Quality of Service Support for Real-time Storage Systems

Zoran Dimitrijević **Raju Rangaswami**

Abstract— **The performance and capacity of commodity computer systems have improved drastically in recent years. However, these systems still lack the support for real-time data access, which is required by an increasing number of emerging applications. In this paper we first present several important storage-bound real-time applications and classify their Quality of Service (QoS) requirements. We then survey the representative work on disk management in the areas of *IO scheduling*, *admission control*, and *data placement*. Finally, we present our approach for providing disk QoS in commodity systems and present key empirical results from the micro-benchmark-based evaluation of our QoS-enhanced Linux kernel.**

Keywords: QoS disk scheduling, real-time storage, Linux.

I. Introduction

The performance and capacity of commodity computer systems have improved drastically in recent years. An increasing number of emerging applications, such as video streaming, video surveillance, virtual reality, scientific and environmental data gathering, digital libraries, or distance learning, require various Quality of Service (QoS) guarantees for data access. For example, they require guaranteed real-time streaming for video or scientific detector data, but guaranteed response time for interactive or high-priority data. These applications increasingly run on commodity systems. However, commodity systems still lack sufficient QoS support for their storage subsystems.

QoS support for real-time storage systems has been an active field of research throughout the past decade. Still, exponential improvements in computational power, disk performance, high-speed local-area networks, and broadband Internet connections are constantly enabling new applications. These applications demand not only large storage space and high-performance access, but also better operating system support for disk quality of service.

A. Storage-bound Real-time Applications

Traditional real-time systems do not use disks to store data and typically operate using only the random-access main memory. An example for these systems are embedded control systems in cars and aeroplanes. On the other hand, the applications that we target have large storage requirements and the only cost-effective solution is to use disks as their main storage medium [26]. Table I summarizes several important storage-bound real-time applications.

Z. Dimitrijević is a Ph.D. candidate in the Department of Computer Science at the University of California, Santa Barbara (email: zoran@cs.ucsb.edu).

R. Rangaswami is a Ph.D. candidate in the Department of Computer Science at the University of California, Santa Barbara (email: raju@cs.ucsb.edu).

Application	Storage access	Bottlenecks
Video-on-Demand	read-only	storage, network
Video surveillance	write-mostly	storage, CPU
Digital libraries	read-mostly	storage, CPU
Distance learning	read-write	network, storage
Virtual reality	read-write	CPU, network, storage
Scientific	write-mostly	storage, network

TABLE I
STORAGE-BOUND REAL-TIME APPLICATIONS.

Video-on-Demand applications provide streaming video concurrently to multiple clients. Clients can issue interactive video requests (for example, fast forward, slow motion, instant replay, or pause/resume). Traditional solutions are designed for local-area network video streaming, and system bottlenecks are in the storage subsystem. With the proliferation of broadband Internet access, global Video-on-Demand systems are becoming more popular. These systems also have bottlenecks in the network management.

Video surveillance applications manage a large number of video streams from surveillance cameras, which need to be reliably recorded to a storage system. At the same time, security personnel need to monitor a subset of video streams in real-time, creating additional read traffic. Emerging solutions for automatic video processing, including suspicious event detection and data mining techniques, also create large computational and database query-processing requirements for video surveillance systems.

Digital libraries manage a large number of heterogeneous multimedia data, including text, images, audio, and video. These heterogeneous data have different QoS requirements and the underlying storage subsystem needs to handle them differently. In addition to mostly read data access for digital libraries, emerging *distance learning* applications need to handle interactive real-time video/audio streaming (both read and write) and dynamic changes to their databases.

Virtual reality applications are still mainly developed in research labs. However, the large storage requirements for represention of complex virtual worlds and the inherent interactivity requirements for storage access make QoS support for their storage systems a necessity.

Scientific applications with real-time storage requirements usually handle a large number of real-time sensors. Data obtained from these sensors have to be reliably recorded to a storage system, since scientific experiments are sometimes hard or impossible to repeat. Examples for these applications are high-energy particle research [1] and the SETI (Search for Extraterrestrial Intelligence) project [2].

B. QoS Requirements for Storage Systems

The general requirements for large-scale storage systems include high availability, high reliability, ease of manageability, large storage space, and high-performance access. With the proliferation of applications that require both large-scale storage and real-time data access, traditional solutions for building storage systems have to be revisited [42], [41].

Traditional storage systems are designed to provide high performance, best effort, and fair service to all clients. This is sufficient for traditional applications that do not require real-time data access. However, in order to support real-time applications, we identify the following requirements for storage access.

