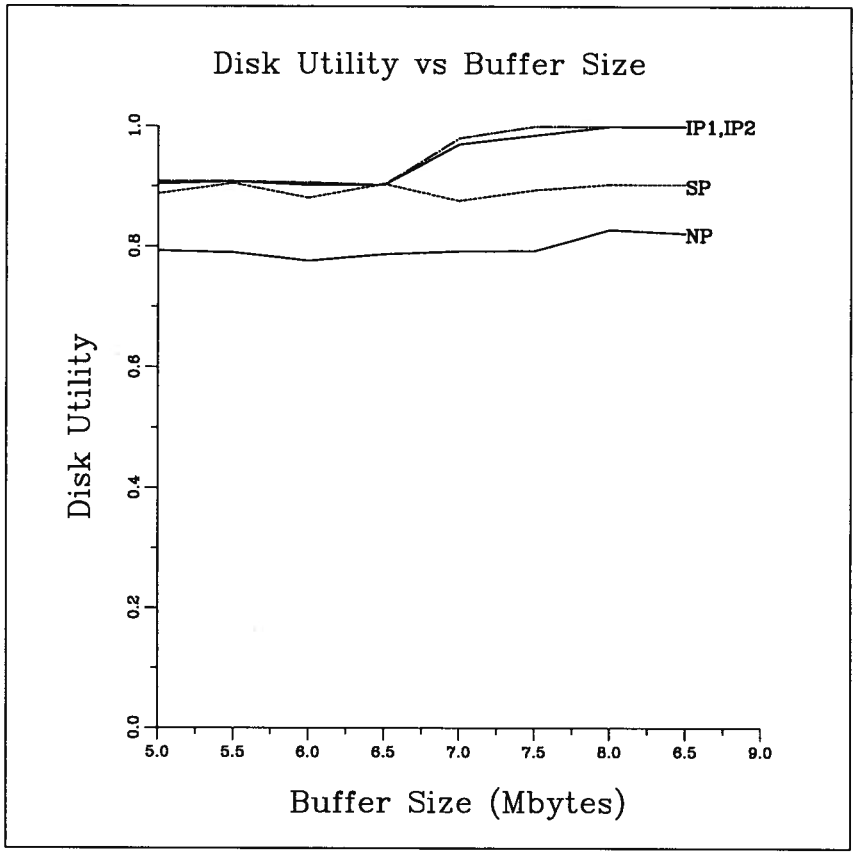

Figure 7.4: SP vs IP1 vs IP2: average disk utilization

simulation. IP2 has much less computing overhead than IP1. The reason is that every time IP2 calculates a prefetch value, it will guarantee a successful admission. On the other hand, due to the reason we discussed at the beginning of Section 5.4, IP1 may have many failed attempts. This increases the computing overhead. In a real system, it will have some impact on performance.

The series of simulations discussed above did not allow buffers to be shared. In another series of simulations, we allowed buffers to be shared, and used the version of admission control that is based on Equation 4.5, not on Equation 2.10. The results of this series of simulations were very similar to those presented above. The only difference

was that buffer sharing saved a few hundred KBs of buffer space, and made it available to perform prefetching. Thus, the point when IP1 and IP2 started to show improvement now began a few hundred KBs earlier than is shown in Figure 7.3. For simplicity, we omit the results here.

7.6 Evaluation of multidisk environment algorithm

In this section, we will give the preliminary simulation results for our multidisk environment algorithm. The algorithm is described as MD in Section 6.2. In this series of simulations, we used a three disk array. We assumed that a stream is placed on only one disk, and there is no duplicated data. There was a system buffer pool shared by all three disks. We set the buffer size to 10MB.

Again, we used 150 queries (roughly 50 queries per disk), each with consumption rate 240 KB/s, and length 90 seconds. Each of the queries was assigned randomly to one of the three disks.

We assumed all the three disks have the same speeds, and in this simulation, we changed the disk speeds from 1000KB/s to 2000 KB/s, reporting the total query finish time.

For comparison, we also reported the query finishing time on a single disk server, which has the same disk speed as the disks in the disk array. In our multiple disk algorithm, the maximum disk space any subsystem can get is 50% of the total buffer space, i.e. 5MB in this situation. To be fair, we set the buffer space of this single disk server to be 5MB too.

In Figure 7.5, the solid line is the total query finish time using this disk array. The dotted line is the result we got using a single disk server.

