Better Admission Control and Disk Scheduling for Multimedia Applications

by

Badrinath Venkatachari

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

January 2002

APPROVED:

Prof. Mark L. Claypool, Thesis Advisor

Prof. Robert E. Kinicki, Thesis Reader

Prof. Micha Hofri, Head of Department

To my Family...Amma, Anna and Mama

Abstract

General purpose operating systems have been designed to provide fast, loss-free disk service to all applications. However, multimedia applications are capable of tolerating some data loss, but are very sensitive to variation in disk service timing. Current research efforts to handle multimedia applications assume pessimistic disk behaviour when deciding to admit new multimedia connections so as not to violate the real-time application constraints. However, since multimedia applications are *soft* real-time applications that can tolerate some loss, we propose an optimistic scheme for admission control which uses average case values for disk access taking advantage of caching in the operating system and on the hard disk controller. We have also optimized the measurement based admission controller to admit variable bandwidth multimedia clients even when the requesting client's requirements cannot be completely met. We assume that the multimedia files have been encoded as multiple layers, each adding resolution to the previous one.

Typically, disk scheduling mechanisms for multimedia applications reduce disk access times by only trying to minimize movement to subsequent blocks after sequencing based on Earliest Deadline First. We propose to implement a disk scheduling algorithm that prioritizes requests to help align service to application requirements. The disk scheduling algorithm uses knowledge of the media stored (like MPEG or Realmedia) and permissible loss and jitter for each client, in addition to the physical parameters used by the other scheduling algorithms, to schedule requests efficiently. We evaluate our approach by implementing our admission control policy and disk scheduling algorithm inside the Linux kernel under a framework called `Clarity` and measuring the quality of various multimedia streams and performance of non-multimedia applications. `Clarity` provides better bandwidth utilization by admitting more multimedia clients than the traditional

pessimistic approach. It also guarantees disk bandwidth availability to clients of a particular class in the presence of a large number of clients of other classes. This results in fewer deadlines violations for multimedia clients, higher throughput and lower average response time for non-multimedia clients. We find that our approach results in improved performance for multimedia and non-multimedia applications. The contributions of this thesis are the development of a new admission control and flexible disk scheduling algorithm for improved multimedia quality of service and their implementation on Linux.

Acknowledgments

For me, it has been a big journey from the start to finish. A journey that has been wrought with endless sleepless nights, disappointments, and a lot of struggle. Needless to say, that the only thing that kept me going was the support of a number of people. First and foremost, Prof. Mark Claypool. Mark, “Without your unstinting support, faith in my work, there is just no way that I could have completed my thesis. You have been always there whenever I needed your help in any form and that is what saw me through times of confusion, and helplessness. I guess no words can adequately describe what you have done for me and my work as an advisor, and companion. I thank you for everything.” I thank my reader, Prof. Bob Kinicki for taking the time to read my thesis carefully and giving numerous suggestions to improve it.

There are a number of friends who have contribute greatly to the closure on my thesis. I would like thank my friend, Gana for all his patience and support during the times I was away in the lab for days together. Many thanks to Nitin for all the time he spent patiently discussing various aspects of my work in its developmental stage.

I thank Srikanth for his support and help with many things including numerous trips to the lab to reboot the machine whenever I lost remote connectivity. Without that, working seamlessly away from school would have been impossible. Many thanks to Pravin, Manish and Akshay for their sustained interest in my work and endless prodding to help me complete my thesis. My very special thanks to Roshan, who literally helped me turn this thesis over to the institute. Without his sustained help at the last minute I would have been late.

Finally, I would also like to extend my gratitute to all my friends who have made my stay at WPI one to remember forever.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"First you guess. Don't laugh, this is the most important step. Then you compute the consequences. Compare the consequences to experience. If it disagrees with experience, the guess is wrong. In that simple statement is the key to science. It doesn't matter how beautiful your guess is or how smart you are or what your name is. If it disagrees with experience, it's wrong. That's all there is to it.*"
>
> Richard P. Feynman

Most traditional applications (e.g. word processors, spreadsheets, graphical editors, file transfer, web servers) often execute, store and retrieve data aperiodically or asynchronously and are not required to have some CPU time or access to disks at regular intervals of time. Additionally, they are loss intolerant but are typically insensitive to variance in delay (jitter). As a result, general purpose operating systems were geared to provide best-effort (throughput-oriented) service to these random-access, often I/O intense applications. In the context of operating sytems, best-effort service means that there there are no guarantees about the timing of execution of or delivery of data to the applications, but simply that the underlying framework (particularly consisting of the process

scheduler, disk scheduler, disk layout and recovery mechanism etc.) tries to do the best it can to meet the service requirements of the applications. While a considerable delay in meeting their requirements can cause severe degradation in performance, relatively minor variations in service do not visibly affect performance. Throughput and response times are the primary measures of performance for these applications.

With the recent advances in computer capabilities, compression technologies and broadband networking audio and video applications have become an integral part of our everyday computational life. It has become necessary for us to provide an integrated environment for the execution of these multimedia applications. It becomes especially relevant in content servers that can serve different kinds of data to various clients. Multimedia applications have quite different resource and performance constraints than do traditional applications [1]. They require time-constrained, fair execution environments and periodic access to disks. Execution of and retrieval of data for these applications have *deadlines* by which the application must get all the resources (data, CPU time etc.) to render video or audio. Most multimedia applications are *soft* real time applications, which means that some loss is tolerable while delay and jitter can greatly reduce performance[2, 3]. Although it is important that a certain amount of data be supplied to the application within a given period of time, it is not necessary that all the requests are satisfied in order to provide reasonable application performance. Omission of a few disk requests does not proportionally translate into degradation of quality. This is especially important in order to cater to the *information-access* needs of a large number of users.

With the explosive growth of the Internet, there has been a huge increase in the amount of content accessed from other computer systems (content providing servers). In order to meet their service [1] requirements of local and remote applications most efficiently, the

[1] *service* is provided by the operating system to applications, in terms of providing CPU bandwidth for execution, making decisions for laying out data and providing disk bandwidth for efficient storage and

operating system needs to be aware of many different kinds of service patterns that the applications might have. For example, an FTP application requires that data get transferred as fast as possible. So throughput would be used to measure the performance of such an application. Interactive applications require a portion of the CPU or fetch data every now and then but do not necessarily complete their task quickly. Continuous media applications, on the other hand, require a guaranteed rate of delivery of data from the disk when playing back files. Service patterns for these applications are different and the operating system needs to have this knowledge incorporated in it to efficiently service requests from these applications.

One of the most ubiquitous components of a computer system is the hard disk, which is used for storage and retrieval of programs and data (both application and system dependent) etc, as well as for essential operating system functions like virtual memory management and more. Therefore, one of major factors impacting application performance is how fast data can be stored and retrieved from the hard disk. More importantly, disk access is orders of magnitude slower than memory access, and it is a bottleneck to overcome in order to provide good application performance by retrieving data quickly and efficiently. Efforts to speed disk accesses included efficient algorithms that sequence disk requests in a way that would minimize time spent in retrieving data. The gap in the operating speeds of hard disks and memory has only widened with increase in computing power and faster memory access, making the problem of optimizing disk access even more critical. Compounding this problem is the fact that the applications developed for present day systems have significantly different service requirements [1].

Traditional execution environments in content servers would be unable to distinguish and support the requirements of multimedia applications and hence would execute as normal processes and retrieve data in a similar fashion to conventional data. Additionally,

retrieval, and managing other hardware components required by applications.

they would be unable to utilize the sequential nature of access of multimedia data to prefetch data into buffer cache or layout data on the disk.

There have been multiple approaches suggested to improve the performance in content servers for a variety of applications ranging from throughput intensive to soft real-time applications. Some important aspects of operating systems that have been considered for providing support for real-time applications are process scheduling, disk layout, recovery and scheduling.

Work on modifying process schedulers to support real-time [2] has attempted to ensure that multimedia clients receive a fair portion of the CPU bandwidth even in the presence of a large number of non real-time clients. However, the problem of providing "fairness" has been dealt with differently by various approaches. The two basic approaches to process scheduling are (1) Earliest Deadline First and (2) Rate Monotonic [4]. The former, as the name suggests, executes tasks depending on the urgency of the task. However, the latter decides on task priorities at admission control or connection time and executes tasks depending on those priorities. Since the two major approaches mentioned above suffer from problems such as not being efficient under over-loaded conditions, being inappropriate for supporting conventional applications together with real-time applications, and not providing *QoS* guarantees for real-time applications, a number of improvements have been suggested.

Weighted fair queuing (stride and lottery) tries to ensure fairness by assigning weights to executing thread classes [5]. Start-time fair queuing guarantee fairness bound using start tags assigned to each thread which are computed using weights and quantum of execution [6]. Eclipse and Nemesis systems provide predictable performance via allocation of processes to reservation domains that are collections of processes and corresponding

[2] The terms *real-time*, *multimedia*, and *continuous media* applications are used interchangebly and refer to the same class of applications.

resource reservations [7, 8]. Move-To-Rear List Scheduling provides strict guarantees regarding the cumulative service obtained by a process while also providing guarantees for more traditional *QoS* parameters like fairness and delay [9]. While these strategies have been designed to provide a fair share (decided by application requirements) of the CPU bandwidth to the multimedia applications, they cannot ensure a guaranteed rate of data delivery from the disk. In order to meet the deadlines imposed by multimedia applications, it is critical to optimize the order in which media blocks are retrieved from the disk and implement mechanisms to handle overload conditions.

[10, 11] discuss strategies to stripe data across multiple disks and/or decide on how data must be stored on the disk so that data retrieval can be speeded up. While efficient techniques for striping and laying out of data on multiple disks can reduce service times significantly, they alone cannot provide bounded disk service times. In order for the system to be able to adapt to varying service loads and handle a large number of multimedia clients efficiently, it is important that they complement disk scheduling algorithms to provide guaranteed *QoS*. Disk scheduling algorithms not only resequence disk requests to align disk service to application needs and increase efficiency of disk access, but also help discard media blocks optimally and spread loss among the various multimedia clients under overloaded conditions. Apart from the techniques mentioned above, there are several striping and disk layout techniques. But their detailed discussion is beyond the scope of this work.

Since processors and memory accesses have been getting faster by the day the number of processes that can be executed in a given time has increased. This means that the bottleneck that needs to be addressed in order to provide better support for real-time applications is the hard disk. Better disk scheduling mechanisms need to evolve to accomodate the service requirements of a wide variety of applications.

Although efficient disk scheduling algorithms, which are aware of application require-

ments would help improve the performance of non real-time and real-time applications, there are other issues that must also be considered in order to provide a stable and reliable environment. One of the biggest problems faced by servers providing text and/or multimedia content is that of overloading [3]. Clearly, since multimedia applications require that data be provided to them periodically, it is very important to shield them from the effects of increased load of non real-time clients. Also, in an effort to meet the *QoS* contract with the multimedia clients being served, it is important to prevent the server from being oversubscribed by other multimedia clients.

An additional problem facing the numerous content serving machines (servers) is that of overload. When there are too many clients that need to be served, the servers tend to slow down. Thus, the kind of service that is obtained from these servers depends on the number of clients currently being serviced by it. While this might be acceptable for normal text content, it is unacceptable for multimedia content which requires periodic delivery of data, since it could result in violation of deadlines. This implies that there needs to be a mechanism in the servers providing multimedia content in addition to text, to isolate the effects of an increase in the number of clients from affecting multimedia clients which are already being serviced. Since the disk bandwidth is limited, there is an upper limit on the number of multimedia clients that can be simultaneously supported while maintaining an acceptable quality of service for all admitted clients. It is very important that the server enforce this limit on the number of clients it services to prevent a potential overload.

Admission control is a mechanism that multimedia servers use to restrict service to a few clients while either having a mechanism to allow re-negotiation with session requesting clients or deny service to the other clients till such time that there is enough bandwidth

[3]For our discussion, we assume that an overloaded condition is characterized by insufficient system resources (e.g. disk bandwidth, CPU cycles) to meet the service requirements of clients

available to serve them.

From the client's perpective, it is important that the server guarantee a certain rate of delivery for multimedia content before starting the transmission. Multimedia clients typically negotiate using what are called Quality of Service (*QoS*) parameters to obtain a certain amount of service in terms of periodicity of data delivery. These *QoS* parameters are commonly expressed as bit rate, block rate and frame rate [12]. These performance parameters supplied by the clients are used for admission control and subsequent service by the server. The server processes a client request based on *QoS* parameters and decides whether or not the performance guarantees for the client request can be met. This is essential to ensuring acceptable deterioration in performance perceived by clients already being served [13, 14, 15, 16].

There are three major approaches to carrying out admission control. The first approach is to provide deterministic guarantees to the clients [17], the strictest form of admission control, since it uses worst case values for retrieving media blocks from the disk. The advantage of this approach is that all admitted clients receive all the blocks and no service agreements are violated. The obvious disadvantage is that it is an overkill for *soft* real-time applications like continuous media applications. Also, this approach admits far fewer clients than can be serviced by the disk resulting in the under-utilization of the disk bandwidth. This is because algorithms like SCAN, CSCAN (widely used scheduling algorithms) tend to resequence requests so as to service all of those which are in the direction of the arm movement, which results in seek and rotational latency times being far less than the worst case ones (maximum possible values for seek times and rotational latencies). Additionally, owing to the sequential nature of access of multimedia files, a number of serially laid out blocks are cached by the disk controller and operating system for future service. When the actual disk requests arrive for those blocks, they are served from the cache and hardly incur any service time in comparison to a disk access.

The second approach is probabilistic in nature [13]. This only provides a statistical guarantee that the deadlines for all the admitted clients will be met. This means that at least a fixed percentage of the blocks are retrieved for each client but not necessarily all of them. One of the biggest advantages of this approach is that it results in an increase in the number of clients that are admitted compared to the deterministic approach, since it is not important that all the blocks are retrieved for all the clients. The obvious problem with this approach is that applications that have very strict deadline and loss requirements cannot be accomodated. In the event that deadlines are violated, there needs to be a mechanism to determine the blocks that can be dropped and to distribute the violation of service guarantees among as many of the admitted clients as possible. Therefore, this approach works very closely with the disk scheduling algorithm.

The third approach is one based on observation [14]. In this approach, the times taken for retrieving various media blocks are recorded. When there is a request for admission, an extrapolation is made from current values for access times to obtain the time taken for the new client. This estimated time is used to either accept or decline a client's request for admission. Although this does not provide very strict service guarantees like the deterministic approach, it still provides a fair amount of improvement over the deterministic approach. This algorithm also works very closely with the disk scheduling algorithm in order to spread the effects of deadline violations among as many clients as possible.

1.1 Disk scheduling for multimedia

Once a client has been admitted for service, careful disk scheduling is very important to attempt to meet the QoS requirements for each of the clients [18, 19, 20, 21]. As already mentioned there is a necessity to provide a unified approach to handle requests from different kinds of applications. Therefore, while the algorithm ensures that disk seeks

are optimized to meet the deadlines of the multimedia clients, it must also optimize for non real-time clients so that the average response times are low and throughput high. Disk scheduling for multimedia applications must also efficiently handle overloaded conditions when all the blocks cannot be retrieved, and an optimal set of blocks need to be retrieved, in order to best meet the QoS guarantees of all admitted clients. In short, while admission control provides real-time guarantees to various applications, it is the job of the disk scheduler to eventually provide real-time guarantees to the clients and enforce whatever policies are required to meet the applications' service requirements.

There have been a number of disk scheduling algorithms proposed that try to optimize the servicing of disk requests. The most basic one is Earliest Deadline First (EDF). In this approach, the requests are sequenced in increasing order of their deadlines. Therefore, a request with a deadline of, say 200 milliseconds later than the present time will be serviced earlier than one with a deadline of 400 milliseconds. However, the employment of strict EDF results in poor throughput and excessive seek times. So while this might be a good idea for serving only multimedia clients, this cannot be used without any modifcations for serving real-time as well as non real-time clients [4].

Many other disk scheduling algorithms start from the Earliest Deadline First algorithm and then try to reduce the amount of disk access time. One of the most used one is SCAN-EDF [18]. It tries to service with the earliest deadline first. But, if several requests have the same deadline, they are serviced by their track locations on the disk or by using seek optimization. This strategy combines the benefits of both real-time and seek-optimizing scheduling algorithms [18, 4, 22]. D-SCAN is a modification of the traditional SCAN algorithm which uses the track location of the read request with the earliest deadline to determine the scan direction [23]. FD-SCAN is similar to D-SCAN, but differs from it in that read requests with feasible deadlines are chosen as targets [23]. Typically, these algorithms do not assign immediate deadlines to requests. Instead, when retrieving

media blocks for multimedia clients in *rounds*, they have all the deadlines set to the end of the round [4]. This lets the disk scheduler perform seek and rotational optimizations for all the multimedia requests in that round. Shortest Seek And Earliest Deadline by Ordering/Value (SSEDO/V) attempts to give requests with earlier deadlines, higher priority. However, in the presence of later deadlines requests closer to the disk arm, they might re-assign priorities to service those first [22].

While all these algorithms try to optimize read and write operations for multimedia clients, they do not necessarily support a variety of applications. It is important that they recognize separate service classes that would enable them to make intelligent decisions about request prioritization.

In order to provide guaranteed service to multimedia applications in the presence of traditional throughput-oriented applications, we have designed an optimistic measurement based admission controller. This determines whether a new session is to be accepted given the current load on the disk. It uses values for recent disk access times to predict the time taken for retrieving all the blocks for the requesting client. If the time thus predicted along with the cumulative disk access times for all multimedia clients does not exceed the alloted time for multimedia clients, the client is admitted for service. We have also designed a disk scheduler that resequences requests so as to align disk service to application requirements. Our disk scheduling algorithm identifies some service classes and tries to categorize applications into these classes. While it is not possible to segregate all applications strictly into as many service classes as service requirements of applications, we have made an attempt to capture some categories into which applications can be broadly fitted. The categories into which we classify applications are (1) Delay sensitive real-time

[4]Broadly speaking, a *round* is any interval of time over which application requirements are defined. The disk scheduler attempts to meet application requirements in this period by keeping statistics, and performing optimizations for disk access.

applications, (2) Loss sensitive real-time applications, (3) Non-essential responsive non real-time applications, and (4) Essential non real-time applications. We have also incorporated the logic to service these classes into our disk scheduler. These classes will be discussed in detail in our approach (Chapter 3).

There are a number of compression schemes used to deliver multimedia streams (MPEG, Real Video). None of the present disk scheduling algorithms attempt to improve average multimedia quality using knowledge of the media being retrieved. We use information on the relative importance of the media blocks being accessed, to schedule disk requests optimally even under over-loaded conditions, so that fewer clients are affected (experience a reduction in performance).

Thus, our disk scheduling algorithm, upon receiving requests for disk blocks, takes into consideration the kind of media being retrieved, loss and jitter that is permissible (depending the applications class of the client) to continue to meet the QoS of a client so as not to degrade its quality too much, in order to best schedule those requests.

Our admission control algorithm is similar to the ones proposed in [13, 14, 24]. However, instead of assuming that each disk access consumes the highest value observed for any disk access (worst-case), we use average case values for disk seek time and rotational latency when retrieving a media block. The average values for disk access are not obtained from any measurements, which would require the disk scheduler to keep maintaining statistics about previous accesses to the disk but simply fixed at the time the operating system boots. This value is obtained from previous characterization of the disk access by the manufacturer. We have also implemented a measurement based admission controller. This uses the actual access times during Direct Memory Access (DMA) transfers from the disk (that constitute status quo measurements) to estimate the amount of time that would be taken by a requesting client. Our measurement based admission controller also attempts to exploit some characteristics of MPEG file format to admit clients reading

MPEG files in an attempt to improve server utilization. While the former method of using average case values for disk access times, would provide an improvement to the existing method of using pessimistic scheduling, the latter method of measurement is expected to provide substantial increase in the number of clients admitted for service and also better service guarantees.

We have implemented `Clarity`, a framework for content servers, consisting of an admission controller and disk scheduling algorithm in Linux. Linux is a UNIX-like operating system originally created by Linus Torvalds with the assistance of developers around the world. Linux natively does not have any special support for multimedia clients either in the form of admission controller or disk scheduling. Our admission controller and disk scheduler run as priviledged processes and are a part of the kernel. We have compared the performance of our admission strategy with others under Linux. We have also carried out performance evaluation of our disk scheduling algorithm with the existing Linux disk scheduler which employs the C-SCAN algorithm to service requests.

In order to evaluate efficiency of our algorithms for responsive non real-time clients, we measure the average response time. This gives an indication of how long a responsive application has to wait before getting data from the disk. Since none of the applications used for our performance evaluation continuously request media blocks over a period of time, we do not concern ourselves with throughput measurements. Also, throughput measurements do not give an accurate picture of how often an application gets data from the disk. Since the goal of our admission controller and disk scheduler is to minimize *QoS* violations for real-time applications, we look at percentage of missed deadlines for various kinds of loads. As mentioned earlier, one of the factors that affects visual quality is jitter. Hence we have tried to evaluate the amount of delay variance in disk service for a client. The efficiency of the admission controller is measured by the number of clients that can be simultaneously serviced while meeting their service requirements and by the

amount of disk utilization achieved by way of admitting clients at bit-rates lower than that asked for by clients without siginificant degradation to perceptual quality.

The rest of the thesis is organized as follows. Chapter 2 briefly presents work that has been done relating to admission control and disk scheduling for supporting multimedia clients. This discusses the pros and cons of various schemes that have been proposed. Chapter 3 details our approach to designing multimedia clients, and designing and implementing the admission control policy and disk scheduler in `Clarity` under Linux. Chapter 5 presents our method of evaluating the algorithms that we implemented and presents results from those performance evaluations. Chapter 6 summarizes the results obtained from evaluation of our approach, the advantages and disadvantages of our approach, and concludes the thesis. Chapter 7 proposes some new directions in which further research could continue.

Chapter 2

Related Research

"Begin with another's to end with your own"

Baltasar Gracian

This chapter discusses the work that has been done relating to this thesis. It outlines the previous contributions towards admission control and disk scheduling for supporting real-time as well as non real-time clients. Although we studied some of the mechanisms in process scheduling, disk layout and recovery used in improving multimedia performance, we will not discuss them in this chapter since they have already been dealt with earlier. We will however, briefly discuss the modifications/extensions made to Linux in order to provide better support for multimedia in Section 2.3.

It is clear that given the diversity of the current and emerging applications, traditional file systems have to be re-designed to support them. There are two ways of enhancing support for multimedia applications:

1. Build multimedia operating systems : There have been a number of efforts in this direction [7, 20, 24]. This involves designing the kernel (file systems, process scheduler, disk scheduler, disk layout and recovery schemes) and all layers above it to

optimize performance for the various applications that are required to be supported. However, the amount of time and effort required to do this is substantial.

2. Modify the existing operating systems : In this approach, modifications could made to one or more of the operating system's core components mentioned above. This task might be an easy or a difficult one, depending on the ease with which changes can be made to the opeating system's design. Since there are a large number of present-day operating systems that lack the capability to support guaranteed service to multimedia applications in the presence of non real-time clients, we think it would be useful to provide increased support for multimedia applications. We shall however, restrict ourselves to the disk scheduling sub-system of an operating system when discussing improvements to enhance multimedia performance.

There has been a lot of research done on building multimedia operating systems and adding support for multimedia in the present-day operating systems. The following sections describe the various approaches taken for enabling disk scheduling support for multimedia in the presence of non-real time applications.

2.1 Disk Scheduling

2.1.1 Multimedia Operating Systems

In [20], Prashant Shenoy et al have described the implementation of Symphony, a physically integrated file system which features a (1) a *QoS*-aware disk scheduler that efficiently supports both real-time and non real-time traffic, (2) a storage manager that supports multiple block sizes and allows control over their placement, therby enabling diverse placement policies to be implemented in the file system [11], (3) a fault-tolerance layer that enables data-specific failure recovery [10], and (4) a two level meta data (inode)

structure that enables data type specific structure to be assigned to files while continuing to support traditional byte stream interface. The disk scheduler in Symphony (*Cello*) identifies three classes of applications which are (1) real-time, (2) throughput intensive, and (3) responsive and tries to align the service to application requirements [21]. We have discussed *Cello* in greater detail under simulation-driven contributions.

While work in [7] concentrates mainly on improving process scheduling (hence not discussed here), [24] has been categorized as more of a change to existing operating systems since the *Fellini* server software has been ported to a number of UNIX-like operating sytems and Windows NT.

2.1.2 Modifications to Present Operating Systems

[25] describes the implementation and evaluation of a Multimedia File System (MMFS) that extends the UNIX file system specifically for interactive multimedia applications. The essential aim is to help in operations like fast forward, reverse etc. while maintaining the synchronicity between audio and video streams being played back. MMFS supports intelligent prefetching, state-based caching, prioritized disk scheduling to enhance performance for multimedia streams. The disk scheduler uses two classes (real-time and non real-time) of disk requests to provide service. The actual scheduling is a variation of SCAN-EDF (discussed earlier). Requests are prioritized so that concurrent non real-time requests do not unduly affect the performance of real-time requests. The framework lets the application specify deadlines for each multimedia read request. In order to service multiple real-time requests, the scheduler employs EDF algorithm and retains SCAN for non real-time requests. One of the biggest drawbacks of this approach is the potential starvation of non real-time requests due to the strict deadline enforcement policy for multimedia requests. Another important aspect of the scheduler is that it does not offer any real-time guarantees to the applications. It is for this reason, that we have also implemen-

tated an admission control scheme.

[24] describes the architecture of the $\mathcal{F}ellini$ multimedia storage system that supports storage and retrieval of both continuous media as well as conventional data (text and image). The algorithms used to retrieve data from the disks provide high throughput by reducing the seek latency time. One of the important features of this implementation is the buffer management scheme that exploits the sequential access pattern of multimedia data in order to determine the buffer pages to be replaced from the cache. This apparently helps reduce disk I/O by increasing cache hits and thus making more disk bandwidth available for conventional data requests. In order to provide rate guarantees to multimedia clients, it uses a pessimistic admission control which is discussed in Section 2.2. This work does not deal with any mechanism to discard blocks when all the clients cannot be serviced and does not recognize any application classes except real-time and non real-time. Cache management in $\mathcal{F}ellini$ is discussed in the section on caching (Section 2.2.2).

YFQ, a disk algorithm that allows applications to set aside portions of the disk bandwidth for exclusive use is introduced in [26]. It has been implemented as part of Eclipse/BSD operating system which has been derived from FreeBSD. YFQ's disk bandwidth reservations can guarantee file accesses with high throughput, low delay and good fairness. YFQ uses weights and length of requests in addition to a work function to decide on which request should be serviced ahead of others. However, such guarantees to individual applications come at the cost of excessive seek and rotational latencies and result in problems in global disk scheduling optimizations. In order to handle this, YFQ has some extensions to the original algorithm like choosing a bunch of requests instead of a single request and re-ordering them to reduce seek and rotational latencies. While it has been shown that YFQ provides higher throughput, it does not have a mechanism to protect clients from one another and no scheme to handle overloaded conditions.

2.1.3 Simulation Driven Disk Scheduling

David Anderson, Ramesh Govindan and Yoshimoto Osawa simulated a Continuous Media File System (CMFS) [17], which supports real-time storage and retrieval of continuous media data on disk. CMFS also addresses several interrelated design issues like real-time semantics of sessions, disk layout and an acceptance test for new sessions apart from a disk scheduling policy. They introduced the concept of using a logical clock to time read and write accesses to disk. The logical clock runs at a fixed rate reading from or writing to a First In First Out (FIFO) buffer allocated for each session, stopping if it catches up with the client. CMFS promises to stay ahead of the logical clock by a certain amount of time. These semantics make it possible for CMFS to handle variable-rate and other nonuniform access of files. This work also proposes an acceptance test to see if a session can be accepted for service. This is discussed in the section on admission control (Section 2.2). However, they try to provide hard guarantees to applications and do not have a policy of dealing with overflow rounds.

In [23], the authors Robert Abbott and Hector Molina discuss methods to improve some of the existing algorithms like D-SCAN, FD-SCAN and EDF through simulations. They consider the problem of having to service read and write requests and investigate the interaction between them. They propose two models to handle real-time read requests and non time-constrained write requests. It is assumed that there is a buffer manager between an application and the disk that issues the actual read and write requests to the disk on receiving similar requests from the applications. While it is important that read requests are serviced in accordance with their deadlines, it is also important that writes to the disk are carried out from time to time depending on the dynamics of the system and request arrival. The authors have proposed using *k*-buffers which can be viewed as buffers managed by the disk handler instead of the buffer manager. So while writing pages, the pages are copied from the dirty buffers in the buffer manager to frames in the

k-buffer. By using *k*-buffers the authors decouple the disk handler from the rest of the system. There are two schemes proposed to handle flushing of buffers for handling writes while meeting the deadline requirements of the real-time applications. They are (1) Space Threshold (looks at when the *k*-buffer is running out of frames to accomodate new write requests), and (2) Time Threshold (creates an artificial deadline for a write event which could consist of multiple writes). Finally, the authors evaluate how this scheme combined with the algorithms mentioned above performs.