- *Differentiated service.* Most real-time applications require both real-time and traditional best-effort data access. This means that a storage system should differentiate IO requests and service them according to their QoS requirements. In effect, this requires that each IO request is associated with its QoS requirements.
- *Guaranteed-latency response.* Some IO requests have to be serviced before their deadlines. Examples for these requests are video data retrieval that must finish before displaying or kernel access to virtual memory on the disk after page faults for high-priority jobs.
- *Guaranteed-rate streaming.* Streaming data with soft or hard real-time guarantees are often read from and written to a storage system. In order to consistently guarantee their streaming rate, a storage system must employ an admission control which ensures that once admitted, streams get sufficient data throughput. Examples for these streams are surveillance video and scientific detector data.
- *Low latency and high-throughput.* Best-effort data still require low latency and high-throughput access, and a storage system must employ scheduling algorithms that provide the high-performance access for best-effort IO requests, while satisfying guarantees for real-time IOs.

C. Paper Outline

The goal of this paper is to provide an overview of current solutions for designing storage systems with real-time QoS requirements and to present our implementation of disk QoS scheduler for the commodity Linux operating system. The paper is organized as follows: In Section II, we survey the representative work on QoS disk management. In Section III, we present the design and implementation of our QoS disk scheduler for Linux suitable for real-time storage systems. In Section IV we illustrate the effectiveness of our Linux-based prototype using microbenchmarks. Section V presents related research. In Section VI, we make concluding remarks and suggest directions for future work.

II. Overview of Real-time Disk Management

In this section we classify the representative work on disk management in the following three areas: *IO scheduling*, *admission control*, and *data placement*.

A. IO Scheduling

Table II depicts several approaches for best-effort, real-time, and heterogeneous (mixed-media) disk scheduling. Disks can be classified as non-volatile storage devices with non-uniform memory access. In terms of data throughput, the best performance is achieved when the disk access is sequential. However, file systems cannot always place and access data sequentially, since various applications have inherent random data access patterns.

Best-effort	Real-time	Heterogeneous
FCFS	Rate-monotonic	GSS
SSTF	EDF	Cello
SATF	SCAN-EDF	ΔL (clockwise FS)
SCAN	EDL	User-safe disk
C-SCAN	Round-robin	MARS
Freeblock	Bubble-up	

TABLE II
Disk schedulers.

Worthington et al. [44] survey the scheduling algorithms for traditional, best-effort disk access. FCFS (First-Come-First-Serve) approach services disk IO requests in the order in which they arrive. Because disk seek times differ drastically and FCFS does not optimize disk seeking, this approach leads to poor utilization of disk throughput when the IO sizes are small and the access is random.

SSTF (Shortest-Seek-Time-First) and SATF (Shortest-Access-Time-First) [19] methods use greedy heuristics in order to minimize disk seeking. SSTF schedules the IO in the waiting queue that requires the shortest seek time relative to the current disk arm position. Similarly, SATF schedules the IO that requires the shortest access time (which includes both seek time and rotational delay) relative to the current disk position. While both methods achieve good disk throughput utilization by minimizing access overheads, they do not prevent starvation. Some IO requests can spend a long time in the queue and the maximum latency is not bounded.

SCAN and C-SCAN algorithms use a simple elevator principle which solves starvation and reduces disk seeking [37], [44]. In SCAN, the disk arm starts from one end of the disk and moves to another, servicing IO requests on the way. In C-SCAN, the disk arm services IO requests only in one direction (usually in increasing order of disk blocks, which means from outer portions of the disk towards the inner ones). SCAN and C-SCAN guarantee non-starvation, but still an IO request can spend a long time in the queue. To prevent this, the OS usually bounds the number of requests that are serviced in one SCAN turn. This also bounds the maximum latency for each IO request. Most commodity operating systems use a variation of this simple elevator principle for best-effort disk scheduling.

SSTF and SATF require a disk model in order to predict disk seek and access times, which is not required for SCAN algorithms. This is the reason why the versions of SCAN and C-SCAN are the most widely currently used disk schedulers. Recently, several scheduling algorithms that rely on detailed disk models are designed to improve disk access [22], [21], [13]. *Freeblock scheduling* [22], [21] uses rotational prediction

to schedule low-priority IOs in the background without affecting other IOs. *Semi-preemptible IO* [13] schedules IO requests using multiple fast-executing disk commands and enables disk access preemption between them.

Real-time disk scheduling algorithms consider additional real-time requirements when servicing disk IOs. These algorithms can be classified in the following two high-level classes.

- *Deadline-based schedulers.* Each disk IO is associated with a deadline. The disk scheduler should service all IOs before their deadlines. Examples for these schedulers are rate-monotonic, EDF, SCAN-EDF, and EDL [26].
- *Cycle-based schedulers.* All real-time disk IOs are serviced in cycles and they all share the common deadline—the disk scheduler should service all IOs before the cycle expires. Examples for these schedulers are round-robin [37], bubble-up [5], and bubble-up 2D [7].

The deadline-based schedulers are more general schedulers since one can implement the cycle-based scheduling using deadlines (by setting deadlines for all IOs in a cycle to the cycle's end). However, for real-time streaming, which is the most common case, cycle-based schedulers are more natural and can provide easier and more efficient admission control.

