System support for providing integrated services from networked multimedia storage servers*

Ravi Wijayaratne
Ensim Corporation
1366 Borregas Avenue
Sunnyvale, CA 94089
ravi@ensim.com

A. L. Narasimha Reddy
Department of Electrical Engineering
Texas A&M University
College Station, TX 77843-3128
reddy@ee.tamu.edu

ABSTRACT

In this paper, we describe our experience in building an integrated multimedia storage system, Prism. Our current Linux-based implementation of Prism provides three levels of service: deadline guarantees for *periodic* applications, best-effort better response times for *interactive* applications and starvation-free throughput guarantees for *aperiodic* applications. Prism separates resource allocation from resource scheduling. Resource allocation is controlled across the service classes by a system-wide policy and service class specific admission controllers. Resource scheduling is done at the resources. This separation allows Prism to be deployed even when the storage system is separated on a network from the file system.

We report on the important aspects of Prism architecture, innovations required to build Prism on top of Linux and lessons learned during the implementation and testing of Prism. We present experimental results to show that Prism achieves its goals in supporting multiple service classes within a single system. We compare Prism against standard Linux operating system to show the impact of our approach.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: Organization and Design

Keywords

Multimedia,Disk,File Systems,Scheduling,Admission Control

1. INTRODUCTION

Multimedia data may be represented in various forms- text, data, image, audio, and video etc. We classify the numerous multimedia data types in to 3 classes based on their presentation characteristics. The *periodic* service class encapsulates all continuous media streams such as video and audio. These streams require real-time service. However periodic streams can tolerate few data blocks missing deadlines at the presentation device and still maintain good presentation quality due to error resilient coding schemes [12, 22]. There is another class of applications which require fast service from the system in order to present a response to the user to preserve its interactive nature. Streams with such service requirements are categorized as *interactive* streams. All data access streams not belonging to any of the service classes are classified as *aperiodic* streams. An integrated multimedia storage server services all 3 classes of data, multiplexing its limited resources among the service classes according to their performance goals.

Prism in an integrated multimedia storage server. We have designed Prism to accomplish the following service objectives.

- Meet soft real-time requirements of periodic streams.
- Provide prompt response for interactive streams.
- Provide best effort starvation free service for aperiodic streams
- Maximize storage system throughput and fairness among service classes.

We named our system after the optical instrument prism, inorder to emphasize the basic design concept. An optical Prism would separate a ray of white light in to its constituent spectrum. Similarly the Prism storage server separates the storage I/O requests arriving at the server in to many service classes and provides service based on the service class the request belongs to.

Current industry trends point to the need for consolidating storage of multiple servers into a single system. This separation of file servers from storage systems requires that appropriate multimedia support be available at both the file servers and the storage systems. In conforming to this trend, we have designed the Prism architecture to enable the placement of functionality specific to the storage device at the storage system and the functionality specific to the file system at the file system host. Prism architecture can also be extended to facilitate multi homed devices.

We have implemented a prototype of Prism on a Linux based PC server. The significant contributions of this paper are the following. First, we present the Prism integrated

*This work was supported by an NSF Career Award, NSF Grants CDA-9601675, CCR-9901640 and by EMC corp.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACMMM2001 September 30 -October 5, 2001, Ottawa, Ontario, Canada
.

multimedia server architecture. Second, we discuss our experience in implementing Prism on a Linux based PC server and present the insights gained during this exercise. We also present performance results to substantiate that Prism outperforms a standard Linux based PC server in providing multiple classes of service. We will also discuss how we interfaced NFS to Prism and present NFS performance on Prism.

The road map of this paper is as follows. In Section 2, we will briefly compare our work with other related multimedia system implementations. In Section 3, we will present a synopsis of the resource scheduling and resource allocation policies discussed in [25, 24] that form the basis for Prism. In Section 4, we will describe the *Prism* multimedia storage system architecture. In Section 5, we will describe the implementation details and our experiences in implementing Prism on Linux 2.2 platform. In section 6, we will present experimental results to compare Prism against standard Linux. Section 7 concludes the paper.

2. RELATED WORK

The multimedia system design points described in [9, 15, 16, 21] are similar to our research focus. The merits of providing integrated services as opposed to having partitioned servers was discussed by Shenoy et al. in [19]. They argue that except for certain limited combinations of loads, the integrated server outperforms the partitioned server with the same amount of resources. Providing services for continuous media from storage systems has been receiving wide attention. Most of the research in multimedia storage systems is focussed on service isolation between streams belonging to only two service classes: namely, periodic and aperiodic [13, 3, 11, 21]. These schemes typically use a deadline driven approach to provide soft real-time service for periodic streams and best effort service for aperiodic streams. In Prism, in addition to providing these services for periodic and aperiodic streams, we provide best-effort better response times for interactive streams.

The QLinux multimedia operating system employs the Cello disk scheduling algorithm [10]. They define 3 service classes similar to the ones we define in this paper. In QLinux, the Cello disk scheduler is implemented between the block device interface and the buffer cache. The disk scheduler is scheduled either by a kernel thread or by the application processes. In Prism, the disk I/O scheduler is implemented between the OS block device interface and the device driver and is scheduled by interrupts. Therefore, Prism is more suitable for a distributed storage environment as the scheduler is completely decoupled from the host operating system.

Eclipse [2] provides differentiated service from the storage system for multiple service classes. Eclipse ensures that all resource consumers get the allocated bandwidth based on cumulative bandwidth consumption. In Prism, we pay attention to performance requirements of service classes over finer time granularity.

3. BACKGROUND

Providing integrated services requires differentiating among service classes and allocating and scheduling requests based on the characteristics of the service required by individual service classes. In our framework, we separate resource allocation from resource scheduling. Resource allocation can be enforced through a system wide policy and service class specific admission controllers. Resource scheduling is done at individual devices. Separating resource allocation and resource scheduling enables, assignment of responsibilities to modules that are better equipped to carry out specific functions. For example, the admission controllers can better use the access stream characteristics and could be placed closer to the application level. The disk I/O request scheduler can optimize overall disk access and meet the response time goals of each service class if placed closer to the device level. Our current implementation of Prism focused on the disk and storage resources. We employed Linux's network QOS infrastructure to schedule the outgoing transfers at the storage server's network interface. Scheduling of buffer and processing resources is not discussed here.