In [21], the disk scheduling algorithm *Cello* attempts to meet the diverse requirements of applications. *Cello* employs a two level disk scheduling architecture, consisting of a class-independent scheduler, which governs the coarse-grain allocation of bandwidth to application classes, and a set of class-specific schedulers, which control the fine-grain interleaving of requests. This mechanism while separating application-independent mechanisms from the application-specific ones, enables the existence of multiple class-specific schedulers. Thus, while real-time applications can use schedulers that would ensure that requests are periodically serviced by the disk, there can be a class-specific scheduler for the responsive applications that see to it that the response time does not exceed a threshold time. Since *Cello* partitions bandwidth among application classes, it ensures that application classes are protected from one another. However, in principle *Cello* is a variation of the SCAN-EDF algorithm. In serving, applications *Cello* considers three classes of applications, (1) Real-time, (2) Throughput intensive best-effort, and (3) Interactive best-effort. The authors show that their scheme works better than SCAN in terms of average response times for non real-time clients and fewer missed deadlines for real-time clients. One of the biggest drawback of this work is the fact that all experiments were carried out using a disk simulator. For the purposes of optimizations, the authors have also over-simplified the view that disk scheduler has of the disk. This scheme presents a very non-practical case since in the real world there are seldom any mechanisms to obtain information that

the authors assume is easily available. The following are some of the other drawbacks of this scheme:

- When creating a queue of application requests, it is assumed that all the multimedia requests (for guaranteed service) that will be made in a particular round is known *apriori* so that the disk requests can be laid out in appropriate fashion. The big problem is that all the disk requests cannot be known *apriori*. Even if one were to design a framework where all the requests were known or generated by the disk scheduler it would mean considerable overhead during disk access, might even be extremely cumbersome to implement or adapt to existing implementation of an operating system and last but not the least, constraining in a number of ways. In a number of cases, especially with VBR (variable bit-rate) clients, number of data requests are different every second. Also, it removes the possibility of applications employing policies, like media scaling in order to be adaptive to over-loaded rounds since it is assumed that the disk scheduler has complete knowledge of all the multimedia requests at the beginning of each round.

- They assume that the duration of each and every request in the queue is known. This information is used to insert non multimedia requests into the service queue, with a view to servicing them only when there is time available between two adjacent multimedia requests. The time taken for various requests is calculated using the present disk head position. The service times of various disk requests cannot be so easily assumed to be deterministic. While the disk access time for a block depends a lot on the present position of the disk head, it must be noted that it is virtually impossible in modern day operating sytems to retrieve the present position of the disk head from the disk controller since it does not provide such information. This makes the authors' approach of calculating the overhead incurred as a result of

inserting a disk request in the service queue using the present position of the disk very unusable. Another important thing to remember is that file systems seldom provide mechanisms for disk schedulers to know where a particular block of data resides on the disk. This information would be required in order to calculate the time taken to read data starting from the present position of the disk head.

- They conveniently ignore the CPU time that would utilized to ensure ordering and re-ordering of the service queue in the presence of a large of requests. In a highly loaded system, it can contribute to substantial CPU time.

Since this work has carried out simulations for only the disk sub-system, it does not consider the feasibility of implementing some of the policies needed to support *Cello*. This in turn results in the exclusion of any overheads (CPU and memory consumption) introduced by *Cello*, which might be absent from a SCAN-EDF implementation, from the statistics shown for various client-load scenarios. Also, this simulation model does not deal with any media-aware mechanisms to decide how discarding of media blocks must take place during overflow rounds.

2.2 Admission Control

In [17], the authors have proposed a criteria for admitting new sessions based on the assumption that all applications need hard guarantees to be satisfied. For this, a sufficient condition is the existence of a static schedule (that cyclically read fixed number of blocks of each session) satisfying the rate requirements of all sessions under worst-case assumptions and for which enough buffer space is available. Quite clearly, when the clients are reading from FIFO buffers, there are some that would read out the buffer faster than others. As long as the minimum time taken (among all the sessions) to read the FIFO buffer

is more than the worst-time it takes to read all the blocks for all the sessions, it is assumed that the new session, given its data requirements, can be supported. One of the disadvantages of this approach is that it is an overkill for multimedia clients which are *soft* real-time applications and can suffer an occasional distortion or loss of information. This drastically reduces the number of such clients that can be serviced and results in server under utilization. Our algorithm tries to overcome this problem by using reported average case values and those obtained from averaging out the DMA times, to admit multimedia clients.

The authors have proposed an observation based admission control algorithm in [14]. The admission of a client is permitted only if the extrapolation from the status quo measurements of the storage server utilization indicate that service requirements of all the clients will be met satisfactorily. While this mode of admission control provides fairly reliable service to clients, there are no absolute guarantees and simultaneous servicing of multiple clients may lead to deadline violation for some of the clients. One of the assumptions of this approach, is that the service times of clients will continue to exhibit similar behaviour even after the admission of a new client. Thus, it is believed that measurement of server performance is good for estimating service times in the presence of a new client. While the approach mentioned above is used for admitting clients that can suffer occasional distortion and loss of information, applications requiring hard guarantees are admitted using theoretical worst-case values of disk parameters. Therefore, support has been provided for both kinds of applications.

[13] discusses a statistical admission algorithm that is used to determine whether in the presence of other clients, a new client should be admitted. This approach attempts to (1) exploit the variation in access times of media blocks from disk as well as the variation in client load induced by variable bit-rate (VBR) compression schemes, and (2) provides statistical service guarantees to each client. The idea here is to use distributions of access

times of media blocks and playback rate requirements of media strands encoded using variable bit-rate compression techniques to provide at least a certain percentage (constitutes the strand's minimum bit-rate requirements) of the media blocks required by each strand. These distributions need to be calculated apriori, possibly at the time of installation by measuring the service times for various placements of media blocks. The reason for doing so is that the service times for media blocks is only dependent on the relative placement of media blocks and not on client characteristics. The main reason behind providing statistical guarantees is that human perceptual tolerances as well as inherent redundancy in multimedia streams allow clients to be tolerant to brief distortions in playback continuity and occasional loss of media information. Thus, this scheme attempts to provide increased usage of server resources while retrieving only a portion of the required media blocks for the admitted clients. The admission criteria used in the presence of n clients is:

$$q * \mathcal{F}_o + (1 - q) \sum_{i=1}^{n+1} f_i \geq \sum_{i=1}^{n+1} p_i * f_i$$

where,

$q =$ probability of an overflow round

$\mathcal{F}_o =$ number of frames that are guaranteed to be retrieved in an overflow round

$f_i =$ number of frames retrieved in each round for client i

$p_i =$ percentage of frames to be retrieved each round to meet client i's service req-
-uirements

We believe that while this approach might provide reasonable guarantees to the clients, its performance is largely based on the ability to collect extensive samples of server performance for varying workloads. In order to make the admission controller much more

reactive to system dynamics, we have designed a measurement-based admission controller.

[24] also describes an admission control policy used in *Fellini*. This work like [17] assumes worst-case values for accessing disks and therefore prevents a number of multimedia clients from getting admitted without deadline violations for already existing clients.

2.2.1 MPEG Layering of Video

In [27], the authors discuss the implementation of a layered scheme for handling requirements of various multicast clients. An MPEG file consists 3 different kinds of frames which constitute different resolutions for the video. Since the data rate requirements for the transfer of MPEG-like video is very high, it is proposed that there be a demultiplexer at the video server that breaks down the various frames in a manner that constitute layers (each one being responsible for increasing visual quality). This scheme allows the selective delivery of the layers and thereby provides a mechanism to carry out controlled degradation of quality at the receiving side in the presence of congestion or increased traffic. We believe that this scheme can be used very effectively for serving MPEG files from the disk. We assume that the files have been encoded as layers and can be retrieved as such. While [28] assumes the presence of only two layers (a base layer and an enhancement layer) we assume that the file has been laid out using information from I, P and B frames. We, however, believe that this still has the same effect of providing video quality enhancement going from one layer to each additional layer.

We exploit the fact that MPEG-like layered clients can be served at bit-rates lower than that constituted by all the layers encoded in the media file. The number of layers served depends on the amount of residual disk bandwidth in the presence of admitted multimedia clients. Additionally, when handling overflow rounds, the disk scheduler attempts to serve only as many layers as can be obtained in the time available. This, therefore, can be used

to provide fairly reliable service to clients that would have been refused admission by all the schemes discussed above.

2.2.2 Caching

Caching has been used extensively for serving content quickly and reliably to clients when there is an overlap in the content being requested by the clients. The content in the cache is present either by way of prefetching or because some client accessed the data and it has not yet been removed from the cache. In the context of an operating system, it is used to serve applications pages from the disk that have been prefetched at an earlier time. This not only reduces the response time to a request but also access to disk. There have been several approaches to increasing performance of multimedia clients. Some of them are discussed in the following paragraphs.

Symphony [20] enables the co-existence of multiple data type specific caching policies. The buffer subsystem maintains two buffer pools: one for deallocated buffers and the other for cached buffers. So while LRU can be employed for text clients some other form of caching can be used for continuous media clients. Thus, it can be expected that there will be good performance for a number of clients.

Another scheme that attempts to use a buffer cache to provide better multimedia performance has been described in [24]. The cache management policy is based on different data access patterns, volume of data and deadline requirements of real-time clients. The authors have also tried to handle scenarios like multiple clients being separated by only a phase difference, in the caching policy for continuous media clients.

It is clear from the above references that there is a need for caching to provide improved performance to applications. It assumes even more significance for multimedia applications, given their deadline requirements and generally higher data rates (esp. for MPEG-like files). In order to fully realize the effect of continuous media specific poli-

cies in terms of an increase in the number of clients served and better performance for admitted clients, one must translate this information for use in admission control.

Therefore, in addition to our briefly discussed schemes, we also propose a scheme for making the admission controller cache aware. There can be a significant increase in number of clients being simultaneously serviced and apparent reduction in the disk service times if data stored in the cache (from previous disk accesses) can be managed and used to admit new clients. This is especially true when there is a lot of locality in the information accessed from a multimedia storage server.

2.3 Linux Implementations

One of the most significant changes made to the standard Linux kernel in order to provide real-time guarantees is RT-Linux [29]. RTLinux is a *hard real-time* variant of Linux that enables control of robots, data acquisition systems, manufacturing plants and other time-sensitive instruments. While RTLinux version 1 was designed to run on low-end x86 machines and provided very a limited set of APIs, RTLinux version 2 is a complete re-write of the earlier version and is designed to support symmetric multiprocessing, run on a larger range of systems, and with extensions for ease of use. RTLinux permits the running of special real-time tasks and interrupt handlers on the same machine as standard Linux [1]. This is possible because the Linux kernel is run at the lowest priority and permits pre-emption by these tasks and handlers at any time irrespective of what Linux is doing. The worst case time between the moment a hardware interrupt is detected by the processor and the moment an interrupt handler starts to execute is under 15 microseconds on RTLinux running on x86.

As mentioned earlier, the attempt to provide hard guarantees to clients would translate

[1] Henceforth, *Linux* shall refer to the standard *Linux* kernel

into fewer clients being serviced. Therefore, there needs to be a mechanism introduced in RTLinux that makes it aware of continuous media clients that can be given fairly reliable service guarantees. Also, there needs to be a mechanism in RTLinux to ensure fairness among clients when sharing disk bandwidth among them. In the case of overflow rounds, there needs to be a mechanism to spread the loss among as many clients as possible. Most importantly, since the real-time clients can pre-empt the Linux kernel running non real-time tasks, there needs to be a scheme to protect non real-time applications and ensure that non real-time requests do not starve in the presence of a large number of multimedia clients. This would require some kind of a selective pre-emption of the Linux kernel depending on the importance or priority of the real-time applications. In this approach, quite clearly applications requiring hard guarantees would have higher priorities than multimedia clients.

Since we believe that the distribution of Linux far exceeds that of RTLinux, we have tried to incorporate some of the above mentioned ideas into Linux. This is also an attempt to add soft-real time capability to Linux. Also, to the best of our knowledge this is first implementation of an admission controller and multimedia disk scheduler on Linux. While [30] describes a Linux implementation for increased multimedia support, we found that only the process scheduler has been modified to provide hierarchical start time fair queuing [6]. The disk scheduler *Cello* described in [21] has not been implemented yet.

Chapter 3

Clarity

Our approach towards supporting multimedia applications on Linux is in two directions. One is in the form of an admission control mechanism and the other in the form of disk scheduling. Both are important in ensuring that a guaranteed level of service can be provided for a certain number of multimedia clients (a number that is dictated by the amount of disk bandwidth available on the disk). The admission controller, as the name suggests, admits and rejects requests for multimedia sessions and the disk scheduler tries to efficiently service disk requests for the admitted clients so as to deliver a certain number of blocks to the application within a given time frame.

The first section of this chapter describes the four service classes that we have identified to categorize applications and our approach to aligning the service provided to applications to those classes. In our context, service to application is primarily in terms of providing enough disk bandwidth to retrieve data from the disk at a given rate. This is followed by the section that describes the layering scheme that has been assumed for MPEG-like clients. Then comes a section on our *admission control* policy which briefly describes the technique used to determine the availability of disk bandwidth for providing the performance specified by a requesting application. The last section describes our

approach to scheduling disk requests in order to meet the QoS requirements of the various applications and balance allocation of disk bandwidth under overloaded conditions to minimally affect application performance. It discusses our approach to modifying the Linux scheduling algorithm in order to provide fair service to applications.

3.1 Service Classes

As mentioned earlier in chapter 1, there are different kinds of applications that an operating system must support and provide good performance. Therefore, there is a very clear need to identify service classes so that applications can be grouped and serviced broadly according to the requirements of that class. This helps us provide low-level optimizations designed to improve application performance within each class.

In order to align applications to their service requirements, the task of identifying application classes needs to be done carefully. A large number of application service classes would result in significant maintenance and computational overhead and having too few of them could result in significantly different applications getting assigned to the same service class, thus preventing significant performance improvement.

Broadly speaking, we have categorized applications into multimedia and non-multimedia applications. This classification is based on the fact that content servers primarily read data from the disk to serve text, graphics and multimedia streams. Writes on content servers are assumed to be performed only when uploading content. The classification results from the basic difference in the type of service required by these classes of applications. Multimedia applications are iso-chronic, needing data supplied to them at regular intervals. On the other hand, the non-multimedia applications require higher throughput and lower response time. In order to further separate application requirements, we have identified four different kind of application classes (2 each for multimedia and non-

multimedia applications) that most of all applications can be categorized into. They are:

1. **Loss-tolerant multimedia applications** - These are multimedia applications that can maintain reasonable quality of service with some loss of data. For these applications, not all disk requests need to be satisfied. The requests that cannot be serviced are dropped when they outlive their usefulness and scheduling is carried for frames that need to delivered shortly after. Most multimedia clients (servers, playback applications etc.) fall into this category. However, it is important to remember that they can tolerate only a certain amount of loss and there must be mechanisms in place to recover from losses that induce deterioration to unacceptable quality levels of transmission. This class includes both constant bit-rate (CBR) and variable bit-rate (VBR) applications.

2. **Delay-tolerant multimedia applications** - While most multimedia applications can tolerate some loss, sometimes it is essential that all frames be received and processed. Typically, when a user is accessing multimedia content that is recorded and not "*live*" or current, the user might want to get all the frames. Another reason for this class is to separate the processing of audio content from the video content so that although there might be deterioration of video, one can ensure that all audio content is read from the disk. The discussion of such mechanisms is however beyond the scope of this work. For these applications each and every requests needs to be satisfied while maintaining as much periodicity as possible. In the presence of delayed service, the disk scheduler must make an attempt to recover from it by having mechanisms that would allow for extra blocks to be processed when there is some extra bandwidth so that normalcy is restored.

3. **Responsive non-multimedia applications** - These are the traditional applications that require disk requests to be satisfied in the shortest time. However, this does

not mean that they require all the data they request immediately. It is adequate to service a small percent of requests from this class, so that the application keeps getting blocks from the disk intermittently, without a prolonged delay.

4. **Essential non-multimedia applications** - This represents a special class of disk requests which need to be satisfied for applications to execute. Although there are no applications per se that fit into this category, all processes acting on behalf of the operating system are automatically included in this class. These applications, if you will, generate requests for swapping out pages, paging etc. that are generated by the operating system to proceed with the execution of processes. The reason these requests generated by these applications are essential is because they cannot always be serviced subject to availability of disk bandwidth to any specific service class. Multimedia applications may sometimes be unable to proceed without the disk request belonging to this class getting serviced. So delaying it due to non-availability of disk bandwidth would mean wastage of disk bandwidth and delay/loss of blocks to be delivered to the application. It is for this reason that no bandwidth is explicitly reserved for servicing disk requests belonging to this category. Also, since these requests are not multimedia requests per se, we have categorized them into a separate class to distinguish requests belonging to this class from the normal requests multimedia/non-multimedia requests that are serviced subject to availability of disk bandwidth. In the absence of this class, there is no way exceptions can be made to deal with system related disk requests. In may be noted that although we do not create such applications, we have mechanisms to identify such requests in `Clarity` so that measurement of time consumed by non-multimedia applications does not include time used by this category.

3.2 Layered Multimedia Streams

When a client requests data, the admission controller tries to find out if there is enough disk bandwidth to service the request for a multimedia session. Although this means that we are essentially preventing the disk from being oversubscribed, we could try to put the "residual" bandwidth to use to serve multimedia clients to improve disk utilization and reduce wastage of bandwidth. We believe that a slightly modified approach to admission control can increase the number of clients accepted and decrease the residual bandwidth which is not utilized. This is especially true for high bit-rate streams like video. When a video client requests a multimedia session, it is quite possible that although there is substantial amount of disk time for servicing some lower bit-rate clients, it is inadequate to service this client completely. Therefore, the video client could have the session rejected. For example, suppose there is enough disk bandwidth to serve clients aggregating 700 *Kbps*, an video client might ask for an average bit-rate of 1.5 *Mbps*. Quite clearly, the admission controller will reject the session based on its average bit-rate requirement and maybe the client will want to try again. While this is definitely beneficial to the clients that have been admitted since they get almost deterministic service without violation of any deadlines, it leaves a large portion of disk bandwidth unused. This might turn out to be a major concern if a number of files being served by a multimedia server are MPEG files or files that require a lot of bandwidth.

In order to deal with this problem, we consider layering of MPEG files. In this approach, an MPEG file is assumed to be consisting of multiple layers, each contributing to a portion of the total bandwidth required. While the base layer provides just the bare minimum visual quality, the additional layers enhance the quality of the video being played. So while it would provide true quality if all the layers are serviced, it is not required that all the frames be delivered to maintain a reasonable level of visual quality [27]. We be-

lieve that in layered MPEG video, acceptable visual quality can be maintained by serving the client with lesser number of frames (fewer I, P and B frames [1] that are actually encoded in the file [31, 27]. In the absence of disk bandwidth we think serving layers in a priority fashion would keep the service predictable, spread the losses more efficiently among the various requesting clients than servicing in a random fashion where the goal is to serve as much to a client as possible.

There are several encoding schemes that can be assumed for MPEG layered streams. Typically, the play-out pattern of the MPEG file is $\{I\ B\ B\ P\ B\ B\ P\ B\ B\ I\}$ and the storage of the stream does not really guarantee that sequential read of blocks will be sufficient to decode them in this sequence. The MPEG file decoder has to access frames out of order to generate other frames. As mentioned in section 2.2.1, B is a bidirectionally predictive frame and needs a frame in the past and one in the future to be generated. This imposes availability of P before the two B frames between the I and P frames can be decoded and played back. While the B and P frames provide incremental quality enhancement over the I frame, they actually contain temporal information apart from spatial information. For our purpose of layering, we have assumed that each B and/or P frame tends to increase clarity and smoothness of the playback (hence results in increased visual quality). Therefore if a B or a P frame was dropped, it would not only result in loss of information between the frames played but also in reduced clarity of images in the streams. Table 3.1 [2] shows the retrieval pattern that can be adopted so that the clients can be served layers of streams instead of a bunch of blocks (dependent on what the clients request) without any limitation on how much the clients can request.

[1] MPEG files are encoded using these frames. I is an ***Intra*** frame and is encoded without any history, P is a ***Predictive*** frame and is predicted from the preceding I or P frame when decoding and B is a ***Bi-directionally predictive*** frame and reconstructed from the closest 2 I or P frames (one in the past and the other in the future).

[2] This table has been adapted from [32]

Layer	Playback Pattern	Blocks Required	% of Actual Blocks Required
5	I B B P B B P B B I	192	100
4	I B P B P B I	161	84
3	I P P I	132	69
2	I P I	90	47
1	I I	63	33

Table 3.1: Layered playback pattern for layered MPEG streams

Quite clearly, this approach reduces the number of blocks that need to retrieved and tries to provide a balance between the visual quality and timeliness of delivery of data blocks for decoding. Some of the more important implications of this approach of retrieving the data blocks (constituting layers) from the disk based on the requirement of a client in a particular round are follows:

1. Loss due to non-availability of disk service time is spread among a number of clients.

2. Unnecessary time spent in retrieving blocks that do not completely constitute a layer is minimized.

3. Request for blocks in a particular round is optimized (if possible) by the requesting application, based on information that it can receive from the disk scheduler about how much it can service in that round. The client can employ techniques like media-scaling in order to change the sequence in which it reads blocks of data in the absence of sufficient disk bandwidth to satisfy all its requests.

3.3 Admission Control

We expressed the need for admission control in order to control the usage and allocation of disk resources for various applications requiring service. Admission control is a key component in multimedia servers, which need to allow the resources to be used by the clients only when they are available. This assumes great significance when the server needs to maintain a certain promised level of service for all the clients being served. If the admission control admits too few clients, it results in wastage of system resources. On the other hand, if too many clients are allowed to contend for resources, then the performance of clients already degrades rapidly in the presence of new clients. Therefore, judicious decision making mechanisms for allocating resources disk bandwidth to clients are needed. Admission control is an integral part of supporting multimedia clients, since it serves as a mechanism to not only identify clients that require multimedia clients that need periodic delivery of a certain amount of data from the disk but also limits the amount of contention for resources like disk that can operate considerably slower than the processor. Here, we assume that we are dealing with only clients that are reading from the disk and not writing to the disk. While the problem essentially remains the same, that of determining whether there is adequate bandwidth available to serve clients that want to write, there are enough differences to warrant our excluding them from our discussion. Figure 3.1[3] represents how the admission controller fits into our framework for providing support for multimedia clients.

Apart from the fact that admission control must be used to control the usage of resources like the disk that are central to meeting the deadlines of the admitted clients, it is also used to ensure that different classes of applications can co-exist without significantly affecting each others performance. For example, in the traditional system disk bandwidth

[3]This figure has been adapted from [33]

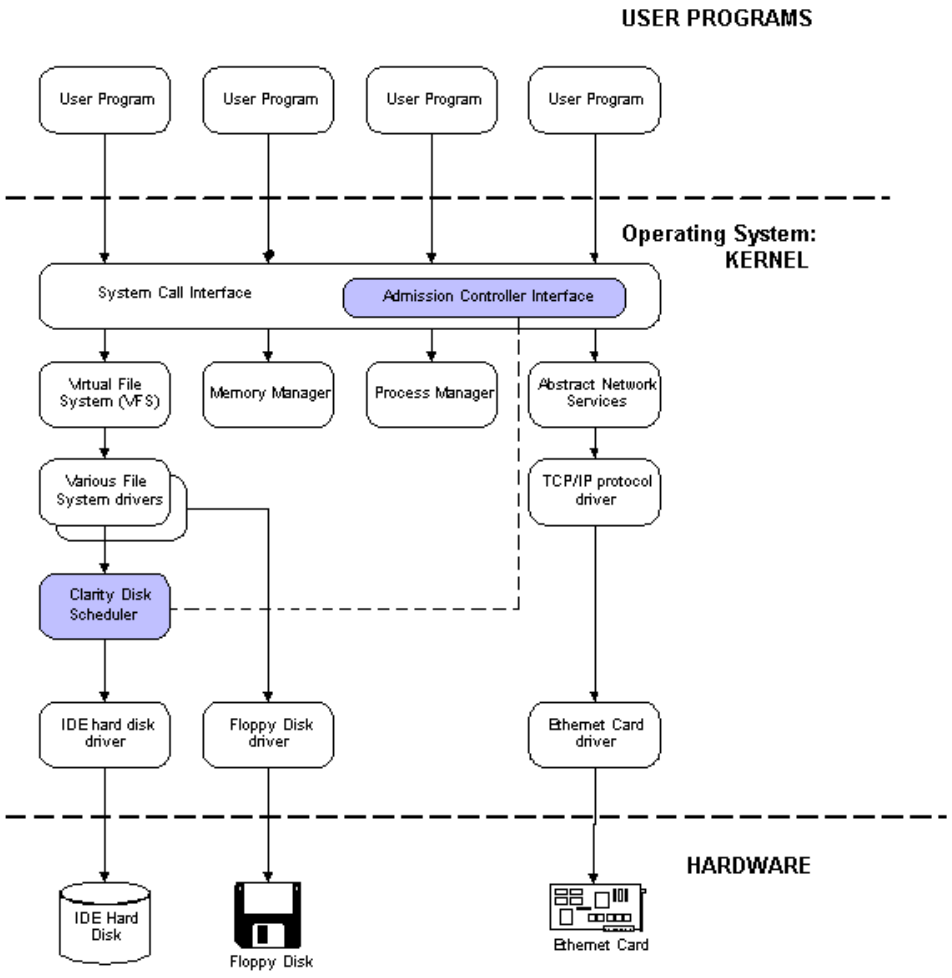

Figure 3.1: Clarity in the Linux kernel

is always shared and so if there are a large number of clients from a particular application class, it is highly likely that disk requests from the other classes will be serviced in a delayed fashion. Therefore, it is very important to reserve bandwidth for the various application classes both during admission and scheduling so that requests belonging to one class do not hamper the servicing of requests from another class. While the admission controller can be largely independent of the disk scheduler, it is such considerations that highlight the need for them to complement each other in servicing client requests

efficiently. For example, there is no point in admitting clients for a certain amount of bandwidth when it is not guaranteed by the disk scheduler. Such implementations cannot provide any guaranteed service for admitted multimedia clients. Therefore, in our approach it is assumed that in a round only a certain fraction (ρ, which is fixed when `Clarity` is activated through a loadable kernel module) of the total time in a round is available for servicing multimedia clients and it is guaranteed by the disk scheduler that under over-loaded conditions this portion of the round timing will be made available to serve multimedia disk requests.

The admission controller can also be used for passing on some information about the various clients being admitted to the disk scheduler which would help the scheduler align service to the application requirements. This can help the admission controller and the disk scheduler complement each other effectively to maintain the *QoS* of the admitted clients. We shall discuss this in further detail in the following sections.

3.3.1 Optimistic Admission Control

There are some applications that are "hard" real-time as opposed to "soft" real-time applications where some deadlines can be missed and can be dealt with depending on whether a loss-tolerant or delay-tolerant service is required by the applications. In "hard" real-time applications, all the deadlines must be met since otherwise highly erroneous computations or unstable conditions could result. In order to service these kinds of applications, admission control has to be designed to take into consideration the worst-case scenario for disk access for blocks so that no retrieval is made beyond the deadline. For such applications, the admission controller has to use the pessimistic approach and assume that each and every data block is going to incur the maximum overhead in terms of seek time and rotational latency.

A number of multimedia applications are loss tolerant applications. Therefore, there

is no need to deliver all the blocks in a round. This would not only result in fewer clients being serviced but possibly gross under-utilization of the system resources (both CPU and disk). This is because lesser clients for disk service would mean lesser clients using the CPU for processing. Also, this would negate the effect of any optimizations and efficient retrieval mechanisms used by the operating system. Therefore, we have designed an optimistic admission controller. While, there might be times when the time for disk access and retrieval of a block will be more than the average value, it is our belief that over prolonged time and high load, the retrieval times average out. In this manner, we also intend to use techniques used by the underlying operating system to cache blocks before they are requested. It is this characteristic of seek times and rotational latencies to average out over a prolonged period of time that we exploit to provide increased utilization and service availability.

While optimistic admission control can be implemented and used largely independent of the disk scheduler, it works better when it complements the disk scheduler in meeting the deadlines of the clients that have been admitted.

Criteria

Suppose there are n clients that have been admitted for multimedia sessions. Let each of the n clients be denoted by $\mathcal{C}_i$. Each of these clients can have different playback rates and therefore needs the disk scheduler to return blocks to them at different rates. Also, the clients might play back streams at rates different than the rate at which disk blocks are retrieved. Let the number of blocks to be retrieved for $\mathcal{C}_i$ in a round be b_i and let the play back rate of the blocks be $\mathcal{B}_{pl}^i$ *bits/sec*. Furthermore, let the block size be $\mathcal{B}$ *bytes*.

Multimedia clients need periodic access to the disks. The idea is to serve blocks to the clients at regular intervals. The duration of the intervals is decided by the data rate that the client is requesting. More specifically, it is dependent on the highest data rate

that has been requested by the admitted clients. A client with a data rate of 20 *Kbps* will have to be served twice as often as a 10 *Kbps* client in a given time. Therefore, it is very important to decide on the duration of a round.

Let the time in a round be denoted by $\mathcal{R}$. This duration $\mathcal{R}$ is decided as follows:

$$\mathcal{R} = \min_{\mathbf{i}\in[\mathbf{1},\mathbf{n}]} \left(\frac{b_i * \mathcal{B} * \mathbf{8}}{\mathcal{B}^i_{\mathbf{pl}}} \right) \tag{3.1}$$

While the play back rate of the client can be different from the rate at which blocks are retrieved, without loss of generality and information we have assumed that they are the same. Therefore, equation 3.1 reduces to a constant time of 1 second for serving clients. This simplifying assumption decreases the computational overhead needed for admitting clients and that for calculating the number of blocks to be retrieved during each round. Also, having smaller rounds can significantly reduce the positive impact of various optimizations employed by the disk scheduler to reduce disk service times. This aspect shall be discussed further in the sections dealing with our approach to scheduling disk requests.