Heterogeneous disk schedulers support scheduling for both best-effort and real-time IOs. In Group Sweeping Strategy (GSS) [9], requests are serviced in cycles, in round-robin manner. To provide the requested guarantees for continuous media data, GSS introduces a *joint deadline* mechanism: it assigns one joint deadline to each group of streams. This deadline is specified as being the earliest one out of the deadlines of all streams in the respective group. Streams are grouped in such a way that all of them comprise similar deadlines.

Cello [35] employs a two-level disk scheduling architecture, consisting of a class-independent scheduler and a set of class-specific schedulers. The two levels of the framework allocate disk bandwidth at two time-scales: the class-independent scheduler governs the coarse-grain allocation of bandwidth to application classes, while the class-specific schedulers control the fine-grain interleaving of requests. Symphony [34] multimedia file system supports diverse application classes that access data with heterogeneous characteristics using Cello framework.

Clockwise [3] is a real-time file system that schedules best-effort and real-time disk requests so that real-time disk requests are serviced before their deadlines and the best-effort requests are serviced as quickly as possible without violating real-time deadlines. It is based on a non-preemptible EDF scheduling.

User-safe disks [27] export a virtual device interface to a number of different clients. They provide the protection necessary to ensure that applications cannot violate the system integrity and ensure the guaranteed QoS for each client's data access.

MARS [4] is a scalable web-based multimedia-on-demand system. To provide fair guaranteed access to storage bandwidth, it uses multiple-priority queues and services them with a *deficit-round-robin* (DRR) fair queueing algorithm within the SCSI driver. Their DRR-based SCSI system provides efficient sharing of resources between real-time and non-real-time disk requests.

In Section III we explain our approaches for scheduling disk IOs with heterogeneous QoS requirements in Linux-based systems.

B. Admission Control

The admission control is a mechanism for deciding if a particular request can be admitted into the system or not. In this paper we classify the disk admission control approaches in the following three categories.

- *Best effort.* Best-effort approaches admit all requests into the system. When the disk cannot service all requests by their deadlines, the system's QoS deteriorates. The systems that use the best-effort approach usually distinguish between different requests, and try to first reduce the QoS for less important requests.
- *Deterministic.* Deterministic approaches admit only requests that can be serviced with their required QoS. These methods use either worst-case assumptions (that usually lead to low disk utilization) or rely on low-level disk models [43], [32], [14] to predict the disk performance. The problem with these approaches is the inherent variability in QoS requirements for various data. For example, compressed videos do not have a constant bit-rate, but in order to deterministically guarantee the real-time streaming, admission control must use the maximum expected bit-rates. This leads to suboptimal disk utilization.
- *Statistical.* Statistical approaches monitor system performance and use various heuristics to predict if they are able to admit a new request or not. This usually means that they can provide only soft guarantees. However, they can utilize the disk better than the deterministic approaches. If the system can provide different QoS for different requests, one can use the deterministic admission control for important requests and various statistical approaches for other IO requests. We proposed several statistical admission control methods in [12] and compared their performance with a deterministic approach.

C. Data Placement

A number of data placement solutions have been proposed for both single disk and multiple disk systems. Initial solutions for placing data on single disk systems were proposed in the UNIX Fast File System (FFS) [23], which proposed the notion of cylinder groups to place related data closer on the disk surface. Log-structured placement [31] proposed performing all stream writes sequentially in large contiguous free space on disk to reduce the overhead for write operations at the cost of sub-optimal stream retrieval. Other single disk placement strategies include multi-zone placement [29], constrained block allocation [28], track extents [33], etc. Multi-zone placement proposes matching of stream bit-rates to zone bit-rates so that the disk throughput is utilized better. Constrained block allocation controls the separation of successive stream blocks on the disk to reduce access overheads. Track extents proposes allocating and accessing related data on disk track boundaries, to avoid excessive rotational latency and track crossing overheads.

For multiple disk systems, the additional design goal is that of load-balancing. Striping proposes scattering bytes (fine-grained striping) or blocks (coarse-grained striping) of data across the disks to increase the throughput of the disk array system using concurrent access. In [6], the authors argue that for streaming applications, operating the disks in the disk array independently rather than striping is better in terms of memory use. Another approach to load-balancing in a disk array system is random duplicated assignment [20], in which each data object is replicated and the copies are placed randomly on two disks. This offers reliability and extendibility apart from lowering response times and RAM costs. Another approach to reliability is using a parity disk as in RAID [25] to detect and correct errors on the disk surface.

III. A Linux QoS Disk Scheduler

Current commodity operating systems are designed to provide efficient best-effort access to its resources. One exception is CPU scheduling, which typically provides preemptible priority scheduling. Various applications successfully utilize this priority scheduling to guarantee various real-time requirements. In addition to the existing CPU scheduler, in this paper we investigate adding a preemptible priority disk scheduler into commodity operating systems.

