UNIVERSITY OF CALIFORNIA
Santa Barbara

Quality-of-Service Scheduling in Storage Systems

by

Zoran Dimitrijević

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Computer Science

June 2004

The dissertation of Zoran Dimitrijević is approved:

Edward Y. Chang, Chair	Date
Klaus E. Schauser	Date
Elizabeth M. Belding-Royer	Date
Tao Yang	Date

June 2004

June 11, 2004

VITA

November 16, 1974	–	Born in Šabac, Serbia, Yugoslavia.
Fall 1993	–	Graduated from "Šabačka Gimnazija," Šabac, Serbia.
Fall 1997	–	Research Intern (IAESTE exchange program), School of Electrical and Computer Engineering, University of Campinas, Brazil.
1998-1999	–	Research Assistant, School of Electrical Engineering, University of Belgrade, Serbia.
1999	–	Dipl.Ing. in Electrical Engineering, School of Electrical Engineering, University of Belgrade, Serbia.
1999-2004	–	Teaching and Research Assistant, Department of Computer Science, University of California, Santa Barbara.
Summer 2000	–	Research Intern, HP Labs, Palo Alto, California.
Summer 2001	–	Research Intern, Sony 550 Digital Media Ventures, San Francisco, California.
June 2003	–	M.S. in Computer Science, University of California, Santa Barbara.
Summer 2003	–	Teaching Associate, Department of Computer Science, University of California, Santa Barbara.
May 2004-present	–	Software Engineer, Google, Mountain View, California.
June 2004	–	Ph.D. in Computer Science, University of California, Santa Barbara.

PUBLICATIONS

Raju Rangaswami, Zoran Dimitrijević, Kyle Kakligian, Edward Chang, and Yuan-Fang Wang. **The SfinX Video Surveillance System**. *Proceedings of the IEEE ICME*, Taipei, Taiwan, June 2004.

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **Preemptive RAID Scheduling**. UCSB Technical Report, April 2004.

Zoran Dimitrijević, Raju Rangaswami, David Watson, and Anurag Acharya. **Diskbench: User-level Disk Feature Extraction Tool**. UCSB Technical Report, April 2004.

Raju Rangaswami, Zoran Dimitrijević, Edward Chang, and S.-H. Gary Chan. **Fine-grained Device Management in an Interactive Media Server**. *IEEE Transactions on Multimedia*, December 2003.

Zoran Dimitrijević, Gang Wu, and Edward Chang. **SFINX: A Multi-sensor Fusion and Mining System**. *Proceedings of the IEEE Pacific-rim Conference on Multimedia*, Singapore, December 2003.

Zoran Dimitrijević and Raju Rangaswami. **Quality of Service Support for Real-time Storage Systems**. *Proceedings of the International IPSI-2003 Conference*, St. Stefan, Serbia and Montenegro, October 2003.

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **Design and Implementation of Semi-preemptible IO**. *Proceedings of the Second Usenix Conference on File and Storage Technologies*, San Francisco, California, March 2003.

Raju Rangaswami, Zoran Dimitrijević, Edward Chang, and Klaus E. Schauser. **MEMS-based Disk Buffer for Streaming Media Servers**. *Proceedings of* *the 19th IEEE International Conference on Data Engineering*, Bangalore, India, March 2003.

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **Virtual IO: Preemptible Disk Access**. *Proceedings of the 10th ACM Conference on Multimedia*, Juan Les Pins, France, December 2002.

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **The XTREAM Multimedia System**. *Proceedings of the IEEE ICME*, Lausanne, Switzerland, August 2002.

Zoran Dimitrijević, Raju Rangaswami, David Watson, and Anurag Acharya. **User-level SCSI Disk Feature Extraction.** Technical Report, UCSB, July 2001.

Alan Messer, Philippe Bernadat, Guangrui Fu, Deqing Chen, Zoran Dimitrijević, David Jeun Fung Lie, Durga Mannaru, Alma Riska, and Dejan Milojičić. **Susceptibility of Modern Systems and Software to Soft Errors**. HP Labs TR HPL-2001-43, 2001.

Deqing Chen, Alan Messer, Philippe Bernadat, Guangrui Fu, Zoran Dimitrijević, David Jeun Fung Lie, Durga Mannaru, Alma Riska, and Dejan Milojičić. **JVM Susceptibility to Memory Errors**. *Proceedings of Usenix Java[tm] Virtual Machine Research and Technology Symposium*, Monterey, California, April 2001.

Igor Ikodinović, Zoran Dimitrijević, Veljko Milutinović, and Antonio Prete. **GPS-enabled Handheld Cartographic Display Systems: A Survey.** Technical Report, School of Electrical Engineering, University of Belgrade, 2000.

Igor Ikodinović, Zoran Dimitrijević, Davor Magdić, Aleksandar Milenković, Jelica Protić, and Veljko Milutinović. **Limes: A Multiprocessor Simulation Environment for PC Platforms.** Book chapter in *Microprocessor and Multimicroprocessor Systems* by Veljko Milutinović, John Wiley & Sons, Inc., 2000.

Igor Ikodinović, Zoran Dimitrijević, Veljko Milutinović, and Antonio Prete. **Proposing the Architecture for a High-performance GPS-enabled Handheld Cartographic Display System.** Technical Report, School of Electrical Engineering, University of Belgrade, 2000.

Miloš Prvulović, Darko Marinov, Zoran Dimitrijević, and Veljko Milutinović. **Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance.** *Newsletter of Technical Committee on Computer Architecture*, IEEE Computer Society, July 1999.

Miloš Prvulović, Darko Marinov, Zoran Dimitrijević, and Veljko Milutinović. **The Split Spatial/Non-spatial Cache: A Performance and Complexity Evaluation.** *Newsletter of Technical Committee on Computer Architecture*, IEEE Computer Society, July 1999.

Zoran Dimitrijević. **Software Environment for the Simulation of Distributed Shared Memory Multiprocessor Systems**. Engineering Diploma Thesis (in Serbian), School of Electrical Engineering, University of Belgrade, July 1999.

Abstract

Quality-of-Service Scheduling in Storage Systems

by

Zoran Dimitrijević

Emerging video surveillance, large-scale sensor networks, and storage-bound Web applications require large, high-performance, and reliable storage systems with high data-throughput as well as short response times for interactive requests. These conflicting requirements call for quality of service (QoS) support. Currently, the most cost-effective non-volatile storage technology for large-volume data is based on magnetic disks.

In this dissertation, we first study the preemptibility of disk IOs. Allowing higher-priority requests to preempt ongoing disk IOs is of particular benefit to delay-sensitive interactive and real-time systems. We present the design and implementation of Semi-preemptible IO [22], which divides disk IO requests into multiple short-duration disk commands to improve the preemptibility of disk access. We propose methods to allow preemption of each component of a disk access — seek, rotation, and data transfer. We analyze the performance and describe implementation challenges. Particularly, we explain disk profiling algorithms for accurate disk-performance modeling. Our evaluation shows that Semi-preemptible IO can substantially reduce IO waiting time with little loss in disk throughput.

We then investigate the effectiveness of preemptive disk-scheduling algorithms to achieve better quality-of-service (QoS) scheduling. Large storage systems are often implemented using Redundant Arrays of Independent Disks (RAID). We present an architecture for QoS-aware RAID systems [24] that use Semi-preemptible IO for servicing their internal disk IOs. We show *when* and *how* to preempt IOs to improve the overall real-time performance of QoS-aware RAID systems. In order to decide when to preempt an IO, we propose preemptive scheduling methods which aim to maximize the total RAID QoS value. In order to decide how to preempt an IO, we introduce two methods for IO preemptions in RAID systems — JIT-preemption and JIT-migration.

Acknowledgments

After leaving Santa Barbara, life seems a little bit lonely. I have a feeling that this is a good moment to acknowledge some of the people who made my PhD studies less lonely and more fulfilling.

Firstly, I wish to thank my parents, Danica and Živadin, and my Brother Dragan for their everlasting care, support, and understanding. You taught me the most important lessons about life.

Secondly, I wish to thank my teachers for their priceless lectures.

I especially thank my advisor, Professor Edward Y. Chang. Ed, you are an extraordinary person and a great advisor. Thank you so much for your precious advice, numerous brainstorming sessions, and your great enthusiasm for research. In many ways we are very different, and yet I feel that without those differences we would not have been able to achieve such a productive collaboration. I know that I do not respond particularly well to management, which is one more reason more to appreciate your tolerance and understanding.