Figure 1 illustrates the system model. The request streams belonging to various service classes have to go through the admission controllers. The admission controllers are employed to limit resource consumption to ensure that one application class does not dominate another application class with lower priority treatment at the resource scheduler. The requests which go through the admission controllers are presented to a local or remote storage device. At the storage devices the requests from different I/O classes are pooled and scheduled according to the response time objective of each service class. Device level optimization techniques such as SCAN optimization are also factored into the scheduling decision. We have proposed an admission controlling algorithm for periodic streams in [24] and disk I/O scheduling algorithm for providing integrated services for disk I/O in [25].

The disk I/O scheduler schedules I/O requests in two time frames: namely mainperiod and subperiod. There are several subperiods per mainperiod. We assume that all periodic requests for the current main period are present before the main period starts. Every main period, we create a combined SCAN ordered list of periodic requests and available aperiodic requests. In the absence of interactive requests, these requests could be served in a SCAN order. In order to provide better response times for interactive requests, every subperiod, we allow the scheduler to disturb the SCAN order. Smaller subperiods allow faster response times for interactive requests at the cost of increased seek times. Every subperiod, we examine the interactive list which is maintained in FIFO order and schedule the first interactive request. An interactive request is scheduled only if there is sufficient slack to schedule periodic requests for the current main period and an aperiodic request is scheduled only if there is sufficient slack time for both periodic and interactive requests. Service class specific admission controllers limit the resource consumption of specific class to assigned levels. This allows starvation-free service for all request classes. The above approach forms the basis for Prism.

4. PRISM ARCHITECTURE

4.1 Design strategy

Prism is a system architecture for a storage system that provides integrated services. In this section, we will present the Prism architecture, its design strategy and design issues. We have paid special attention to minimize the changes to the host operating system(Linux) and the Clint side oper-

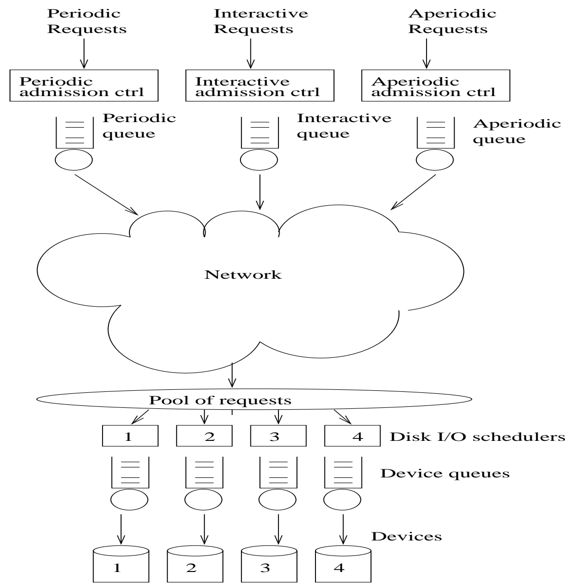

Figure 1: Prism resource allocation and resource scheduling model

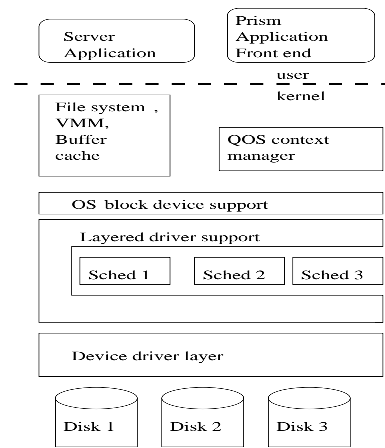

Figure 2: Prism server component layout

ating system and applications.

4.2 Components of Prism architecture

Prism provides service for periodic, aperiodic and interactive streams [1]. It is suitable to be deployed in a file server which services a mix of service classes in one system. It consists of the following components.

- A QOS context management module.
- An API for communicating the service class to the QOS context manager
- File system and kernel support to accommodate prioritizing requests and maintaining QOS parameters of various streams.
- An architecture for layered drivers.
- A scheduler to schedule I/O requests to the storage system.
- Application layer support for providing integrated services.
- Client side support for interacting with Prism server.

Figure 2 illustrates the layout of the above components.

4.3 Application layer support, Prism API and the QOS context manager

In Prism, the QOS aware file I/O operations are carried out on opened instances of files. In most operating systems, an opened instance of a file is identified by a file handle or a file descriptor. There is a broad selection in associating the service class to the accessed stream. In one extreme, we can maintain the service class as part of the persistent data of the file such as in the inode structures. Since persistent file information is typically shared by active processes in most operating systems, all streams that access a particular file will get only one level of service at any time. Such a scheme would limit the ability of a file to be opened in multiple service classes by multiple users. On the other end of the spectrum of options, the QOS information can be sent with each I/O call. The server will need to maintain only minimal QOS state in this scenario. However sending QOS information per call entails changing file systems, communication protocols and even applications. Furthermore it is likely that the admission control process will be in the data transfer path making the I/O call overhead prohibitively large.

Since Prism maintains the QOS context based on the opened instance of a file it is sufficient to provide a simple API only to start and stop a stream. This API must create the QOS context for the started stream using the QOS context manager. A disk I/O request has to percolate through several levels before it is satisfied by the disk subsystem. In a typical operating system these levels may include system call interface, file system, buffer cache, virtual memory manager, operating system block device support and the device drivers [7]. If QOS aware operations are carried out at each of these levels, access to the QOS context of the data object is required. The task of the QOS context manager is to make the service class information of each opened file available to various kernel entities at each layer. This operation necessarily entails maintaining state of the accessed stream inside the kernel. If the operations are carried out in the server itself, the service class can easily be associated with the accessed stream by simply labeling the stream. However it is challenging to maintain QOS context for stateless applications such as NFS [6] [2] by associating the service class with opened stream for each independent I/O operation in the absence of a session state. In such a case, some involvement of the server front end may be required to create the session QOS context.