A. Kernel-level Scheduler

The disk scheduling in commodity operating systems has not been radically changed since the early days of Unix systems [44]. The standard best-effort disk scheduling is not appropriate for multimedia applications, especially not for applications with harder real-time requirements [17]. In this section we explain our implementation of the priority-based disk scheduler [15]. We employ a scheme in which all best-effort disk IO requests share the common priority level, and the user-level applications can specify both higher and lower priority for some of their IOs. Being able to specify higher-priority requests can substantially reduce the response time for latency-critical jobs. Similarly, being able to specify lower-priority requests enables an application to perform background operations which do not disturb more important operations. In addition to simple priorities, a user-level daemon can periodically set a best-effort slack time. When this slack time is available, the kernel schedules best-effort requests before the guaranteed-rate requests (effectively performing the priority inversion). This is useful for reducing the average response time for interactive best-effort applications. (In Section III-B we explain in details how to use this slack time to implement a user-level cycle-based scheduler.)

Disk Meta-scheduler

Hierarchical schedulers are introduced to simplify the design of real-time schedulers and enable dynamic scheduling [30]. The main reasons for using the hierarchical approach in our implementation are 1) to enable modularization and 2) to integrate priority scheduling without modification of the existing best-effort disk scheduler.

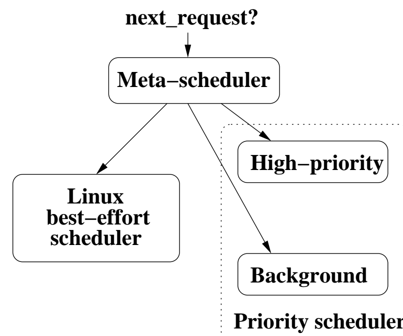

Fig. 1. UCSB-IO hierarchical disk scheduler.

Figure 1 depicts the decision process when the disk device driver requests the next IO. The meta-scheduler first checks if there is any high-priority request (with higher priority than the default best-effort priority). If there is, it invokes our priority scheduler. Otherwise, it invokes the default best-effort disk scheduler. If there are no best-effort IO requests in the queue, the meta-scheduler checks if there is any low-priority request, and if it exists, invokes the priority scheduler to service it. These lower priority requests are serviced in chunks [13] and can be preempted if the higher-priority IO request arrives.

B. User-level Scheduler

Dedicated systems with QoS requirements have been implemented on top of commodity systems (for example, XTREAM [12]). However, in the time-sharing multiprogrammed systems the design is complicated by the fact that multiple processes are sharing the storage subsystem. Current commodity operating systems provide only the best-effort fairness to all processes. This means that the application with QoS requirements has to ensure that no other processes use its critical resources at the same time. In this section we propose one possible user-level QoS scheduler on top of our kernel-level priority scheduler. We support the following three classes of IO requests: *guaranteed-rate IO*, *interactive IO*, and *background IO*.

Guaranteed-rate IO: Streaming applications require guaranteed-rate disk IO in order to get jitter-free video playback or recording. In our implementation, we employ time-cycle-based disk scheduling because of its simple design, predictable behavior, and simple admission control. Figure 2 depicts time-cycle-based scheduling with three guaranteed-rate streams. The scheduler maintains a bubble slot[5] and services best-effort IO requests when they do not violate the streaming guaranteed-rate requirements.

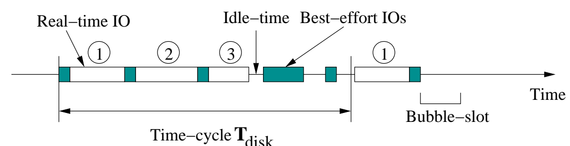

Fig. 2. Time-cycle-based disk QoS scheduler.

The user-level time-cycle based scheduler uses the following kernel methods: disk IO tagging, disk priority scheduler,

best-effort slack-time management, and asynchronous IO. (In Section III-A we have explained the first three methods. Asynchronous IO is already available in commodity operating systems.) At the beginning of each cycle, the user-level scheduler asynchronously submits all real-time IO requests for the whole time-cycle. Admission control explained in Section III-B ensures that these IO requests will be serviced before the end of the cycle. The scheduler also sets the slack-time register for best-effort IOs at the beginning of each time cycle. The kernel disk meta-scheduler services the next best-effort request only when it has sufficient slack time or when all real-time requests are already serviced. The user-level scheduler can choose to service an interactive or other important request at any moment by specifying high priority for that particular disk IO.

Interactive IO: We support two types of interactive requests: best-effort and guaranteed-latency IO requests. Best-effort interactive requests are serviced before higher-priority guaranteed-rate requests when the slack time is available. Guaranteed-latency requests are serviced with higher priority than the guaranteed-rate requests. The scheduler reserves time for interactive guaranteed-latency requests and uses admission control to ensure that they do not violate the real-time requirements for guaranteed-rate IO.