The one and only criterion that is used to determine whether a client requesting a multimedia session can be admitted is the availability of disk bandwidth to satisfy the *QoS* requirements specified by the client. Let there be n clients in the system, when a new client $(n+1)$ requests admission. If the bit-rate that this client is asking for is r_{n+1}, then the number of blocks to be retrieved for this client b_i can be calculated using the block size $\mathcal{B}$ as:

$$b_{n+1} = \left(\frac{r_{n+1}}{\mathcal{B} * \mathbf{8}} \right) \tag{3.2}$$

Furthermore, let ρ be the fraction of every round (round time $\mathcal{R}$) that is available for servicing multimedia disk requests. Therefore, in order to service b_i blocks, the disk takes average case values for seek time and rotational latency. From this, we can derive that the time spent in disk seek for i^{th} client is $b_i * (\mathcal{T}_{avg}^{seek})$. Similarly, the amount of time spent in moving the required block under the read arm on an average is $b_i * (\mathcal{T}_{avg}^{rot.\ lat})$. This leads to define our admission control criteria, which tests for the existence of residual bandwidth sufficient to meet the minimum quality requirements of the client, in the manner shown below:

$$\sum_{\mathbf{i=1}}^{\mathbf{n+1}} b_i * (\mathcal{T}_{avg}^{\mathbf{seek}} + \mathcal{T}_{avg}^{\mathbf{rot.\ lat}}) \leq \rho * \mathcal{R} \tag{3.3}$$

We assume that the data rate specified by the client when requesting a multimedia session has already taken into consideration the amount of loss per round that is acceptable for the session. However, the client can make a request for blocks that constitute the actual data rate i.e. without any losses, at the beginning of each round from the disk scheduler. Although this data-rate may be higher than the data rate at which the client was admitted, the disk scheduler always attempts to first satisfy only as much as the client was admitted for. Also, while the disk scheduler might try to satisfy the request for a client completely, the disk scheduler is not at all constrained to do so and might decide to use the bandwidth for servicing whatever it deems important.

There are a number of applications that are constant bit-rate applications. For these it is easy to take into consideration the number of blocks that need to be retrieved in a round. However, there are a number of multimedia clients that are variable bit-rate applications. They playback a constant number of frames with the size of frames between rounds varying. This makes the number of blocks retrieved in a round a variable. In order to support variable bit-rate applications we carry out admission control using the average

bit-rates necessary for those applications. We have mechanism to deal with the fact that the highest bit-rates for these applications can be significantly different from their average bit-rates and we shall discuss it later in our approach to disk scheduling (Section 3.4).

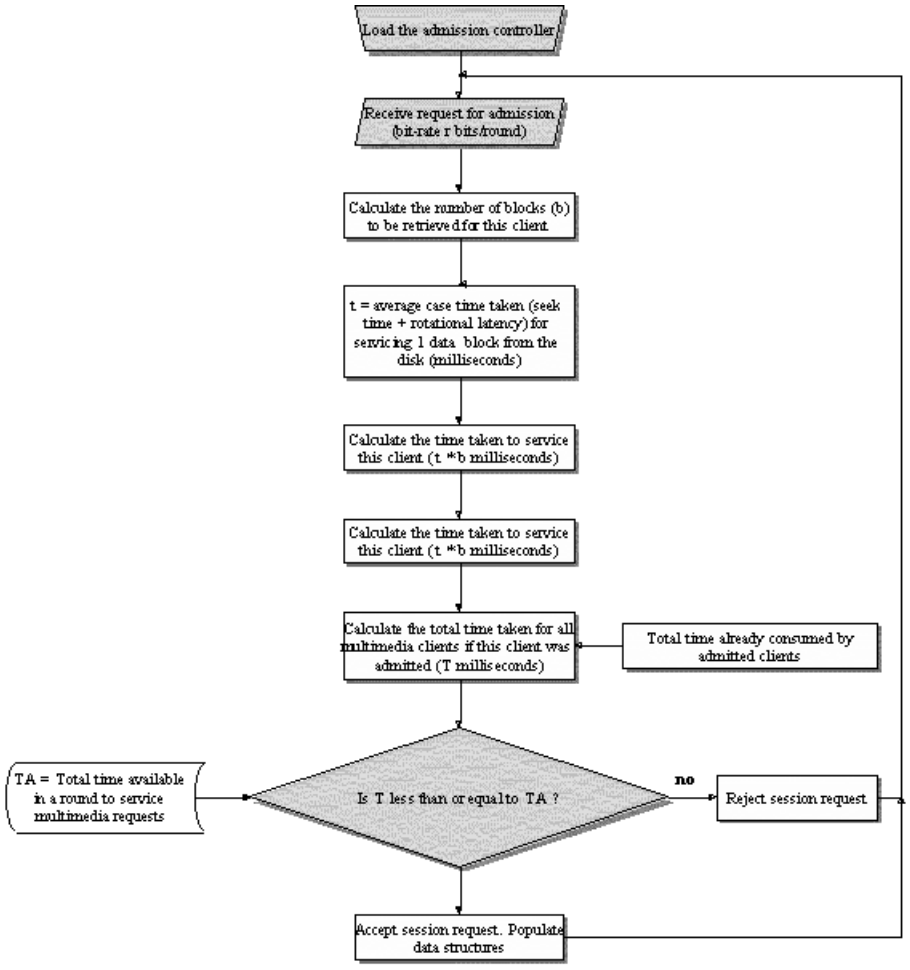

Figure 3.2: Flow-chart for admission control

Our algorithm to admit a requesting client is explained in a step by step manner in figure 3.2. In order to illustrate our admission control policy, let us assume that there are already 5 clients that have already been admitted and they consume in all 400 *millisec-*

onds. Without loss of generality, we now suppose that there is another client that wants multimedia session with a bit-rate of 80 *Kbps*. Assume that the time taken reserved for disk access in a round for the i^{th} client is given by t_i. Furthermore, assume that the average seek time for a disk block $\mathcal{T}_{av}^{seek}$ is 4.5 *milliseconds* and the average rotational latency for it $\mathcal{T}_{av}^{rot.\ lat}$ is 12.0 *milliseconds*. Suppose that the fraction of bandwidth in a round available for serving multimedia requests ρ is 0.5 of the round timing $\mathcal{R}$. We assume that $\mathcal{R}$ is 1 *second*. Therefore, in a round the time available to service multimedia disk requests is 500 *milliseconds*. The following are the steps carried out to determine whether there is enough bandwidth to service the requesting client:

1. Number of blocks to be read every round for the new client is b_6 = 80 $\div$ 8 = 10
2. Time taken in the round for reading b_6 blocks = 10 $*$ $(\mathcal{T}_{av}^{seek} + \mathcal{T}_{av}^{rot.\ lat})$ = 10 $*$ (12 + 4.5) = 165 *milliseconds*
3. Total time taken to satisfy all the clients with their minimum bit-rates = $\sum_{i=1}^{6} t_i$ = 400 + 165 = 565 *milliseconds*
4. But we know that the amount of time that we have in a round is 500 *milliseconds*. Since the time required is more than the time available for servicing multimedia applications, the client is **denied** admission.

3.3.2 Measurement-based Admission Control

While we feel that it is a good idea to expect that disk access time to statistically tend towards an average value instead of worst-case values, there are other important considerations that actually reduce the disk access times substantially. It is important to realize that for every block of data, a request is sent to the disk, a seek is made to the appropriate *track* containing data and then the appropriate *sector* is brought under the read arm by

rotating the disk by the required amount. While this approach is adequate for comparing approaches like optimistic and pessimistic admission control by simulation and looking at the relative improvement in performance, in reality present-day operating systems perform a number of optimizations to reduce disk requests sent and hence the disk service time. We believe that by employing the optimistic admission control scheme, which uses fixed average disk access time to calculate the cumulative time of access for a client, we heuristically take into account the effects of caching on the disk controller, in operating system and effects of the dynamics of disk access over a fairly extended period of time. However, we do not completely consider the effects on disk access time due to optimizations in the operating system and their combination with one or more factors mentioned above. That is why we have designed a measurement-based admission controller which uses average values of disk access obtained from actual measurement to calculate cumulative disk access times for various clients.

Modern day operating systems including *Linux* makes use of temporary storage in memory called *cache*. When a client wants to retrieve a block of data, *Linux* sets up the request for not only the requested block but also for some adjacent blocks too, the time for reading which is substantially lower than if a new request was issued to read each one of them. This only complements existing disk controllers many of which intelligently cache an entire track when fetching data from the actual physical disk. These are blocks that the operating system and the disk controller anticipate would be required by the client (the word 'client' means different entities for the disk controller and the operating system) sometime in the near future. So when the client asks for a block that has been *cached*, *Linux* simply retrieves it from memory. This prevents the generation of a separate request to the disk and avoids all the overheads of seek times and rotational latencies. Quite clearly, caching provides many magnitudes of service improvement. This is because (1) retrieval of a block that is cached when reading another does not incur any substantial

seek time and rotational latencies since it is contiguous with the one requested for and (2) when a cached block is served, it is from memory.

We believe that there is a lot of merit to using values from measurement of times for disk block retrieval to admit clients. This is because they reflect disk access times obtained as a result of various optimizations in the operating sytem, like file layout, block pre-fetching, caching etc. to enhance performance. Since these optimizations lower disk access times, this approach would provide a further improvement in terms of the number of clients admitted for guaranteed service. In order to carry out measurement-based admission control, we keep track of the time taken for various disk requests to be serviced and use the average time taken to read a block of data to calculate the time that would be required to satisfy a requesting client's data rate.

Our approach to a measurement-based admission control mechanism heavily depends on the disk scheduler and its ability to record and maintain information on the time taken for the various disk reads. In order to make the system responsive to changing client load, it is important for the disk scheduler to constantly communicate with the admission controller. While the admission controller can make decisions about admitting or rejecting client requests independent of the disk scheduler, having the disk scheduler inform the admission controller about its load and performance makes the admission controller sensitive to the effects of the system as a whole on the disk. Since not all traffic passes through the admission controller (text clients, traffic for paging, swapping etc.), this approach helps tremendously improve system responsiveness to shifting system loads and helps the admission controller make well-informed decisions. Since typical processors are very fast compared to the disk, typically disk reads are handled by a DMA controller. This is so that the CPU does not spend cycles waiting for the disk read to complete and can proceed with normal execution of other programs that need CPU bandwidth till such time that the disk block is fetched. It is then that the CPU is interrupted by the DMA

controller and the blocks are returned to the requesting application. The CPU can now continue from where it left off in the program that it suspended waiting for disk I/O completion. Our approach is to keep track of these DMA transfers for both multimedia and non-multimedia blocks, count the number of blocks and using the total time taken average out the access time for a block. In order to more accurately measure the average, we also take into consideration time for data access over a number of DMA transfers. The exact number of such DMA transfers taken to calculate the average would probably need to be finetuned and is a trade-off between how well one needs to have the average reflect load on the system at different times versus the CPU cycles needed to calculate the average. We determined through our experiments that averaging over around 30 DMA transfers provides reasonably accurate disk access times without consuming too much CPU bandwidth. There are a number of approaches that can be used to calculate the average time for retrieving a block of data (it must be remembered, that we are not concerned about the size of each block of data). For example, one of the most commonly used method is divide the total time taken by the total number of blocks read to obtain the average time per block. We will attempt to obtain the averaging mechanism that is most accurate for our purposes.

Suppose that there are *n* clients that have already been admited for multimedia sessions with the i^{th} client having a bit-rate of r_i. As in the discussion of optimistic admission control, let another client $(n+1)$ request for a multimedia session at a bit-rate r_{n+1}. Then the number of blocks to retrieved for this client in a round is given by equation 3.2. Suppose that the average time for a block access is $\mathcal{T}_{avg}$, measured over a certain number of DMA transfers and ρ is the fraction of every round (round time $\mathcal{R}$) that is available for servicing multimedia disk requests. The admission control criterion then can be represented by the following equation:

$$\sum_{\mathbf{i=1}}^{\mathbf{n+1}} b_i * \mathcal{T}_{avg} \leq \rho * \mathcal{R} \tag{3.4}$$

One of the biggest advantages of this approach is that it tries to account for the time reduction associated with pre-fetching blocks contiguous with the one(s) requested by an application. For example, let us assume that an application requests block 1, which is contiguous to block numbers 2, 3 and 4. So when an applications asks for block 1, the operating system pre-fetches blocks 2, 3, and 4. Suppose the access time for block 1 is 15 *milliseconds*, then reading times for the contiguous blocks may be around 20 *milliseconds* since by the time the operating system pre-fetches these blocks, the disk controller has already ensured that the blocks are retrieved from the disk. Quite obviously, this is a huge improvement from a situation where requests to the disk are generated as and when clients request data. This is because by the time the request is sent, the *cache* on the disk controller may have been over-written by newer data and the new request would have to be serviced by actually positioning the arm and moving the platter on the disk. Assuming that this happens, it would result in an overall time of 60 *milliseconds*. So, if there are a number of blocks pre-fetched into the cache, subsequent access for those blocks report almost insignificant access times. So the average access time for these blocks get lowered to an great extent. Essentially, the measurements are then an indication of how fast contiguous blocks can be read and how intelligently the operating system pre-fetches the data blocks.

In order to make use of disk scheduling optimizations and its effect on the time taken for transferring a block of data, it is very important that measurements be made at appropriate places in the kernel. We shall discuss this aspect further in the chapter on implementation (Chapter 4).

3.3.3 Adaptive Measurement-based Admission Control

In this approach, we have essentially used the measurement-based approach to performing admission control for multimedia clients. However, we have attempted to use a different form of admission control mechanism for MPEG kind of clients. These clients, as described in section 3.2, can be served in layers, with each layer possibly contributing to enhancing the clarity of video playback.

For such layered clients, we typically assume that there are no loss percentages that the client tries to specify. This is so that the streams can be served in various number of layers, the loss percentages of which are different. Our approach in this form of admission control is to typically look at how many blocks can be served to the requesting client by iteration. Suppose that a client requests a rate of r_i and needs b_i blocks, then we try to find out the aggregate number of blocks (which can be different from b_i) that can be served towards a certain number of layers. We use the criterion for measurement-based admission control (equation 3.4) to determine the maximum number of layers for which the client can be admitted. For the purpose of performing admission control, we assume that the bit-rates for the layers are some fixed percentage of the amount of data needed by a client in any single round. So when an MPEG client requires service, the admission controller checks to see if there is enough bandwidth to serve the client completely using its average bit-rate. If it is not feasible to support the required bit-rate, the admission controller scales down the average bit-rate requested (this represents the dropping of a layer) and repeats until such time that the client is either admitted or it has dropped all layers. Subsequently, the disk scheduler serves blocks constituting the maximum number of layers for which the client was admitted.

Let us assume that that N is the maximum number of the layers that can be serviced for the stream that the client is asking service for. Further, if l_{max} is the maximum number of layers that can be serviced without violating the admission control criteria, let r_l^{max} (in

bps) be the rate associated with l_{max}. Equation 3.5 shows the admission control criteria that used to determine whether or not a client can be admitted for service.

$$\sum_{\mathbf{i=1}}^{\mathbf{n}} b_i * \mathcal{T}_{avg} + r_l^{max} \leq \rho * \mathcal{R}, \qquad \mathbf{0} < l_{max} \leq \mathbf{N} \tag{3.5}$$

3.3.4 Cache aware measurement-based admission control

Modern day operating systems use *cache* to increase the overall speed of execution and overall response time of applications. Cache is used not only for pre-fetching blocks of data from the disk so that they are quickly retrieved when the application asks for them but also for keeping often used data in general.

In the earlier description of our approach to admission control, we have assumed that all the content that needs to be fetched has to be from the disk. While this is a valid assumption to make in a number of circumstances, there are others in which there is lot of *locality of reference* of data. This means that there is some content or part of it that clients are most interested in and want to access more than other content hosted by the multimedia server. This is a typical scenario when there is an audio or video clip of some important event(s) that a lot of people are interested in. Therefore, the traffic to those few files is very high compared to some other clip that is a little old. When there are a certain number of files on a multimedia server that are accessed very often compared to some others, some or all of the content can be kept in the cache and served from there. Modern multimedia servers come with huge caches and some of these clips can easily be accommodated in part or fully in the cache. Also, while it might be very beneficial to store the video streams that are getting accessed in the cache completely, limitations on how much of each stream can be stored in the cache necessitates explicit caching mechanisms to store multimedia streams.

We believe that with careful caching policies, it is possible to admit clients requesting

sessions when some other client is just a little ahead in time and reading the same file. A client which requests for data which is in the cache can be served from there instead of being refused admission for lack of disk access time. The situation is analogous to one where there are a number of applications asking for a shared page which has been cached. Even when the page changes, requests for it can be serviced from the cache instead of from the disk till such time it is written to the disk and removed from the cache. The number of clients serviced would then depend a little lesser on the disk bandwidth and more on the efficiency of admission control and cache management policies.

Let us assume without loss of any generality, that there is a stream $\mathcal{S}$ that is accessed by a number of people. During admission control, cache-unaware admission controllers will try to find out if there is enough residual disk bandwidth before admitting clients by calculating the amount of time required to service each and every required block request from the disk. They would reject sessions to clients trying to access $\mathcal{S}$ once all the disk bandwidth has been reserved or when the total bandwidth requested is more than that available. However, as mentioned earlier, what they fail to take into consideration is the fact that $\mathcal{S}$ can be present in part or fully in the cache and there is little or no disk bandwidth needed to service requesting clients. So using this mechanism, we can admit a client reading $\mathcal{S}$ in the presence of others reading it, without significantly increasing disk traffic. A light coupling of the admission controller and the caching mechanism for multimedia clients can result in *locality of information access* being exploited to a very great extent to provide far more clients with acceptable video content than is possible with traditional admission control policies. Apart from the ability to interact with the admission controller, the cache manager must have provisions to share the cache among multiple frequently accessed video or audio streams in order that multiple streams are allowed to be cached and results in increased cache-hit ratio. The amount of caching that is done for each stream could depend on the number of clients accessing them, the rate at

which the streams are being played and how far apart in time are clients reading the same stream etc.

We briefly outline some of important considerations for the cache management scheme to take into account to efficiently support (provide reasonably guaranteed service) multiple multimedia clients, which could have locality of access or are trying to access a stream some part of which is already present in the cache:

- The cache should be shared among multiple streams.
- Since the cache need not store the entire stream in the cache at the time of admission, a mechanism to pre-fetch data blocks for frames in the future into the cache may be required to increase the cache-hit ratio for future accesses.
- Pre-fetching of data blocks, in order to provide guaranteed service to the client throughout the session, must be carried out at a certain pre-determined rate and not simply where there is free disk bandwidth. This rate would probably be take into consideration how far separated in time the clients reading the same file are with respect to each other.
- Caching mechanism should not make any policy decisions about whether a client needs to be admitted or not.
- There must be an clearly defined interface to the admission controller to help it obtain information about the state of cache and about the content of various streams in it. This would help it make decisions on admitting a client.
- The admission controller must be able to influence what streams are cached and how much of data is cached for any stream if required.

We have described below the admission control mechanism that uses the cache present in any modern operating system. We have also outlined any support functionality required

and suggested possible directions for cache management in order to help this mechanism work.

Let us assume that the system has been servicing multimedia clients for sometime. This would mean that there is data for various streams in the cache. For cache replacement, we assume that a cache eviction scheme like LRU (*Least Recently Used*) is not used. Although, this is a very widely used scheme and works well for most purposes, it might not be suitable for our purposes since we would like to have data pertaining to earlier frames in the playback in the cache as opposed to the later ones. Therefore, when pages need to be selected from the cache for replacement, we assume the employment of a scheme where the pages for frames later in time are removed before those earlier. This lets us assume that the pages that carry data for frames at the start of the stream will be replaced after those that carry data for frames further down in time. This ensures whenever data for a stream is present in the cache, it is for consecutive frames starting from the beginning of the stream. This helps us keep the cache management simple and avoids our having to deal with complications like having some intermediate part of the stream in the cache during admission control. Let us assume that a client *i* needs a multimedia session. Let c_i be the number of blocks of data for a stream *i* in the cache. Let us further assume the cache manager can obtain information on the size of the stream from the file system data structures. Let the total size of the stream (file) be F_i blocks. Further, let the bit-rate of the client be, as in the previous discussions, r_i *bps*. Now, the original time taken for playing back the stream would have been

$$t_{\mathbf{orig}} = \frac{F_i}{r_i} \tag{3.6}$$

The total time taken to read the entire stream in the presence of cache data (assuming a constant bit-rate stream) is

$$t_i = \frac{(F_i - c_i)}{r_i} \tag{3.7}$$

Since the cache already stores some of the data that the client needs in the coming rounds, we can think of this time over which the cached data will be played out as the *buffer time*. Quite intuitively, it is unnecessary to completely commit all the resources required by the client to reading the disk, since some of the data has already been cached. Therefore, an interesting approach would be to find a reduced rate at which the client can be satisfied throughout the duration of the playback. This reduced rate would have to consider the proportion of cached data and clients, if any, in the system that are playing back the same stream. The latter consideration could make available data to the client which is lagging in time if the cache manager made sure that the disk blocks read and stored in the cache by the client ahead in time is made available to the one lagging. This way, practically all disk bandwidth usage can be eliminated for the client lagging in time for the playback of the same stream. However, the problem of capturing the ability to do this at the time of admission control (resource commitment) still remains.

Since the amount of data to be read from the disk is (F_i - c_i) over a period of t_{orig}, the rate of reading from the disk would then be:

$$r_i^{'} = \left(\frac{F_i - c_i}{F_i}\right) * r_i \tag{3.8}$$

It is clear from the formula above that the reduced rate would be lower for higher values of c_i. That is, more the data in the cache the lower would be bandwidth necessary to meet the deadlines for the client.

In our previous discussion for specifying a reduced rate in the presence of data in the cache, we wanted to be able to accomodate and make use of the clients playing back the same stream but separated only slightly (possibly less than the average time of a page in the cache) in time. Suppose at the time client *i* was attempting play back of stream $\mathcal{S}$ there was another client C_a, just ahead in time to which sufficient resources were committed for guaranteed service. Let us suppose that the cache replacement policy took into account

the number of simulataneously accessing clients before evicting the pages belonging to a stream, so that some percentage p_i of the number of pages required by the client *i* is provided from the playback of $\mathcal{S}$ for C_a. Then the reduced rate $r_i^{'}$ can be simply approximated to be a function of p_i as

$$r_i^{'} = (\mathbf{1} - p_i) * \left(\frac{F_i - c_i}{F_i}\right) * r_i \tag{3.9}$$

It is clear from equation 3.9 that by increasing the percentage of the data available to client *i* from the playback of C_a, we can greatly reduce the amount of disk bandwidth committed to client *i* at the time of admission. Another way to look at p_i is to consider it as the probability that a page required (that can potentially be marked non-swappable) can be provided to anyone needing it within a certain time limit. It can be intuitively thought of as being similar to cache-hit ratio provided by the underlying cache management scheme. This percentage can be determined statistically through simulation or by gathering statistics in the kernel for such instances. Further discussion of the statistical anaylsis for the determination of the probability with which pages can be made available is beyond the scope of this work.

We have attempted to outline in somewhat detail an admission control scheme not only to increase the number of clients serviced, but also reduce disk access by employing a cache management scheme and using that in conjuction with the admission controller. However, we have not implemented this scheme in the framework of `Clarity`. Our discussion of a cache-aware admission controller not only gives more closure to the treatment of various admission control policies proposed by us, but also serves to provide a point of further research and analysis.

3.4 Disk Scheduling

One of the most important mechanisms to support multimedia clients is efficient servicing

of disk requests so that data blocks are delivered to the applications in a timely fashion (according to their bit-rate requirements). While the admission controller restricts the number of simultaneous multimedia clients in the system, it is up to the disk scheduler to ensure that the admitted clients get the quality of service they were promised during admission. Additionally, the disk scheduler also helps perform optimizations to the number of blocks needed for retrieval depending on the type of the client. One of the biggest problems faced by disk schedulers are overflow rounds. These are rounds where the number of blocks to be retrieved for the admitted clients is more than the number of blocks that can be retrieved by the disk scheduler. The challenge here is to reduce the service time by *dropping* block requests. However, this is a non-trivial task and requires careful considerations so that dropping of the minimum number blocks requests will bring down the service time below the available time.

Given the fact that different applications might have different service requirements, it is important to incorporate as much knowledge as possible into the framework supporting these clients. We provide a 2-tier approach for supporting multimedia applications, in the form of an admission controller and a disk scheduler. While the admission controller tries to make policy decisions about accepting or rejecting requests for multimedia clients, the disk scheduler is responsible for enforcing those policy decisions. In order to complement each other effectively, it is important for the admission controller and the disk scheduler to have as much information as possible about the various service requirements and inherent characteristics of multimedia files being read by the clients. We use information about files being retrieved to increase multimedia clients being admitted for service and provide improved service to the admitted clients in terms of more effective sequencing of requests and discard of requests in overflow rounds etc.

Our approach has been largely motivated by the fact that in order to support multiple application requirements, the disk scheduler needs to be as intelligent as possible. It

should be able to identify the application requirements and translate them in mechanisms to be carried out in order that a large proportion of clients that are being serviced are satisfied.

In order to support multimedia applications efficiently, we have incorporated the following features into our disk scheduling algorithm:

- Use of multiple service (application) classes in order to match application needs to static system descriptions of those. This helps `Clarity` align disk service to application requirements by resequencing the order in which disk requests are serviced.
- Protection of service classes from one another.
- Protection of multimedia clients from one another.
- Adaptability to changing system load.
- Minimization of seek and rotational latencies in accessing media blocks.
- Awareness of the presence of caching policies in the operating system.
- Ability to exploit inherent media characteristics to enable mechanisms for handling loss.
- Ability to support variable bit-rate applications.
- Efficiency in computation.

The following paragraphs discuss each of the above mentioned points in detail and describe our approach to design in order to support these policies.

3.4.1 Alignment of disk service through Service Classes

In order to align service provided by the disk to the various applications, we follow a two-step method. Firstly, we have identified different classes into which we attempt to group applications as described earlier in section 3.1. This gives us a mechanism to broadly determine what the disk requirements are going to be and how (if any) re-sequencing of disk requests from these applications needs to take place. Secondly, we carry out application-specific optimizations like reading data in layers for MPEG clients and averaging out loss for real video clients by dropping frames that are delayed at the end of the round. While it is highly desirable to have each and every application completely meet its requirements in terms of loss, delay and jitter, such a service can be only provided at the risk of the disk scheduler working with possibly too many variable parameters, a very large number of classes, and a huge overhead of computation. Therefore, we have adopted this 2 step mechanism to achieve alignment of disk service to application requirements.

3.4.2 Protection of Service Classes

One of the greatest concerns in handling multiple kinds of applications is that of *fairness*. It is very important that each application get its fair share of the bandwidth that it was admitted for. As long as the traffic for each of the classes is low, there is little concern about any of the classes not getting its share of the disk bandwidth. However, when traffic of any one application or a class of applications generates a huge number of disk requests, it is possible they consume much more than their share of the bandwidth leaving others deprived of any bandwidth at all. While this does not degrade performance of best-effort applications too much, it could mean missing of a number of deadlines for the multimedia applcations. As a result, the clients belonging to other application classes suffer resulting in degraded performance. Our approach to handle this situation involved

bandwidth allocation to the various classes. We reserve a portion of the disk bandwidth every round for servicing multimedia and non-multimedia requests. Servicing requests for clients is purely subject to availability of bandwidth for the application's class. In the absence of adequate bandwidth, the request is queued till such time that bandwidth is available. We shall discuss means by which bandwidth can be become available to an application in later paragraphs. Also, one has the option to further partition the bandwidth allocated to the multimedia class among the delay and loss tolerant applications. Thus, we have a policy in place for ensuring that too much traffic from one type of multimedia service class does not result in significantly reduced bandwidth for others.

In our approach, the disk bandwidth that is available to the disk scheduler is statically divided among the various classes. The percentage of disk bandwidth to be allocated to various applications can be determined by an analysis of different contents on the server and their bandwidth consumption. Any class that exceeds its quota of disk service time is denied service unless there are no clients left that need disk service. For our purposes, we have two classes, which are: (1) real-time class (2) non real-time class. We keep track of the amount of time in a time period that the disk has spent servicing requests from these two classes. When the disk scheduler realizes that a particular class has consumed all of its fair share of bandwidth, it blocks all the disk requests for applications belonging to that class until either the time period expires or bandwidth becomes available. This ensures that an increased number of clients requesting data from the disk do not adversely affect the bandwidth guarantees made to other service classes.

3.4.3 Protection of Clients Within a Service Class

Apart from ensuring reservation of bandwidth to classes, we also try to protect applications from the same multimedia class from one another. The idea is to give each application only as much as it has asked for before preceeding with the service of any more

blocks for that application (for example, when the operating system is trying to cache). Our approach is to let the application tell the scheduler before reading how much it needs. Using this information, the disk scheduler can then block requests from an application that has obtained all the data blocks it needs. It must be remembered, that when the application informs the scheduler at the beginning of each round how much data the scheduler needs to read, it is up to the disk scheduler to guarantee anything more than minimum that the client was admitted for. This mechanism prevents a client from asking for too much after admission at a rate far lower.

In order to illustrate this mechanism, let us suppose that there are three applications that have been admitted with 20 *KBps*, 40 *KBps*, and 50 *KBps* respectively. Let us assume without any loss of generality that none of the clients took into consideration the loss percentage in stating this data rate requirement during admission for a multimedia session. Assuming that the block size of the file system in which these files have been stored is 1 *KB*, and they need to read in 20, 40, and 50 blocks of data on an average every round. However, these applications might want to read more depending on whether they are constant or variable bit-rate applications. Now at the beginning of every round, let us assume each of the clients states that it needs 25, 45, and 55 blocks respectively. During the course of a round, we keep track of the number of bytes retrieved for each client. It is not necessary that all data requirements for a client are met before the other clients are serviced. Service will mostly take place in an interleaved fashion with each slice of execution ensuring partial fulfilment of the bit-rates of each client. Suppose that the client that needed 45 *kbps* gets all the guaranteed data (40 blocks) but wants more. When the request for the 41^{st} block comes in, the disk scheduler checks to see if all the clients that were admitted have been satisfied by going through the linked list of clients. If they have not been, it blocks the requests for this client. This procedure is continued, until clients that have not obtained their data are forced to be executed by the process scheduler

since the others have been blocked. When the last client has its data requirements met, `Clarity` signals all the multimedia clients to compete for the remaining time slice (if any available) to try and meet their data requirements for that round. In this manner, we ensure that no client gets more than the share of the bandwidth allocated to its class.