Before starting a data transfer, the client has to establish a data transfer session with the server. At the Prism server there is a server process for each service class listening to the incoming requests for that service class. Once a request arrives, this process would spawn another process which would henceforth handle the data transfer session.

[1]Prism can be easily extended to up to 31 service classes

[2]NFS version 2

This server side process carries out the admission control procedures. If the stream is admitted, the server side process will use the Prism API and create the QOS context for the opened session. The peer process may then optionally go on to carrying out service class specific operations during the life time of the session. The peer process carries out the functionality of Prism server front end depicted in Figure 2.

The Prism server front end is better implemented at the application level instead of the kernel level since it carries out activities intimate to the data contents of access streams. For example the admission control operations for periodic streams described in [24] use the access patterns of periodic data streams. Furthermore, in the application layer there is sufficient flexibility to modify the procedures depending on how much they get involved in the data transfer process. The design guideline is to minimize such involvement. For example, in the case of the admission control procedure for periodic streams presented in [24], admission control function is carried out only one time per session at the start of the periodic stream. Therefore, elaborate procedures can be used to ensure proper admission procedures. Whereas a rate controller for interactive requests which participate in the I/O data transfer process can be simplified to minimize its impact.

4.4 File system support,kernel support and the layered driver support

We mentioned in section 4.3 that an I/O request needs to go through several levels in the kernel before the it is satisfied. At each of these layers QOS aware operations need to be carried out. Such QOS aware operations may vary from simply labeling a data object with the appropriate service class to doing a detailed transformation to data. Resolving the service class for each access stream poses a challenge as the stream context is lost at the lower levels of the kernel. For example, in Linux[3] operating system the access to stream context is lost at the buffer cache level. Furthermore, the access to the process context is lost at the device driver level [4]. Therefore, Prism kernel and file system support components need to resolve the service class of each data object at each layer of the operating system. Our design guideline was to resolve the service class as early as possible and copy the service class to each layer below.

For algorithms that scheduled multimedia disk I/O requests, the scheduler has to have the capability of accessing all disk I/O requests. The disk I/O traffic is generated by several entities in a typical operating system. The I/O traffic may be generated by explicit disk I/O requests such as `read()` or `write()` system calls or may be generated by operating system events such as page faults or may be generated by house keeping procedures such as buffer cache synchronization [7]. All these requests go through the operating system block device support layer. Therefore the best location to place the Prism scheduler is between the operating system block device support routines and the disk device driver. To plug in such a scheduler layer there has to be some facilities in the operating system. For example in Microsoft Windows NT version 4.0 operating system such facility exists in the form of filter drivers [1]. Filter drivers can be gracefully plugged in at boot up time by modifying the I/O request packet redirecting information database of the I/O

[3] Version 2.2

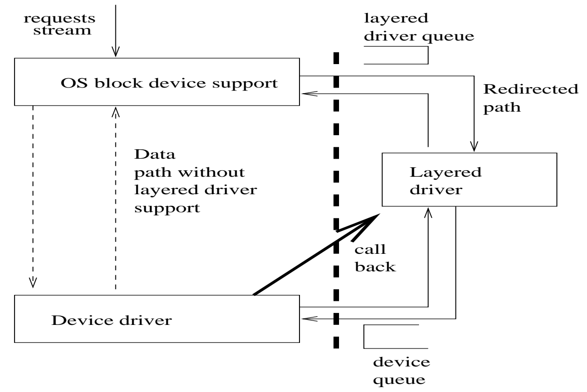

Figure 3: Layered driver support

manager. This can be done by modifying the configuration registry. Some operating systems do not have the layered driver architecture built in. For such platforms a layered driver architecture needs to be created to deploy Prism. To our best knowledge Linux version 2.2 operating system did not have such layered driver architecture support.

Figure 3 illustrates the configuration of the layered driver architecture. The normal data path is depicted in the left hand side of the figure. The system configuration is modified to redirect the data path through the layered driver as shown in the right hand side of the figure. The redirection information is modified when the layered driver is instantiated in the system, done as a system administration task. The device driver may want to communicate the device queue status to the layered driver. The call back facility, which is initialized at the driver instantiating time can be used for this purpose. Unlike processes in the operating system, there are no explicit schedulers for layered drivers. Layered drivers will typically be scheduled in the interrupt context. The main function of the call back facility is to have a way of communicating to the scheduler, any relevant event that occurs inside the device driver.

4.5 I/O request scheduler

As illustrated in Figure 3 the scheduler uses the layered driver architecture to trap all I/O requests to the disk device. A kernel timer facility is necessary for the Prism disk I/O scheduler [25] to function. Most contemporary kernels provide such timer facilities. The scheduler routines that get invoked when a subperiod or a mainperiod event occurs need to be reentrant as these routines would typically be scheduled in the interrupt context. The layered driver call back facility can be used to create a subperiod event when the device driver request queues become empty. The scheduler implements its own queues which holds the I/O requests till they are scheduled to the device. A dispatcher invoked at every subperiod would schedule requests from the internal scheduler queues to the device queues. There needs to be an instance of the scheduler per disk. If the scheduler is implemented using the layered driver architecture it can be a self contained module. There for it is possible to detach the scheduler from the Prism server and place it at a remote storage device away from the file server. This flexibility will be advantageous if Prism is deployed in a network attached storage device or a storage area network environment.

4.6 Client side support

The minimal requirement from the client is to inform the Prism server, which service class it opens the stream in. All streams opened without such a specification will belong to the aperiodic service class. Once the client application decides the service class, there should be some protocol to communicate the service class to the Prism server and establishing the QOS context of the opened stream. This simple functionality can be easily implemented as a set of shared library calls which the client application needs to use. In this case, the client side binaries will have to be updated. Moreover, in Prism the QOS context establishment process is independent of the data I/O operations. Therefore the client can establish the QOS context as a separate control activity and use legacy applications to transfer data and still receive the required QOS from the server.