Background IO: Background IO requests are for data which do not need a prompt service. For example, the application may choose to perform data mining, backup copy, or logging using the background IO. Our scheduler services background IO using the low-priority disk access.

Disk Admission Control

The user-level disk scheduler uses the following analytical model to decide if the new stream can be admitted in the system. Given N streams to support, in each IO cycle the disk must perform N IO operations, each consisting of a latency component and a data transfer component. Let T_{disk} denote the disk cycle time. Let L_{disk_i} denote the disk latency to start IO transfer for stream i. Let r_{RT} denote the fraction of time reserved for real-time streams within each disk cycle. Let B_i denote the bit-rate for a stream i. Let R_{disk_i} denote the disk throughput for the zone on which stream i resides. Then T_{disk} can be written as

$$r_{RT} \times T_{disk} \geq \sum_{i=1}^{N} L_{disk_i} + \sum_{i=1}^{N} \frac{T_{disk} \times B_i}{R_{disk_i}}.$$

The above equation can be simplified as

$$T_{disk} \quad \geq \quad \frac{N \times \bar{L}_{disk} \times R_{disk}}{r_{RT} \times R_{disk} - N \times \bar{B}} \qquad (1)$$

where $r_{RT} \times R_{disk} > N \times \bar{B}$. When the application asks for a specific guaranteed-rate, the scheduler checks to see whether Inequality 1 is satisfied. If not, the application is notified that it cannot get the required disk bandwidth. More details are in our related paper about the XTREAM system [12].

C. Linux Implementation Details

Figure 3 depicts a simplified architecture of the Linux 2.5 development kernel. Currently, the upper-layers in the official

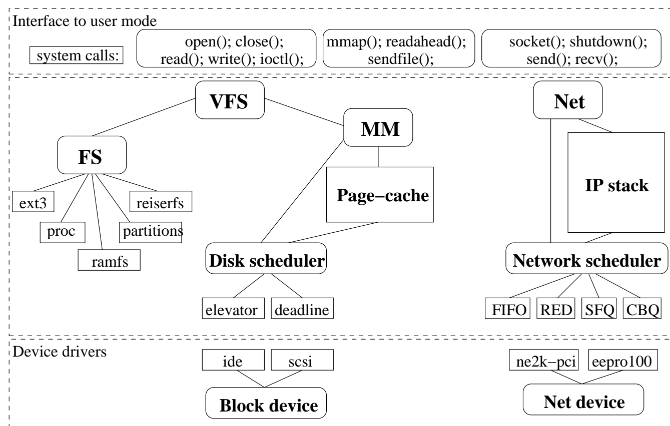

Fig. 3. Simplified architecture of Linux 2.5 kernel.

Linux kernel do not pass file information to the disk scheduler layer. We enable specifying QoS parameters per file descriptor, and pass the pointer to these QoS parameters to each IO request in the disk queue (similar to the approaches in semantically-smart disk systems [38]). After the *open()* system call, file accesses get the default best-effort QoS. We introduce several new *ioctl()* commands which enable an application to setup different QoS parameters for its opened files. These additional *ioctl()* commands are summarized in Table III.

Ioctl command	Argument	Description
IO_GET_QOS	**struct ucsb_io ***	Get file's QoS
IO_BESTEFFORT		Set best-effort QoS
IO_SOFT_RT	**int *rate**	Set guaranteed rate
IO_PRIORITY	**int *priority**	Set priority

TABLE III
ADDITIONAL *ioclt()* COMMANDS.

Disk Profiler

We improve the basic time-cycle model by introducing realistic multi-zone disk modeling [14] and maintain the bubble-up slot [8] to minimize latency for interactive streams.

To provide guaranteed-rate disk access, our scheduler relies on the realistic disk model obtained using an automatic disk profiler [14]. In our previous work we have implemented and tested our user-level disk QoS scheduling [12]. However, the kernel disk scheduler can achieve harder real-time guarantees. In order to support interactive, high-priority IO requests, we use chunking when accessing large sequential disk regions [10], [11], [13]. This way, the higher-priority request can preempt the lower-priority one.

Programming Model

To minimize changes in the Linux kernel and enable seamless porting of existing applications to utilize the operating system QoS extensions, we employ the following strategies:

- Applications provide explicit parameters for disk-access QoS (using *ioctl()* system calls and the */proc* file-system interface).