3.4.4 Adaptability to Changing System Load

We also implemented a mechanism to make allocation of bandwidth/time slice to real-time and non real-time class adaptive to varying loads (dynamic). When starting a time period, the time allocated to the various service classes is static as decided at the time of kernel compilation. As the round proceeds, we keep track of the amount of disk service time that is getting used for each of the service classes using the method outlined in description of measurement of time for disk access and depending on the dynamics of disk usage we carry out the process of re-assigning disk bandwidth for the remainder of the round. This can be done in a number of ways. One is the aggresive approach, where the amount of unused bandwidth is completely assigned to the other application classes. In our approach, it would amount to all the unused bandwidth being passed to either the non real-time or real-time class depending on which is under-utilizing its quota. Our scheme is to use a semi-aggresive policy for re-assignment of disk bandwidth reservation. Using this mechanism, we re-allocate a part of the bandwidth reserved for the class that we believe would under-utilize its bandwidth to the one that might need it. Needless to say, that this is based on heuristics that a vast difference in service utilization times (derived from measurements at regular intervals during a round) is likely to translate into bandwidth wastage by one of the classes at the end of the round.

3.4.5 Minimization of Seek and Rotational Latencies

Linux uses CSCAN algorithm to service disk request. We have modified this algorithm to incorporate knowledge of multimedia clients, so that it is a variation of the original algorithm. Our approach is a variation of CSCAN-Earliest Deadline First (EDF). We assume that the deadlines for all the disk requests is the end of the *round* and perform CSCAN optimizations to sequence the requests so that they incur minimum seek and rotational latencies. Furthermore, requests that belong to the different service classes are treated differently as far as their insertion into the queue is concerned. For example, the delay sensitive clients are given preference over loss sensitive clients, since their deadlines need to be met as much as possible. However, in order to ensure that delay tolerant clients that have already been delayed do not suffer monotonically increasing delay in the future rounds, the insertion of their requests is carried out in preference to the delay intolerant ones. Thus, in this sense it is a variation of the CSCAN-EDF algorithm. However, it must remembered that all the requests belonging to either classes are treated more broadly as multimedia requests which are different from non real-time requests and optimizations carried out accordingly.

3.4.6 Awareness of Caching Policies in Linux

Caching is a very important part of an operating system. It prevents excessive disk requests being sent and significantly reduces the service times of disk access. When a disk request is received for a certain number of blocks, *Linux* requests those blocks for the client from the disk. When the blocks are received, the disk scheduler passes them onto the requesting client. *Linux* also maintains a copy in memory of the blocks that were retrieved. This is so that any subsequent access (before the page is removed from cache) to this page, is handled by retrieving data from the page in memory. This avoids the huge

cost of sending another disk request. Another important aspect of having a cache is the fact that it acts as a storage for blocks that are adjacent to the block that is requested by a client. Whenever a disk request for a block is received, *Linux* creates a request for accessing the adjacent blocks on the disk in the hope that they will be subsequently be needed by the same application. We make use of this mechanism when scheduling disk requests, since we can avoid having to deal with requests that can be satisfied in the cache itself. This reduces any computational overhead associated with processing requests whose blocks are in the cache and is expected to reduce any performance hits accompanying making decisions about how requests that require access to the disk need to be handled.

3.4.7 Ability to Exploit Inherent Media Characteristics

As mentioned before, media like MPEG can be handled differently from other streams. The fact that they are layered can be exploited by the disk scheduler and the application to ensure that there is a certain controlled degradation of quality when there are *overloaded* rounds. Overloaded rounds occur when the disk scheduler lacks the bandwidth to schedule multimedia requests so that the appropriate clients' *QoS* requirements can be met. This typically happens in the presence of variable bit-rate streams like MPEG that could read different number of blocks every round although the frame rate is a constant. As a result when a number of MPEG clients read substantially more than the average bit-rate that they were admitted with, `Clarity` might lack the disk bandwidth to meet the deadlines for all the clients. When this happens, the disk scheduler has to drop requests. `Clarity` attempts to spread the loss as evenly as possible by dropping disk requests from all the admitted clients.

In case of MPEG-like clients, the requests can be dropped in such a manner that they constitute a layer's worth of data. For CBR clients, it would amount to something like

media scaling, with lesser than normal amount of data being read from the disk. With layered streams, the disk scheduler does not attempt to drop requests intermittently so as they add upto a layer's worth of data. It just decides that the client would read one layer less than it normally would. The client can employ whatever techniques it chooses to read data efficiently so that it does degrade the video quality too much. One of the very important things that we have avoided doing is imposing exactly what data is available to the clients. There are no restrictions imposed by this approach about what frames can be read given the amount of bandwidth that can be used to do it.

3.4.8 Efficiency in Computation

Given the complexity of our framework, it is quite obvious that there is a lot of computation overhead involved in providing efficient service. In order to keep `Clarity` efficient, it is important to achieve the right balance between how much CPU power can be used in making various decisions versus how much can be sacrificed for simplicity in design, code and execution. We have made a conscious attempt to make simplifying assumptions where we believe that the merits of computation do not exceed the time overhead associated with it. For example, we have assumed that the duration of a round, although dependent on the client with the highest bit-rate, is a constant (1 *second*). This helps us avoid a lot of computational overhead associated with calculating round times, synchronizing clients, making synchronized measurements and validating those measurements by possible adjustments in timing in the kernel. We also attempt to limit the increase in the size of the kernel arising from addition of the admission controller and disk scheduler. We shall evaluate the increase in kernel code and image size as a result of `Clarity`.

Chapter 4

Implementation of Clarity

"STRATEGY is; A style of thinking, a conscious and deliberate process, an intensive implementation system, the science of insuring future success"

Pete Johnson

In order to evaluate our approach and the performance of our framework, we implemented an admission controller and a disk scheduler in the Linux kernel. Our design and implementation does not mandate the usage of either the admission controller or the disk scheduler unless required by the user. This is so that normal system operations can be carried out in the kernel with support for multimedia applications. Apart from describing the implementation of the admission controller and the disk scheduler, this chapter also details some of the secondary issues like timing, measurement of DMA transfers etc.

4.1 System Configuration

All our implementation and experimentation was carried out using a Pentium processor running at 166 MHz. The linux kernel reported 66 bogomips for the processor speed after

calibration. The system had 96*MB* of RAM and a hard disk capacity of 4*GB*. The Linux kernel version was 2.0.36.

4.2 Admission Controller

The admission controller has been implemented as a loadable kernel module [34, 35]. This means that while it is not part of the kernel at all times, it can be loaded whenever admission control needs to be used for admitting multimedia clients into the system. Once it has been loaded, it can interact with the rest of the kernel just like it is a part of the kernel. It can share data structures with the disk scheduler if required. This is especially useful in order to perform measurement-based admission control. When being unloaded the admission controller relinquishes all the control that it has over clients being admitted and activates all the clients whose disk requests are pending.

As described earlier in Figure 3.1, we see that from an application's perspective, the admission controller is the interface to the disk scheduler. We have provided this interface in the form of system calls that can be used by applications to reserve disk bandwidth and get guarantees for delivery of data.

There are two ways in which support for admission control can be added to the set of system calls supported by Linux. The first method is to modify the existing system calls to perform admission control. This would mean change in the parameter list of the system calls since there a number of parameters that need to be supplied to the admission controller in order to help it determine whether the client can be accepted for a multimedia session or not. Similarly, the return values could also change since the return value from the system calls need to be interpreted by the multimedia clients differently from the way normal clients do it. This is because the return values could contain information on how much of the requested bandwidth has been allocated or the bit-rate that has been

admitted. However, if this is done it would interfere with the execution of normal applications which use system calls like open(), close() etc. since they are not aware of the extended parameter list or the different return values. Hence, this approach of modifying the existing system calls would essentially break all applications written using them and is not an acceptable solution. The second method is to introduce new system calls. These calls would have additional parameters and return values that can be understood by multimedia clients requesting sessions. They would then incorporate admission control into the functionality that exists in the system calls they are replacing. This way we can separate the set of system calls used by normal applications and multimedia clients. We decided to implement our admission controller in the latter way so as to minimally impact any existing applications and to aggregate changes related to `Clarity` separately.

There are a number of system calls that we have introduced in order to allow clients to register themselves as multimedia clients. This is so that the kernel knows that they are multimedia clients to provision bandwidth based on their requirements. Due to the manner in which we have made system calls available to clients, we do not mandate the usage of our framework for multimedia clients. However, it must be remembered that for the clients that do not make use of these system calls, there are no guarantees for delivery of audio and/or video content. Since there is no minimum rate of data delivery for these clients, it could result in poor performance. This is since the disk requests from such applications will be serviced as normal non real-time requests and will have to compete with other similar applications for disk bandwidth. However, when using system calls for setting up a multimedia session, it is quite possible that due to lack of bandwidth the session request is rejected and the client has to try at a later time in order to get data.

The following section gives a brief description of all the system calls that have been introduced. These system calls are considered a part of the admission controller.

4.2.1 System Calls

We have provided two system calls that CBR clients are expected use to request a multimedia session. They are:

(a) **sys_mmdopen**(`const char*` *filename*, `int` *flags*, `int*` *client_id*, `uint` *min_rate*)

This system call is used by a multimedia client to open a file specified by *filename* using the *flags* that the `sys_open()` operation takes as an argument. The *client_id* is a value that is returned to the client to be used for reading data during the multimedia session as shall be explained later. The client, in fact uses a combination of the file descriptor and client id to read every round. Additionally, the client specifies the minimum bit rate that it wants for the duration of the session. It is assumed that *min_rate* is used to specify the rate that the client would want after losses in the form of dropped requests. Thus, the client could potentially ask for more than this rate every round. It is up to the disk scheduler to provide more than this although the rate specified here is what the disk scheduler tries to give all participating clients before any client is served more. This call is used by clients that are **delay tolerant** and **loss intolerant**. This function performs the task of admission control by finding out if there is enough bandwidth to admit the client. If the client is admitted, then it is returned a valid file descriptor and client id. Additionally, disk bandwidth is also reserved for the admitted client. In case of insufficient bandwidth availability, it returns -1 for the client id and invalid file descriptor (same as that returned by `sys_open()`).

(b) **sys_mmlopen**(`const char*` *filename*, `int` *flags*, `int*` *client_id*, `uint` *min_rate*)

The parameters for this function mean the same as they do for `sys_mmdopen()`. However, in contrast to `sys_mmdopen()`, this call is used by clients that are **loss tolerant**

and **delay intolerant**. All admitted clients are marked as being loss tolerant so that when the disk scheduler encounters overloaded rounds, it drops requests from these clients instead of waiting until bandwidth is available. This call also returns -1 for the client id and invalid file descriptor (same as that returned by `sys_open()`) when the bandwidth requested is not available.

In addition to the two calls for CBR clients, we have provided system calls for MPEG-like VBR clients to use. This is because there is a difference in the way admission control is carried out for those clients since it is assumed that the MPEG files are present in a layered format. The following two system calls are for admitting MPEG clients.

(c) **sys_mpegdopen**(`const char*` *filename*, `int` *flags*, `int*` *client_id*, `uint` *min_rate*, `int` *max_layer_required*)

This system call is used by a multimedia client to open a file specified by *filename* using the *flags* that `sys_open()` operation takes as an argument. The *client_id* is a value that is returned to the client to be used for reading data during the multimedia session as shall be explained later. The client, in fact uses a combination of the file descriptor and client id to read every round. Also, the client specifies the minimum bit rate that it wants for the duration of the session. It is assumed that *min_rate* is used to specify the rate that the client would want after losses in the form of dropped requests. Thus, the client could potentially ask for more than this rate every round. It is upto the disk scheduler to provide more than this although the rate specified here is what the disk scheduler tries to give all participating clients before any client is served more. The extra parameter in this system call is *max_layer_required* which indicates the number of layers from the file that need to be returned every round. This call is used by clients that are **delay tolerant** and **loss intolerant**. This function performs the task of admission control by finding out if there is enough bandwidth to admit the client. If the client is admitted, then it is returned a valid

file descriptor and client id. Additionally, disk bandwidth is also reserved for the admitted client. In case of insufficient bandwidth availability, the cass reduces the numnber of layers to be supplied and tries to ascertain if the bit-rate requirements for those layers can be supported. This process is repeated till either admission control succeeds or no layers can be served. In this , the function returns -1 for the client id and an invalid file descriptor (same as that returned by `sys_open()`.

(d) **sys_mpeglopen**(`const char*` *filename*, `int` *flags*, `int*` *client_id*, `uint` *min_rate*, `int` *max_layer_required*)
All the parameters mean the same as they do for `sys_mpegdopen()`. The only difference being that this call is used by clients that are **loss tolerant** and **delay intolerant**, with the clients marked as such after upon admission for helping the disk scheduler take appropriate action in over-loaded rounds. All functions for admission control performed in `sys_mpegdopen()` are also carried out by this function.

All the above mentioned system calls have been inserted statically in the system call table. Since there were problems with allocating random contiguous positions in the system call table, we had to allocate indices carefully so as not to disrupt normal system operations. Table 4.1 shows the assignments of indices in the system call table.

Since all the mechanisms for handling admission control have been implemented as system calls, our admission controller is essentially a part of the kernel. While it is possible to implement the admission controller as a user level interface, for reasons of simplicity and necessity of communication with the disk scheduler, we have implemented it in the kernel.

The admission controller constantly interacts with the disk scheduler to measure the times taken by the various disk requests so that it can admit clients more intelligently.

System call name	Assigned index
sys_mmdopen	169
sys_mmread	172
sys_wfround	173
sys_mmlopen	179
sys_mpeglopen	180
sys_mpegdopen	181
sys_mmclose	184
sys_reserve	185

Table 4.1: System call table allocation for the various system calls

Thus, there are some data structures shared between the admission controller and the disk scheduler. The disk scheduler attempts to schedule disk requests optimally, and also communicates to the admission controller about the various times it is taking for multimedia clients. The admission controller then uses this information to correlate to the times that it had been using for admission control and corrects them accordingly. Thus, there is a feedback mechanism set up between the admission controller and the disk scheduler to constantly measure and fine tune performance.

4.3 Disk Scheduler

The disk scheduler is central to providing good performance to the multimedia clients that have been admitted. It serves to not only re-sequence disk requests in a manner that will reduce the overall time for service, but also supports a certain guaranteed rate for the multimedia clients. It is also important that the disk scheduler, while attempting to meet the deadlines of all audio and video clients, does not neglect the non-real time clients. It

is imperative they do not suffer by having huge response times.

4.3.1 System Calls

Quite obviously, the system call `sys_read()` is insufficient to let the multimedia clients read at a certain rate. Since there is no new information that can be passed on to the disk scheduler from the application, the disk scheduler will be unable to distinguish between multimedia and non-multimedia requests. Hence, we have introduced a new system call sys_mmread() that can be used to mark the requests as multimedia requests. The following is the description of the system call:

(a) **sys_mmread**(`FILE *`*fp*`, int` *client_id*`, int` *class*`, char *`*buf*`, int` *count*)
In this system call, *fp* refers to the file pointer that is returned from one of the system calls that is used by multimedia clients to request for a multimedia session. The *client_id* is also returned as one of the values by them. This is used by the disk scheduler to provision bandwidth, queue, deny or drop the request from the client indicated by the *client_id*. *class* refers to whether the client is delay or loss tolerant. *buf* is the buffer that holds the data of *count* bytes which has been requested by the client. By using this system call, all information that is required to prioritize disk requests from this client are passed on to the disk scheduler. This call functions pretty much like `sys_read()` in that, it does not obtain all the data that the client might want in one invocation. The clients need to repeatedly use this function call to get all the data they want in a round. Since the amount of buffer available in a system might be limited, it is not feasible to reserve huge buffer space for a client so that only one call needs to be made. This could mean that another client is denied service until such time that buffers allocated to a particular high bit-rate client are freed. This mechanism of having to read each and every time is in consonance with the way it is normally done by any client and helps multiple high bit-rate clients to

function without buffer issues in a low RAM environment. The number assigned to it in the system call table has been mentioned in Table 4.1.

When clients request service from `Clarity`, the CBR clients do not have a problem specifying the bit-rate they want. However, the VBR clients have a constant frame-rate and possibly varying bit-rates. In order to support VBR clients, `Clarity` provides a system call that lets the clients specify in each round how much it wants service for. A description of `sys_reserve()` follows.

(b) **sys_reserve**(`int` *client_id*, `double` *blocksRequested*)
The first parameter is the *client_id* that is obtained from the client's call to one of the four functions to initiate a session. The second parameter as the declaration states is the number of blocks the client wants to read in the present round. This is essentially so that the disk scheduler can attempt to serve more than the average number of blocks to the VBR client. Of course, the availability of extra disk bandwidth is always subject to the other clients meeting their *QoS* requirements. It helps the disk scheduler make better decisions about using residual bandwidth when all the multimedia clients have been serviced, when the client informs the scheduler right at the beginning of the round. While in practice, the client can inform the scheduler at any point in the round, it is helpful if it does so at the beginning so that the disk scheduler can attempt to service more than the client's minimum date requirements.

4.3.2 Implementation Considerations

In the following paragraphs, we shall discuss implementation issues and details pertaining to the disk scheduler that helped us provide the features mentioned in the chapter on our approach (Chapter 3). First, we shall discuss factors that directly affect the way our scheduler works and are very relevant to a complete understanding of our implementation.

Playback of Multimedia Files

The disk scheduler gives data to the clients which then play it back. Typically, in order to prevent the clients from missing deadlines, the scheduler must be ahead of them. This is to say that by the time the client finishes playing out the data for a particular round completely the disk must have already put enough data in the client's buffer that it will continue to play it back unhindered. Simply put the disk must write to the client's buffer earlier than it attempts to read from it. Our disk scheduler does not maintain ring buffers that would allow the data being read being stored in one half of the ring while the client reads from the other half. We assume that this mechanism has been implemented in user space by the application that is requesting data on behalf of the actual client. Therefore, the applications [1] working on behalf the actual clients store data in their internal buffers and manipulate the buffers so that data is constantly fed to the clients at the required rate. So as far as the disk scheduler is concerned, it needs to deliver a certain amount of data in the time period (a point discussed in the following section) to the application. The disk scheduler does not worry about the actual playback itself.

Determination of Time Period

The determination of the time period over which the clients will be serviced is very important. It is the time in which the following tasks (a subset of all the tasks performed by our disk scheduler) are performed:

1. Attempt to deliver all the data requested by the client. At the end of this time, it is assumed that the time available to fetch data in that round is over and the disk scheduler re-initializes the blocks read for each clients to start servicing the clients

[1] Henceforth, we shall use "*clients*" to mean applications working on behalf of the actual clients (local or remote).

all over again for another 1 *second* (not necessarily in the same order as in the previous round since *Linux* does not guarantee it).

2. Measurement of disk access times taken, blocks read, blocks not serviced, deadlines missed etc. for various clients.

3. Determination of excessive bandwidth usage by any class/individual clients is made so that classes/clients can be protected from each other.

4. Optimizations in measurements and/or service for clients being serviced in that round.

We observed that data rates are specified as the amount of data (in bits, bytes, or any other multiples of these) per second. Therefore, in order to help reduce the complexity of measurements and calculations (to provide reasonable performance with a low end machine as the server), we have assumed that the time period is 1 *second*. Since the disk scheduler does not directly concern itself with the playback of the file, it aims to deliver whatever the application supporting a client determines must be delivered for a smooth playback.

Alignment of Clients with the Time Period

Since we have mentioned that we service clients in *rounds*, there needs to be a mechanism to deliver data, maintain statistics and measure performance over a time period that starts at the time the client is granted admission and starts requesting data. Quite obviously, if we maintain a timer for each and every client the system CPU will soon be loaded and will be unable to handle the client requests efficiently. This is since every time a client's time period gets over, an interrupt will have to be generated to signal end of *round*, values will have to be recorded and variables reset. Also, the process of adapting to the changing

workload will be an extremely complicated process. In short, it will be extremely inefficient to have separate timer mechanisms for each client. We have deviced a method by which clients will be in *sync* so it can be safely assumed that timing is the same for all clients. The effect of synchronizing the clients is achieved by the implementation of a system call `sys_wfround()`. This considerably reduces the complexity of maintaining state information, makes it easier to adapt to the load, measuring performance and making other service decisions.

As soon as a client is granted admission, the client calls `sys_wfround()`, which makes the admission controller put it on a `WAIT_QUEUE`. This `WAIT_QUEUE` is shared by (1) clients that have been admitted in that *round*, and (2) clients that have finished receiving their service for the ongoing *round*. When the 1 *second* timer interrupts signalling the end of the *round*, the disk scheduler wakes up all the clients on this `WAIT_QUEUE`. This is done so that when the next *round* starts the new clients are woken up along with the old clients for service so that there is no delay in letting the new clients request disk service. We assume that the client is well-behaved in this respect and do not concern ourselves with a situation when the client does not put itself on wait for the end of the round.

Measurement of Disk Access Time

While it is theoritically possible to predict the disk access time from the application of various schemes that model a hard drive, in reality it is very difficult to make measurements for a particular block. In many cases, when the disk controller receives a request for reading a block, it caches the entire track. If there is a request for a block on the track that it cached before its expiry from the controller's cache, it is serviced from there. In our implementation, we mark each and every `request` as being a non real-time or a real-time request. When a DMA transfer completes and the kernel is interrupted,

`end_request()` is called. `end_request()` marks all the `buffer_head` structures dirty (having been written to) so that the application that owns the `buffer_head` structures can pick up the data from them. We count the number of `buffer_head` structures that are freed for the non real-time and real-time clients. When all the `buffer_head` structures associated with a `request` have been marked, we subtract the time (obtained with `do_gettimeofday()`) when the DMA transfer began from the time that it ended. The time taken for requests from each class is calculated as a weighted average of the time taken for the DMA transfer.

Measurement of Bandwidth Consumpion by Service Classes

This is an aspect that is very important from the point of implementation. Since there is little precious bandwidth available to the various service classes, it is very important that measurements are as accurate as possible. In our implementation, the time taken by various requests is not predictive but measurement based. This translates into times for service being available only after the actual disk service has taken place. While this might seem fine, there is a subtlety involved that we have to be careful about. Suppose that there is a request that is formed out of a number of requests. In Linux, a request can be composed of reads to adjacent blocks (requests for which could come from different clients). Suppose servicing this request takes 40 *milliseconds*. Since the requests are serviced through DMA transfers, the CPU is potentially capable of putting more requests on the service queue before the disk notifies it of the previous request having been serviced. Suppose a request (R) is put on the disk service queue during the service of the previous request (P). Let us further assume that both requests belong to the same class (but different applications). If the aggregation of disk service times till request P is nearly equal to the class time slice available in the round, it is quite possible that servicing R will result in overconsumption of bandwidth by the class to which P and R belong. This

is since a request is definitely processed if it is moved to the service queue for the disk. While it is possible that a check can be made to see if the bandwidth for the class that the request belongs to has been exceeded before passing it onto the disk controller, it is not a very clean solution since the `request` could have been coalasced out of multiple smaller requests from clients belonging to different classes. Furthermore, it would be extremely complicated if we were to take out only those smaller requests which belong to the class that has already consumed its fair share and re-arrange others. In fact, it would break the consistency of the entire `request` structure since no longer are blocks adjacent to each other. In order to avoid such complications, we felt it is a better idea to check before a request from a client is sent for adding into the disk service queue where the request could either get merged with existing ones (since it is for a block adjacent to some already asked for) or get added as a separate `request` itself.

In order to prevent over-usage of bandwidth, we have made our measurement of disk access time for a particular request *predictive*. This is to say that we use the average value for disk access from previous DMA transfers and calculate the amount of time that will be consumed if a particular request is let through. If this exceeds the class's share of the disk bandwidth, it is blocked. Otherwise, it is sent for addition to the disk service queue. We also keep making adjustments to the average disk access time per block by small amounts (ϵ), which is obtained by comparing the predicted value to the value obtained from the latest DMA transfer. This is to make predictions of disk usage take into consideration the changing state of the system.

Protection of Service Classes

As explained earlier in our approach, it is very important that presence of clients belonging to one class should not be allowed to affect the performance perceived by the clients of another class. While, it is not possible to completely eliminate effects of classes on

each other we have attempted to minimize the performance degradation of any client due to other clients in the system. Using the procedure for measurement of disk time usage outlined in the previous section, when it is found that a particular class has used up its fair share of the bandwidth, the scheduler blocks all the requests from that class and puts the client on a `WAIT_QUEUE`. When either the round gets over or there is additional bandwidth available, the scheduler starts processing all the client requests that had been blocked.

Protection of Clients within a Service Class

While it is very important that different kinds of clients receive good service, it is equally important to ensure that clients in one class do not force others in the same class to perform poorly due to their over using the bandwidth available to the class. This is a very important aspect that schedulers fail to consider while supporting multimedia clients. While every class is provided with a certain percentage of the bandwidth every round, there is no mechanism to limit the amount of data that is requested by a client in each round. Also it could be disastrous to hope that the clients are all going to be well-behaved and request only as much as they specified at the time of admission. Therefore, as described in detail in section 3.4.3 we have incorporated a mechanism in the disk scheduler to limit the service provided to any one client. When a client has obtained its stated requirements for a round and wants to read more, the disk scheduler checks to see if all the remaining clients in the same class have been satisfied (if they have any stated minimum requirements). If they have not been, it blocks the requests for the requesting client and puts it on a `WAIT_QUEUE` (defined in the *Linux* kernel [36, 37, 38]). This procedure is continued until clients that have not obtained their data are forced to be executed by the process scheduler since the others have been put on a `WAIT_QUEUE`. When the data requirements of all the clients are met, the disk scheduler wakes up all the multimedia clients using the

`wake_up()` call so that they can compete for the remaining time slice (if any) to try and meet their data requirements for that round.

Essential clients versus Non-essential clients

While the disk is generally used to store and retrieve data from the files, most (and it might not be inaccurate to say all) modern computers use the disk for Virtual Memory Management (VMM). This means that apart from disk requests being sent by normal non-real time clients, they could well be sent for loading a page into the main memory on a page fault. While, it might be acceptable to delay processing requests from a non-real time client simply reading from a file, VMM related requests might have to be serviced irrespective of whether there is enough disk bandwidth to service non-real time clients or not. In such cases, bandwidth from any class that has it needs to be used to service the page fault since the page to be retrieved could be critical to the running of another application (which could well be a real-time application). We identify such disk requests as *essential* non-multimedia disk requests. Therefore, we have classified non-multimedia clients as essential and non-essential and allow disk service to the essential ones under all circumstances.

Adaptability to Changing System Load

In our semi-aggresive implementation, we re-evaluate the amount of time that is necessary for different classes in the remaining part of the period every 200 *milliseconds*. With sufficient load of both classes of clients, one would expect a fairly even distribution of disk usage every 200 *milliseconds*. However, depending on the amount of disk requests generated by each class in the round, we attempt to re-allocate bandwidth so that in the absence of disk requests from clients of one class and a large under-utilization of disk bandwidth, disk can be used for servicing other classes if required in the future. When

we detect that a class is consuming less than its share of bandwidth we extrapolate the bandwidth requirements for a class from the measurement obtained so far in the round. This takes into consideration the amount of time that the disk has been idle. We then re-assign the excess bandwidth and increase the time slice of a class that needs bandwidth. The amount of bandwidth (in *milliseconds*) that is re-assigned can vary depending on the statistical analysis of client load depending on usage characteristics in various times. However, if it is found that some of the multimedia clients are not meeting deadlines due to re-assignment of some of the originally assigned bandwidth, `Clarity` gives back the required bandwidth to the multimedia clients to avoid deadline violations.

We carry out this process of re-assignment in order to maximize the *disk utilization* and let classes that need disk bandwidth take it from those that do not. It must be remembered that this method is predictive in nature and a surge in the number of clients in a class during the course of a round can cause this mechanism to breakdown. However, we believe that this method of re-allocation is both useful and practical because the effects of this mechanism are localized to a single round and in most of the cases, usage of bandwidth measured at regular intervals during a round are a good representation of the load on the system due to different clients.

Minimization of Seek and Rotational Latencies

As mentioned earlier, *Linux* uses CSCAN algorithm to service disk request. Our implementation is a variation of CSCAN-Earliest Deadline First (EDF). We simply assume that the deadline for all the multimedia requests in the end of the *round* but do not assign deadlines for each and every request. This has been specifically done to avoid a fairly significant overhead of calculating and possibly re-calculating the deadlines for all the requests in the queue as and when insertion take place. Also, we are prevented from assigning each request a deadline since the disk scheduler does not know *apriori* how many

requests are going to be received each round (new clients can join, variable bit-rate applications can request more data etc.). This would mean the scheduler cannot determine how soon a request needs to be scheduled to meet the requirements of all other multimedia requests in that *round*. Additionally, this would not allow the disk scheduler to coalesce requests for adjacent blocks to be combined into a single request (`struct request`). This merging of requests is carried out since the time for reading a sequence of blocks is far less than that taken for reading them through individual calls. This also tends to make use of hard disk controller optimizations where entire tracks are read and cached on the hard drive when a block of data is requested so that requests for subsequent blocks are served from the cache rather than the disk. This results in a huge reduction of service times.