5. IMPLEMENTATION

We have implemented Prism in Linux version 2.2 platform. We used the Linux standard ext2 file system for the testbed and NFS as a test application. In this section we will summarize our implementation experience under several headings. In each section we will distinguish between the lessons learnt and our innovations.

5.1 Application support and QOS context manager

INNOVATION:QOS context manager
The QOS context manager is built into the kernel as a part of the Virtual File system Switch(VFS) layer. When the QOS context manager is initialized an association is made between the file system in-core super block data structure and the QOS information including a linked list of empty QOS nodes. A QOS node per opened stream is identified by the `<user id,group id,client id,file inode,access key>` tuple. All this information is available at the application. The access key is used to ensure that the client does not violate his QOS agreement. Access key is also used to identify between two occasions of the same client opening the same file in multiple service classes. Creation of the access key is up to the application front end. The QOS context manager simply stores it. To reduce the service class resolving overhead we maintain the active QOS nodes in a hashed linked list, hashed by the inode number and the address of the in-core file system super block data structure.

When describing the Prism architecture we explained that application support depends on the service class. The aperiodic service class is the default service class. Therefore it does not need special treatment at the application level. For periodic and interactive service classes the application layer front end has to participate in two stages– QOS context establishment stage and data transfer stage.

For both service classes we use a similar strategy for QOS context establishment. If the application and the Prism server are sharing the same context of a machine it is easy to create an association between the opened stream and the QOS context node by creating a link to the QOS node from the opened file data structure. However when the client and the server are not sharing the same context such as in the case of a remote client, we have to provide a representation of the opened stream at the server side. This can be done using several techniques. In Prism, we do this by creating a transient symbolic link to the opened file at the server. We then instruct the client to access the symbolic link to receive the desired QOS.

When the I/O request passes the admission controller tests, the server front end process creates a symbolic link to the data file. In the ext2 file system the symbolic link gets an inode number different to the inode number of the file. The inode number of the symbolic link can be used as a part of the access key to identify the unique QOS node of the opened file. For example when user A wants to play the game *NASCAR2000* he would communicate to the interactive server front end, the name of the game and the service class. The location of the file *NASCAR2000.exe* is transparent to the user. The Interactive daemon would create the symbolic link *NASCAR2000userA.exe* to the file *NASCAR2000.exe* and instruct user A to open the file *NASCAR2000userA.exe* to receive interactive service. The data transfer action for the link *NASCAR2000userA.exe* need not make any adjustments to accommodate interactive service.

INNOVATION: Client-Pull in a Server-Push architecture
The disk I/O scheduler assumes server control on the time the I/O requests would arrive at the scheduler. Furthermore, by expecting the client to conform to the access pattern of its attribute file at the server, the admission control assumes server control on the client data retrieval. Both these assumptions fit the server-push access paradigm rather than the conventional client-pull access paradigm.

In the conventional client-pull service model, server control can be achieved by having a separate synchronizing process between the client and the server. Alternatively the server-push paradigm gives full control to the server to decide when the data should be dispatched and in what pattern. Most clients conform to the former paradigm while it is easy to deploy Prism in a paradigm which conform to the latter. To bridge the gap, for each periodic stream active in the server we create a prefetch thread. The prefetch thread always stays ahead of the client and retrieves the data blocks in the same pattern as the client. When the client's request arrives at the server it would see a hit in the buffer cache. If the client decides to work ahead too aggressively it will be penalized as the client accesses surpasses the server accesses and the requests would experience disk I/O latencies. By adopting this strategy, we conform the server to the server-push paradigm while the client remains conforming to the conventional client-pull paradigm. The added advantage of the prefetch process is prefetching file meta data. Since most operating systems cache meta data such as inodes, indirection blocks etc., the prefetch process ensures that the meta data will be present in the caches before the client request arrives. Therefore as far as the client is concerned, all the data it requires are in memory when the client request arrives. Hence, the storage system has effectively *pushed* the data to the client.

INNOVATION: QOS context in stateless applications
In NFS, most operations are stateless. The instance of the open file is identified by the file handle which is opaque to the user [6]. The challenge in deploying a stateless protocol such as NFS in the Prism file server is to associate each stateless I/O operation with its QOS context. In the Linux NFS server daemon implementation the inode number is a part of the file handle [5]. The contents of the file handle

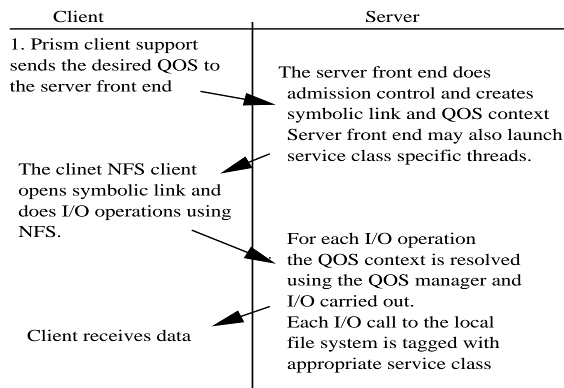

Figure 4: NFS I/O operations

are visible to the server. The Prism client side module initially contacts the Prism server front end and establishes the QOS context for the requested stream. Figure 4 illustrates the sequence of events that occurs in the process of a file I/O operation. The symbolic link is the representative of the opened stream at the server side. The application front end will use the access key field of the QOS context manager to associate the symbolic link to the file. NFS lookup procedure at the server can be modified to force the use of the inode number or the name of the symbolic link for the file handle, if a QOS node is present for that symbolic link. The access key of the QOS node can store the inode number of the data file. When the NFS I/O request arrives, the dispatcher routine can use the inode number stored in the QOS context manager to access the data file. Alternately, the name space of the Prism server side file system can be used to attribute a service class to individual files. This approach is pursued in [2, 20]. Name space based approaches allow service class to be implicitly informed to the server. If such a name space based approach is adopted, we can deploy Prism without having to do any changes at the client side.