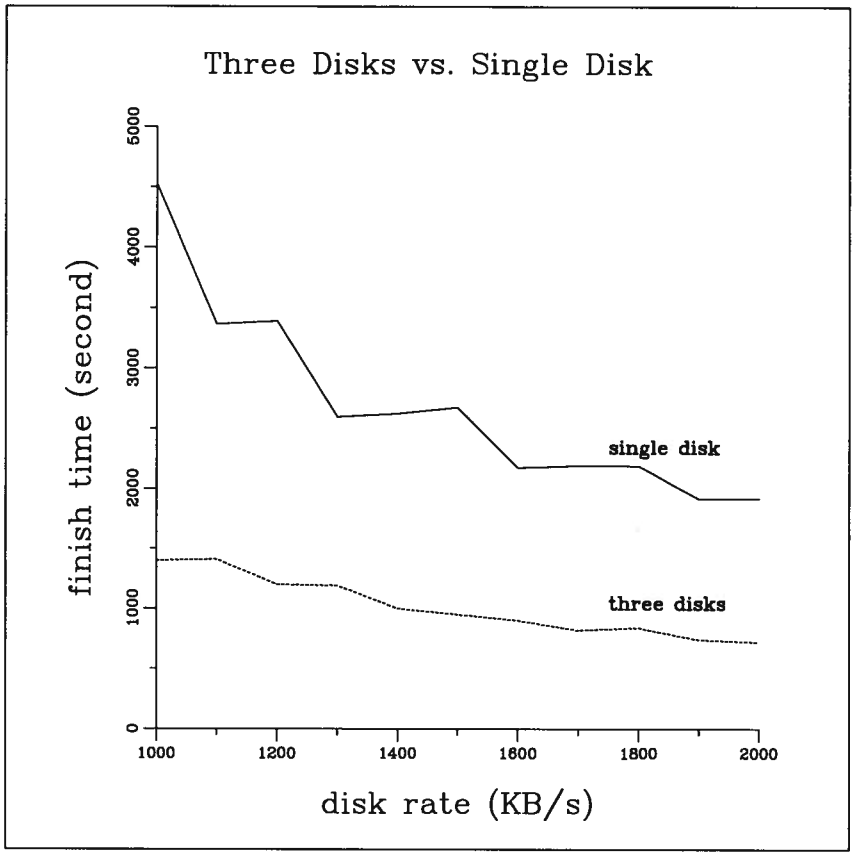

Figure 7.5: The result of the multiple disk algorithm

Figure 7.5 shows that by using this disk array, we can achieve a speedup factor closes to three with various disk speeds.

Chapter 8

Conclusions

8.1 Summary

Designing a high performance multimedia file server is a research topic of great interest and value. In this thesis, we studied one of the key problems encountered in such systems. Given a fixed amount of buffer space and disk bandwidth both pre-determined at design time, we investigated how to maximize the throughput of the system. Our approach was to maximize the utilizations of buffers and disk. To achieve this goal, we proposed and studied two schemes, buffer sharing and prefetching.

Buffer sharing in the fixed reading order scheme was discussed in Chapter 4. We first analyzed a simple case in which all the streams have the identical playback rates. Analysis showed buffer sharing could lead to 50% saving when the disk utilization approaches 1. Removing the restriction that all the streams should have the same playback rates, we further analyzed buffer sharing in the general case. Simulation results showed buffer sharing could lead to 30 to 40% savings in general cases.

Prefetching is the other scheme we used to improve disk and buffer utilization. In Chapter 5, we first discussed the benefits of prefetching. A query can be helped by prefetching in many ways. A simple prefetching scheme SP was given, and the performance was analyzed and simulated. Not satisfied with the performance result of SP, we further proposed a more intelligent prefetching scheme IP1. This scheme has much

better performance than SP and can fully utilize the disk and buffers. IP2 is a further revision of IP1 and the admission control overhead of the algorithm itself is reduced. Simulation results show intelligent prefetching schemes IP1 and IP2 could lead to up to 40% improvement in system performance.

Some preliminary investigation on how to apply our buffer sharing and prefetching schemes to a multiple disk environment was described in Chapter 6. Our analysis showed there is no major difficulty in this extension. Our simulation results also indicated this.

8.2 Future works

There are at least two main categories for future research.

- Improvement of current work
- New directions

8.2.1 Improvement of current work

For the study of buffer sharing, one thing we should do is to design an implementation scheme. In the buffer sharing case, there is no dedicated buffer set for each stream. Buffers are assigned dynamically and globally. Designing a good implementation scheme is a challenging goal.

In this thesis, we only considered the case in which the reading order of all the streams is fixed. How to apply buffer sharing in a variable reading order scheme is an interesting and challenging topic. If we can do buffer sharing in a variable reading order case, we will get both the advantages of buffer sharing (reducing buffer usage) and of variable order reading (optimizing disk reading).

For the study of prefetching, as in the buffer sharing case, we also need to design

an efficient implementation scheme which can fully achieve the result we obtained by theoretical analysis. In addition, we need to find out under what condition prefetching will give the optimal result. Further study of the behaviors of IP1 and IP2 is also needed. We believe further studies on these problems will lead to some interesting results, and will give us a better understanding of the buffer-disk relation in multimedia file server systems.