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include "../linux/include/linux/ucsb_io.h"

#define O_DIRECT        040000 /* direct disk access hint */

static int io_size = 32*4096;

main(int argc, char **argv){
        int ret,size;
        int rate,priority;
        struct ucsb_io qos;
        int fd;
        char *buf;

        ret = posix_memalign(&buf, 4096, io_size);
        if (argc<2) {
                fprintf(stderr,"Usage: %10s <file>\n",argv[0]);
                exit(-1);
        }
        fd = open(argv[1],O_RDONLY | O_DIRECT);
        if (fd<0) {
                fprintf(stderr,"Cannot open file: %s\n",argv[1]);
                exit(-1);
        }
        priority = 100;
        ret = ioctl(fd,UCSB_IO_PRIORITY,&priority);
        if (ret) {
                fprintf(stderr,
                "WARNING: Cannot set UCSB_IO priority!\n");
        }

        while ((size=read(fd,buf,io_size)) > 0){
                do {
                        ret=fwrite(buf,1,size,stdout);
                        if (ret<=0) break;
                        size-=ret;
                } while(size>0);
        }
}
```

Fig. 4. Example code for a high-priority Linux *cat*.

- The kernel provides implicit optimizations for the file system's data placement.

An application opens a file using unmodified *open()* system call. The file gets the default best-effort disk QoS. After opening, the application can change the file QoS at any time using *ioctl()* system call. The QoS setting is per file descriptor, and the application can access the same file using different QoS. The application can setup the priority and rate for each file. The kernel notifies application if the QoS is granted using *ioctl()* error codes.

Additionally, the application can change the disk scheduler's parameters using the */proc* interface (for example, the scheduler's slack time). The application can also use the */proc* interface to get QoS usage statistics or to setup a debugging level.

We have changed three system calls to provide disk QoS: *read()*, *write()*, and *ioctl()*. Figure 4 depicts an example application which performs the high-priority *cat*. The disk priority is specified using an *ioctl()* on the opened file.

IV. Experimental Results

In this section we present a micro-benchmark evaluation of our kernel-level disk priority scheduler. We have implemented the scheduler in Linux 2.5.67. Our testbed was an 800 MHz Pentium III system with one IDE (ST330630A) and one SCSI disk (ST318437LW). In this draft we will present only the results for guaranteed-rate streaming. In the final version of this paper we plan to add the results for guaranteed-latency disk IO.

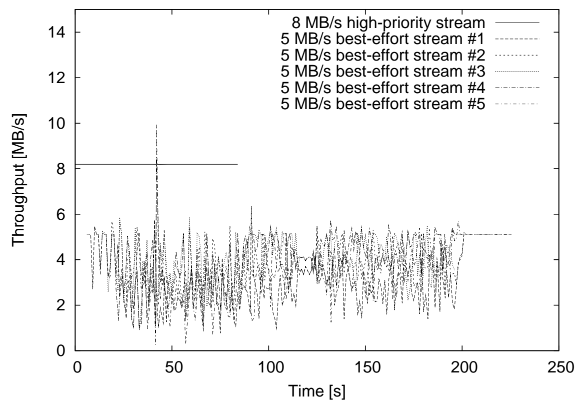

Fig. 5. High-priority vs. best-effort constant-rate streaming.

Figure 5 depicts the difference between a high-priority constant-rate stream (8 MB/s) and best-effort constant-rate streams (5 MB/s). We read data from the ext2 file system on the SCSI disk. Each stream was accessing a different sequentially-placed 700 MB file. In the first 5 seconds, the only stream in the system was the high-priority one. Additional 5 MB/s best-effort streams were introduced after 5, 10, 15, 20, and 25 seconds. The disk was not able to provide the requested throughput for all streams, but our system was able to provide the requested 8 MB/s for the high-priority stream. The cycle time for the high-priority stream was 4 seconds.

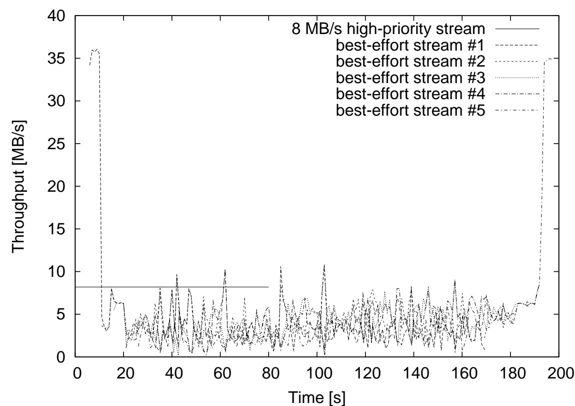

Fig. 6. High-priority constant-rate streaming in the presence of best-effort streams.

Figure 6 depicts the streaming throughput for a high-priority constant-rate stream (8 MB/s) and best-effort streams. Each best-effort stream tries to read as fast as possible. Again, at the beginning of the experiment we had only one high-priority stream and introduced additional best-effort streams after 5, 10, 15, 20, and 25 seconds. We can see that the first best-effort stream started with a high streaming rate. When the number of concurrent best-effort streams increased, the disk seeking overhead reduced the data rate, but without reducing the rate for the high-priority 8 MB/s stream. The cycle time for the high-priority stream was again 4 seconds.

Figure 7 depicts the average response time for both high-priority and low-priority disk IOs. Our micro-benchmark consisted of one high-priority constant-rate stream (4 MB/s) and

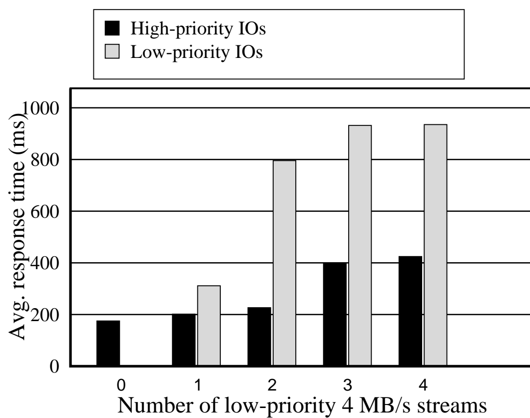

Fig. 7. Average response time for disk IOs.

several low-priority constant-rate streams (also 4 MB/s). The cycle time for all stream was the same (4 seconds). The priority scheduler was able to provide substantially better response time for high-priority IOs.