The periodic admission controller was implemented as a part of the server front end. A very rudimentary client server QOS context establishment protocol was implemented to test the system. However Prism architecture is flexible to accommodate elaborate QOS negotiating protocols such as RTSP [18]. The Prism API was implemented as wrappers to the `open()` and `close()` system calls. The QOS service class information is passed on as an additional argument for these calls.

5.2 Kernel support and the scheduler

INNOVATION: Kernel modifications to copy service class

When the I/O request transfers from one kernel layer to the other certain amount of context is lost. At the lowest layer the device driver only receives a request to perform an I/O operation to a set of logical blocks. At this level it is difficult to make an association of the block addresses to the opened instance of the file. Therefore the service class has to be copied to every new data structure created at each layer of the kernel. Several kernel data structures needs to be modified to achieve this. A detailed description of the modification to kernel data structures can be found in [23].

LESSON: Problem with synchronous data retrieval and file prefetches

Most contemporary operating systems throttle the rate at which each process can send I/O requests to the storage device to achieve fairness among buffer cache space and I/O bandwidth allocation among processes. Furthermore, to take advantage of the spatial locality of file access data blocks are prefetched. These prefetch algorithms favor sequentially accessed files. In typical prefetch algorithms the prefetch window is aggressively increased if prefetched pages see hits in the buffer cache.

In Linux, the first page of a multiple page request is sent to the storage device synchronously and the subsequent pages are requested asynchronously as a file prefetch operation [14, 4]. In the context of Prism, this action gives rise to two adverse effects. The scheduler assumes that periodic requests will be present at the storage device before the mainperiod starts. However due to the asynchronous operation the requests may arrive at the storage device at various times disrupting the schedules. Furthermore, in the scheduler, periodic requests receive soft real-time service. Since periodic files are sequentially accessed, it is possible that prefetch algorithms favor periodic streams over the other access streams and send to the storage device unanticipated loads. This would disrupt the schedules created at the scheduler. Therefore, for periodic services, we bypassed the operating system throttling and prefetch process and implemented mechanisms to request the exact number of pages per main period from the I/O device in batches.

The synchronous retrieval of file data gives rise to another problem. If two read requests arrive for the same page within a short window of time from different service classes, one process will wait for the page to arrive in memory while the other process actually retrieves the data. There could be a priority inversion in this context. The effects of priority inversion will be reflected in the case when the preceding read request is of a low priority service class than the one waiting for the page. We have not provided for this case in our implementation. However if the page is locked and it is of a lower priority service class we can force the system to reissue the request with the new service class instead of waiting for the page. The first request that completes the page can release the lock. However sufficient controls will have to be implemented to nullify the effects of completion of the second request for the same page.

LESSON: Request Coalescing across service classes

In many operating systems, storage device support routines coalesce requests to take advantage of disk device optimization by batching requests. Furthermore, requests coalescing may also salvage some limited request headers required to schedule other requests. When requests are coalesced the QOS context of some of the requests may be lost since the data structures holding the service class information is detached from these requests. Therefore it is undesirable to coalesce requests belonging to different service classes. In our implementation, we prevent coalescing requests belonging to different service classes.

INNOVATION: Scheduler implementation for Network Attached Storage(NAS)

The scheduler was implemented as a loadable module. When the module is loaded it modifies the layered driver redirection information in the system block device data base. The operating system block device support routines then for-

wards all requests destined to the disk to the scheduler. We use kernel timer facilities to invoke the main period scheduler. The subperiod scheduler gets invoked every time the device driver request queues get empty and when there are requests to be scheduled. The sub period scheduler is implemented as a bottom half. The call back facility in the layered driver is used to invoke the subperiod scheduler.

The layered driver architecture and the scheduler implementation facilitates decoupling of the scheduler from the host server and placing it in a remote network attached storage device. The layered driver architecture enables the scheduler to be an independent entity instead of being a part of the kernel. Furthermore, the call back procedure can easily be used as a flow control mechanism to synchronize the flow of I/O request batches between the host and the NAS device. The scheduler is independent of the file system or the buffer cache and deals only with the block device support layer data structures. Therefore the scheduler can concentrate on device specific optimizations and scheduling.

6. PERFORMANCE EVALUATION

6.1 Testbed

We have implemented Prism architecture on a PC server running Linux version 2.2 with the configuration described in table 1 [8]. In our testbed, periodic applications read 50MB MPEG files. The access pattern was derived after modifying the mpeg frame traces obtained from [17]. To read the 50MB file according to the access trace the experiment had to be run between 2.5 to 5 minutes. The block size for periodic streams were set to 64KB. The aperiodic and interactive access streams were simulated by bursty traffic sources. The off time for these sources were randomly selected with an average of 70ms. The number of data bytes requested on each interval was randomly distributed between 4 and 12 KB. These random reads simulate typical file system accesses. The performance data, unless otherwise stated, was measured at the application. The admission controllers were disabled when required.

6.2 Performance of periodic streams

Figure 5 illustrates the variation of the percentages of I/O calls and data bytes missing deadlines with the number of periodic streams in the system. We had 10 aperiodic and 10 interactive streams active in the system. The periodic admission controller was disabled to test the contribution of the scheduler. We were able to admit up to 10 periodic streams, with only 12% of I/O s missing deadlines. For Linux, at 10 streams, up to 69% of I/O s and up to about 77% of bytes missed deadlines. Only 3 periodic streams can be admitted for Linux, missing deadlines less than 12% . For a larger numbers of periodic streams, the deadline misses in Prism are caused by the load of the periodic streams themselves.