I wish to thank the other great members of my dissertation committee. Klaus, I have truly enjoyed every moment I had a chance to talk and work with you. You really know how to read my mind and how to motivate me. Thank you very much for everything, especially for teaching me how to think in a scalable way. I definitely hope our paths will cross again sometime soon. Elizabeth, thanks for teaching me about wireless protocols. Tao, thanks for teaching me more about distributed, parallel, and scientific computing.

On my very first day at UCSB, Anurag found me and made me his TA for the operating systems course. Acha, I really enjoyed working with you. Thank you for teaching me more about real systems design and implementation. Now, after nearly five long years, I am very excited and looking forward to continuing our collaboration at Google.

I wish to thank dear Helen Gordon for her invaluable proof-reading and numerous patient comments about my writing. I know that I still make the same old mistakes and often introduce new words from my angloserbian vocabulary. I hope that I will be a good student and remember your lessons whenever I write even the least important write-only technical document.

I wish to thank Marijana Vidojević and Dragan Dimitrijević for their great help in the final proof-reading of this dissertation.

Thirdly, I wish to thank my friends for being with me all these years.

My greatest thanks go to my friend, colleague, and lab-mate Raju Rangaswami. Raju, without you and our long discussions, my time at UCSB wouldn't at all be the same. Thank you for being patient and enthusiastic even during the most off-the-topic discussions. I wish to thank my long-time friend and computer guru — Davor Magdić. Davor, thanks for all those long philosophical discussions in Java Jones and Mojo coffee shops. I wish to thank my old friends for their precious thoughts and the long phone discussions; especially Zoran Kuzmanović, Mirko Ristivojević, Jelena Pješivac-Grbović, Vladan Velisavljević, Goran Gogić, and Aleksandar Milenković.

I wish to thank all my friends from UCSB; especially Abhishek, Aleksandra, Ankur, Arun, Bob, Costin, Chris, David, Doca, Dragan, Gang, Gwen, Hong, Jelena, Josep, King-Shy, Kizo, Kyle, Matz, Max, Mihailo, Mike, Milan, Nadja, Neb, Panda, Prashant, Sezgin, Srini, Tanja, Una, Vladimir, and Yi. My great thanks goes to all Santa Barbarians from former Yugoslavia who made me feel very close home. I know that wherever we end up, we will somehow continue our BBQ tradition.

I have been incredibly lucky to enjoy working in a real academic environment and interacting with friendly people from the UCSB Computer Science Department. I especially wish to thank Teo, Kevin, Pete, Divy, Amr, Rich, Tevfik, Ambuj, Yuan-Fang, Giovanni, John, Subhash and all the other professors and students for their advice and their time. And last, but certainly not least, I wish to thank all staff members for making CS department a pleasant and efficient workplace; especially our dear Mary Jane, Juli, Amanda, and Sandy.

I guess I am feeling lucky that my life is not so lonely. Thank you all!

This work was supported in part by a SONY/UC DiMI grant, an IBM gift, the NSF Career grant IIS-0133802, the UCSB Dean's Fellowship, the UCSB Computer Science TA and Tuition Fellowships, the UCSB Work-study Award, and the NSF CISE Infrastructure grant EIA-0080134.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Emerging applications such as video surveillance, large-scale sensor networks, storage-bound Web applications, and virtual reality require high-capacity, high-bandwidth RAID storage to support high-volume IOs. All these applications typically access large sequential data-segments to achieve high disk throughput. In addition to high-throughput non-interactive traffic, these applications also service a large number of interactive requests, requiring short response time. The deployment of high-bandwidth networks promised by research projects such as OptIPuter[87] will further magnify the access-time bottleneck of a remote RAID store, inevitably making the access-time reduction increasingly important.

What is the worst-case disk-access time, and how can it be mitigated? On an idle disk, the access time is composed of a seek and a rotational delay. However, when the disk is servicing an IO, a new interactive IO requiring short response time must wait at least until after the ongoing IO has been completed. For the applications mentioned earlier, the typical IO sizes are of the order of a few megabytes. For example, while concurrently servicing interactive queries, the

Google File System [35] stores data in 64 MB chunks and video surveillance systems [17, 70] record video segments of several megabytes each. Another example is a virtual-reality flight simulator from the TerraFly project [16], which continuously streams the image data for multiple users from their database of satellite images. Simultaneously, the system must support interactive user operations.

Traditionally, disk IOs have been thought of as non-preemptible operations. Once initiated, they cannot be stopped until completed. Over the years, operating system designers have learned to live with this restriction. However, non-preemptible IOs can be a stumbling block for applications that require short response time. In this dissertation, we propose methods to make disk IOs semi-preemptible, thus providing the operating system with a finer level of control over the disk-drive.

Preemptible disk access is desirable in certain settings. One such domain is that of real-time disk scheduling. Real-time scheduling theoreticians have developed schedulability tests (the test of whether a task set is schedulable such that all deadlines are met) in various settings [46, 48, 51]. In real-time scheduling theory, *blocking*[1], or priority inversion, is defined as the time during which a higher-priority task is prevented from running due to the non-preemptibility of a low-priority task. Blocking degrades schedulability of real-time tasks and is thus undesirable. Making disk IOs preemptible would reduce blocking and improve the schedulability of real-time disk IOs.

For example, suppose that the disk is servicing a long low-priority sequential write when a higher priority read IO arrives. Now, a simple priority-based scheduler will always preempt the long sequential write access (and incur a preemption overhead) regardless of whether the read IO arrives soon after the beginning of the low-priority write or nearly at the end of the long write. How-

[1]In this dissertation, we refer to blocking as the *waiting time.*

ever, preempting a nearly-completed write access may not be profitable. Such a preemption is likely to be counter-productive — not gaining much in the read response time, but incurring preemption overhead. This example shows just one simple scenario where additional mechanisms can lead to performance gains for storage systems.

Thesis Statement

In this dissertation we aim to prove the following thesis: With the low-level disk model, it is feasible to enable effective preemptions of disk IOs and to employ this explicitly-preemptive approach in designing quality-of-service disk-scheduling algorithms.

1.2 Contributions

In summary, the main contributions of this dissertation are:

- **Preemptible disk scheduling.** We investigate the preemptibility of disk access and show that a high-level of preemptibility can be achieved for current disk drives. We design and implement *Semi-preemptible IO* [20, 22] prototype.

- **Preemptive RAID scheduling.** We investigate the effectiveness of preemptive disk-scheduling algorithms to achieve better quality of service in RAID systems [23, 24]. We present an architecture for QoS-aware RAID systems that use Semi-preemptible IO for servicing internal disk IOs. We show *when* and *how* to preempt IOs to improve the overall performance of the RAID systems.

- **Disk profiling.** We implemented Diskbench [25, 29], a tool for low-level disk profiling and modeling. Based on accurate low-level disk models, we investigated admission control methods for real-time streaming applications [21, 69] and Semi-preemptible IO [22, 20].

1.3 Dissertation Outline

In Chapter 2, we present several important classes of real-time applications and classify their data QoS requirements.

In Chapter 3, we explain the architecture of magnetic disks in details. We present algorithms and methods used for low-level disk profiling and modeling.

The main focus of this dissertation is on methods and ideas presented in Chapters 4 and 5.

In Chapter 4 we investigate the preemptibility of disk access and present the design and implementation of Semi-preemptible IO [22].

In Chapter 5 we focus on QoS RAID scheduling. We first extend our Semi-preemptible IO to RAID systems. We then present an approach for preemptive RAID scheduling. Additionally, we present the kernel-level implementation of our QoS extensions for Linux [19, 24, 27].

In Chapter 6, we survey the previous work relevant to disk modeling and profiling, preemptible disk access, and preemptive RAID scheduling.

In Chapter 7, we summarize the main contributions of this dissertation and suggest directions for future work.

Chapter 2

Quality of Service in Storage Systems

In this chapter, we present several classes of storage-bound applications and classify their data QoS requirements. We then explain the underlying technology for accessing current storage systems. Finally, we present two example applications with real-time QoS requirements.

2.1 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.

2.1.1 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 of these systems is the embedded control system 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 [62]. Table 2.1 summarizes several important storage-bound real-time applications.

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 2.1: 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 [50, 70] 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 representation 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 of these applications are high-energy particle research [8] and the SETI (Search for Extraterrestrial Intelligence) project [81].