V. Related Work

Multimedia real-time systems were the focus of several relevant survey and trend studies [39], [18], [16], [26]. Plagemann et al. [26] survey related work in operating system architectures, CPU scheduling, disk management, memory management, and low-level bus, cache, and device management. Gemmell et al. [16] concentrate on disk file systems and scheduling for continuous media applications. In this paper, we also focus on disk QoS management, but we are interested in heterogenous media applications, which manage interactive, continuous, and best-effort data.

While it is the case that most current commodity operating systems do not provide sufficient support for real-time applications, several research projects are committed to implementing real-time QoS support for open-source commodity operating systems [24], [36], [40]. Molano et al. [24] presented their design and implementation of a real-time file system for RT-Mach (which is a microkernel-based real-time operating system from CMU). Shenoy et al. [36] and Sundaram et al. [40] presented their QoS extensions for Linux operating system (QLinux). Our goals are similar to QLinux since we want to add QoS support for Linux disk access. However, in this paper we investigated an approach with minimal changes in Linux kernel space, which is sufficient for an efficient implementation of QoS disk scheduling.

VI. Conclusion and Future Directions

In this paper we have first presented several important storage-bound real-time applications and classified their QoS requirements. We have then surveyed the representative work on disk management in the areas of *IO scheduling*, *admission control*, and *data placement*. Finally, we have presented our approach for disk QoS in commodity systems and key empirical results from the micro-benchmark evaluation of our QoS-enhanced Linux kernel.

We plan to further study the QoS support required for multi-disk RAID systems. Our next milestone is to design a preemptible priority-based RAID scheduler and to investigate its benefits for real-time applications.

Acknowledgments

This work was partially supported by NSF grants IIS-0133802 (Career) and IIS-0219885 (ITR), and a SONY/UC DiMI grant. The authors wish to thank our advisor Prof. Dr. Edward Chang and our colleagues Mohit Sang, Krishna Ramachandran, and Steven Maglio for their valuable help and suggestions.

References

[1] CERN. *http://www.cern.ch*.
[2] SETI Institute. *http://www.seti.org*.
[3] P. Bosch and S. J. Mullender. Real-time disk scheduling in a mixed-media file system. *IEEE RTAS*, May 2000.
[4] M. M. Buddhikot. Project Mars: Scalable, high performance, web based multimedia-on-demand services and servers. *University of Washington PhD Dissertation*, August 1998.
[5] E. Chang and H. Garcia-Molina. Bubbleup - Low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98, November 1997.
[6] E. Chang and H. Garcia-Molina. Effective memory use in a media server. *Proceedings of the 23rd VLDB Conference*, pages 496–505, August 1997.
[7] E. Chang, H. Garcia-Molina, and C. Li. 2d BubbleUp - Managing parallel disks for media servers. *Proceedings of the 5th Foundations on Data Organization*, pages 221–230, November 1998.
[8] E. Chang, H. Garcia-Molina, and C. Li. 2d Bubbleup - Managing parallel disks for media servers (extended version). *Stanford Technical Report SIDL-WP-1998-0090*, February 1998.
[9] M.-S. Chen, D. D. Kandlur, and P. S. Yu. Optimization of the grouped sweeping scheduling (gss) with heterogeneous multimedia streams. *Proceedings of ACM Multimedia*, August 1993.
[10] S. J. Daigle and J. K. Strosnider. Disk scheduling for multimedia data streams. *Proceedings of the IS&T/SPIE*, February 1994.
[11] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Virtual IO: Preemptible disk access. *Proceedings of the 10th ACM Conference on Multimedia*, pages 231–234, December 2002.
[12] Z. Dimitrijevic, R. Rangaswami, and E. Chang. The XTREAM multimedia system. *Proceedings of the IEEE Conference on Multimedia and Expo*, pages 545–548, August 2002.
[13] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Design and implementation of Semi-preemptible IO. *Proceeding of the 2nd Usenix FAST*, pages 145–158, March 2003.
[14] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya. Diskbench. *UCSB Technical Report, http://www.cs.ucsb.edu/∼zoran/papers/db01.pdf*, November 2001.
[15] Z. Dimitrijevic, R. Rangaswami, M. Sang, K. Ramachandran, and E. Chang. UCSB-IO: Linux kernel extensions for QoS disk access. *http://www.cs.ucsb.edu/∼zoran/ucsb-io*, 2003.
[16] D. J. Gemmell, H. M. Vin, D. D. Kandlur, and P. V. Rangan. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, 1995.