Assigning the end of the *round* as the deadline lets us perform CSCAN optimizations to sequence the requests so that they incur minimum seek and rotational latencies. We have not explored the option to allow coalescing of only those requests that belong to the same class. This mechanism would partly allow us to assign deadlines to the various multimedia requests, and insert non real-time requests whenever there is a *slack*. It is our belief that the gains from optimizations arising from assignment of deadlines to the individual requests do not outweigh the computational overhead incurred in the process (especially on slow machines).

Awareness of Caching Policies in the Operating System

We have carefully avoided having any core scheduling functionality before the caching layer so that all of it is below the layer that caches the disk blocks in memory. All our disk related functionality has been incorporated in the `ll_rw_blk.c` which contains the routines for low level read and writes like `ll_rw_block(), make_request()` `and add_request()` When a client requests data, `Clarity` receives the request only

if the requested block is not in memory already. This mechanism is especially helpful for handling MPEG clients, since they do a lot of random access in order to generate frames for display. Thus, all such random seeks (if fairly close to the present position in the file being read) can be handled by the cache without sending a disk request. This implementation is also consistent with our view that caches can be effectively used to service clients separated in time and reading the same file and has been done with a view to supporting cache-based admission controllers proposed by us in chapter 3.

Using Inherent Media Characteristics to Service Requests

In this work, we have tried to differentiate between two very different classes of multimedia applications, namely layered (MPEG) and non-layered (RealVideo, RealAudio etc.). In order to service MPEG clients, the disk scheduler treats the file as a layered stream with *I*, *P* and *B* frames contribution to increased frame resolution. However, it must be remembered that the disk scheduler does not really understand the encoding/decoding process for the files and that the layering is understood purely in terms of reduction in the number of disk blocks returned to the application. The disk scheduler estimates the amount of time that it would take to process the remaining requests using the average time per block. When it determines that the *round* is going to be over-loaded, it drops layers for the MPEG clients. After dropping out a layer for each of the MPEG clients, it then drops the remaining blocks that cannot be serviced from the other clients uniformly. By doing this, we ensure that the application can decide whether or not to drop a layer by reducing the number of requests sent to the disk scheduler. This can help it ensure that unnecessary data is not obtained for MPEG streams since when retrieving only a part of the frame (might need a complete set of blocks before decoding them into *I*, *P* and *B* frames) will not substantially increase visual quality. Our design ensures that any of the media specific optimizations (mostly by the applications) are carried out before the

request is put onto the system disk queue.

Ability to Support Variable Bit-rate Applications

While the mechanism to support variable bit-rate applications has not been completely implemented in the disk scheduler, we attempt to support variable bit-rates by providing mechanisms that can be used by the clients to read data at different rates during the course of playback. Our kernel exports `sys_reserve()` (a system call that can be used to convey the number of bytes the client needs in the immediate *round*). The disk scheduler uses the average value of the playback to first service the agreed upon data for all the clients, before servicing clients with increased bit-rate requirements. While this allows the clients to ask for more data than guaranteed, the actual amount of data received will depend on the current load of the disk. Protection of service classes ensures that in the presence of non real-time clients, and relatively few multimedia clients, variable bit-rate clients can be adequately supported.

Efficiency of Computation

Theoretically speaking, there can be a number of optimizations made and the best decisions taken regarding servicing a particular client/request. While these can be implemented when the experiements are being conducted through simulations, actual implementations must take due consideration of the fact that optimizations can consume considerable amount of system resources like CPU. Therefore, they should balanced optimizations against their benefits. It is very important that restrictions imposed by memory and processor speed are considered key factors while deciding on the amount of computation to be carried out. We avoid complex re-arrangement, insertion of requests into the system queue. We have preferred to keep re-ordering of requests less mathematical while writing our disk scheduler within the framework offered by the existing *Linux* disk

scheduler. Most of our computations are restricted to maintaining statistics to make our scheduler as adaptive and intelligent as possible. We try to take a decision about whether a request needs to be serviced prior to putting it on the request queue, which saves precious bandwidth used to insert it into the system disk queue. The following are the time complexities of some of the commonly performed operations in our scheduler.

Operation	**Time Complexity**
Determination of blocks serviced for a client	O(1)
Insertion of request into a `WAIT_QUEUE`	O(1)
Removing client requests from `WAIT_QUEUE`	O(n)
Determination of average disk access time per block	O(1)
Insertion into the system disk queue	O(n)
Re-allocation of disk bandwidth	O(1)
Updation of client information (blocks read, allowed etc.)	O(1)

Table 4.2: Time Complexity for Various Commonly Performed Operations in the Disk Scheduler

In table 4.2, n refers to the number of multimedia clients that have been admitted for service.

We have carried out all the implementation using a 2.0.36 [2] kernel for an Intel x86 architecture. All the software and their versions needed to compile this kernel can be obtained along with the distribution from various web-sites.

[2]The current kernel version is 2.4.17

Chapter 5

Performance Evaluation and Results

> *"Experimental confirmation of a prediction is merely a measurement. An experiment disproving a prediction is a discovery."*
>
> Enrico Fermi, Italian physicist

We present an evaluation of our proposed admission control and disk scheduling schemes.

The measurement of performance of the different admission control schemes help us better understand how various sytems can be better utilized to service multimedia clients by allowing partial delivery of data that is requested by the clients and by exploiting commonly implemented features in an operating system, like caching. The following metrics were used to evaluate our admission controllers (optimistic and measurement-based) and compare their performance with pessimistic admission control schemes.

- Number of Clients Admitted: The total number of clients that were admitted for a multimedia session out of a large number of requesting clients.
- Percentage of Deadlines Missed: In admitting the multimedia clients, there could be a number of deadlines missed. We measure the percentage of deadlines that

were not satisfied (which reflects on the quality of service received by the admitted clients).

- Disk Bandwidth Partitioning: In the absence of the disk bandwidth partitioning being enforced by the disk scheduler, it is interesting to see how well the admission controller in conjunction with the disk scheduler can estimate the amount of time consumed by the multimedia clients admitted for service. This is especially important since an inaccurate estimation process could either deny service unnecessarily to multimedia or non-multimedia clients or result in a high percentage of deadline violations for multimedia applications or increased response time for non-multimedia ones.

The measurement of performance of the admission controller has been largely independent of the disk scheduler and dependencies have been noted wherever applicable. In order to measure the performance of our disk scheduling algorithm, we have focussed on the following issues:

- Raw Throughput Performance: This experiment is aimed at measuring how normal applications would behave with `Clarity`, in the absence of any multimedia clients.

- Protection of Service Classes: When there are a large number of clients belonging to one particular class, the efficiency of `Clarity` in dividing bandwidth among them in trying to ensure that performance of the other classes does not suffer.

- Intra-Service Class Protection: This focusses on how *QoS* is maintained for different multimedia clients in the presence of VBR (variable-bit rate) clients. VBR clients can read more than the average number of blocks in a round, which can cause other multimedia clients to miss their deadlines or suffer loss.

- Bandwith Re-allocation: This focusses on how *Clarity* re-assigns bandwidth when the bandwidth available to one class is under-utilized and is required for satisfactory performance of applications belonging to another class.

- Overloaded Rounds: There are instances when a number of VBR clients trying to use more than their share of the disk bandwidth can cause the round to be overloaded. In such cases, `Clarity` attempts to drop blocks and informs the application about such an action so that the application can then decide what it needs to do to minimize the impact of loss or delay etc.

5.1 Determination of Running Time for Clients

When designing the clients, it was very important to identify the minimum duration for which the clients must run so that we can obtain stable statistics. Typically when multiple clients are started for service there is an initial period of time when the kernel is setting up data structures for the new processes. It takes some time for the system to achieve a state of equilibrium so that measurements between one run and another, under the same set of constraints or conditions, are consistent.

Therefore, in order to determine the minimum running time for the clients, we executed clients and measured the average response time and throughput over various durations of client execution. The clients are not configured to read any specific number of blocks and were all started at the same time. All the clients, when started, just go on reading from the disk continuously and at the end of the run, record their average throughput and response times. This very closely simulates scenarios that would occur during our testing, where we will have to start a certain number of clients at the same time. The reason we cannot always start one client at a time is because of limited space. Since starting each client separately would mean that clients at the beginning of the test must read files

that are very large in order for them to last the entire duration of the test. Due to limited disk space, it was not possible for us to create a number of very large files ($>$ 70 MB).

We measured the average response time and average throughput for 10 clients. This was done in order to ascertain the minimum duration for which both these values would be stable. Figure 5.1 shows the values obtained for various running times for 10 clients. It can be seen that the values are not consistent for execution times from 5 *seconds* to 20 *seconds*. However, when the times of execution goes beyond 20 *seconds*, it is seen that there is very high degree of consistency in measured values. It may be remembered that for all our experiments we rely on the extended filesystem (ext2fs) implemented in Linux for allocation of disk blocks.

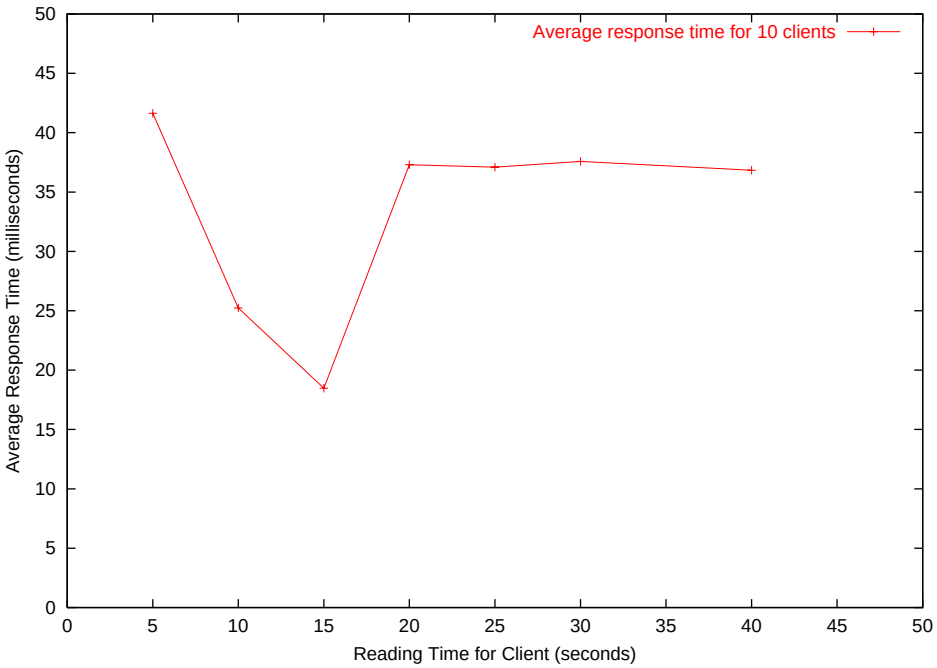

Figure 5.1: Determination of Client Execution Time: Average response times versus Client execution times for 10 clients

A similar pattern is observed when looking at the average throughput for 10 clients. It can be seen that while the average throughput is not predictable for execution times of

5 *seconds* to 20 *seconds*, it is predictable and consistent thereafter. Figure 5.2 shows the average throughput in *MBytes/sec* over different client execution times.

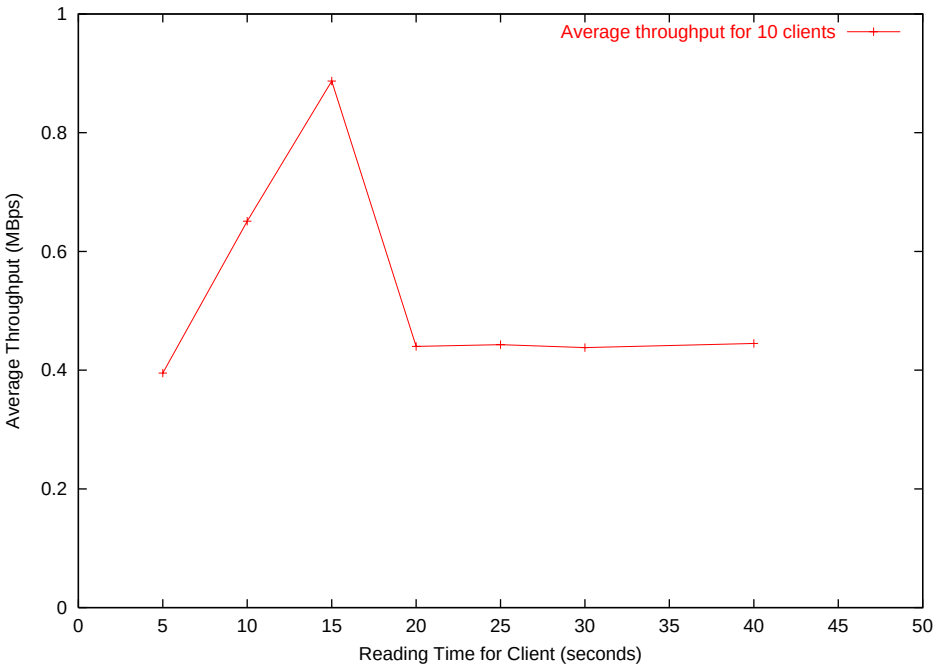

Figure 5.2: Determination of Client Execution Time: Average throughput versus Client execution times for 10 clients

From the pattern seen in the two graphs, we decided to have client execution times of at least 25 *seconds*. However, we have not restricted the running times for each client to 25 *seconds* and there are a number of experiments where we have more than a single client (i.e. a group of clients) execute for a minimum of 25 or 30 *seconds*. This was done due to the limited amount of disk space available, since running each client for 30 *seconds* for every experiment would mean the clients starting initially would require huge files.

5.2 Admission Control

In order to measure the performance of our schemes with that of pessimistic admission

control scheme, we did measurements of various parameters. Since the amount of time taken to retrieve data blocks from the disk is dependent on the size of each block of data, we have attempted to measure the performance of different admission control schemes for varying blocks sizes of 1*KB*, 2*KB*, and 4*KB*.

Number of Admitted Clients

In order to facilitate testing, we have multimedia clients attempting to start sessions using `Clarity's` framework. They are CBR (constant bit-rate) clients that attempt to read the same number of blocks from the disk every round. However, each client attempts to read a different file so that sufficient disk requests are generated and measurements from disk access are used for admission control. We also made sure that the size of a file was fairly large (> 35 *MB*) in order to prevent substantial caching of the file, which would in turn reduce the number of disk requests. The bit-rate for each multimedia client is 12*KBps*.[1] The test was designed to start a multimedia session every 30 *seconds* (for reasons explained earlier) and record the number of clients that are admitted for a multimedia session. Upon, admission a multimedia client attempts to read as many blocks as mentioned at the time of admission every round. Once the client has read the required number of blocks, it waits for the kernel to signal end of the round before reading data for another frame. This arrangement keeps all clients in sync with start and end of a round [2] and also ensures accuracy of measurements made during any particular round for various multimedia clients.

Figure 5.3 shows the comparitive performance of pessimistic, optimistic and measurement based admission controllers for a block size of 1*KB*.

Since the pessimistic admission control assumes that every access to the disk requires

[1] All multimedia clients read in *kilobytes (KB)*

[2] For definition please see footnote on page 10 in Chapter 1

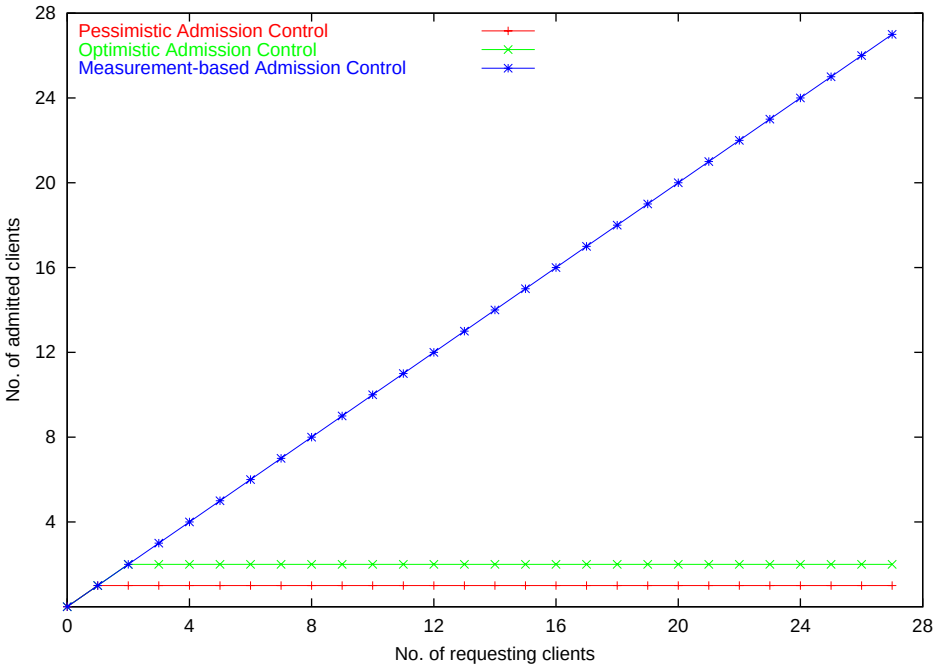

Figure 5.3: Number of clients admitted by pessimistic, optimistic and measurement-based admission controllers for a block size of 1*KB*

the worst time observed, it is able to admit only a single client. It does not take into account the fact that when hard disk is accessed entire tracks can be cached and subsequently provided with very little access time. Therefore, while it is able to provide guarantees to hard real-time clients, it results in a very low number of soft real-time clients being admitted. On the other hand, assuming that each access to the disk is going to be closer to the average value, the optimistic admission controller, clealy performes better by admitting two clients. However, even the optimistic admission controller also assumes that each request needs to be sent to the disk for service. Since it assumes that for every request sent disk access converges to an average value it performs better than pessimistic. As it evident from the graph, it the measurement-based admission control that does extremely well by admitting all the 27 [3] clients that request for service. Measurement-based admis-

[3]The number of clients used to request service was restricted due to storage limitations. This does not

sion controller measures the time taken for blocks to be read from the disk, which enables it to to take advantage of system optimizations like pre-fetching of disk blocks both by the operating system and the hard drive controller. We predict, based on measurements of bandwidth usage, that the measurement-based admission controller admitted just as many as can be accomodated for a disk block size of 1*KB*.

Figures 5.4 and 5.5 compare the performance of the admission controllers for 2*KB* and 4*KB* block sizes respectively for clients reading at 12*KB* per second.

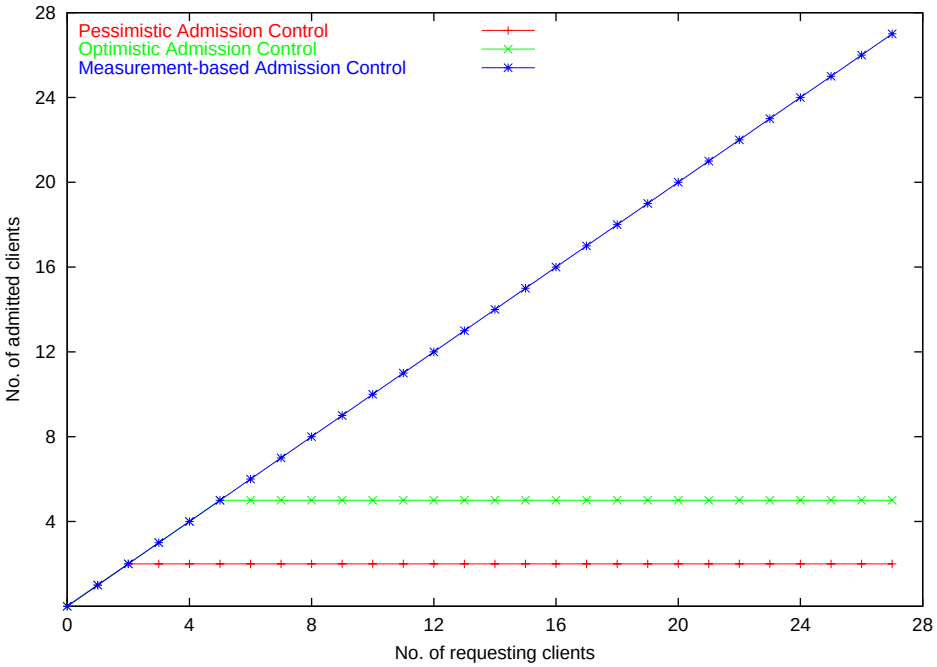

Figure 5.4: Number of clients admitted by pessimistic, optimistic and measurement-based admission controllers for a block size of 2*KB*

It may be noted from the graphs that for all admission controllers the equation of the line depicting the number of admitted clients versus the number of requesting clients would be $x = y$. This means that until such time that the admission controller rejects

necessarily represent the actual maximum number of clients that can be accomodated in the system by measurement-based admission controller.

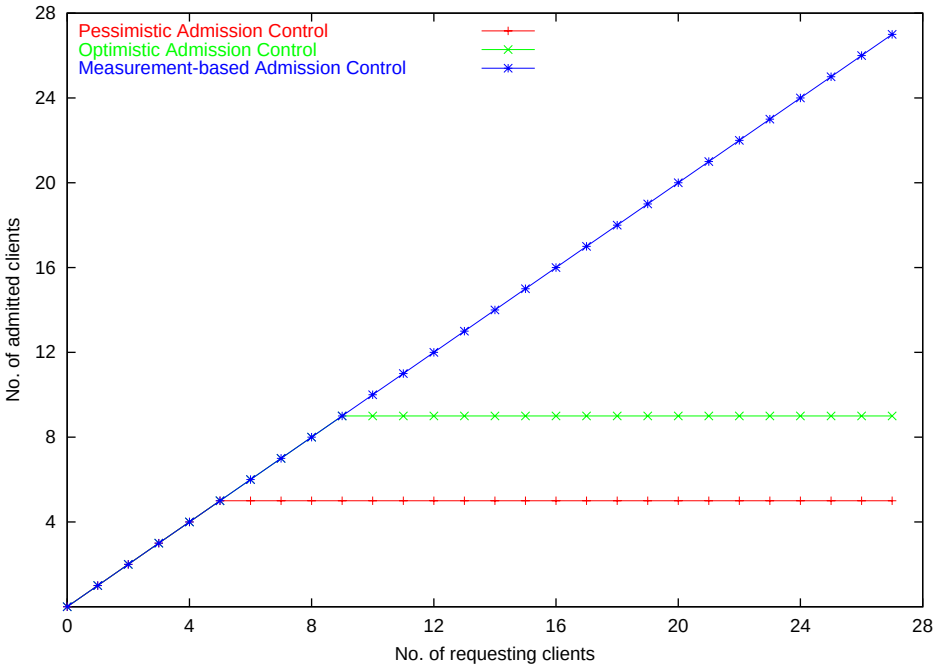

Figure 5.5: Number of clients admitted by pessimistic, optimistic and measurement-based admission controllers for a block size of 4*KB*

request for a multimedia session, the number of admitted clients is the same as the number of clients that requested for a multimedia session. Once a client is rejected, the number of admitted clients stays the same although the number of requesting clients keep increasing. This is depicted by flattening of the graph at the point where a client requesting a session is denied admission, i.e. does not become an admitted client.

Since the time for retrieving blocks from the disk decrease with increase in block sizes, the pessimistic and optimistic admission controllers admit more clients with increasing block sizes. In the 2*KB* block size case, while the pessimistic admission controller admits 2 clients, the optimistic admission control scheme does much better by permitting 5 clients into the system. This is because as the disk block size increases, using average values for disk access provides much lower predicted values for disk bandwidth consumption than using worst-case values. As observed earlier, the measurement-based admission

scheme serves by far the most number of multimedia clients by admitting all the 27 requested sessions. From the plot of disk bandwidth consumption in Figure 5.6, we predict that the measurement-based admission scheme would admit around 30 clients. In the $4KB$ case, it is observed the pessimistic admission control scheme permits 5 clients with the optimistic scheme permitting 9 sessions. This is because for a given bit-rate, increasing block size results in decrease in the number of requests to the disk. Therefore, the disk bandwidth required to read the same amount of data decreases under both pessimistic and optimistic admission schemes. The measurement-based admission controller admits all the 27 requested multimedia sessions. While we could determine the maximum number of clients that the measurement-based admission scheme would admit experimentally, we predict that it would be around 50.

It is observed that all the admission control schemes permit an increasing number of sessions with increase in block size. The difference in the number of clients admitted by optimistic and pessimistic admission control is also found to increase with block size. Although we did not measure performance with adaptive measurement-based scheme, we expect that it would admit at least as many clients as the non-adaptive measurement based admission control.

Percentage Deadlines Missed

It quite obvious from the previous graphs that the measurement-based admission control scheme performs very well by admitting a large number of clients. But what is of greater interest is if admission for a large number of clients results in a high percentage of deadlines missed. It is definitely not acceptable to have a high percentage of admission which results in a large number of those admitted clients not receiving guaranteed service, since our case for recommending a measurement-based admission controller, which utilizes average values of measured times for disk access, is that it not only permits a large number

of clients but also meets their quality requirements. In order to determine the efficiency of the measurement-based admission controller, we compare the percentage of deadlines missed by the various clients during the course of their playback. It was observed that all clients admitted using the measurement-based admission controller receive all the bandwidth they require to maintain their quality of service. This is shown by the fact the deadlines met for all the clients is 100%. This is also true of clients admitted with the pessimistic and optimistic criteria.

As a point of further interest, we also take a look at the amount of time that is consumed in each round that the clients execute in, since not missing deadlines is not an accurate indication of the amount of time taken in each round to maintain the quality of service guaranteed to each client. This is because the disk scheduler is not constrained to restrict the amount of time given to servicing multimedia clients, although it does provide the average time per request to the admission controller. Since there are no non-multimedia clients in the system, all the available time in a round can be used for servicing multimedia clients to meet their deadlines. Therefore, an accurate representation of admission control efficiency is a combination of the fact that all deadlines are met and the time estimated time by the admission controller is close to the time available (and preferably less than the time allocated for multimedia clients, which in this case is 500 *milliseconds*). Figure 5.6 shows the amount of time taken to service all the admitted multimedia clients in each round (with a disk block size of 1*KB*). It can be seen that in admitting 27 [4] clients, the admission controller, in majority of the rounds, does not force the disk scheduler to spend more than 500 *milliseconds* in servicing all the requests from these clients. There seems to be a certain fraction of rounds where the disk scheduler spends more than 500 *milliseconds* servicing client requests. In addition to measurement with a block size of 1*KB*, the total time taken for servicing multimedia clients was also taken with varying block

[4] Please refer to footnote on page 90 for explanation on the number of clients

size of 2 and 4*KB*. The results obtained were similar to the 1*KB* case, but it is seen the disk scheduler spends far less than time allocated for multimedia clients, in servicing the same number of clients. Clearly, the measurement based admission controller not only increases the number of clients serviced, but does so within the constraints of bandwidth availability for all multimedia clients.

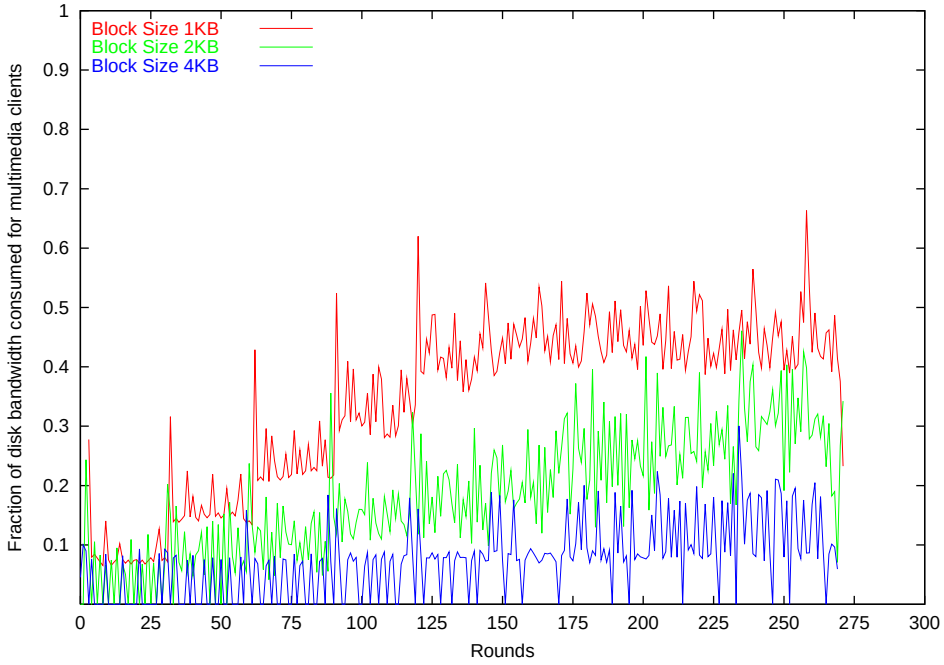

Figure 5.6: Measurement-based admission control: Disk bandwidth consumed for various disk block sizes

We next take a look at the comparitive performance of our measurement-based admission controller and its adaptive variation. In this case, we admit MPEG clients which have very high bit-rate requirements. Also, since we observed in the previous experiments that 12*KBps* were not sufficient to load the disk scheduler, using higher bit-rate clients helped us observe the limit for admitting such clients. We compare the performance based on 2 factors: the number of clients admitted and the fraction of disk bandwidth consumed. For this experiment, the clients had a bit-rate of 192*KBps* or 1.5*Mbps*. This represents

the bit-rate requirements of MPEG-like clients. The measurement-based admission control uses disk access times provided by the disk scheduler to grant or deny admission for a multimedia session. On the other hand, the adaptive admission controller attempts to find the maximum rate at which such MPEG-like clients (which are layered and so can provide good picture quality even at reduced rates) can be admitted. Figure 5.7 shows the number of clients admitted for various block sizes of 1*KB*, 2*KB*, and 4*KB*.