2.1.2 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 [98, 99].

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 requires 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.

2.2 Storage

The architecture for data storage is usually based on multiple layers of mediums with different price-performance characteristics. Modern processors operate on data in their local registers. In order to efficiently access data in main memory (usually implemented as DRAM modules), designers employ multiple layers of fast caches between the CPU registers and the main memory. These caches are often implemented on the same die as CPUs and provide various levels of access performance. Both the main memory and caches are considered as volatile memory, since they lose content after losing their power supply.

2.2.1 Magnetic Disks

Currently, the main non-volatile storage is based on magnetic disks. For large-volume data, the main storage is usually implemented using redundant arrays of independent disks (RAID) [14, 31]. Disks provide the best price-performance ratio for many storage-bound applications. D. A. Thompson and J. S. Best [95] published an in-depth study about the trends and the future of magnetic data storage. We present a detailed study on the architecture and profiling of magnetic disk drives in Chapter 3.

Backup Storage

For backup storage, current systems use both magnetic tapes and optical disks. Additionally, due to reduced cost of magnetic disk storage [95, 39], many applications can use magnetic disks exclusively to backup their data.

2.2.2 Emerging Technologies

Even after the long reign of Moore's Law, the basic memory hierarchy in computer systems has not changed significantly. At the non-volatile end, magnetic disks have managed to survive as the most cost-effective mass storage medium, and there are no alternative technologies which show promise for replacing them in the next decade [95]. Disk access times are improving at the rate of only 10% per year. For more than a decade they have continued to lag behind the annual disk throughput increase of 40% and capacity increase of 60% [39, 95]. Due to the increasing gap between the improvements in disk bandwidth and disk access times (both seek time and rotational delay), achieving high disk throughput necessitates accessing the disk drive in larger chunks.

Micro-electro-mechanical-systems (MEMS) based storage is an emerging technology that promises to bridge the performance gap between magnetic disks and DRAM [7, 97]. MEMS devices are predicted to be an order of magnitude cheaper than DRAM, while offering an order of magnitude faster access times than disk drives [40, 78]. These devices offer a unique low-cost solution for streaming applications. In our related work [69], we have investigated the possibility of using MEMS storage for buffering and caching streaming multimedia content. We proposed an analytical framework to evaluate the effective use of MEMS devices in a streaming media server. Specifically, we derived analytical models for studying two MEMS configurations, *using MEMS storage as a buffer* between DRAM and disk, and *using MEMS storage as a cache*. Summarizing our findings, MEMS storage can improve the performance of streaming media servers by providing low-access latency and high throughput for accessing streaming data.

2.3 Overview of Disk Management Techniques

In this section, we survey the representative work in disk scheduling, disk admission control, and data placement.

2.3.1 Disk Scheduling

Table 2.2 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 2.2: Disk schedulers.

Worthington et al. [102] survey the scheduling algorithms for traditional, best-effort disk access. FCFS (First-Come-First-Serve) approach schedules 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) [45] 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 [86, 102]. 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, 52, 53]. *Freeblock scheduling* [52, 53] uses rotational prediction to schedule low-priority IOs in the background without affecting other IOs. *Semi-preemptible IO* [22] 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 [62].

- *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 [86], bubble-up [9], and bubble-up 2D [11].

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) [12], 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 [84] 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 [83] multimedia file system supports diverse application classes that access data with heterogeneous characteristics using Cello framework.

Clockwise [5] 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 non-preemptible EDF scheduling.

User-safe disks [64] 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 [6] 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.

2.3.2 Disk Admission Control

Admission Control is a mechanism for deciding if a particular request can be admitted into the system or not. We classify the disk admission control approaches into 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. 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 [25, 76, 101] to predict 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 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 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 while designing Xtream [21] and compared their performance with the deterministic approach.

2.3.3 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) [57], which proposed the notion of cylinder groups to place related data closer on the disk surface. Log-structured placement [72] 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 [68], constrained block allocation [65], track extents [77], 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 [10], 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 [49], 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 [60] to detect and correct errors on the disk surface.

2.4 Example Applications

In this section, we present our two testbed applications for QoS disk schedulers: the Xtream Multimedia System [21] and the SfinX video surveillance system [30, 70].

2.4.1 The Xtream Multimedia System

Xtream [21] is a video streaming system that supports guaranteed-rate IO for both write (for example, recording by a surveillance camera) and read streams (for example, audio or video playback).

The Xtream service model consists of one or more clients connecting to a server to request multimedia data stored on the server's disk drive. The client could be desktop requesting a video-on-demand service, a surveillance camera recording video, or simply a web-browser requesting html data. In this model, we assume that no bottleneck exists in the interconnection network between the server and the clients.

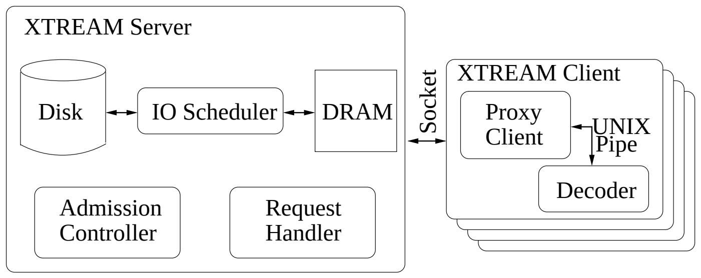


Figure 2.1: Xtream system architecture.

As shown in Figure 2.1, the Xtream clients include a *proxy* component which connects to the server on their behalf and also performs data buffering to mask network bandwidth variations. The client is designed such that it can operate with any encoder or decoder application that supports a UNIX pipe-like interface.

The Xtream server runs entirely in user space. Its two functions are to decide if it can admit a new stream and to maintain the QoS for existing streams. The three requirements of the Xtream server — *high throughput*, *low initial latency*, and *guaranteed IO* — are addressed by the three components within Xtream. The *IO Scheduler* uses the time cycle model [66] for servicing disk IOs; the *Request Handler* preempts the IO scheduler for servicing new requests promptly; the *Admission Controller* guarantees QoS for soft-real-time streams while ensuring that non-real-time data retrievals are not starved. Xtream uses a *Disk Profiler* [28] to obtain a realistic model required to predict disk performance and provide real-time streaming guarantees.

IO Scheduler

Xtream adopts a single-thread IO paradigm wherein the IO scheduler performs all disk IOs inside a single thread. It uses the time cycle model [66], which divides time into basic units called time cycles (T). In each cycle, Xtream services exactly one disk IO per stream. The size of the IO is chosen so that the display buffer does not underflow before the next IO for the same stream is performed. Unlike that in the original time cycle model, the scheduling order for stream IOs may vary between cycles. Using a *double buffer* for each stream, which can sustain playback for as much as two time cycles, makes the initial latency bound independent of the number of streams being serviced and reduces it to the duration of a single disk IO (see Section 2.4.1). In contrast, the simple multi-threaded approach services each stream using a dedicated thread. Four advantages of the single-thread IO paradigm used in Xtream are:

Deterministic execution: Since a single thread is performing all disk IOs, the IO schedule is deterministic, which enables soft-real-time guarantees. In the multi-threaded IO model, the OS scheduling determines the IO order, and we cannot predict when any IO will be serviced.

Controlled IO variability: IO variability is defined as the fluctuations in time between successive IOs for the same stream. Large IO variability requires more in-memory buffering and increases the system cost. The single-thread model controls IO variability by performing at least one IO for each stream in each cycle. This approach is not possible in the simple multi-threaded design.

Contiguous IOs: Since the operating system might break up a large IO request into multiple small ones, an IO operation for a single stream might incur multiple disk accesses simply due to thread-switching in a multi-threaded design. However, in the single-threaded design, the operating system cannot interleave IOs for different streams, which ensures that an IO operation to the disk is indeed sequential.

Fairness: In the single-thread IO model, we can incorporate service for non-real-time requests simply by reserving a fixed portion of each cycle for non-real-time jobs.

A similar approach is employed in the design of the Google File System [35].

Request Handler

When a new request arrives in the Xtream system, the request handler module is invoked to service it. The request handler, in turn, invokes the admission controller to determine if the new request can be serviced. If it can, the request handler preempts the IO scheduler as soon as it finishes its current IO job. It then adds the request to the head of the IO service queue, which is used by the IO scheduler to determine the service order. We can make the following observations for this approach:

- The initial latency does not depend on the number of streams in the system. It is simply the sum of the maximum time required to service a single IO for any existing stream and the time required to perform the initial IO for filling up the buffer of the new stream. This approach comes at the cost of double buffering, which frees the IO scheduler from having to maintain the same IO order between time cycles. If required, the initial latency can be further decreased by using preemptible disk access methods proposed in [22].