For the study of how to extend our work to a multiple disk environment, our research is just a preliminary study. There are lots of issues we have not considered very carefully. Some examples are: What are the advantages and disadvantages of sharing a global buffer pool? How will the system bus affect the performance? What are good ways to organize metadata? What are good ways to coordinate prefetching on all the disks? Further work is necessary for all these topics.

8.2.2 New directions of research

There are several important issues we have not considered in this thesis. The first problem is synchronization. A query may require data that are stored on different media. A typical example is a query that requires video, audio and text at the same time, but these are stored separately. The synchronization requirements may vary from application to application. How to handle synchronization is one of the most important issues in a multimedia system. Many studies have been conducted in this area. However, we need to find out how our buffer sharing and prefetching schemes work when handling synchronization.

Another important issue is the impact of networking. A multimedia file server is usually operated in a network environment. The server is at one location, and the user

can be at any location which can be reached through the network. First, the network bandwidth may be a big problem to the server throughput. Second, some problems caused specially by network, like network buffering, data loss, network jitter, etc., will also affect the operation of the server in some way. When designing a network multimedia file server, all these factors must be taken into consideration.

How to apply real-time scheduling theory in multimedia file server design is also an interesting research direction. Multimedia systems have some inherent real-time requirements. Scheduling a query for a video stream is quite similar to scheduling a real-time task. But multimedia systems have some special requirements which an ordinary real-time system does not have. It will be very interesting to study how to apply real-time scheduling theory in multimedia systems, and what kind of modifications and adjustments we have to make for this special environment.

Bibliography

[Adi93] Chris Adie. A survey of distributed multimedia research, standards and products. Technical report, Edinburgh University Computing Service, January 1993.

[AOG92] David P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. A file system for continuous media. *ACM Transaction on Computer Systems*, Vol. 10, No. 4:311–337, November 1992.

[CKY93] Mon-Song Chen, Dilip D. Kandlur, and Phipli S. Yu. Optimization of the Grouped Sweeping Scheduling(GSS) with Heterogeneous Multimedia Streams. In *Proc. of ACM Multimedia'93*, pages 235–242, July 1993.

[Gal91] Didier Le Gall. MPEG: a video compression standard for multimedia applications. *communications of the ACM*, Vol.34, No.4:46–58, April 1991.

[GC92] Jim Gemmell and Stavros Christodaulakis. Principles of delay-sensitive multimedia data storage and retrieval. *ACM Transactions on Information Systems*, Vol.10, No.1:51–90, January 1992.

[Gem93] D. James Gemmell. Multimedia network file servers: Multi-channel delay sensitive data retrieval. In *Proc. of ACM Multimedia'93*, pages 243–250, July 1993.

[JSM91] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive schduling of periodic and sporadic tasks. In *Proc. of the 1991 IEEE Symposium on Real-time Systems*, pages 129–139, 1991.

[LS93] P. Lougher and D. Shepherd. The design of a storage server for continuous media. *The Computer Journal*, Vol.36, No.1:32–42, January 1993.

[NFS91] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of ACM-SIGMOD*, pages 387–396, May 1991.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of ACM-SIGMOD*, pages 109–116, 1988.

[Pol91] Vassilios G. Polimenis. The design of a file system that supports multimedia. Technical Report TR-91-020, Computer Science Division, EECS Department, University of California at Berkeley, 1991.

[RS92] Lawrence A. Rowe and Brian C. Smith. A continuous media player. In *Proc. 3rd Int. Workshop on Network and OS Support for digital Audio and video*, November 1992.

[RV91] P. Venkat Rangan and Harrick M. Vin. Designing file systems for digital video and audio. In *Proc. of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, pages 13–16, October 1991.

[RV93] P. Venkat Rangan and Harrick M. Vin. Efficient storage techniques for digital continuous multimedia. *IEEE transactions on Knowledge and Data engineering*, August 1993.

[RVR92] P. Venkat Rangan, Harrick M. Vin, and Srinivas Ramannathan. Designing an on-demand multimedia service. *IEEE communications Magazine*, Vol.30, No.7:56–65, July 1992.

[RW93] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. In *Proc. of ACM Multimedia'93*, pages 225–233, July 1993.

[TB93] K. Tindell and A. Burns. Scheduling hard real-time multi-media disk traffic. Technical Report 204, Department of Computer Science, York University, United Kingdom, 1993.

[Y+89] Clement Yu et al. Efficient placement of audio data on optical disks for real-time applications. *Communications of the ACM* , Vol.32, No.7:862–871, July 1989.