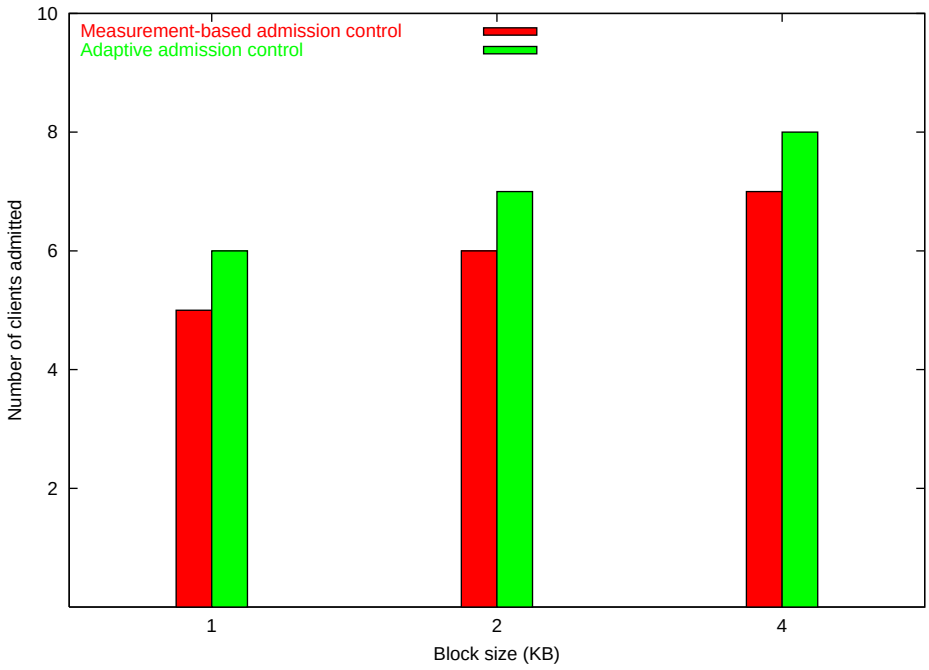

Figure 5.7: Comparison of measurement-based admission control and its adaptive variation for various block sizes: Number of MPEG-like clients admitted

As can be expected, the number of client admissions are directly proportional to the block sizes (at least for the block sizes that we have tested with). This is because time for retrieval of data blocks decreases with an increase in the block size, since an access to the disk fetches more. However, it is to be noted that there is performance gain only up to a certain block size. Beyond that the fact that the disk takes very long for even a single block of data can result in deterioration of disk performance. While both adaptive

and non-adaptive measurement-based admission controllers admit an increasing number of clients, it is observed that the adaptive admission controller always admits more clients that the normal measurement-based admission controller. In this case, the difference in the number of clients admitted is 1. Since in the absence of sufficient disk bandwidth, the adaptive scheme attempts to find the maximum bit-rate that can be supported, most of the residual bandwidth would be used for admitting the last client. However, it must be remembered that variations in disk access times, bit-rates for serving various clients can result in a much higher number of clients being admitted by the adaptive scheme.

Apart from the fact the adaptive variation of the measurement-based admission controller admitted more clients it is able to do so while consuming less that the amount of bandwidth allocated to the multimedia clients. This is especially important since it would be prohibitive to incur penalties in terms of disk times used for multimedia clients by way of admitting clients at slightly lower bit-rates. Figure 5.8 shows the amount of time consumed for block sizes of 1 *KB*, 2*KB* and 4*KB*. The figure shows the fraction of the total disk bandwidth that is consumed by multimedia clients. During the initial stages, when more and more clients are getting admitted, the graphs rises since it results in increased disk access. However, when no more clients can be admitted, the amount of disk bandwidth used seems to flatten and shows very little increase or decrease. However, it can be clearly seen that adaptive measurement-based admission control can be successfuly used to not only increase the number of clients serviced, but also increase disk utilization.

From our performance measurements, it is clear that the optimistic admission controller based on fixed average disk access times performs better than the traditional pessimistic admission controller. It not only admits more clients but also does not violate any deadlines for the clients admitted. The measurement-based admission controller performs much better than both the optimistic and pessimistic admission controllers in terms of the number of clients admitted for service. It is found that averaging the measured

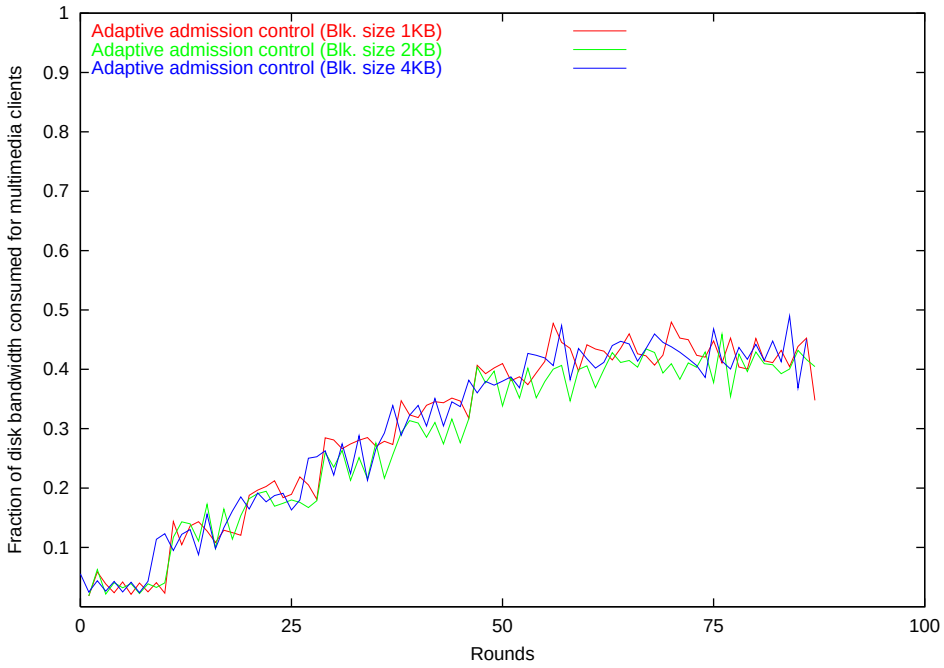

Figure 5.8: Fraction of total bandwidth consumed by MPEG-like clients under the adaptive measurement-based admission control scheme

disk access times reported by the disk scheduler, and using that to admit clients results not only in increased client admission but also led to better utilization of the bandwidth assigned to multimedia clients. This implementation is also in contrast to the idea of using worst-case observed times for admitting clients, although the worst-case times would still probably be less than the fixed worst-case times reported for retrieving a disk block of data. In order to improve performance further, we use adaptive techniques applied to the measurement-based admission controller. This attempts to exploit the inherent property of clients that access layered streams, where admission to a reduced number of layers would still not result in very poor video quality. We have carried out experiments based on percentage of client admission, deadlines missed as a result of client admission, fraction of bandwidth consumed by multimedia clients and conclude that our approach outperforms existing techniques for admission control. It must be remembered, however, that

the optimistic admission control can function independently of the disk scheduler since it does not need any information from the disk scheduler about the disk access times. On the other hand, the measurement-based admission controller needs support from the disk scheduler to find out the disk access times for a block of data. The disk scheduler does not need to provide other services like inter-service class protection and intra-service class protection.

5.3 Disk scheduling

We carried out performance measurements of `Clarity's` disk scheduling mechanism with emphasis on some of the points that were mentioned at the beginning of this chapter. While we cannot simulate all possible scenarios which this framework might be required to perform under, we have attempted to highlight some of the important scenarios that merit attention and detail.

Raw Throughput

`Clarity's` framework has been designed with an intent to add support to the Linux kernel to allow various types of clients to have their application requirements met. For the purposes of our evaluation we have considered only two major application classes, multimedia and non-multimedia clients (traditional non-real time clients). It is therefore very important that in the absence of the multimedia clients `Clarity` perform as well as the original disk scheduling framework for non-multimedia clients. For non-multimedia applications, two of the most important factors for performance are response time and throughput. Our first test of `Clarity's` ability to handle non-multimedia clients involved loading the system with only non-multimedia clients and measuring how the clients performed. Since we have an intelligent disk scheduling framework in order to

meet the service requirements of multimedia and non-multimedia clients, `Clarity` incurs additional overhead in terms of managing information for each clients in its various internal data structures, re-sequencing disk requests, running timers so that the system performance is constantly monitored and maintaining statistics on how well the application requirements are being met. Therefore, it is of interest to see how much overhead we incur as a result of the adding this framework.

Our first experiment consisted of adding non-multimedia clients, a certain number at a time in order to simulate the presence of the number of clients in the system. The clients once started just go on reading from the disk for as long as they run. Since we want to be able to load the disk, we have attempted to keep the clients very thin with a view to making them consume as little CPU time as possible. Figure 5.9 shows the results of our experiment for measuring the average throughput of the clients. The graphs shows a plot between the number of clients in the system on the X-axis versus the average throughput seen by the clients on the Y-axis.

The graph shows that we introduce three clients at an interval of 30 *seconds* until we reach a maximum of 15 clients. As the number of clients increases the average throughput of each clients reduces. The experiement was repeated under Linux and `Clarity` for various block sizes of 1*KB*, 2*KB*, and 4*KB*. It is observed that in each case the average throughput experienced by the clients under `Clarity` matches that observed under the Linux scheduling framework. This asserts the fact that our framework does not results in loss of service in the absence of multimedia clients and will continue to perform as well as the traditional system.

Apart from measuring the average throughput we also measured the average response time seen by each client in the system. The response time is the difference in the time between making a request and getting the requested block of data. This experiment also involved introduction of three clients at 30 second intervals until the system had 15 clients.

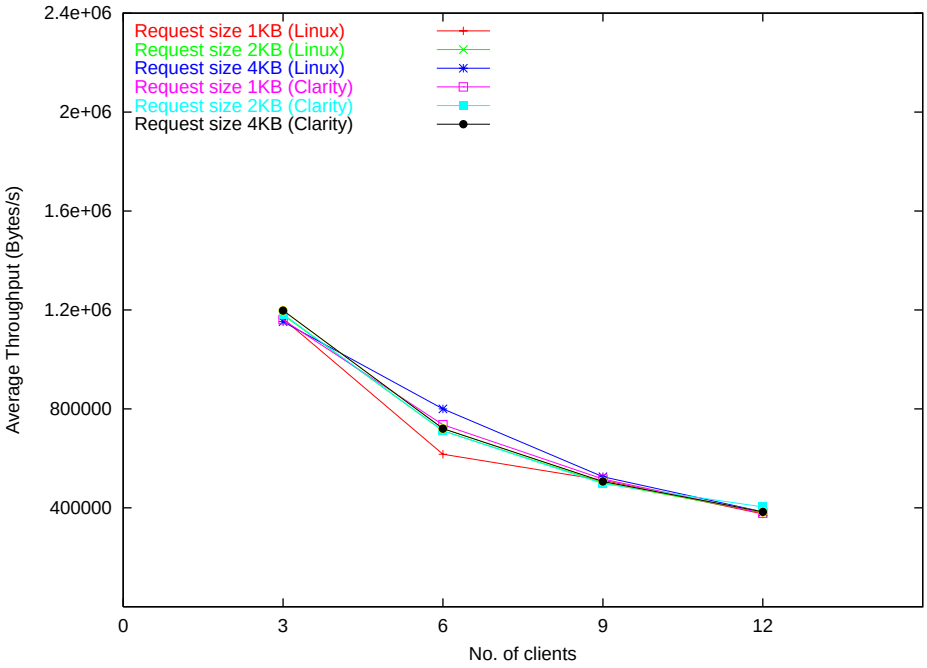

Figure 5.9: Performance comparison of Linux and `Clarity` in the presence of only non-multimedia clients: Average throughput

Our measurements, in this case, were also made for different block sizes of 1*KB*, 2*KB*, and 4*KB*. Figure 5.10 shows the results of this experiment. It is clear from the graph that the average response time seen by the clients under `Clarity` is again very comparable to that seen by them under the traditional Linux environment.

The graph shows that as the number of clients go on increasing the average response time of each client increases, as can be expected. This also tallies with the fact that the there is a reduction in the average throughput seen by each client as the number of clients in the system increases.

We note that all the measurements were made at the application level and not at the disk scheduler level. If we take measurements at the level of the disk scheduling, it would hide any CPU intensive computation that `Clarity` might be do. If `Clarity` consumed too much CPU resources to perform its functions, while we might be able to

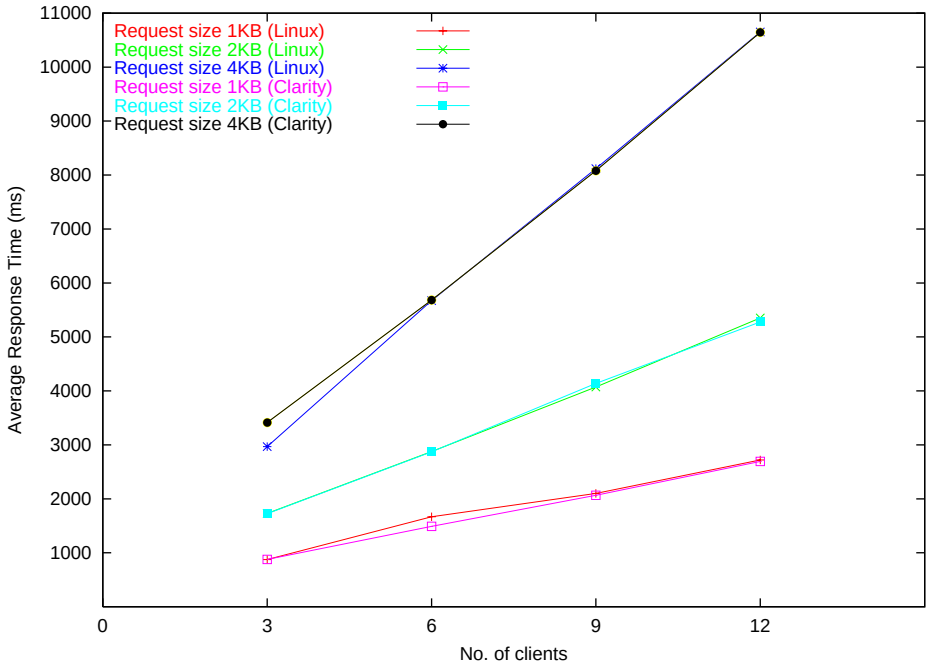

Figure 5.10: Performance comparison of Linux and `Clarity` in the presence of only non-multimedia clients: Average response time

get comparable performance for data retrieval at the scheduler level, `Clarity` could very easily fail to give the data back to the applications as quickly as Linux does it. It is only because `Clarity` is computationally efficient that it is able to match *Linux's* performance even at the application level.

Inter Service Class Protection

While `Clarity` performs very well in the absence of multimedia applications, our primary goal is to be able to support both multimedia and non-multimedia applications at the same time. There are a number of scenarios that are of interest in terms of performance monitoring. One such scenario is the presence of an excess number of clients of one class. Under traditional Linux there is no differentiation of service and all the clients compete equally for disk bandwidth. However, it is of interest to see how well `Clarity` offers

inter-service class protection. In other words, does `Clarity` ensure that multimedia clients meet all their deadlines when there are excess non-multimedia clients and maintain acceptable response time and throughput for non-multimedia clients in the presence of too many multimedia clients.

Our experiment involved starting with two multimedia clients at the beginning (*t=0 seconds*). These clients can be admitted using anyone of the previously discussed admission control schemes. For the purposes of our evaluation, we used the adaptive measurement-based admission controller. Each of the clients attempted to start a sessions requesting 192*KB* every second. Once the multimedia clients had run for 30*seconds*, non-multimedia clients were introduced. As in the previous experiments, these clients just read a file from start to finish. They had no upper bound on the number of data blocks they should read in each round. We introduced 6 non-multimedia clients at an interval of 30 seconds. The disk bandwidth was configured to be divided equally among real-time and non-real time applications, i.e. ρ, the fraction of bandwidth available to service applications with guarantees, is 0.5. Due to disk space limitations, we introduced only twelve non-multimedia clients in all. But we believe that this generated a very good load in which to test `Clarity's` ability to efficiently partition bandwidth. We would like to point out that the following anaylsis holds good for both loss-tolerant and delay-tolerant multimedia applications. We will, however, make an attempt to identify situations that are applicable to only one or the other wherever necessary.

Figure 5.11 shows the performance of the disk scheduler under Linux. It is observed that initially the plot of the disk bandwidth consumed by multimedia clients is steady. In the absence of any other clients in the system, the disk scheduler is able to meet all the deadlines of the multimedia clients. Figure 5.12 shows a plot of the percentage deadlines missed in the presence of different number of non-multimedia clients. Please note that measurements were made only at points explicitly marked (6 and 12 clients). Also, this

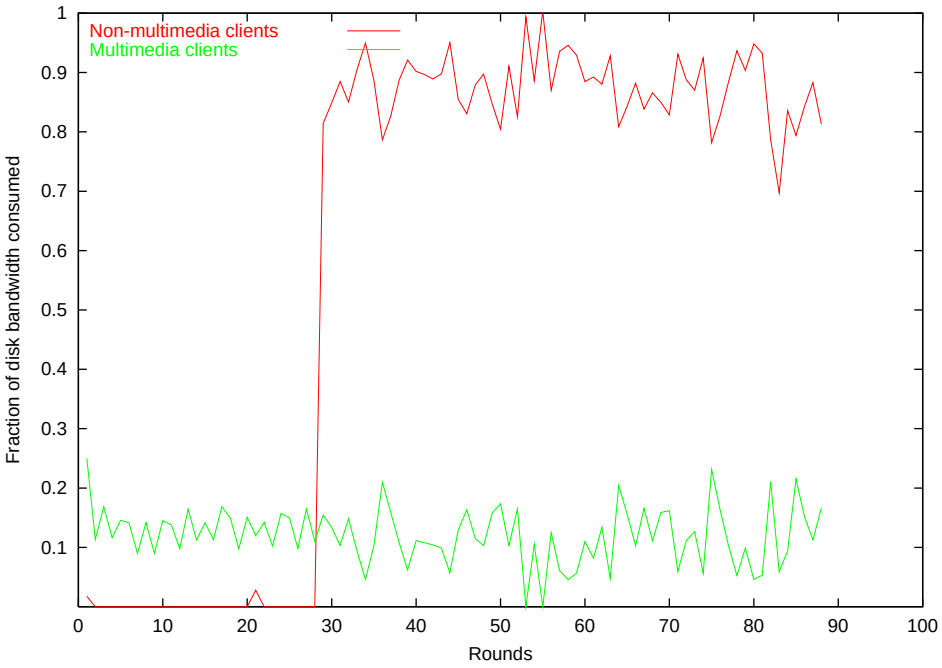

Figure 5.11: Bandwidth partitioning under Linux in the presence of excess non-multimedia clients

measurement holds for both delay-tolerant and loss-tolerant applications. While deadlines missed seems to apply to more intuitively to loss-tolerant clients, with delay-tolerant clients, we can still consider delayed frames as having missed their deadlines for being displayed.

However, in order to keep representation of results simple and its understanding intuitive, we have used straight lines to join only those points at which measurements were made. While we expect the percentage of deadline violations to be in the vicinity of the line, we claim no accuracy of data at intermediate points. In figure 5.11, the introduction of 6 non-multimedia clients is reflected by a spike in the amount of disk bandwidth consumed by non-multimedia clients around the 30^{th} round. We observe the assignment of disk bandwidth to the multimedia clients becomes irregular and starts oscillating. It also leads to a number of deadlines violations because of the fact that the newly introduced

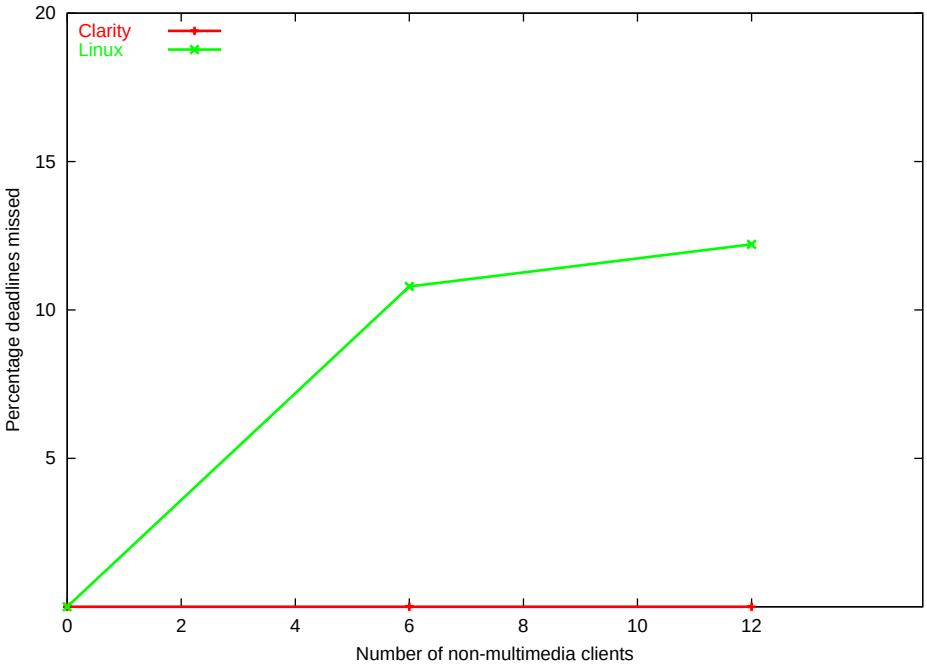

Figure 5.12: Percentage of deadlines missed under Linux and `Clarity` in the presence of excess non-multimedia clients

non-multimedia clients consume all the disk bandwidth. It can be observed from figure 5.12, that the percentage of deadlines missed is a little more than 10% in the presence of 6 non-real time clients. When the number of non-real time clients is increased to 12, it observed that percentage deadlines missed further increases. Figure 5.12 shows that more than 12% of the deadlines are missed.

We repeated the same experiment under the `Clarity` framework. Figure 5.13 shows how disk bandwidth allocations was carried out throughout the duration of the test. We observed that the behaviour of multimedia clients before introduction of non-multimedia clients is similar to that under Linux. This is because the multimedia clients are consuming much less than their share of the bandwidth and there are no non-multimedia clients to compete for bandwidth. All their deadlines are met and the allocation of disk bandwidth is steady. When 6 non-multimedia clients are introduced, there is a spike in the

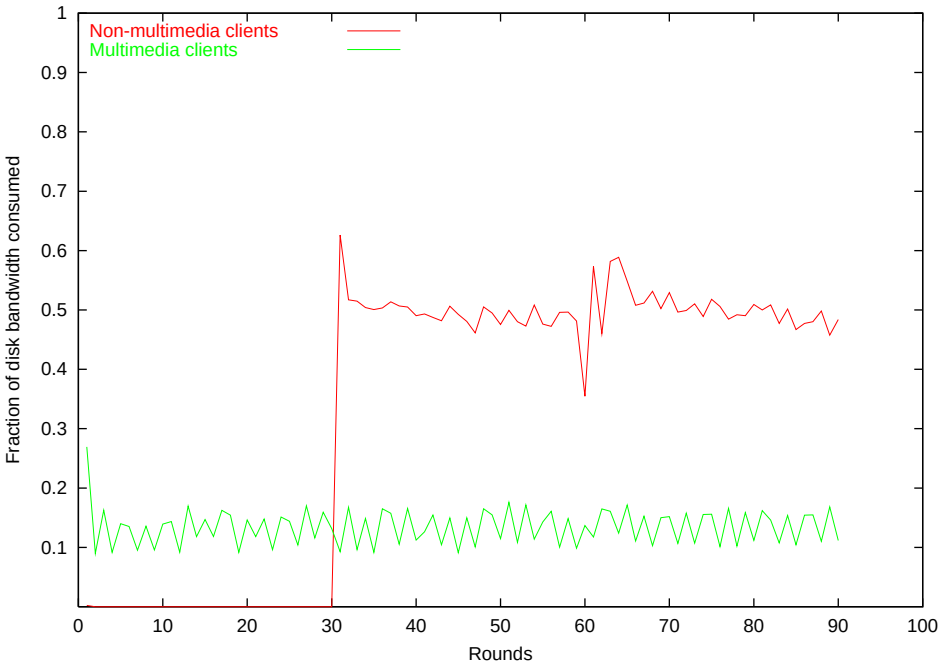

Figure 5.13: Bandwidth partitioning under `Clarity` in the presence of excess non-multimedia clients

amount of disk bandwidth used for non-multimedia clients. However, the non-multimedia clients are not allowed to over-run the multimedia clients by consuming all the disk bandwidth. `Clarity` restricts the disk bandwidth consumed by the non-multimedia clients to roughly their allocated portion of 500 *milliseconds* ($\rho * 1000.0$). The reason for slight deviations above 500 *milliseconds* can be attributed to the cumulative effect of small inaccuracies associated with measurement of individual DMA times. It is therefore observed that the disk bandwidth allocation to the multimedia clients continues to be steady even after there are sufficient non-multimedia attempting to access the disk continuously. A plot of the percentage deadline violations in figure 5.12, shows that there are no deadline violations in the presence of 6 non-multimedia clients. When the number of non-multimedia clients is further increased to 12, the pattern of bandwidth allocation to the multimedia clients does not seem to suffer any distortion. `Clarity` continue to protect the multime-

dia clients with service guarantees from the effects of too many non-multimedia clients. Additionally, figure 5.12 shows that `Clarity` is able to keep the percentage of deadline violations to 0%.

In addition to observing the deadlines missed for the multimedia clients, we also observed the jitter experienced by the multimedia clients under Linux and `Clarity`. This is especially relevant for delay-tolerant multimedia applications, which do not want loss but would like to receive all the frames over possibly an extended period of time. With the loss-tolerant applications, it would not matter much since frames could be lost during the course of service without degrading the video quality too much. We measure jitter as variation in the service times for the delivery of consecutive frames. In order to playback the video without significant jitter, we would expect that the frame be delivered to the application with a maximum tolerance of 1 second. This would however, in reality depend on the amount of buffering that the client has. Figure 5.14 shows the plot of jitter observed for the two delay-tolerant multimedia clients throughout the duration of the test.

During the time that there are only multimedia clients in the system, the jitter observed by the multimedia clients under Linux and `Clarity` are similar. However, around the 30^{th} round when the non-multimedia clients are introduced, the performance of the multimedia clients rapidly degrades. This can be seen by the large amount of jitter seen by the multimedia clients. On the other hand, the introduction of more and more non-multimedia clients seems to increase the jitter seen by the multimedia clients only very marginally when they are running under `Clarity`. It can be seen that while there is an increase in jitter under `Clarity` it is kept well within a second (which is the length of a round).

Quite clearly, Linux fails to protect the multimedia applications from the effects of a large number of non-multimedia applications competing for disk bandwidth. This is primarily because Linux has no framework to indicated preferential service and no notion

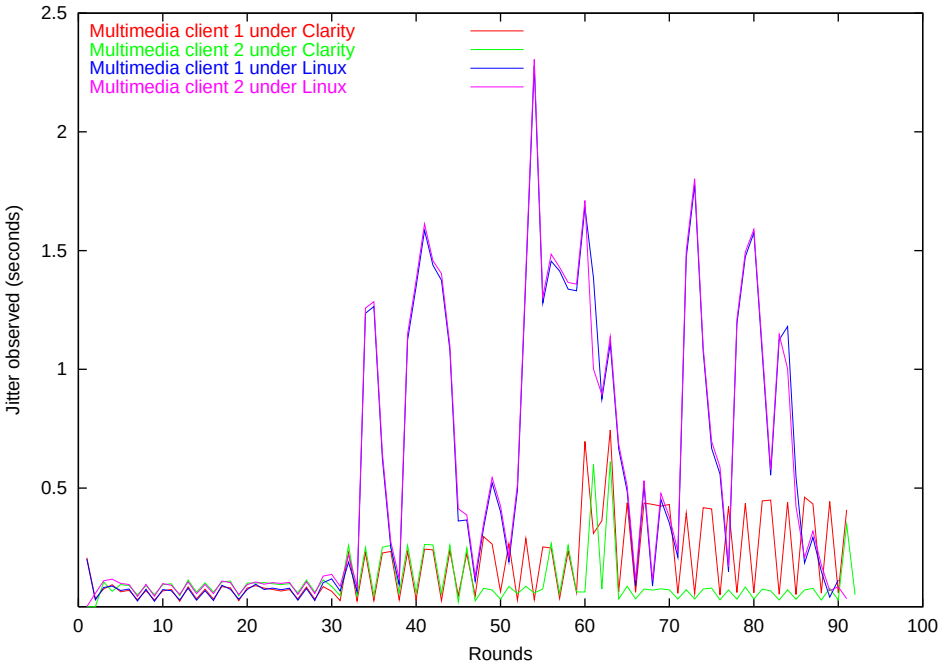

Figure 5.14: Jitter comparison for multimedia clients under Linux and `Clarity` in the presence of excess non-multimedia clients

of differentiated services. However, by incoporating that notion into `Clarity` we are able to provide guaranteed bit-rates to multimedia clients in the presence of a relatively large number of clients belonging to another class.

While it is important that multimedia clients are protected against the presence of excessive non-multimedia clients, it is equally important that we protect non-multimedia clients in the presence of excessive multimedia clients. Therefore, we conducted an experiment with the previous scenario reversed. In this case, the number of multimedia clients far exceeded the number of non-multimedia clients. Under such circumstances, one would expect the throughput and average response time of the non-multimedia clients to degrade. This is since more and more disk requests would be serviced for multimedia clients.