- The double buffering scheme also frees IO scheduler from using *fixed-stretch* [10], in which the IO for a stream must be started exactly at the same time relative to the beginning of each cycle. In a system which services both real-time streams and non-real-time requests, a fixed-stretch IO restriction might lead to under-utilization of disk bandwidth because of variability in both the number of streams and their bit-rates. In contrast, the double buffering scheme can tolerate these phenomena easily.

The Xtream admission controller is explained in Section 3.3.2 in Chapter 3.

2.4.2 The SfinX Video Surveillance System

In a surveillance system, video signals are generated by multiple cameras with or without spatially and temporally overlapping coverage. These signals need to be compressed, fused, stored, indexed, and then summarized as semantic events to allow efficient and effective querying and mining.

The target application that we intend to support would be capable of not only viewing video streams in real-time, but also supporting scan operations (like rew, ffwd, slow-motion, etc.) on the video streams. In addition, it would also support video analysis in the form of database queries. A query, for instance, can be worded like this: "select object = *'vehicles'* where event = *'circling'* and location = *'parking lots'* and time = *'since 9pm last night'*." Another example-query might be "select object = *'vehicle A'* where event = *'*'* and location = *'*'* and time = *'since 9pm last night'*."

In this section, we introduce the hardware and software architecture of the SfinX video surveillance system [30, 70]. Figure 2.2 depicts a typical hardware architecture of SfinX. Cameras are mounted at the edges of a sensor network to collect signals (shown on the upper-right of the figure). When activity is detected, signals are compressed and transfered to a server (lower-left of the figure). The server fuses multi-sensor data and constructs spatio-temporal descriptors to depict the captured activities. The server indexes and stores video signals with their meta-data on RAID storage (lower-right of the figure). Users of the system (upper-left of the figure) are alerted to unusual events and they can perform online queries to retrieve and inspect video-clips of interest.

Figure 2.3 depicts the software architecture of SfinX. Video signals are captured by the *video capture* module. At the same time *tracking* algorithms are employed to track objects in the captured video streams and the video stream is

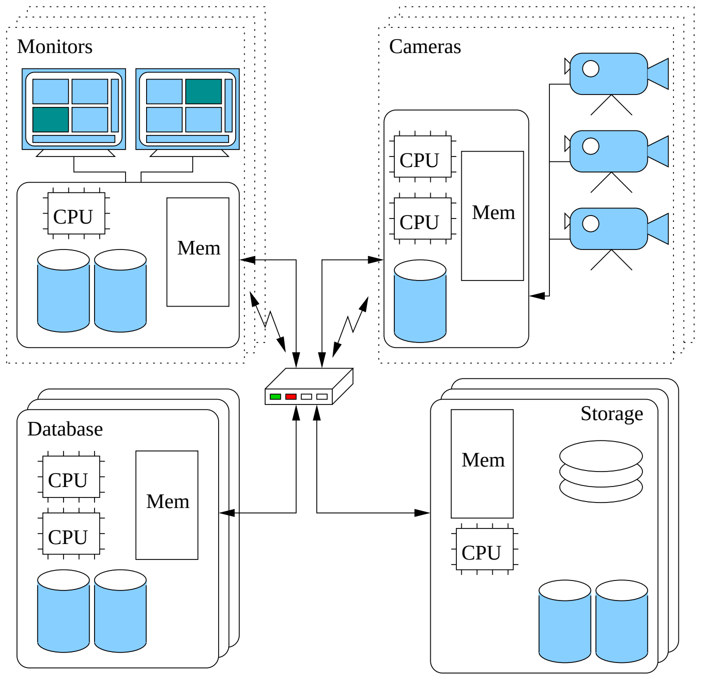


Figure 2.2: SfinX hardware architecture.

encoded and sent off to be stored onto Xtream [21], a *real-time streaming storage* system. To aid in effective tracking of occluded objects and to obtain consensus on object position in ambiguous situations, a *multi-tracker* module combines the tracking information from different cameras which cover a common physical area and feeds back global information to the individual camera tracking modules. Multiple multi-trackers also exist, which track objects in physically disjointed areas.

Using the global tracking information and object representation created by the multi-tracker modules, the *fusion and representation* module maps the trajectory of each object as it moves through the entire scene. The representation module represents the trajectory of each object using *sequence data representation* [103]. This information is stored in the *events database* for future reference.

The user-interface consists of two distinct components. First, using the *real-*

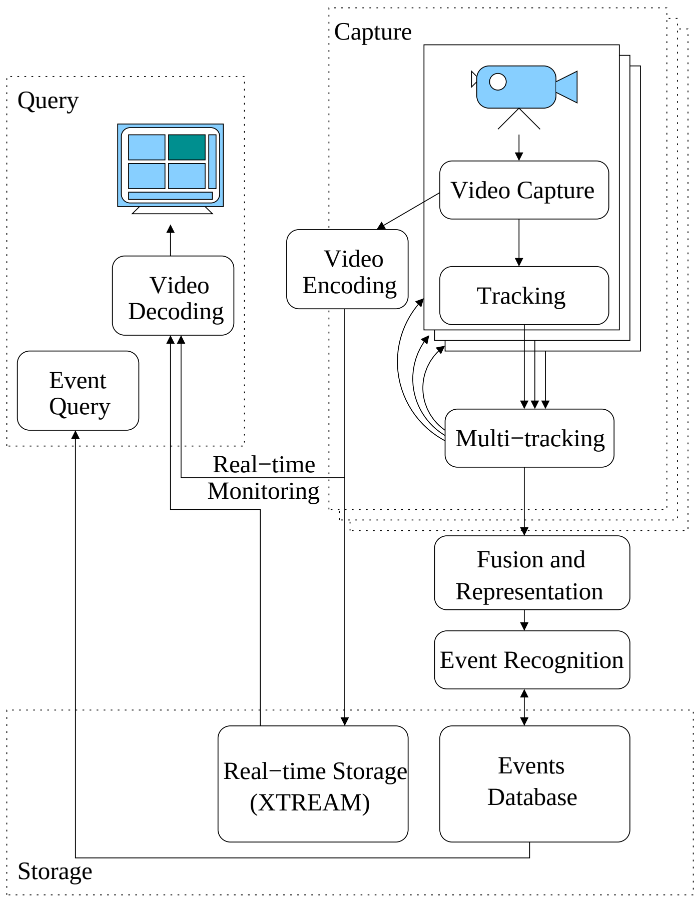


Figure 2.3: SfinX software architecture.

time monitoring component, a user can view live camera feeds and interact with them in order to scan through the video streams. This helps the user to immediately track objects by moving through any stream at will. Second, the viewer can also analyze the stored video streams by performing database queries. Controlling the query semantics, the user can get detailed information from the database.

System Components

We now present the major components of the SfinX system. We analyze each component of the software architecture and describe the interaction between various components.

Video Capture

For capturing video streams, we propose using multiple, cheap, off-the-shelf video cameras for each physical location requiring surveillance. These cameras share data between themselves to perform their functions with greater accuracy. Similar to a previous study [105], we use a single high-end camera per location possessing zoom and motion capabilities for tracking objects or humans in close-up. The most important problem in capturing useful information from a scene is that of camera callibration [32]. Ideally, this must be an automatic process, that maps the camera co-ordinates to co-ordinates in the physical location. In addition, the close-tracking high-end camera must be perfectly callibrated at all times in spite of zoom and motion operations.

Encoding and Real-time Storage

The video stream obtained from each camera is encoded using standard encoding algorithms like H.263, MPEG1, or MPEG4. Each stream is then stored using a real-time storage system like Xtream [21] for future viewing purposes. The storage system provides real-time stream retrieval and supports scan operations like rew, ffwd, and slow-motion. The main sub-components of the real-time storage component are: *data placement, admission control, disk scheduling*, and *backup manager*.

The *data placement module* makes decisions about data placements using global knowledge about all storage nodes and the QoS requirements for each IO request. The placement decisions can be short-term (for example, for each database update) or long-term (for example, the placement for the next one hour of a particular video stream). The data placement module consults the *admission control* module to check if a particular placement satisfies the real-time access requirements. It also manages data redundancy for reliability.