[17] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, and L. A. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer Magazine*, pages 40–49, May 1995.
[18] G. Gibson, J. Vitter, and J. Wilkes. Strategic directions in storage io issues in large-scale computing. *ACM Computing Survey*, 28(4):779–93, December 1996.
[19] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. *HPL Technical Report*, February 1991.

[20] J. Korst. Random duplication assignment: An alternative to striping in video servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 219–226, November 1997.

[21] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. *Proceedings of the 1st Usenix FAST*, January 2002.

[22] C. R. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle. Towards higher disk head utilization: Extracting free bandwith from busy disk drives. *Proceedings of the OSDI*, 2000.

[23] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for unix*. *ACM Transactions on Computer Systems 2*, 3:181–197, August 1984.

[24] A. Molano, K. Juvva, and R. Rajkumar. Guaranteeing timing constraints for disk accesses in RT-Mach. *Proceedings of the Real Time Systems Symposium*, 1997.

[25] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of ACM SIGMOD Conference on the Management of Data*, pages 109–116, 1988.

[26] T. Plageman, V. Goebel, P. Halvorsen, and O. J. Anshus. Operating system support for multimedia system. *Computer Communications*, pages 267–289, 2000.

[27] I. A. Pratt. The user-safe device i/o architecture. *University of Cambridge King's College Ph.D. Dissertation*, August 1997.

[28] P. V. Rangan and H. M. Vin. Efficient storage techniques for digital continuous multimedia. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):564–573, August 1993.

[29] R. Rangaswami, Z. Dimitrijevic, E. Chang, and S.-H. G. Chan. Fine-grained device management in an interactive media server. *To appear in IEEE Transactions on Multimedia*, 2003.

[30] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. *Proceedings of the IEEE RTSS*, December 2001.

[31] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. 1991.

[32] J. Schindler and G. R. Ganger. Automated disk drive characterization. *CMU Technical Report CMU-CS-00-176*, December 1999.

[33] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. *Proceedings of the 1st Usenix FAST*, January 2002.

[34] P. J. Shenoy, P. Goyal, S. S. Rao, and H. Vin. Symphony: An integrated multimedia file system. *Proceedings of the Multimedia Computing and Networking (MMCN)*, 1998.

[35] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Proceedings of the ACM Sigmetrics*, June 1998.

[36] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Proceedings of ACM SIGMETRICS Conference*, June 1998.

[37] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison-Wesley, 1998.

[38] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. *Proceedings of the 2nd Usenix FAST*, March 2003.

[39] R. Steinmetz. Multimedia file systems survey: approaches for continuous media disk scheduling. *Computer Communications*, pages 133–44, March 1995.

[40] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin. Application performance in the qlinux multimedia operating system. *ACM Multimedia*, 2000.

[41] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. *Proceedings of Intl. Workshop on Quality of Service (IWQoS'2001)*, June 2001.

[42] J. Wilkes. Data services – from data to containers. *Keynote speech at the 2nd Usenix FAST*, March 2003.

[43] B. L. Worthington, G. Ganger, Y. N. Patt, and J. Wilkes. Online extraction of scsi disk drive parameters. *Proceedings of the ACM Sigmetrics*, pages 146–156, 1995.

[44] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of the ACM Sigmetrics*, pages 241–251, May 1994.

Zoran Dimitrijević received the Dipl.Ing. degree in Electrical Engineering from the School of Electrical Engineering, University of Belgrade, Serbia in 1999. Currently, he is a Ph.D. candidate in the Department of Computer Science at the University of California, Santa Barbara. During 1999-2000, he was a Dean's Fellow at the University of California, Santa Barbara. His research interests include operating systems, storage systems, multimedia applications, parallel computing, and computer architecture.

Raju Rangaswami received the B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, India in 1999. Currently, he is a Ph.D. candidate in the Department of Computer Science at the University of California, Santa Barbara. During 1999-2000, he was a Dean's Fellow at the University of California, Santa Barbara. His research interests include multimedia applications, file systems, operating systems, and storage.