Figure 6 illustrates the influence of the aperiodic traffic on the periodic streams. There were 4 periodic streams and 10 interactive streams in the system. The fraction of deadline missed for both I/O s and bytes was less than 2%. For Linux, the percentage of data and I/O s missing deadlines increased up to 25.5% for I/O s and 34.9% for bytes. Similar performance figures were experienced with the variation of the number of interactive threads. This clearly demonstrates that the periodic streams are well insulated from aperiodic

Table 1: Testbed parameters

Processor	Intel Pentium 233 MHz, 4.3ns clock
Disk capacity	4.3GB
Maximum heads	15
Maximum cylinders	8896
Track to track seek time	3ms
Average seek time	11ms
Maximum seek time	21ms
Average latency	5.5ms
Spindle speed	90 revolutions per second
Head switch time	890μs
Sectors per track	63
Sector size	512
Minimum media to buffer bandwidth	68 M Bits/s
Maximum media to buffer bandwidth	135.5 M Bits/s
Total memory	196 MB

Table 2: Testbed parameters

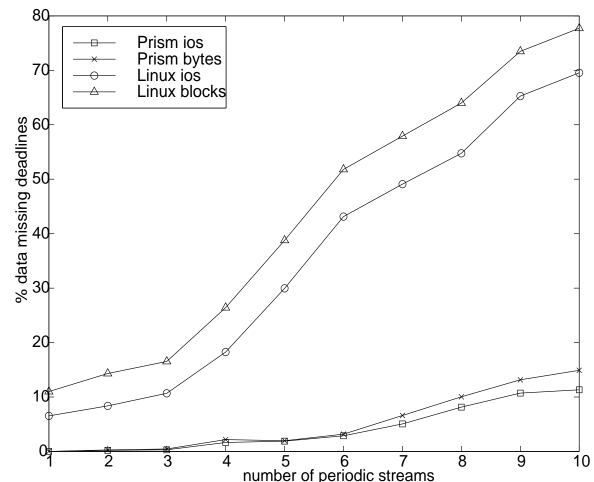

Figure 5: Periodic stream throughput with periodic admission control disabled

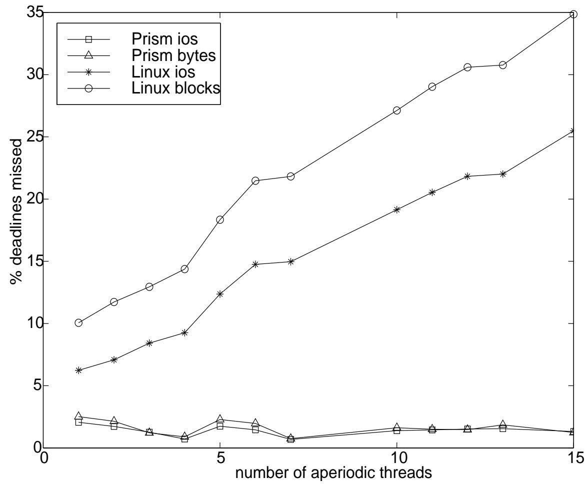

Figure 6: Influence of aperiodic traffic

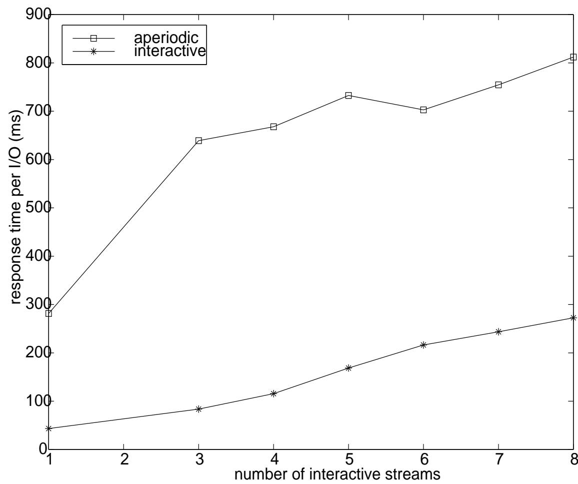

Figure 7: Average response times for aperiodic and interactive classes

and interactive traffic. Increase in traffic of either service class cannot cause periodic streams to miss deadlines.

6.3 Performance of Interactive and Aperiodic streams

Figure 7 illustrates the aperiodic and interactive response time variation with the number of interactive streams in the system. The periodic and aperiodic load were set to 3 streams and 8 streams respectively. The interactive stream response time is much less than that of the aperiodic streams because interactive requests receive preferential best effort service at the scheduler.

At the Prism scheduler, interactive requests get precedence over aperiodic requests if the available time to service interactive requests in the current main period is not sufficient. Therefore if interactive requests arrive at the scheduler without bounds they may starve the aperiodic requests of service. Therefore we limit the number of interactive requests coming into the scheduler by a leaky bucket rate controller. If the interactive arrival rates increase to an extent beyond the amount, the allocated bandwidth could sustain

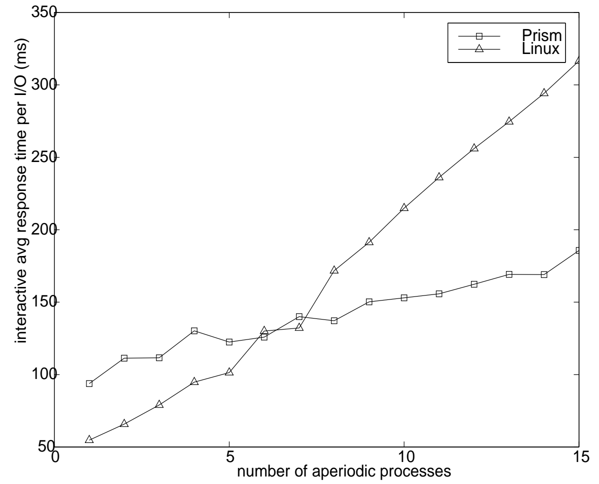

Figure 8: Influence of aperiodic traffic on interactive response time

interactive requests would see queuing delays at the rate controllers. In Figure 7 the interactive average response times grows gradually with the amount of interactive traffic in the system due to queuing delays at the rate controller.

Figure 8 illustrates that the amount of aperiodic traffic has minimal effect on the interactive response times whereas for Linux since both service classes are treated the same way the interactive response time increases. The slight increase in average interactive response times is due to the increase in batch sizes scheduled to the disk each subperiod due to the increase in aperiodic traffic.