The *disk scheduling module* is responsible for local disk scheduling and buffer management on each storage node. SfinX uses time cycle scheduling [67] for guaranteed-rate real-time streams. The basic time cycle model is extended to support non real-time IO requests with different priorities (high-priority, best-effort, and background IO). To achieve short latency for high-priority requests while maintaining high disk throughput, SfinX uses preemptible disk scheduling [22].

The *backup manager* module is responsible for deciding which data to copy from main storage to backup and when. The volume of video data in SfinX is large, of the order of TB/day. Since the main SfinX storage is designed to be reliable, backup is mainly used to filter its data and to keep only the important data in main storage.

Tracking and Multi-tracking

Tracking refers to the process of following and mapping the trajectory of a moving object in the scene. Moving objects in each camera feed are tracked using real-time tracking algorithms [88, 17]. Using the information about motion trajectory, the high-end camera may be used to follow the moving object in close-up.

Multi-tracking combines the tracking information from different cameras monitoring the same physical location. It uses the global knowledge thus obtained to aid in tracking objects which are occluded for individual cameras. It can also use this global information to reach consensus when individual tracking modules disagree on object positions. The multi-tracker feeds this global information back to the individual camera tracking modules. Each physical location employs a multi-tracker to combine the information from individual cameras at that location.

Fusion and Representation

Using the global tracking information and object representation created by the multi-tracker modules, the *fusion and representation* module maps the trajectory of each object as it moves through the entire scene. The representation module represents the trajectory of each object using *sequence data representation* [103]. To arrive at a reasonable representation, the trajectory of each object is smoothed using Kalman filters [47] to obtain a piecewise linear trajectory. This piecewise-linear trajectory is then represented using sequence data representation.

Event Recognition

Event recognition translates to the problem of recognizing spatio-temporal patterns under extreme statistical constraints. It deals with mapping motion patterns to semantics (for example, benign and suspicious events). Recognizing rare events comes up against two mathematical challenges. First, the number of training instances that can be collected for modeling rare events is typically very small. Let N denote the number of training instances, and D the dimensionality of data. Traditional statistical models such as the Hidden Markov Model (HMM) cannot work effectively under the $N < D$ constraint. Furthermore, positive events (i.e., the sought-for hazardous events) are always significantly outnumbered by negative events in the training data. In such an imbalanced set of training data, the class boundary tends to skew toward the minority class and hence results in a high incidence of false negatives.

Querying and Monitoring

Monitoring allows retrieving videos efficiently via different access paths. Video data can be accessed via a variety of attributes, e.g., by objects, temporal attributes, spatial attributes, pattern similarity, and by any combination of the above. We support retrieval of videos with trajectories that match a given SQL query definition. At the same time, the storage system must also support viewing of stored videos. The infrastructure also supports real-time monitoring of camera streams. However, simultaneously supporting high-throughput writes (recording encoded videos) and quick response reads (retrieving video segments relevant to a query) presents conflicting design requirements for memory management, disk scheduling, and data placement policies at the storage system.

2.5 Summary

In this chapter, we have first presented a brief overview of QoS requirements for several classes of storage-bound applications. We then briefly surveyed data-management techniques for accessing storage systems — particularly disk scheduling, admission control, and data placement techniques. Finally, we presented two example applications with real-time QoS requirements.

Chapter 3

Disk Modeling and Profiling

In this chapter, we explain the architecture and performance of magnetic disks in detail. We present several approaches for disk modeling and explain methods for automatic low-level disk profiling. The disk profiler automatically extracts low-level block mappings, rotational factors, seek curves, and buffer parameters.

3.1 Introduction

The performance gap between hard drives and CPU-memory subsystems is steadily increasing. In order to bridge this gap, operating systems can use advanced disk management strategies [42, 45, 53, 93, 102], which often require detailed knowledge about disk parameters, such as zoning, bad-sector locations, and disk latency. Some of these parameters can be obtained from disk manufacturers. However, the information they provide can be imprecise and static, or simply unavailable. For instance, disk vendors usually give out only the maximum, minimum, and average data transfer rates and seek times. In addition, some dynamic information such as the locations of bad sectors cannot be known prior to actual use. As a consequence, the effectiveness of most disk management strategies can be compromised due to disk model imperfections.

For optimal disk performance, it is necessary to tune disk access to the requirements of the application. For example, a multimedia streaming server must predict disk performance to maintain real-time streaming without underutilizing the disk. However, disk abstractions (for example, SCSI and IDE interfaces) hide low-level device characteristics from the operating system and virtualize access to the device in the form of logical disk blocks. Such device abstractions make the task of tuning disk operation to match application requirements (and thus improving IO efficiency) difficult. In this chapter we present Diskbench, a tool for disk profiling. Diskbench consists of two applications, Scsibench and Idextract. Scsibench [29] runs in user space on Linux systems and accesses SCSI disks through the Linux SCSI generic interface [38]. Scsibench uses interrogative and empirical methods for feature extraction in a manner similar to the previous work done in disk profiling [1, 101, 76, 91]. Idextract uses Linux raw disk access and empirical methods to extract features from any disk-like device (the approach used by Patterson et al. [91]). Scsibench is open source and available for download [26].

Using Diskbench, we can obtain many low-level disk features including 1) rotational time, 2) seek curve, 3) track and cylinder skew times, 4) caching and prefetching techniques, and 5) logical-to-physical block mappings. Diskbench also extracts several high-level disk features useful for real-time disk schedulers. In this chapter we present two important high-level features: optimal chunk size for sequential disk access and admission control curves for cycle-based real-time disk schedulers. In addition, we show that the access time (including seek time and rotational delay) between two disk accesses can be predicted with high accuracy using a disk model obtained through Scsibench profiling. Scsibench can additionally perform a trace-driven execution with accurate timing mechanisms (for example, SCSI read, write, seek, enable/disable cache, etc.). Using knowledge about disk features and trace support, system or application programmers can obtain information about the precise distribution of time spent by the disk performing various operations. Bottlenecks can thus be identified, and scheduling can be adjusted accordingly to utilize the disk more efficiently. We successfully used Diskbench during the design and implementation of three real-time storage systems: Xtream [21], SfinX [30, 70], and Semi-preemptible IO [20, 22].

The Xtream multimedia system [21] provides real-time video streaming capability to multiple clients simultaneously. For video playback, the disk management must guarantee that all IOs meet their real-time constraints. If the system does not have information about low-level disk features, it must assume the worst-case IO time or use statistical methods. These pessimistic and statistical estimates of disk drive performance lead to sub-optimal performance of the entire system. In contrast, Xtream uses Diskbench to obtain the required disk features for making accurate admission control decisions.

The SfinX video surveillance system [30, 70, 103] is an ongoing project at UCSB, which uses Diskbench for its storage subsystem. The SfinX' storage subsystem needs to support real-time streaming (the write streams from surveillance cameras as well as the read streams for surveillance monitoring and data mining), various high-priority IOs for interactive operations, and a high volume of traditional best-effort IOs.

Semi-preemptible IO [20, 22] is an abstraction for disk IO, which provides preemptible disk access with little loss in disk throughput. Semi-preemptible IO relies on accurate disk-access predictions. The implementation of Semi-preemptible IO was made feasible due to Scsibench, which extracts essential disk information for accurate disk-performance modeling.

3.2 Disk Architecture

Before we get into the details of disk features that are of interest when designing high-performance real-time systems, we provide a brief overview of disk architecture. The main components of a typical disk drive are:

- One or more *disk platters* rotating in lockstep fashion on a shared *spindle*,
- A set of *read/write heads* residing on a shared arm moved by an *actuator*,
- *Disk logic*, including the disk controller, and
- *Cache/buffer memory* with embedded replacement and scheduling algorithms.

The data on the disk drive is logically organized into disk *blocks* (the minimum unit of disk access). Typically, a block corresponds to one disk *sector*. The set of sectors that are on the same magnetic surface and at the same distance from the central spindle form a *track*. The set of tracks at the same distance from the spindle form a *cylinder*. Meta-data such as error detection and correction data are stored in between regular sectors. Sectors can be used to store data for a logical block, to reserve space for future bad sector re-mappings (spare sectors), or to store disk meta-data. They can also be marked as "bad" if they are located on the damaged magnetic surface.