For this test, we had two non-multimedia clients start at *t=0 seconds*. These two

clients continuously read from the disk and have no upper limit on how much read in a round. This is so that they attempt to consume all the bandwidth allocated to the non-multimedia service class every round. Along with these two non-multimedia clients, we also admitted three multimedia clients. As the test progressed, we introduced three multimedia clients at a regular interval of 25 seconds with each multimedia client attempting to read 192*KB* every round. Here again, the disk scheduler was configured to partition disk bandwidth equally among multimedia and non-multimedia clients and we used measurement-based admission control to admit clients. However, we noted that measurement-based admission control scheme would not allow more than six multimedia clients, reading at 192 *KBps*, to exist simultaneously at any given point in time (Figure 5.7). Therefore, it was concluded that to force consumption of more than 500 *milliseconds* by multimedia clients each round (to simulate excessive multimedia clients), we would need more than 6 clients reading at 192 *KBps*. Therefore, we decided to design multimedia clients that would request for data rate lower than 192 *KBps* in order to be admitted but after admission attempt to read at 192 *KBps*. The test was configured to admit nine multimedia clients in all.

It was observed that in the presence of excessive non-multimedia clients, Linux does not offer protection to the non-multimedia clients and their performance degrades with an increasing number of multimedia clients in the system. Figure 5.15 summarizes the bandwidth consumed by each service class under Linux.

When only three multimedia clients are admitted along with two non-multimedia clients, there is no impact on the non-multimedia clients. This is because the multimedia clients do not need more than their allocated share of the disk bandwidth every round. However, when six clients are admitted the non-multimedia clients start to see degradation in their performance. Linux does not offer any protection from multimedia clients since there is no bandwidth partitioning. Each class consumes disk bandwidth almost

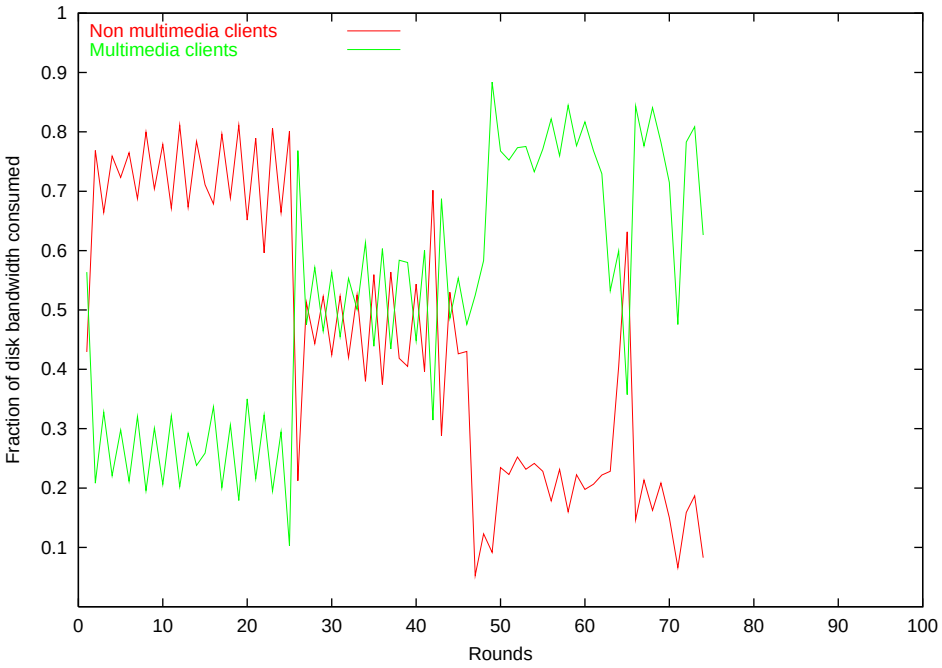

Figure 5.15: Bandwidth partitioning under Linux in the presence of excess multimedia clients

in proportion to its requirement. However, when nine multimedia clients are admitted, they consume most of the disk bandwidth every round. The performance of the non-multimedia is seen to degrade further. In direct contrast, `Clarity` provides inter service class protection by making sure that the multimedia clients do not consume bandwidth that was allocated to the non-multimedia clients. It can be inferred from Figure 5.16 that bandwidth allocation under `Clarity` is very fair[5].

When only three multimedia clients are admitted, `Clarity` protects the multimedia clients from the non-multimedia clients since the load due to multimedia clients is not enough to consume close to 500 *milliseconds*. However, when six multimedia clients are present, it ensures that disk allocation to non-multimedia clients is maintained. However,

[5]For our purposes, fairness of service is determined by how close disk bandwidth allocation or consumption is to the percentage of bandwidth allowed for that class.

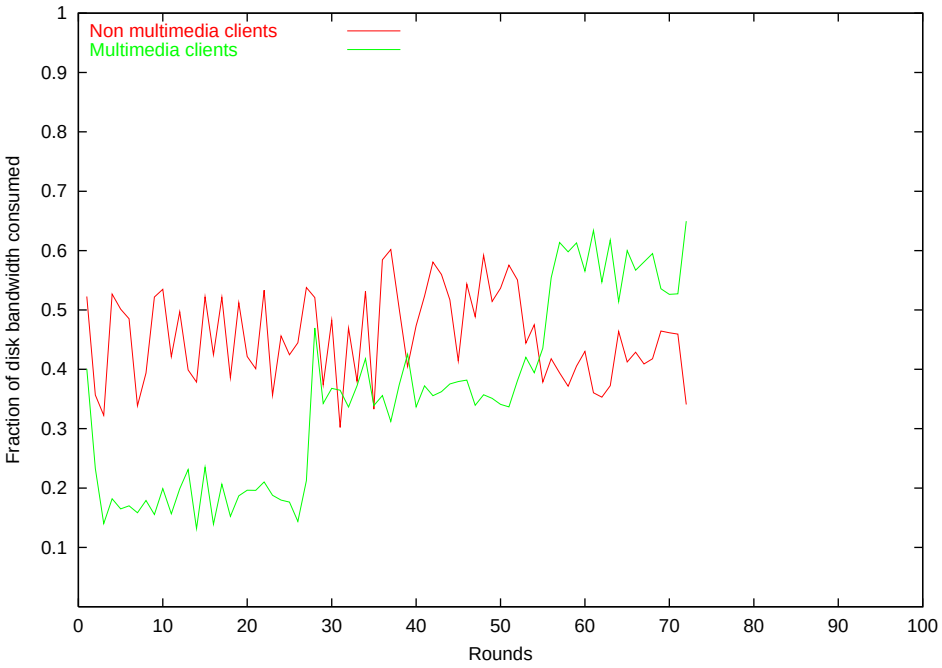

Figure 5.16: Bandwidth partitioning under `Clarity` in the presence of excess multimedia clients

when nine clients are present in the system, it is observed that there is a further degradation in the throughput for non-multimedia clients. Under `Clarity` it is observed that the multimedia clients are restricted from stealing bandwidth from the non-multimedia applications. However, a large number of multimedia clients seem to result in slightly lesser CPU cycles being available for the non-multimedia clients. This can be noticed from the fact that sum of fraction of the disk bandwidth consumed by multimedia and non-multimedia clients is less than 1. During the test-run it was observed that there are some essential non-multimedia requests generated by the system, that contribute in some part to the non-multimedia clients getting a little less than their allocated share of the bandwidth. However, the performance (average throughput and average response time) of non-multimedia clients under `Clarity` is much better than that under Linux.

Figure 5.17 characterizes the average throughput observed by the non-multimedia

clients with an increasing number of multimedia clients under Linux and `Clarity`. When three multimedia clients are admitted, there is no impact on the performance of the non-multimedia clients, since the multimedia clients do not load the system. In fact, the throughput under Linux is seen to be better than under `Clarity`. This is since non-multimedia clients consume all the bandwidth they require while the multimedia clients consume much less than their allocated bandwidth under Linux. `Clarity` does not allow the non-multimedia clients to consume more than their share of the disk bandwidth in order to provide inter- service class protection to the multimedia clients, resulting in lower throughput. However, as soon as there are six multimedia clients in the system, they tend to consume more than 500 *milliseconds* every round. This results in a drastic reduction of throughput for non-multimedia clients under Linux.

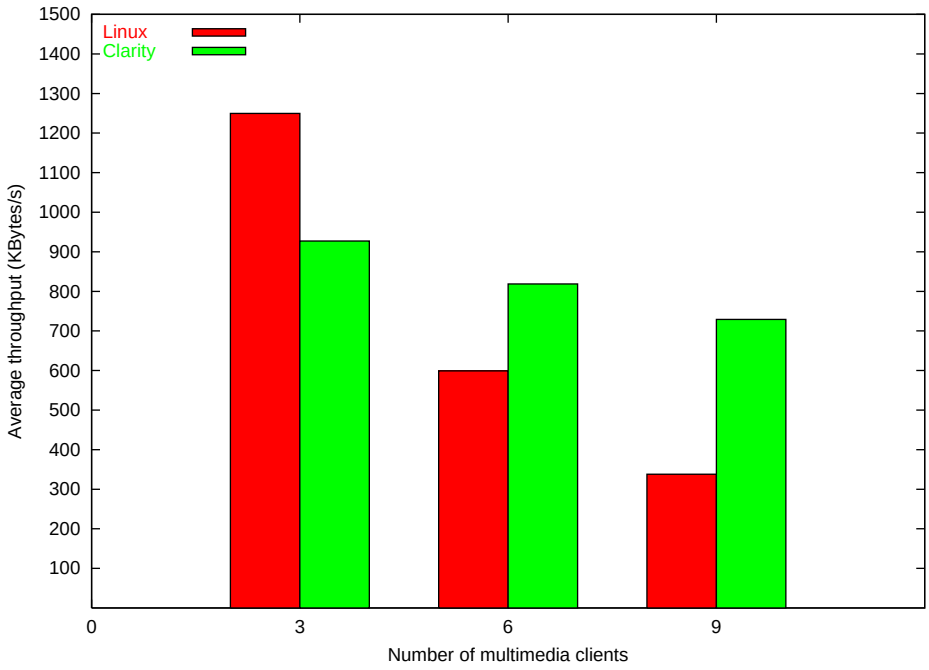

Figure 5.17: Average throughput for non-multimedia clients under Linux and `Clarity` in the presence of excess multimedia clients

In contrast, `Clarity` protects the non-multimedia clients from losing their share of

disk bandwidth to the multimedia clients. This results in very little decrease in throughput for non-multimedia clients. However, nine are admitted in the system it is observed that the throughput for the non-multimedia clients drops significantly under `Clarity` also. As explained earlier, some essential non-multimedia requests were generated and the bandwidth for those are not subtracted from the bandwidth allocated to the multimedia clients. Also, in the presence of applications other than just the test clients, an increased number of multimedia clients consuming more CPU cycles result in a slight decrease in the number of requests that the non-multimedia clients can make even though it is apparent that their disk bandwidth is protected. This is because the multimedia clients do not consume much more than their allocated bandwidth.

A very similar pattern is observed when plotting the average response times for the non-multimedia with an increasing number of multimedia clients. Figure 5.18 shows the plot of the average response time in the presence of three, six and nine multimedia clients.

While the average response time under Linux is much better than under `Clarity`, in the presence of just three multimedia clients, it is seen to quickly degrade in the presence of six or more clients. However, `Clarity` not only protects the multimedia clients when non-multimedia clients attempt to use the disk for more than 500 *milliseconds*, but also protects non-multimedia clients when the load due to the multimedia clients is substantially higher. There is a degradation seen in the average response time of the non-multimedia when there are six and nine multimedia clients under both `Clarity` and Linux. However, the average response time in the presence of six and nine multimedia clients under Linux is nearly 1.5 *milliseconds* and 6 *milliseconds* respectively more than under `Clarity`.

It has been very clearly established from the preceding graphs that `Clarity` provides inter service class protection to both multimedia and non-multimedia classes so that performance of either is not affected in the presence of a large number of clients from

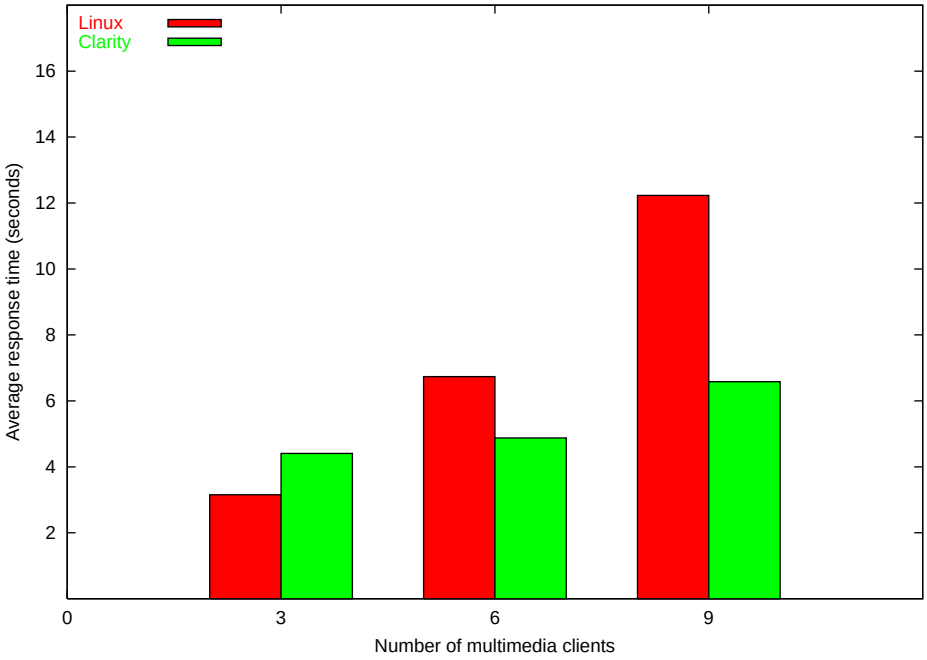

Figure 5.18: Average response time for non-multimedia clients under Linux and `Clarity` in the presence of excess multimedia clients

other classes. Under Linux, it is seen that multimedia clients lose a large percentage of deadlines in the presence of a large number of non-multimedia clients. Also, the average throughput and response time of non-multimedia clients in the presence of an increasing number of multimedia clients is degraded.

Intra service class protection

In order to illustrate that `Clarity` provides intra service class protection, we use the preceding experiment where we attempt to measure the performance of non-multimedia clients in the presence of an excess number of multimedia clients. It may be recalled that intra service class protection attempts to meet the minimum requirements of the multimedia clients in overloaded rounds. In this experiment, we admitted multimedia clients at rates of 60 *KBps*, which is lower that the rate at which they read from the disk in

every round (192 *KBps*). With 500 *milliseconds* being inadequate to service all the requests from the multimedia clients, `Clarity` will attempt to provide at least 60 *KBps* to each client before processing requests for more bandwidth. During the preceding experiment, we measured the number of blocks obtained by each multimedia client to see if the minimum bit-rate for all the clients had been met. Figure 5.19 summarizes the results obtained.

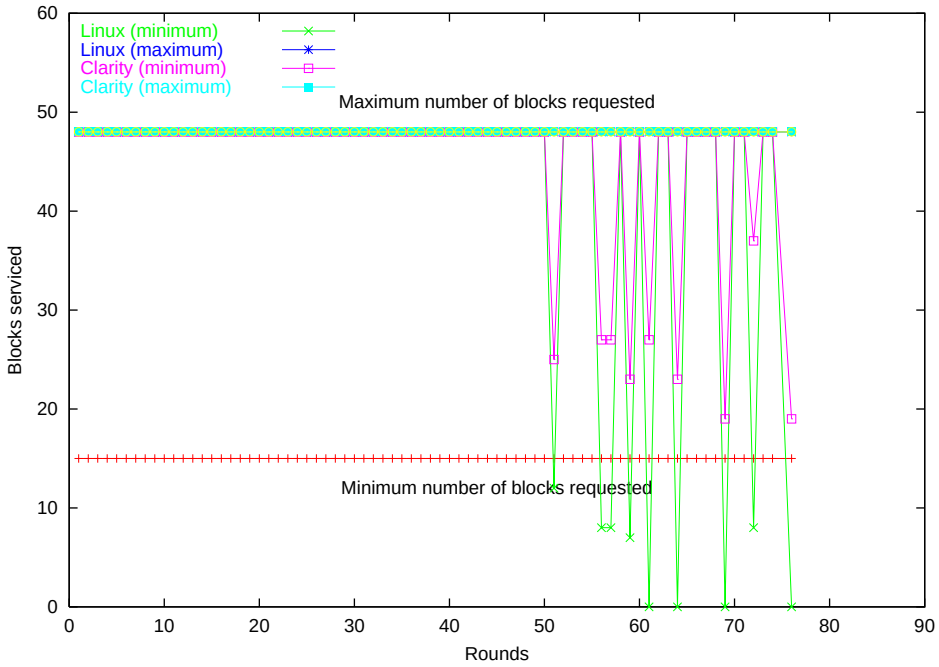

Figure 5.19: Blocks serviced under Linux and `Clarity` in the presence of excess multimedia clients

It is seen that while Linux meets all the deadlines for the clients when there are just one to six multimedia clients in the system, it results in a slight performance degradation for some of the multimedia clients when the number of clients admitted increases to nine. This is because it does not attempt to provide any minimum guarantees and serves clients as and when requests are generated. While it does satisfy most clients completely, it neglects some other clients so that they do not receive even their minimum. On the

other hand, `Clarity` provides not only inter service class protection but also protection to multimedia clients from each other by providing minimum guarantees to each client. This is seen from the fact that for all the admitted clients, the minimum number of blocks delivered by `Clarity` to any client is greater than or equal to the minimum number of blocks requested by it at the time of admission. When all the clients cannot be provided their requested bit-rate, `Clarity` makes sure that all the clients first get their negotiated bit-rate before proceeding to service any extra requests from them. It was interesting to note that the number of clients completely satisfied under Linux is more than those under `Clarity`. However, the fact that some multimedia clients might not have their guarantees met in the presence of an large number of multimedia clients indicates performance degradation for multimedia clients under Linux.

Re-allocation of bandwidth

In this experiment, we attempt to show how intelligently `Clarity` re-allocates bandwidth from a class with reduced load to another class that needs the bandwidth. The test consisted of admitting two multimedia clients at *t=0 seconds*. They are admitted to read at a rate of 192*KBps*. We use such a high bit-rate client so that we simulate a condition where it is not very easily apparent to `Clarity` there is a substantially reduced load. At the same time, we limited the number of clients to two so that some re-allocation can take place, the intent being to check if `Clarity` is able to re-allocate bandwidth when the multimedia clients do not consume all their bandwidth. We introduce six non-multimedia clients all of which read continuously from the disk. There is no limit on the number of blocks to read every round. This is done to simulate a condition where the non-multimedia clients need more than their allocated bandwidth of 500 *milliseconds*. After six non-multimedia clients have run for 25 seconds, we introduced an additional six non-multimedia clients. This is done not only to increase the bandwidth required by the

non-multimedia clients but also to observe how much bandwidth gets re-allocated to the non-multimedia clients when their load on the disk increases.

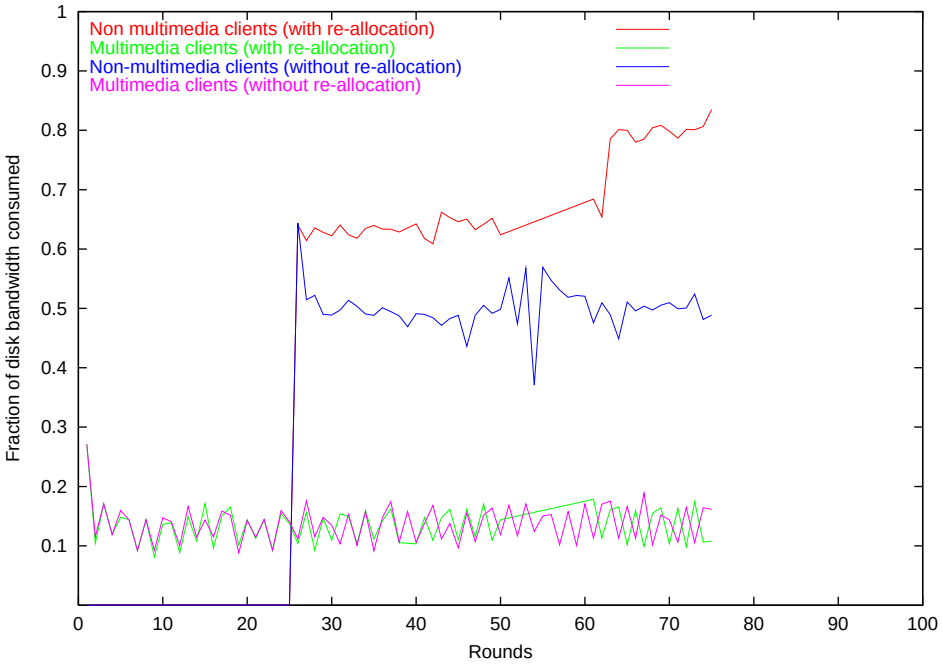

Figure 5.20: Disk utilization with and without bandwidth re-allocation policies under `Clarity` when multimedia service class is under-utilized

Figure 5.20 shows the disk utilization for the two classes under `Clarity` in the absence and presence of bandwidth re-allocation strategies. It seen that throughout the duration of the test there is no perceptible change in the fraction of bandwidth utilized by the multimedia clients. This is because `Clarity` always attempts to provide the minimum bandwidth required by multimedia clients in order to meet their bit-rate requirements. When non-multimedia clients are introduced after 25 seconds, with re-allocation of bandwidth disabled, it is observed that the fraction of bandwidth usage for non-multimedia clients is restricted to 500 *milliseconds* since `Clarity` provides inter service class protection. This is independent of the load generated by the non-multimedia and multimedia clients. Therefore, the maximum fraction of the bandwidth that the non-multimedia

clients can consume in the absence of bandwidth re-allocation is around 500 *milliseconds*, as is seen from Figure 5.20.

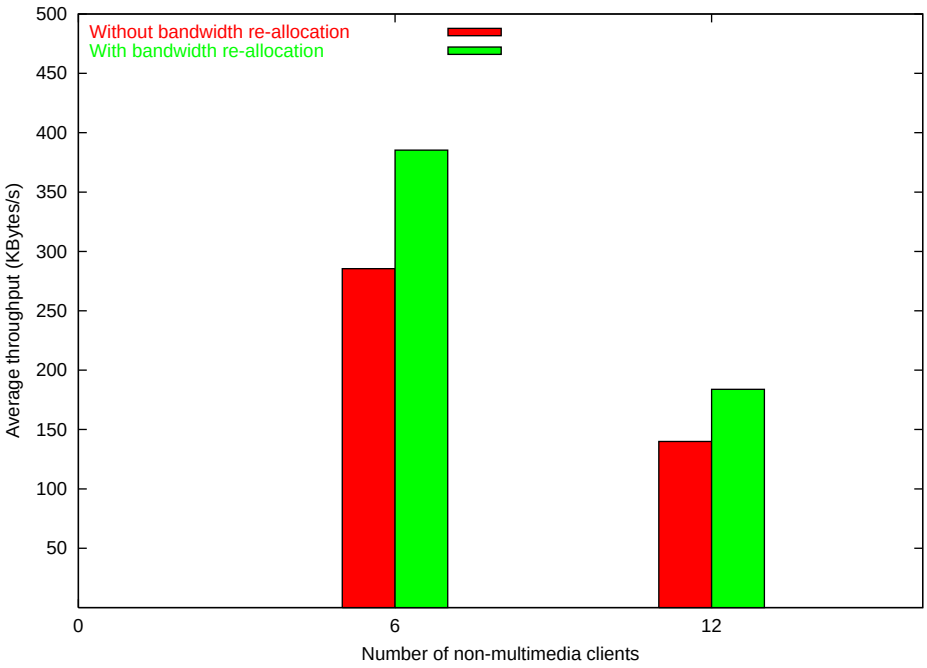

Figure 5.21: Average throughput with and without bandwidth re-allocation policies under `Clarity` when multimedia service class is under-utilized

However, when bandwidth re-allocation is permitted it is noticed that the fraction of bandwidth used by non-multimedia clients increases. `Clarity` re-allocates the bandwidth that is not used by multimedia clients to the non-multimedia clients when its measurements indicate that they need more than their allocated share. This results in higher throughput and lower response times for the non-multimedia clients. This can be inferred from Figures 5.21 and 5.22. However, it is also observed that the fraction of disk bandwidth used by multimedia clients does not change and all their deadlines are met. This is because `Clarity` keeps measuring the amount of bandwidth required by the multimedia clients and re-allocates bandwidth only when all the deadlines for the multimedia clients are met. The amount of bandwidth consumed by non-multimedia clients increases when

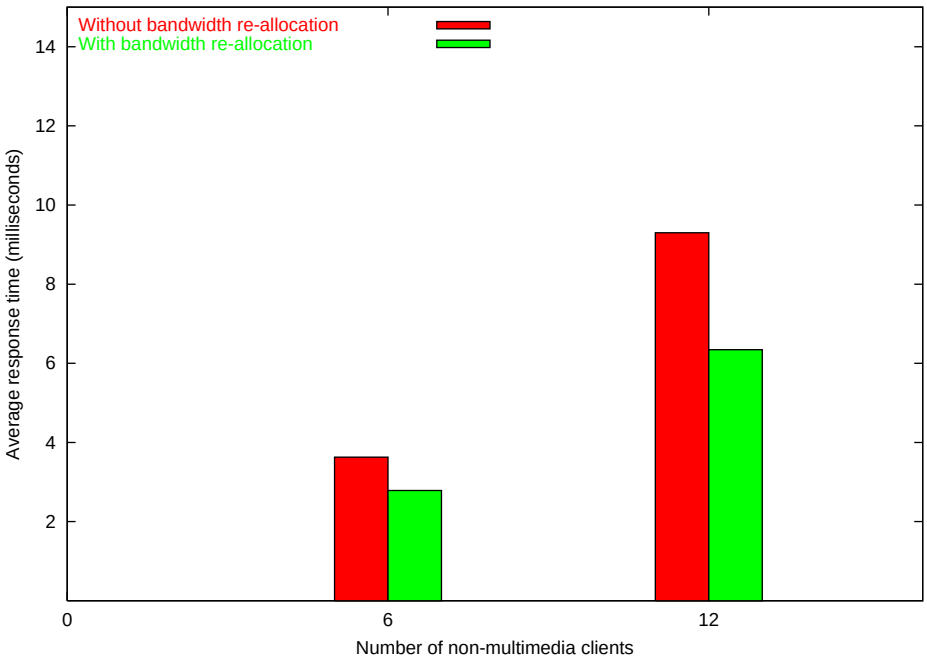

Figure 5.22: Average response time with and without bandwidth re-allocation policies under `Clarity` when multimedia service class is under-utilized

there are 12 non-multimedia clients, since `Clarity` re-allocates more to them when their load increases. However, all the deadlines for the multimedia clients were met.

Apart from the fact, that there are no deadline violations resulting from disk bandwidth re-allocation, we can also see from Figure 5.23, that the jitter resulting from re-allocation is comparable to that when there is no re-allocation. This is because `Clarity` ensures that the non-multimedia clients do not consume disk bandwidth required by the multimedia clients who are yet to receive their guaranteed service. The measurement of jitter was made by each application running as a multimedia client and not the kernel. Therefore, it shows that `Clarity` does not consume too many CPU cycles in order to carry out re-allocation of disk bandwidth.

While we have seen how efficiently `Clarity` re-allocates bandwidth from multimedia to non-multimedia clients, it is also very interesting to consider measuring perfor-

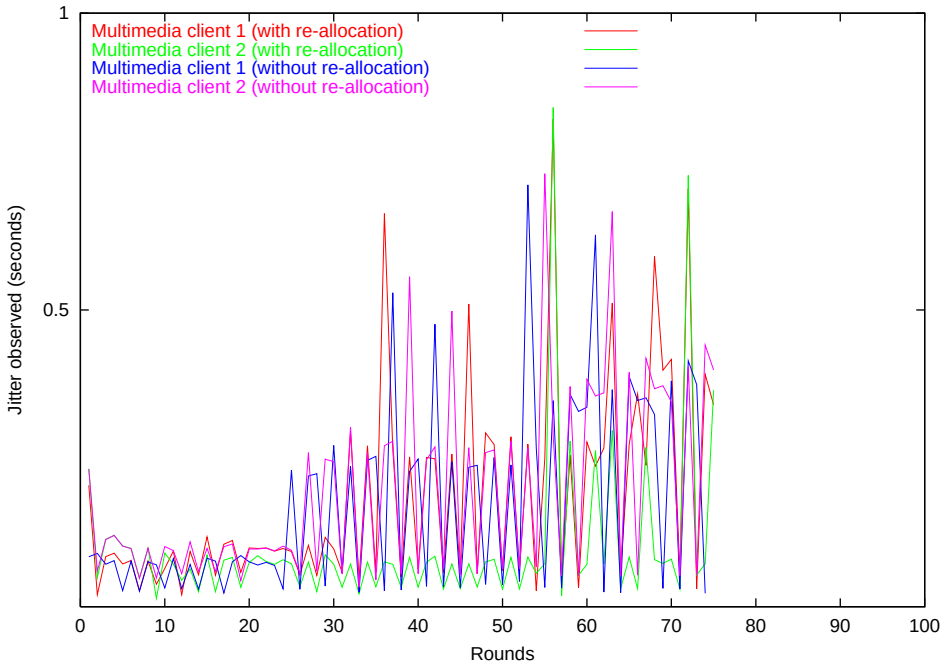

Figure 5.23: Jitter observed by multimedia clients with and without bandwidth re-allocation policies under `Clarity` when multimedia service class is under-utilized

mance in the reverse scenario. When `Clarity` attempts to re-allocate bandwidth from multimedia clients to non-multimedia clients, it can make an accurate prediction of the amount of disk bandwidth required to satisfy the multimedia clients. This is because all the multimedia clients need to reserve bandwidth at the beginning of each round by telling the disk scheduler how much they intend to read. However, when there are a large number of multimedia clients, it is very difficult for `Clarity` to decide what the required bandwidth for non-multimedia clients is. This information is absolutely essential for re-allocation of bandwidth to take place. We performed an experiment where we introduced a large number of multimedia clients and had non-multimedia clients read very little every round. We measured how well `Clarity` re-allocates bandwidth to the multimedia clients.