It is also important that the frequency of interactive requests experiencing high response time be as low as possible. Our experiments showed that when the server is busy the interactive percentile maximum response times were less for Prism than that of a Linux PC file server. Furthermore the cumulative frequencies for response times less than 50ms, 200ms and 500ms thresholds were much high for Prism than for the PC Linux server.

6.4 Resource sharing

The responsibility of the admission controller is to ensure that all service classes receive the allocated storage I/O bandwidth if bandwidth was scarce. If there is slack bandwidth, the scheduler must make sure that low priority requests use the idle resource. We allocated 50% for periodic streams, 25% if interactive streams and 25% for aperiodic streams. The system traffic consists of system disk I/O traffic such as accesses cause by page faults and file meta data retrieval. The utilization was measured at the device driver to isolate effects of other operating system activities.

Figure 9 illustrate how various service classes share the disk I/O bandwidth. The amount of aperiodic traffic cannot considerably influence the periodic or interactive disk utilization. However aperiodic disk utilization increases with the aperiodic load demonstrating the work conserving nature of the Prism disk scheduler. Although periodic streams have 50% of the bandwidth, the disk utilization for periodic streams is about 40%. This is primarily due to the VBR nature of the streams and the use of worst case service time estimates in the admission controllers. Because

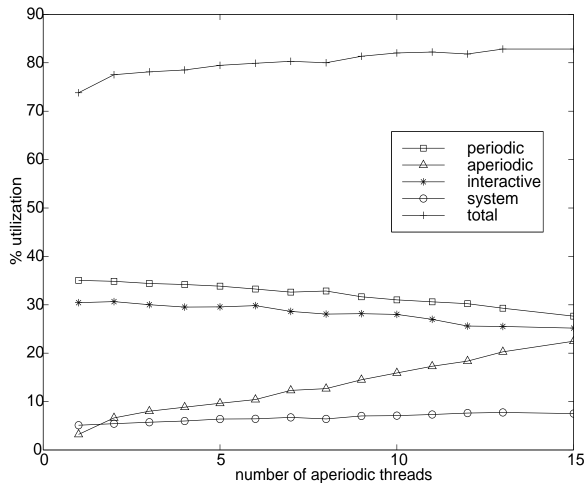

Figure 9: Influence of aperiodic traffic on disk I/O bandwidth utilization

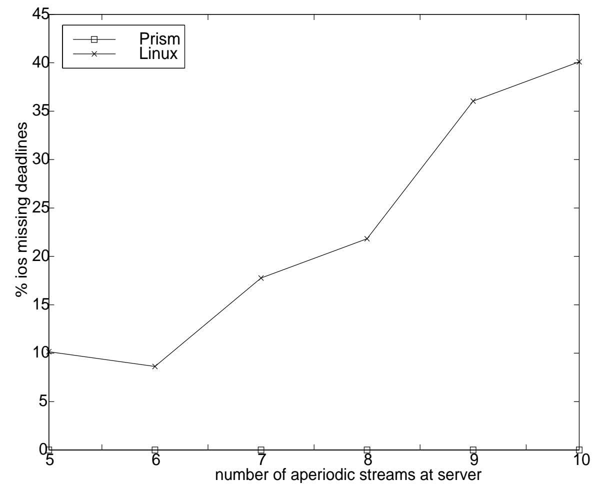

Figure 10: Periodic service class performance on NFS

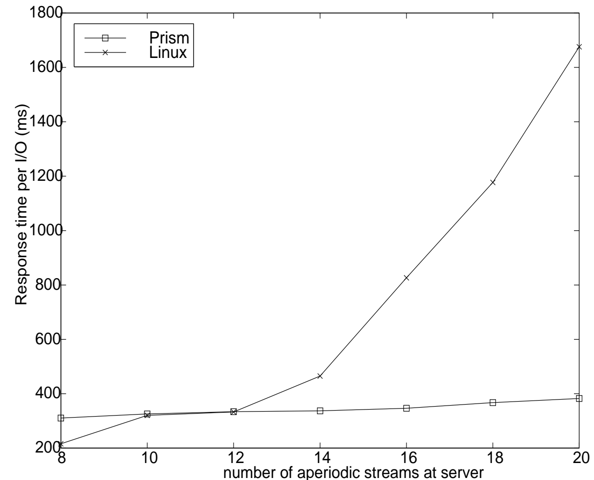

Figure 11: Interactive service class performance on NFS

interactive requests get preferred best effort service, the interactive requests receive more than the allocated share of bandwidth. However, each time the SCAN schedule is disturbed to service interactive requests the seek time is most often charged to the interactive requests. Since the seek operation is shared by the cluster, the other requests experience a reduction in service times. Therefore the actual utilization of interactive requests is less than what is shown in the figure.

We adhere to the overall bandwidth allocations fairly well. The purpose of the bandwidth allocation is to ensure that interactive requests and aperiodic requests receive fair service at the disk. Our response time results show that we accomplish this objective. Furthermore it is possible for aperiodic requests and interactive requests to utilize the unutilized bandwidth as seen Figure. 9.

6.5 NFS performance

We invoked the kernel level NFS daemon on the Prism server. The Prism client and the NFS client were implemented on a 500MHz machine with 128M Bytes of memory and 10 G Bytes of disk space. The client and the server were connected via a public network with 10M Bytes/s of maximum bandwidth. The experiments were conducted during a period when the network was relatively less busy. Even though the network was not busy, network latencies affected our measurement, as we retrieved several hundred MB of information from the server at server speeds. Two steps were taken inorder to make the server the bottleneck instead of the network. First, the client application was delayed until sufficient buffer was built between the client and the server so that the client application would always stay ahead of the sever. This is synonymous with the steps taken by many streaming applications to provide for network jitter. The second step was to create a server load by invoking I/O intensive processes at the NFS server similar to ones we used to test the stand alone Prism server in the previous sections.