The *storage density* (amount of data that can be stored per square inch) is constant for magnetic surfaces (media) used in disks today. Since the outer disk tracks are longer, they can store more data than the inner ones. Hence, modern disks do not have a constant number of sectors per track. Disks divide cylinders into multiple *disk zones*, each zone having a constant number of sectors per track (and hence having its own performance characteristics).

The *rotational speed* of the disk is constant (with small random variations). Since the track size varies from zone to zone, each disk zone has a different *raw* *bandwidth* (data transfer rate from the disk magnetic media to the internal disk logic). The outer zones have a significantly larger raw disk bandwidth than the inner ones.

When the disk head switches from one track to the next, some time is spent in positioning the disk head to the center of the next track. If the two adjacent tracks are on the same cylinder, this time is referred to as the *track switch time*. If the tracks are on different cylinders, then it is referred to as *cylinder switch time*. In order to optimize the disk for sequential access, disk sectors are organized so that the starting sectors on two adjacent tracks are skewed. This skew compensates for the track or cylinder switch time. It is referred to as *track skew* and *cylinder skew* for track and cylinder switches respectively.

The *seek time* is the time that the disk arm needs in order to move from its current position to the destination cylinder. In the first stage of the seek operation, the arm accelerates at a constant rate. This is followed by a period of constant maximum velocity. In the next stage, the arm slows down with constant deceleration. The final stage of the seek is the settle time, which is needed to position the disk head exactly at the center of the destination track. Since the disk seek mainly depends on the characteristics of the disk arm and its actuator, the seek time curve does not depend on the starting and destination cylinders. It depends only on the seek distance (in cylinders).

The disk magnetic surfaces inevitably contain defects because the process of making perfect surfaces would be too expensive. Hence, disk low-level format marks bad sectors and skips them during logical block numbering. Additionally, some disk sectors are reserved as spare, to enable the disk to re-map bad sectors that occur during its lifetime. The algorithm for spare sector allocation differs from disk to disk. In order to accurately model the disk for intelligent data placement, scheduling, or even simple seek curve extraction, a system needs detailed mapping between the physical sectors and the logical blocks. In addition to mapping, a system must be able to query the disk about re-mapped blocks.

Re-mapping occurs when a disk detects a new bad sector.

The *disk cache* is divided into a number of *cache segments*. Each cache segment can either be allocated to a single sequential access stream or can be further split into blocks for independent allocation. In our study, the cache parameters of interest are segment size, number of cache segments, cache replacement policies, modeling prefetching algorithms, and write buffer organization. The disk uses prefetching algorithms to improve the performance of sequential reads. The write buffer is used to delay the actual writing of data to the disk media and to enable the disk to re-schedule write IOs (hence optimizing throughput). The buffer is also used to optimize sequential write access.

3.3 Disk Profiling

In this section, we present methods for extracting certain disk features using a combination of interrogative and empirical methods. Interrogative methods use inquiry SCSI commands [43, 79] to get required information from the disk firmware. Empirical methods measure completion times for various disk access patterns, and profile the disk based on these measurements.

3.3.1 Low-level Disk Features

We now present the methods Diskbench [28] uses to extract low-level disk features. In some of our extraction methods we assume the ability to force access to the disk media for read or write requests (hence, avoiding the disk caching and buffering). Most modern disks allow turning off the write buffer. In the case of SCSI disks, this can be done by turning off the disk buffers, or by setting the “force media access bit” in a SCSI command [79]. In the case of IDE disks, this can often be done using their OS drivers.

Rotational Time

Since current disk drives operate with a constant rotational speed,[1] whenever the interrogative SCSI command for obtaining rotational period (T_{rot}) is supported by the disk, it will return the correct value. In the absence of the interrogative command, we can also use the following empirical method (described in Worthington et al. [101]) to obtain T_{rot}. First, we ensure that read (or write) commands access the disk media (by turning off the disk caching mechanisms). Next, we perform n successive disk accesses to the same block, and measure the access completion times. The absolute completion time for each disk access is

$$T_i = T_{end\_reading} + T_{transfer} + T_{OS\_delay_i}. \tag{3.1}$$

$T_{end\_reading}$ is the absolute time immediately after the disk reads the block from the disk media. $T_{transfer}$ is the transfer time needed to transfer data over the IO bus. $T_{OS\_delay_i}$ is the time between the moment when the OS receives data over the IO bus, and the moment when the data is transfered to the user level Diskbench process. Since the disk needs to wait for one full disk rotation for each successive disk block access, the time between the two accesses can be expressed as

$$T_{i+1} - T_i = T_{rot} + (T_{OS\_delay_{i+1}} - T_{OS\_delay_i}); \tag{3.2}$$

$$T_{n+1} - T_1 = n \times T_{rot} + (T_{OS\_delay_{n+1}} - T_{OS\_delay_1}). \tag{3.3}$$

The rotational period for current disks is much longer than the OS delay and other IO overhead (not including the seek and rotational times). Thus, we can measure the rotational period as

$$T_{rot\_measured} = T_{rot} + \frac{\Delta T_{OS\_delay_{n+1,1}}}{n} = \frac{T_{n+1} - T_1}{n}. \tag{3.4}$$

For large n, the error term ($\frac{\Delta T_{OS\_delay_{n+1,1}}}{n}$) is reduced in comparison to T_{rot}.

[1]To support the low-power mode, disks may occasionaly spin at a slower rate or stop spinning completely. However, when servicing disk requests, current disks operate with a constant rotational speed.

OS Delay Variations

In order to estimate variations in operating system delay for IO requests, we use the following method. First, we turn off all disk caching and disk buffering. Then, we read the same block in successive disk rotations (as in the empirical method for extracting T_{rot}). We measure completion times for each read request. Assuming that the rotational period (T_{rot}) is constant, variations in $T_i - T_{i-1}$ from Equation 3.2 enable us to estimate the distribution of $\Delta T_{OS\_delay_{i,i-1}} = T_{OS\_delay_i} - T_{OS\_delay_{i-1}}$. Thus, by measuring variations in $T_{i+1} - T_i$ from Equation 3.2, we can estimate variations in the operating system delay.

Mapping from Logical to Physical Block Address

Most current SCSI disks implement SCSI commands for address translation (Send/Receive Diagnostic Command [79]) which can be used to extract disk mapping. However, in the case of older SCSI disks, or for disks where address translation commands are not supported (for example, IDE disks), empirical methods are necessary.

Interrogative Mapping

For interrogative mapping, we use an algorithm based on the approach described in Worthington et al. [101]. Using the interrogative method, a single address translation typically requires less than one millisecond. But, since the number of logical blocks is large, it is inefficient to map each logical block. Fortunately, modern disks are optimized for sequential access of logical blocks.[2] Due to this, logical blocks on a track are generally placed sequentially. Thus, we can extract highly accurate mapping information by translating just one ad-

[2]Additionally, most disks use the skipping method to skip bad sectors during low-level disk format, instead of re-mapping them.

dress per track, except when anomalies are detected (tracks with bad or spare blocks).

Since the number of re-mapped blocks is small compared to the total number of blocks, we propose the following two data structures to store the obtained mapping information. In the first data structure, we store the mapping that existed immediately after the low-level disk format. Since there are no re-mapped blocks, we simply store the starting logical block number and the track size (in blocks) for each track. The second data structure is used to store information about the re-mapped blocks. Thus we only need to update the second data structure periodically, using the inquiry SCSI command which returns the list of current bad-block locations.

Figure 3.1 presents a simplified algorithm for the interrogative extraction employed in Diskbench. Using the SCSI command for physical-to-logical address translation, we extract the LBA for the first sector (sector zero) of each track. If sector zero is marked as bad, we continue performing address translation for subsequent sectors until we obtain a valid logical block number. When we come across a cylinder in which the number of sectors that lack a valid LBA (bad or spare blocks) is above a fixed threshold ($K_{threshold}$), we mark that cylinder as logically bad.[3] We do not use these cylinders in our seek curve extraction method. Tracks which have a substantial number of blocks without a valid LBA are usually the ones containing mostly spare sectors.

[3]These cylinders are not necessarily bad, since they usualy just contain a large number of spare blocks. However, our disk profiler does not use them for the sake of simplicity.