We started two non-multimedia clients at (*t=0 seconds*). These clients did not read

continuously. Instead, they read 100 *KB* each every second. This simulates a lightly loaded non-multimedia application class. We allowed them to run for 25 seconds before introducing multimedia clients. The multimedia clients are admitted without any admission control. This was essential in order to overload the disk with requests from multimedia clients. They read at a rate of 192 *KBps*. The bandwidth allocation to multimedia and non-multimedia clients was 500 *milliseconds* each. We introduced 3 multimedia clients at intervals of 25 seconds, till there were nine clients in all. Figure 5.24 shows the disk bandwidth utilization by multimedia and non-multimedia clients throughout the duration of the test.

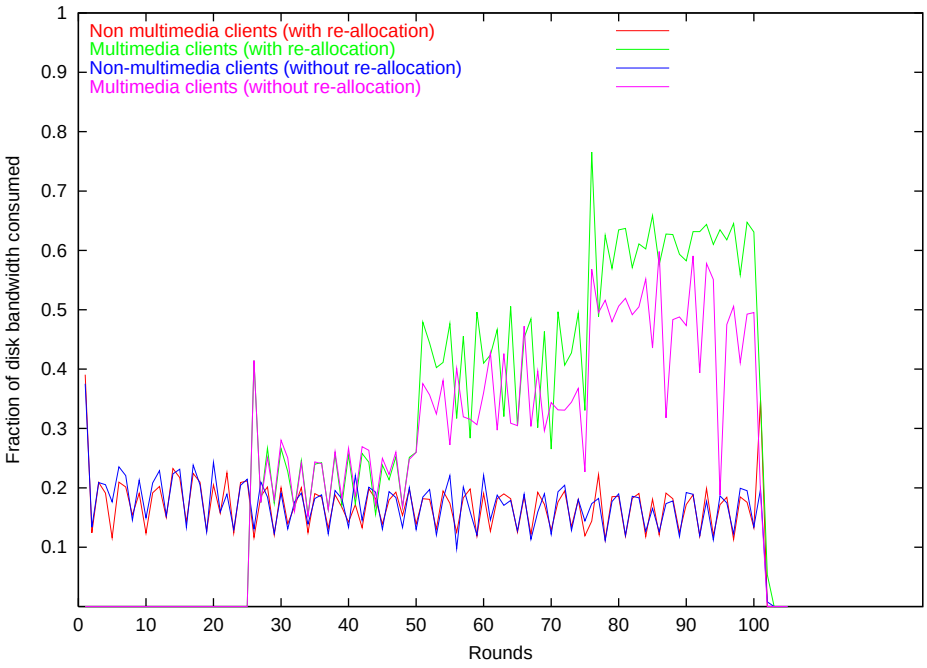

Figure 5.24: Disk utilization with and without bandwidth re-allocation policies under `Clarity` when non-multimedia service class is under-utilized

The fraction of disk bandwidth consumed by non-multimedia clients remains the same throughout the test since they are consuming much less than their allocated bandwidth. During the first 25 seconds of the test, non-multimedia clients consume the same frac-

tion of the bandwidth in the presence and absence of re-allocation policies in Clarity. When three multimedia clients are introduced, they do not completely consume the bandwidth allocated to the multimedia clients. Therefore, bandwidth re-allocation does not result in any improvement for multimedia clients. However, when there are six multimedia clients, they attempt to consume more than their allocated share. While in the absence of bandwidth re-allocation, Clarity attempts to restrict them to their allocated share, it does not do so when bandwidth re-allocation is permitted. Clarity attempts to re-allocate bandwidth, whenever it is permitted to, by monitoring the disk utilization due to non-multimedia clients. It can be observed that the multimedia clients get a higher portion of the bandwidth than when there is no re-allocation. This difference in the bandwidth consumption in the presence of re-allocation is seen to increase further in the presence of nine multimedia clients. Since nine multimedia clients require more bandwidth than six, the effect of Clarity's allocation policy has a very significant effect. In the absence of re-allocation, Clarity simply restricted each class to its allocated bandwidth causing a substantial wastage of disk bandwidth.

In order to make sure that bandwidth was not being re-allocated by Clarity at the cost of reduced non-real time performance, we measured the throughput and average response time for the non-multimedia clients in the presence of an increasing number of multimedia clients. Figures 5.25 and 5.26 show comparison of throughput and average response time observed by the non-multimedia clients.

Figure 5.25 shows that the throughput observed by the non-multimedia clients in the presence of re-allocation was almost the same as that observed when there was no re-allocation of disk bandwidth. When Clarity re-allocated bandwidth it does so carefully so as not to degrade the performance of non-multimedia clients. The values remain comparable even when more multimedia clients are introduced into the system. The average response times observed in Figure 5.26 are only slightly more when re-allocation

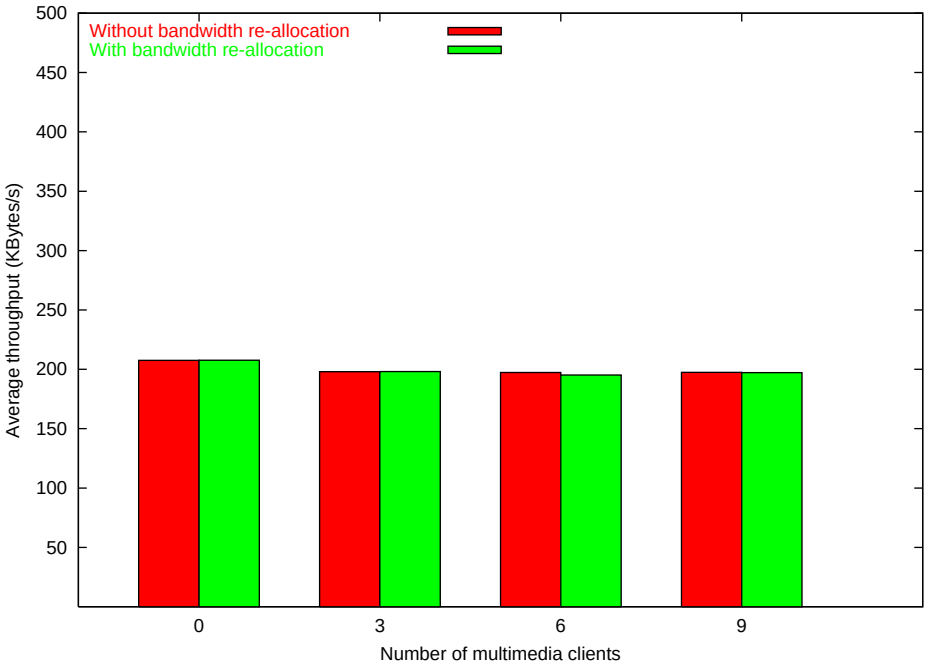

Figure 5.25: Average throughput with and without bandwidth re-allocation policies under `Clarity` when non-multimedia service class is under-utilized

is carried out. In the absence of multimedia clients, they are almost the same. When three multimedia clients are introduced, only a very small difference is observed. With the introduction of six and nine multimedia clients, there is very little increase observed in the difference in the average response times and the response times in the presence and absence of bandwidth re-allocation remains very comparable. When there is no re-allocation of bandwidth the disk requests from non-multimedia clients wait much shorter in the disk queue since the multimedia clients are restricted to their allocation share. However, when re-allocation takes place, the multimedia clients generate more disk requests thus increasing the average response time. We see that there is no sudden increase in the response time due to bandwidth re-allocation to multimedia clients. Thus, `Clarity` not only improves disk usage by re-allocating disk bandwidth from an under-utilized non-multimedia class to a loaded multimedia class, but also efficiently prevents performance

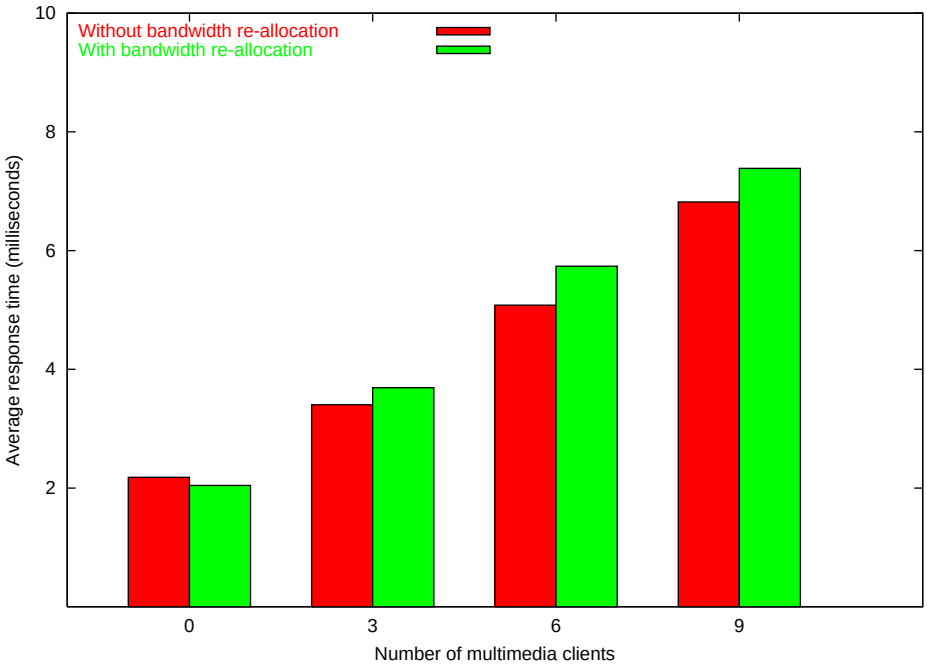

Figure 5.26: Average response time with and without bandwidth re-allocation policies under `Clarity` when non-multimedia service class is under-utilized

degradation of the non-multimedia clients.

In addition to the fact there is no significant impact on the non-multimedia clients, we observed that re-allocation results is a big performance gain for the multimedia clients. Although there was no necessity to re-allocate in the presence of three clients, `Clarity` dramatically improved performance in the presence of six and nine multimedia clients. Figure 5.27 compares the loss experienced by the multimedia clients in the presence and absence of bandwidth re-allocation.

In the presence of three clients there is no loss observed by the multimedia clients since they do not even consume their allocated share of 500 *milliseconds*. However, when the clients increase to six the multimedia client attempt to use more than 500 *milliseconds* but are restricted to reading less than that. Difficulties in measuring DMA times, leads the disk scheduler to restrict the usage of bandwidth to a little less than the allocated

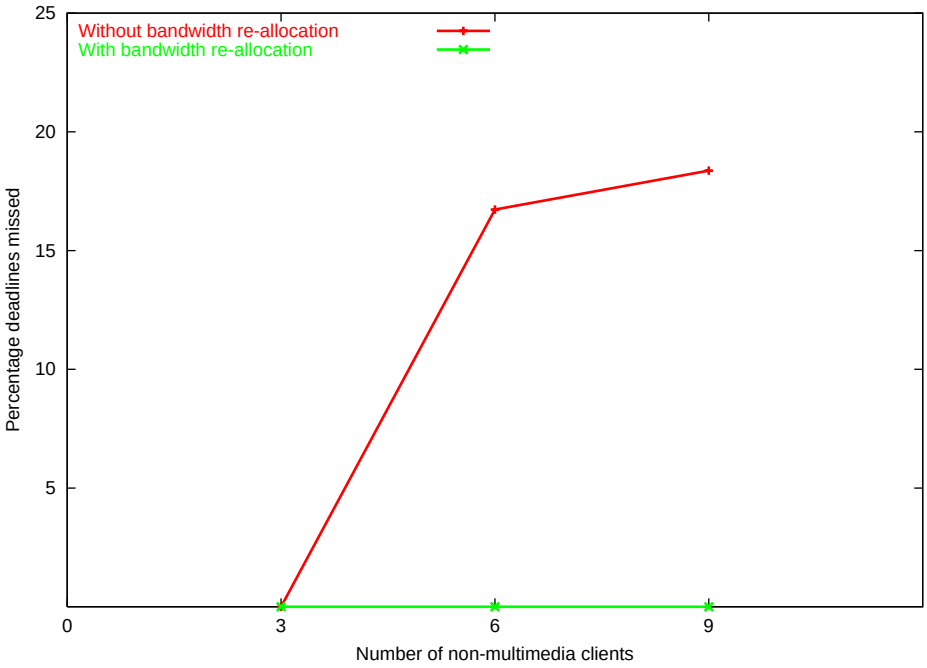

Figure 5.27: Percentage of deadlines missed under Linux and `Clarity` in the presence of excess non-multimedia clients

share. However, when disk bandwidth is permitted to be re-allocated, `Clarity` provides additional bandwidth to the multimedia clients. This prevents any deadline violations by them. When there are nine clients, re-allocation results in the multimedia clients getting much more than 500 *milliseconds* resulting in their meeting all deadlines.

Chapter 6

Conclusion

Most present day operating systems are designed to be general purpose operating systems, which provide best-effort service to applications that run on them. With the rapid emergence of advanced computing capabilities, compression technologies and broadband video and audio, it has become essential that an operating system provide an integrated enviroment for a variety of applications, which include multimedia and non-multimedia applications. This becomes especially relevant for content servers which can handle varied kinds of data (text, graphics, multimedia streams etc.)

While the non-multimedia applications are typically I/O intensive requiring high throughput and low response times when retrieving data from the hard disk, the multimedia applications are isochronous. They require time-constrained, periodic access to the hard disk in order to meet their requirements for rendering video and playing back audio streams. Also, most multimedia applications are soft real-time applications that can be resilient to a small amount of loss but so much to delay and jitter. Thus, an operating system must be designed to tune system performance in order to optimally meet the service requirements of multimedia and non-multimedia applications.

We have presented and evaluated our approach to better admission control and disk

scheduling with `Clarity`. `Clarity` has been designed as a framework for content servers to provide differentiated services for multimedia and non-multimedia clients. All the implementation was carried out under Linux.

We have presented an optimisitic admission controller by using fixed and measurement-based values and compared its performance with current admission control policies which assume worst-case values for disk access. We evaluated our approach for admission control against the pessimistic admission control scheme using a number of criteria including number of simultaneously admitted multimedia clients, bandwidth consumption, and percentage of deadlines violated for all multimedia clients admitted for service. The admission controller was implemented in system calls, which were added to the Linux kernel. This was in addition to minor modifications to existing system calls.

It is clear from graphs in Chapter 5 that our approach clearly outperforms the pessimistic admission controller. While the optimistic admission controller based on fixed average disk access times performs marginally better than the pessimistic variation, its performance increases substantially with increase in disk block size. An improvement over the optimistic admission controller that we proposed and implemented was the measurement-based admission controller. It uses average values of measured DMA times for predicting disk usage when admitting a client and outperformed the pessimistic admission controller by not only admitting far more clients for service, but also by meeting all their deadlines. We also formulated and implemented an adaptive admission controller based on measurements, which can be used when admitting layered streams where each layer contributes to enhancement of video quality. When there is insufficient bandwidth to admit a client reading a layered stream at its full bandwidth requirement, our adaptive scheme increases disk bandwidth utilization but admitting it at a slightly lower rate by exploiting the fact that it is layered. In addition to the implementation of all the above mentioned admission control schemes, we also made a detailed analysis of a cache based admission controller.

In this approach, we attempted to make use of the cache to keep multimedia streams in memory intelligently so that streams separated by only a small gap in time can be served from the memory instead of the disk. This would not only dramatically decrease disk usage, leaving room for other applications, but also work extremely well in conditions where the number of clients is very large but the scope of access is limited to a few video streams. This represents a very typical scenario on a multimedia server where a large number of clients try to access a rather small group of important news or video clips at similar but not identical times.

We also proposed and implemented a disk scheduling algorithm as a part of `Clarity` under Linux. Our disk scheduler has been designed to provide differentiated service to different kinds of clients. We have carried out all our experiments by making use of only two classes: multimedia and non-multimedia. `Clarity` provides classfication of services, inter service class protection, intra service class protection, and bandwidth re-allocation.

We compared `Clarity's` disk scheduling algorithm with the disk scheduling framework under Linux under a number of different conditions. In the absence of clients belonging to different classes i.e. all clients being just non-multimedia clients, `Clarity` matched the throughput and average response time provided by Linux. In the presence of multimedia applications, `Clarity` protected guarantees given to multimedia clients from the effects of an increased load due to non-multimedia clients. Also, `Clarity` ensured that competing multimedia clients did not cause an increased deadline violation for other clients belonging to the same class. We also demonstrated that `Clarity` re-allocates bandwidth very efficiently from one class to another when bandwidth of an under-loaded class can be used to service bandwidth starved clients belonging to another class. This resulted in higher throughput and lower average response time for non-multimedia clients when bandwidth was re-allocated from the multimedia clients. On the

other hand, when bandwidth was re-allocated from non-multimedia clients to multimedia clients, it resulted in no deadline violations for multimedia clients while keeping the throughput and average response times for non-multimedia clients comparable to those under Linux. From the graphs in Chapter 5, it is clear that our algorithm out-performs the CSCAN disk scheduling scheme in Linux to provide fairly reliable service to *soft* multimedia applications by meeting their deadlines, while providing higher throughput and lower average response time to normal applications under different conditions.

Chapter 7

Future Work

In order to better understand the requirements for handling various kinds of audio and video applications within the framework of existing operating systems and for building new ones, we believe that there are a number of areas that merit further research. This chapter essentially provides some pointers to pursue further investigation for bulding better video/audio servers.

We have proposed a cache based admission control. There are a number of areas where this can be studied further. Existing caching techniques like First In First Out, Least Recent Used etc. might not be suitable in their present form. They may require changes because the cache must be made sharable between as many clients as possible. Additionally, instead of evicting files completely or arbitrarily from the cache, there might be some merit to retaining small portions of the file that was already there in cache when deciding what needs to be removed from the cache during cache replacement. For example, it might be a good idea to retain initial portions of a file as opposed to the later portions since it could increase the chances of clients getting admitted. Additional resources required may subsequently become available. Thus, the pages storing later portions of data for a multimedia stream can be marked as more eligible for replacement than the

others. It would be very interesting to see what caching mechanisms can be developed so that cache hits for the multimedia clients can be maximized while sharing the available memory efficiently between the various applications. Naturally, an increase in cache hit percentage would have a direct bearing on the number of clients that can be admitted and served without missing deadlines. Apart from the cache management itself, work can also be done on defining the admission control criteria. Our approach takes into consideration the amount of cached data the requesting client can access if admitted. This is so that there is enough time to put the cache all the data that the client will require in the future. Our approach also considers how far ahead of the requesting clients are others reading the same file. There could be other considerations that can be incorporated to better represent the requirements of the clients that have been admitted and the ones that need a session.

While we have categorized clients and services to provide differentited service, the list is not exhaustive. Further research could be carried out to understand and incorporate optimization techniques for other forms of video and audio encoding techniques. This could lead to further classification of service and coupled with class specific optimizations in the disk scheduler, it could result in service that is flexible and application-specific.

We have created a framework under `Clarity` where the application has to repeatedly request data. It might be interesting to explore an implementation where the kernel returns all the data in one buffer without any necessity for going back and forth. This would mean that the client is expected to have some sort of a ring buffer where one half gets written into and the other half read from. While this might result in the kernel consuming more memory, it might result in faster service and save CPU cycles involved in switching between user and kernel spaces (since the kernel itself issues requests for a certain number of blocks to be read). However, there needs to be a mechanism for the client to inform the disk scheduler how much it needs at the beginning of the round, so that the disk scheduler can schedule requests from various clients efficiently.

While we introduced the concept of assuming that variable bit-rate clients (like MPEG) have layered streams from which to read, further study of different layering schemes can lead to better optimizations on the machine that acts as a multimedia server. This is especially important when reading blocks and dropping requests, where some could be more important than others, during overloaded rounds. However, this kind of preferential treatment to blocks needs extensive support from the file system, since it is only in the file system that we can store some meta information about blocks. This will facilitate their usage when requests are made for disk blocks.

One of the most important aspects of data storage that affects disk retrieval speed is the manner in which data has been organized on the disk (in short the file system). Typically, if the data belonging to a file are spread out in the disk, then a typical access to the disk is more likely to look like a random seek than a sequential seek. Random seeks not only increase retrieval times but also can result in little or no utilitization of any caching policy the operating system might employ to reduce subsequent accesses to the same data. The greater the sequential storage of files, the lower the times for retrieval, since subsequent accesses to the data in a file has a greater chance of being serviced from the disk cache and/or operating system's cache. Therefore, a direction for further research could be the design of a file system that is multimedia friendly and works well in conjunction with the the way disk and operating sytem caches work. Such a file system could be used to employ various schemes for storage. For example, since some blocks of data could be more important than others, this information can be stored in the filesystem and can be used to group more important blocks together. This can help `Clarity` achieve lower disk access times for data that contributes significantly to the video/audio stream. Also, it can help the disk scheduler, such as a part of `Clarity`, make decision easily and quickly about how it can drop disk requests in over-loaded round with minimal impact to the quality of the playback.

Apart from the disk scheduler, the process scheduler to a great extent determines the priority given to multimedia applications. In fact, the process scheduler can greatly enhance the performance of multimedia applications by giving real-time guarantees to applications for execution. We believe that for a system to be truly flexible and service-oriented, knowledge of applications should be present in both the process and disk scheduler. Application specific optimizations in the process scheduler would in turn affect disk scheduling in a number of ways. Real-time scheduling guarantees to multimedia applications become especially important when the load is heavy. This ensures that an application will give sufficient opportunities to request all needed data. A framework like `Clarity` can make use of the guarantees provided by the process scheduler to schedule requests Just In Time (JIT) for their service so that all the remaining bandwidth can be used for scheduling requests from non-multimedia applications. This can be done by calculating the slack time that is available to each disk request and inserting non-multimedia requests whenever there is a slack. This can lead to better disk and processor utilizations, better throughput and average response times for non-multimedia applications and lower deadline violations for multimedia clients.

Bibliography

[1] P. R. Barham, “A Fresh Approach to File System Quality of Service,” in *Proceedings of NOSSDAV’97*, (St. Louis, Missouri), pp. 119–128, May 1997.

[2] M. L. Claypool and J. Reidl, “End-to-End Quality in Multimedia Applications,” in *Handbook on Multimedia Computing*, ch. 40, Boca Raton, Florida: CRC Press, 1999.

[3] M. L. Claypool and J. Tanner, “The Effects of Jitter on the Perceptual Quality of Video,” *ACM Multimedia Conference*, October 1999.

[4] R. Steinmetz, “Analyzing the Multimedia Operating System,” *IEEE Multimedia*, 1995.

[5] C. A. Waldspurger and W. E. Weihl, “Stride Scheduling Deterministic Proportional Share Resource Management,” Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

[6] P. Goyal, X. Guo, and H. R. Vin, “A Hierarchical CPU Scheduler for Multimedia Operating Systems,” in *USENIX 2nd Symposium on Operating Systems Design and Implementation*, (Seattle, Washington), October 1996.

[7] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz, "The Eclipse Operating System: Providing Quality of Service via Reservation Domains," in *Proceedings of the 1998 USENIX Annual Technical Conference*, (New Orleans, Louisiana), June 1998.

[8] I. Leslie, D. McAuley, R. Black, T. Roscoe, B. P., D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE Journal on Selected Areas in Communication*, vol. 14, pp. 1280–1297, September 1996.

[9] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz, "Move-To-Rear List Scheduling: A New Scheduling Algorithm for Providing QoS Guarantees," in *Proceedings of the 5th International Conference on Multimedia*, pp. 63–73, November 1997.

[10] P. J. Shenoy and H. M. Vin, "Efficient Striping Techniques for Multimedia File Servers," in *Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 25–36, May 1997.

[11] P. Goyal, S. Rao, and H. M. Vin, "Optimizing the Placement of Multimedia Objects on Disk Arrays," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pp. 158–165, May 1995.

[12] S. Childs, "Portable and Adaptive Specification of Disk Bandwidth Quality of Service," *Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '99)*, June 1999.

[13] H. M. Vin, P. Goyal, A. Goyal, and A. Goyal, "A Statistical Admission Control Algorithm for Multimedia Servers," in *Proceedings of ACM Multimedia'94*, (San Francisco), pp. 33–40, October 1994.

[14] H. M. Vin, A. Goyal, A. Goyal, and P. Goyal, "An Observation-Based Admission Control Algorithm for Multimedia Servers," in *Proceedings of the First IEEE Inter-*

national Conference on Multimedia Computing and Systems (ICMCS'94), (Boston), pp. 234–243, May 1994.

[15] P. Mohapatra and X. Jiang, "An Aggresive Admission Control Scheme for Multimedia Servers," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pp. 620–621, 1997.

[16] A. K. Atlas and A. Bestavros, "Slack Stealing Job Admission Control," Tech. Rep. 98-009, Computer Science Department, Boston University, May 1998.

[17] D. Anderson, Y. Osawa, and R. Govindan, "A File System for Continuous Media," *ACM Transactions on Computer Systems*, pp. 311–337, November 1992.

[18] A. L. N. Reddy and J. Wyllie, "Disk Scheduling in Multimedia I/O System," in *Proceedings of ACM Multimedia'93*, (Anaheim, CA), pp. 225–234, August 1993.

[19] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling Algorithms for Modern Disk Drives," in *Proceedings of ACM SIGMETRICS'94*, pp. 241–251, May 1994.

[20] P. J. Shenoy, P. Goyal, S. S. Rao, and H. M. Vin, "Symphony: An Integrated Multimedia File System," in *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking (MMCN'98)*, (San Jose, CA), pp. 124–138, January 1998.

[21] P. J. Shenoy and H. M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems," in *Proceedings of ACM SIGMETRICS Conference*, (Madison, WI), pp. 44–55, June 1998.

[22] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley, "Performance Evaluation of New Disk Scheduling Algorithms for Real-time Systems," *Journal of Real-Time Systems*, pp. 307–336, 1991.

[23] R. K. Abbott and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," in *Proceedings of 11th IEEE Real-Time Systems Symposium*, pp. 113–125, December 1990.

[24] C. Martin, P. S. Narayan, B. Özden, R. Rastogi, and A. Silberschatz, "The Fellini Multimedia Storage System," *Journal of Digital Libraries*, 1997.

[25] T. N. Niranjan, T. Chiueh, and G. A. Schloss, "Implementation and Evaluation of a Multimedia File System," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, (Ontario, Canada), pp. 269–276, 1997.

[26] J. Bruno, J. Brustoloni, E. Gabber, M. McShea, B. Özden, and A. Silberschatz, "Disk Scheduling with Quality of Service Guarantees," in *Proceedings of International Conference on Multimedia Computing and Systems*, vol. 2, (Florence, Italy), pp. 400–405, June 1999.

[27] X. Li, S. Paul, P. Pancha, and M. Ammar, "Layered Video Multicast with Retransmission (LVMR): Evaluation of Hierarchical Rate Control," in *Proceedings of Seventeenth Annual Joint Conference of the IEEE Computer and Communication Societies*, vol. 3, pp. 1062–1072, 1998.

[28] D. Saparilla and K. W. Ross, "Optimal Streaming of Layered Video," in *Proceedings of Nineteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, vol. 2, pp. 737–746, 2000.

[29] V. Yodaiken and M. Barabanov, "RTLinux Version Two," 1999. URL: http://www.rtlinux.com/archives/design.pdf.

[30] P. Goyal, P. Shenoy, H. Vin, J. K. Sahni, R. Srinivasan, and T. R. Vishwanath, "QLinux: A QoS Enhanced Kernel for Multimedia Computing." URL: http://www.cs.umass.edu/ lass/software/qlinux, 1999.

[31] X. Li, S. Paul, and M. Ammar, “Layered Video Multicast with Retransmission (LVMR): Evaluation of Error Recovery Schemes,” in *Proceedings of IEEE 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 161–172, 1997.

[32] J. Chung and M. Claypool, “Better-Behaved, Better-Performing Multimedia Networking,” in *SCS Euromedia Conference*, May 2000.

[33] L. Wirzenius, J. Oja, and S. Stafford, “Linux System Administrator’s Guide,” 2001. URL: http://www.ibiblio.org/pub/Linux/docs/LDP/system-admin-guide/sag-0.7.pdf.

[34] O. Pomerantz, “Linux Module Programmer’s Guide,” 1999. URL: http://www.linuxdoc.org/LDP/lkmpg/mpg.html.

[35] D. A. Rusling, “The Linux Kernel,” 1996. URL: http://www.linuxdoc.org/LDP/tlk/tlk.html.

[36] A. Rubini, *Linux Device Drivers*. O’Reily & Associates, February 1998.

[37] R. Card, E. Dumas, and F. Mevel, *The Linux Kernel Book*. John Wiley & Sons, 1999.

[38] M. K. Johnson, “Linux Kernel Hacker’s Guide,” 1996. URL: http://www.linuxhq.com/guides/KHG/HyperNews/get/khg.html.