Figure 10 illustrates the periodic stream performance for NFS on Prism. We had one periodic stream active at the NFS client. The periodic block size was 64KB. Up to 2.5s worth of data was buffered at the client before the periodic stream started to counter the network delay variation. The experiment was conducted form Prism and Linux. For Prism, no data blocks missed deadlines (all the datapoints are on the x-axis in the Figure). However, for Linux the fraction of data blocks missing deadlines increased from 10.15% to 40.10%.

To determine the performance of interactive request streams on NFS we created 2 periodic threads at the server. There were 3 interactive threads at the NFS client. We varied the number of aperiodic threads accessing data from the Prism server and measured interactive performance at the client application. The experiment was carried out for 100s. Figure 11 illustrates the variation of the average interactive response time per I/O at the NFS client with the number of aperiodic threads at the server. The average interactive response time increases only from 310.6ms to 382.5ms when the client load is increased from 8 to 20 aperiodic streams. Compared to more than 600% gain for Linux. Prism outperforms Linux after a load of 12 aperiodic streams at the server.

7. CONCLUSIONS

In this paper, we have presented the Prism architecture for supporting integrated multimedia services in a single storage system. We reported on our experience on building Prism on top of Linux. We presented experimental results to compare Prism with standard Linux. Our experience suggests that a multimedia file system should (a) allow more than one block request per process to be outstanding, (b) allow file prefetches to be turned off, (c) allow request coalescing to be turned off and (d) allow new device schedulers to be easily loaded into the system. Our approach of allowing the service class to be visible to the devices (a) enabled our Prism storage system to be separated on the network from the file system and (b) allowed device specific optimizations to be employed in supporting the service classes. The results have shown that Prism provides sufficient separation of service classes and that it is possible to simultaneously achieve multiple performance (in particular, bandwidth and delay) objectives in a single storage system.

8. REFERENCES

[1] A. Baker. *The Windows NT device driver book*, chapter 3-7. Prentice hall publications, 1997.

[2] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz. Resource management for qos in Eclipse/BSD. *FreeBSD Conference*, 1999. http://freebsdcon.org/1999/exhibitors/; accessed April 2000.

[3] W. J. Bolosky, R. P. Fitzgerald, and J. R. Doucer. Distributed schedule management in the tiger video file server. *in Proc. ACM Symposium on Operating Systems Principles*, pages 212–223, October 1997.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*, chapter 5–9. Oreilly Publication, 2000.

[5] N. Brown. The linux kernel nfsd implementation. *The Linux knfsd release notes*, 1999. http://www.cse.unsw.edu.au/ neilb/oss/linux-commentary/nfsd.html; accessed Januray 2000.

[6] B. Callaghan. *NFS Illustrated*, chapter 5. Addison Wesley publishers, 1995.

[7] R. Card and E. D. amd F. Mevel. *The Linux Kernel Book*, chapter 3–16. Jhon Woley and Sons Publications, 1998.

[8] S. Corporation. Disk drive product information for IDE WDCAC34300 and IDE WDCAC325 IDE disks. *Disk drive product information*, 1997. http://www.seagate.com; accessed November 20 1997.

[9] D. J. Gemmel, H. M. Vin, D. D.Kandlur, P. V. Rangan, and L. A. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer*, 28(5):40–49, May 1995.

[10] P. Goyal, J. K. Sahni, P. Shenoy, R. Srinivasan, H. Vin, and T. R. Vishwanath. QLinux: A QOS enhanced Linux kernel for multimedia computing. *QLinux home page*, 1999. http://www.cs.umass.edu/ lass/software/qlinux/; accessed Januray 2000.

[11] R. Haskin. The shark continuous media file server. *in Proc. IEEE COMPCON*, pages 12–17, February 1993.

[12] F. Kuo, W. Effelsberg, and J. J. Garcia-Luna-Aceves. *Multimedia Communications - Protocols and Applications*, chapter 1–8. Prentice Hall, 1998.

[13] C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz. *The Fellini multimedia storage system*, chapter 5. Kluwer Academic Publications, 1996.

[14] R. Net. Linux 2.2.14 source code. *Linux source code navigator*, 2000. http://zaphod.redwave.net/linux/kernel-2.2/; accessed May 2000.

[15] T. N. Niranjan, T. Chiueh, and G. A. Schloss. Implementation and evaluation of a multimedia file system. *in Proc. IEEE International Conference on Multimedia Computing Systems*, pages 269–276, June 1996.

[16] T. Roscoe. The structure of multi-service operating system. *Ph.D. Thesis, Univ. of Cambridge*, April 1995.

[17] O. Rose. Statistical properties of mpeg video traffic and their impact on traffic modeling in atm systems. Technical Report 101, Institute of Computer Science,University of Wurzburg, February 1995.

[18] H. Schulzrinne, A. Rao, and R. Lanphier. Rtsp:real time streaming protocol. Technical Report RFC2326, Internet Engineeering Task Force, April 1998.

[19] P. Shenoy, P. Goyal, and H. M. Vin. Architectural considerations for next generation file systems. *in Proc. ACM International Conference on Multimedia*, pages 457–467, October 1999.

[20] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible location and transport of wide-area resources. *Proc. of USENIX Symp. on Internet Tech. and Systems*, Oct. 1999.

[21] M. Vernick, C. Venkatramani, and T. c. Chiueh. Adventures in buliding the Stony Brook video server. *Proc. of ACM Multimedia '96*, 1996.

[22] A. Watson and M. A. Sasse. Measuring perceived quality of speech and video in multimedia conferencing applications. *in Proc. ACM Multimedia Conferance*, pages 55–60, September 1998.

[23] K. B. R. Wijayaratne. *Prism: A file server architecture for providing integrated services*. PhD thesis, Department of Computer Science, Texas A&M University, Texas, USA, 2001.

[24] R. Wijayaratne and A. L. N. Reddy. Techniques for improving the throughput of vbr streams. *ACM/SPIE Multimedia Computing and Networking*, 3654:216–227, January 1999.

[25] R. Wijayaratne and A. L. N. Reddy. Providing qos guarantees for disk I/O. *Multimedia Systems*, 8(1):57–68, January 2000.