Procedure: *Interrogative Mapping*

- **Variables:**

  1. *cyl_num* : Total number of cylinders
  2. *track_per_cyl* : Number of tracks per cylinder
  3. i : Cylinder number
  4. j : Track number
  5. *cyl_info[i]* : data structure for cylinder i info
  6. *cyl_info[i].logstart[j]* : Starting LBA for track j on cylinder i
  7. *cyl_info[i].logsize[j]* : Track j size in blocks

- **Execution:**

```
1. for $i = 0$ to cyl_num do
2.     for $j = 0$ to track_per_cyl do
3.         for $k = 0$ to $K_{threshold}$ do
4.             cyl_info[i].logstart[j] = phys_to_log(i,j,k)
5.             if valid(cyl_info[i].logstart[j]) then break
6.         if not_valid(cyl_info[i].logstart[j]) then
7.             mark track as bad.
8. sort by cyl_info[i].logstart[j]
9. calculate cyl_info[i].logsize[j]
```

Figure 3.1: Interrogative method for disk mapping.

Empirical Mapping

For disks that do not support address translation, the profiler needs to perform empirical experiments in order to extract the mapping. We employ an empirical method that is similar to the approach presented in Patterson et al. [91]. We additionally use the first derivative of access times for easier automatic mapping and various heuristics to prune the mapping errors near track boundaries.

In the first step, we measure the time delay in reading a pair of blocks from the disk. We repeat this measurement for a number of block pairs, always keeping the position fixed for the first block in the pair. In successive experiments, we linearly increase the position of the second block in a pair. Using this method, our tool extracts accurate positions of track and cylinder boundaries. Time $T(i)$ defined in Equation 3.5 is the completion time measured at the moment when the user process receives data for logical block address i. (The variables on the right side of Equation 3.5 are defined in Section 3.3.1.)

$$T(i) = T_{end\_reading} + T_{transfer} + T_{OS\_delay}. \tag{3.5}$$

The access time ($T_a(x, 0) = T_{end\_reading}(x) - T_{end\_reading}(0)$) is the time needed to access block x after accessing block 0. It includes both seek time and rotational delay, but does not include transfer time and OS delay. Equation 3.7 presents the first derivative of $\Delta T(x, 0)$ defined in Equation 3.6.

$$\begin{aligned} \Delta T(x, 0) &= T(x) - T(0); \\ \Delta T(x-1, 0) &= T_a(x-1, 0) + (T_{OS\_delay_{x-1}} - T_{OS\_delay_0}); \\ \Delta T(x, 0) &= T_a(x, 0) + (T_{OS\_delay_x} - T_{OS\_delay_{0'}}). \end{aligned} \tag{3.6}$$

$$\begin{aligned} \Delta &= \Delta T(x, 0) - \Delta T(x-1, 0); \\ \Delta &= T_a(x, 0) - T_a(x-1, 0) + (\Delta T_{OS\_delay_{x,0'}} - \Delta T_{OS\_delay_{x-1,0}}). \end{aligned} \tag{3.7}$$

Whenever the variations in the OS delay are small, Δ is small and proportional to $T_{rot}/track_{size}$ for the case when both block $x-1$ and block x are on the same track. Whenever the disk can perform the access to block $x-1$ without the additional rotational delay after the seek, and the access to block x with the delay of a full T_{rot}, Δ is proportional to $T_{rot}/track_{size} \pm T_{rot}$. (The sign of Δ depends on T_{OS_delay} since it might happen that we occasionally need less time to access block x.) Whenever blocks x and $x-1$ reside on different tracks, or cylinders, Δ is proportional to the track or cylinder skew time (T_{skew}). Since the cylinder skew is usually larger than the track skew, we can use the positions of block x (when Diskbench experience skew times) to find out accurate track and cylinder boundaries. We define the normalized first derivative in Equation 3.8. This way we eliminate the $\pm T_{rot}$ factor from Equation 3.7, which helps Diskbench to automatically extract accurate disk mapping.

$$norm(\Delta) = (\Delta T(x, 0') - \Delta T(x-1, 0)) \bmod T_{rot}. \tag{3.8}$$

The $norm(\Delta)$ is useful for automatic extraction only if the OS delay is small compared to the skew times. Since the OS delay is a random variable, we perform several measurements whenever $|\Delta|$ is greater than a specific threshold (for example, $0.02 \times T_{rot}$) and stop the measurement when the difference between two consecutive Δ's is less than a specific threshold (for example, 5%).

Seek Curves

Seek time is the time that the disk head requires to move from the current to the destination cylinder. We implement two methods for seek curve extraction. The first method uses the SCSI seek command to move (seek) to a destination cylinder. The second one measures the minimum time delay between reading a single block on the source cylinder and reading a single block on the destination cylinder to obtain the seek time. In order to find the minimum time, we can measure the time between reading a fixed block on the source cylinder, and reading all blocks on one track in the destination cylinder. The seek time is the minimum of the measured times. Since LBAs increase linearly on a track, we use an efficient binary-search method in order to find the minimum access time (hence, the seek time). $T_{seek}(x, y)$ returns seek time (in μs) between logical blocks x and y. This function is symmetrical, i.e., $T_{seek}(x, y) = T_{seek}(y, x)$. The seek time obtained using this method also includes the settling time for the disk head.

Disk Buffer/Cache Parameters

Most disk drives are equipped with a read cache. The read cache improves the disk performance by allowing for data prefetching (for sequential access) and by allowing frequently used disk blocks to reside in the buffer (eliminating disk seeks).[4] We now present methods to extract the cache segment size, the number of segments, and the cache segment replacement policy.

Read Cache Segment Size

The method for extracting the cache segment size consists of three steps. First we read a few sequential disk blocks from a specific disk location. Next we wait for a long enough period of time (several disk rotations) to allow the disk to fill up the cache segment with prefetched data. Finally we read consecutive disk blocks occurring immediately after the first read and measure the completion time. If the block is in the cache, the completion time includes only a block transfer time (from the cache to the OS through the IO bus) and a random T_{OS_delay}. If the block is not in the cache, the completion time also includes seek time and rotational delay, as well as data transfer time. The seek time

[4]Since the disk read cache is small compared to the OS cache, disks usually optimize their cache algorithms to support efficient access to multiple sequential streams via prefetching methods.

and rotational delay are the dominant factors in this IO completion time. We can thus detect the size of a cache segment by detecting when completion times include long mechanical seek operations.

Number of Cache Segments

In order to extract the number of disk cache segments, we need to be able to clear the cache. We “clear” the cache by performing a large number of random sequential reads for different logical blocks, which effectively clears the cache by polluting it. In this extraction method we linearly increase the number of sequential streams accessing the disk. In each iteration for each stream we read a few blocks (from locations which are not used during the cache pollution phase). We then wait a sufficient amount of time so that the disk can fill the cache segment with the prefetched data. After this step, the disk cache allocates one cache segment for each stream.

We perform read requests for all streams and measure IO completion times. If the completion times are smaller than a specific threshold, we assume that all blocks were in the cache, and that the number of cache segments is greater than or equal to the number of streams in this iteration. When we detect that one read request requires an amount of time exceeding the threshold, we deduce that the number of cache segments is equal to the number of streams in the previous iteration. We repeat the entire experiment to confirm that the excessive time is not caused by a large random OS delay.

By changing the access pattern in the previous experiment and taking into the account which streams are not serviced from the cache, Diskbench can deduce the policy used for the cache segment replacement.

Write Buffer Parameters

Most disk drives are equipped with a write buffer to improve the disk write performance. Whenever the disk has sufficient space in the write buffer, the write command is completed as soon as data is transferred to the disk's write buffer.[5] The disk writes this data to the disk magnetic surface at some later time, aiming to improve the disk write performance.

Figure 3.2 depicts an empirical method for extracting the write buffer size. Between iterations, we allow the disk to purge the contents of the write buffer to the disk media. Before issuing the write IO, we also seek to a cylinder far away from the write request's destination. When the write request size is smaller than the write buffer, we expect that the write completion times will increase linearly, proportional to the throughput of the IO bus. When the write request size is greater than the write buffer, the completion time will incur seek and rotational delays. Diskbench can detect this using simple heuristics. Using this method we can also measure the throughput of the IO bus.

3.3.2 High-level Disk Features

Most real-time schedulers rely on simple disk models to reduce the problem complexity. In this section, we present several high-level disk features that are used for data-placement algorithms [68], rotationally-aware schedulers [42, 45, 53], preemptible schedulers [20, 22, 23, 24], and admission control methods [21].

Disk Zones

Using extracted disk mapping, Diskbench implements methods to extract zoning information, including 1) precise zone boundaries, 2) track and cylinder

[5]However, most disks allow applications to turn off the write buffer in order to maintain data consistency.

Procedure: *Write Buffer Size*

- **Variables:**

  1. *max_size* : Maximal estimated write buffer size
  2. *start* : Starting LBA for write request
  3. *long_seek* : LBA for a block with large $T_{seek}(start, long\_seek)$
  4. i : Write request size iteration
  5. T_1, T_2, T_{prev}: Time registers

- **Execution:**

```
 1. $T_{prev} = 0$
 2. for $i = 1$ to $max\_size$ do
 3.     disk_seek($long\_seek$)
 4.     wait($20 \times T_{rot}$)
 5.     $T_1$ = get_time()
 6.     disk_write($start$, $i$)
 7.     $T_2$ = get_time()
 8.     if $T_2 - T_1 - T_{prev} > \frac{T_{rot}}{10}$ then
 9.         return $i - 1$
10.     $T_{prev} = T_2 - T_1$
```

Figure 3.2: Empirical method for extracting write buffer size.

skew factors for each zone, 3) track size in logical blocks, and 4) sequential data-throughput of each zone. The algorithm used to extract zoning information scans the cylinders from the logical beginning to the logical end based on the disk mapping table.

Due to the presence of bad and spare sectors, some tracks in a zone may have a smaller number of blocks than the others. Since we store only the track size (in logical blocks) for each track, we might detect a new zone incorrectly. In order to minimize the number of false positives, we use the following heuristics. First, we ignore cylinders with a large number of spare sectors. Second, during the cylinder scan, we detect a new zone only if the maximum track size in the current cylinder differs from the track size of the current zone by more than two blocks. Third, we detect a new zone only when the size of the new zone (in cylinders) is above a specific threshold.

Rotational Delay

In order to optimize disk scheduling, the OS may require accurate seek time and rotational delay characteristics of a disk [45, 53, 57, 102]. We can predict the rotational distance between two LBAs using the following disk features:

- disk mapping extracted in Section 3.3.1, and
- skew factors for the beginning of each track, relative to a chosen rotational reference point.

We choose the disk block with LBA zero as the reference point. Let c_i be the track's cylinder number, t_j the track's position in a cylinder, and $track_{size}(c_i, t_j)$ the track's size in logical blocks. Let $LBA_{start}(c_i, t_j)$ be the track's starting logical block number, and T(LBA) the time after access to a specific LBA is completed. The skew factor of a track is defined as

$$s_{c_i,t_j} = [T(LBA_{start}(c_i, t_j)) - T(LBA_0)] \bmod T_{rot}. \qquad (3.9)$$

If the number of spare and bad sectors is small, we can accurately predict the rotational distance between two LBAs (x and y) using the following equations:

$$X = T_{rot} \times \frac{[x - LBA_{start}(c_x, t_x)] \bmod track_{size}(x)}{track_{size}(x)} + s_{c_x, t_x}; \tag{3.10}$$

$$Y = T_{rot} \times \frac{[y - LBA_{start}(c_y, t_y)] \bmod track_{size}(y)}{track_{size}(y)} + s_{c_y, t_y}; \tag{3.11}$$

$$T_{rot\_del}(y, x) = (Y - X) \bmod T_{rot}. \tag{3.12}$$

Using the seek time $T_{seek}(y, x)$ defined in Section 3.3.1 and the rotational delay prediction from Equation 3.12, we can predict the access time to a disk block y after access to a block x, $T_a(y, x)$ as

$$T_a(y, x) = T_{rot\_del}(y, x) + T_{rot} \times \left\lceil \frac{T_{seek}(y, x) - T_{rot\_del}(y, x)}{T_{rot}} \right\rceil. \tag{3.13}$$

Sequential Throughput and Chunking

The maximum IO size in current schedulers for commodity operating systems is bounded to reasonable small values (approximately between 128 and 256 kB). For large sequentially-placed data, the sequential access is divided into multiple "chunks" [18, 22]. In this section, we present a method to extract optimal chunk size for sequential disk access. Figure 3.3 illustrates the effect of chunk size on disk throughput using a mock disk. The optimal chunk size lies between a and b. For chunk sizes smaller than a, due to the overhead associated with issuing a disk command, the IO bus is a bottleneck. Point b in Figure 3.3 denotes the point beyond which the performance of the cache may be sub-optimal. Points a and b in Figure 3.3 can both be extracted using Diskbench.

Disk Admission Control

In this section, we present a framework for disk admission-control methods based on disk profile for cycle-based real-time disk schedulers [9]. The cycle-

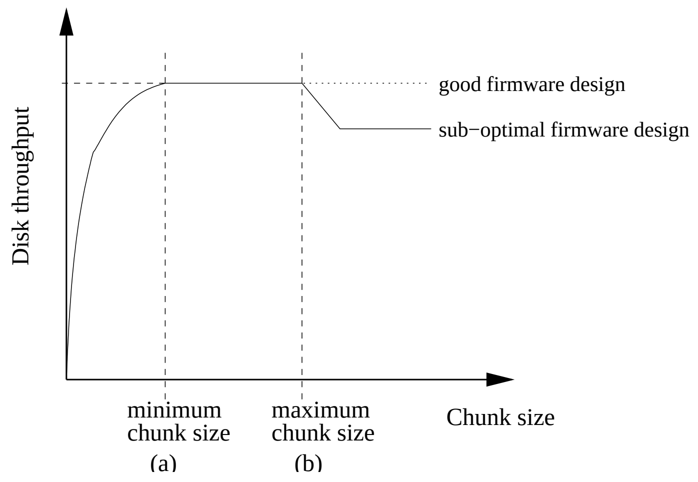


Figure 3.3: Effect of chunk size on disk throughput.

based disk scheduler employed in Xtream [21] uses the following analytical model to decide if the new stream can be admitted to 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 ratio of time reserved for real-time streams and the disk cycle time. Then T_{disk} can be written as

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

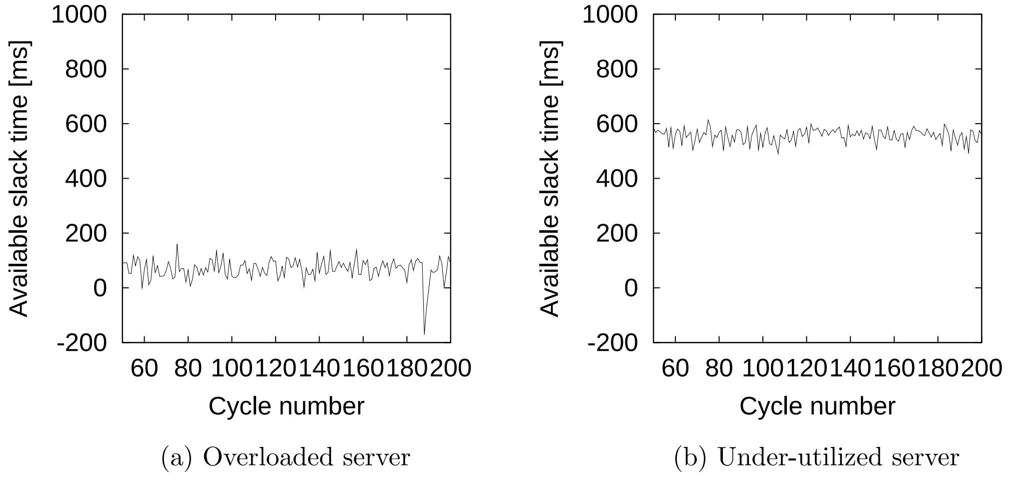
The above equation can be simplified as

$$T_{disk} \quad \geq \quad \frac{N \times \bar{L}_{disk} \times R_{disk}}{r_{RT} \times R_{disk} - N \times \bar{B}} \tag{3.14}$$

where $r_{RT} \times R_{disk} > N \times \bar{B}$. When the application asks for a specific guaranteed rate, the scheduler checks that Inequality 3.14 is satisfied. If it is not, the application is notified that it cannot get required disk bandwidth.

The admission controller must ensure that the Xtream server will not be overloaded if a new stream is admitted. At the same time, it should not deny service to a new request that will not overload the server. The two main objectives of the admission controller are maintaining QoS and avoiding under-utilization of the server.

Figure 3.4 depicts the available *slack* in each time cycle for two scenarios. Figures 3.4(a) and 3.4(b) illustrate the variations of available slack when the Xtream server is slightly overloaded (i.e., cannot maintain real-time guarantees) and under-utilized (i.e., can admit more streams) respectively. Only when the available slack is always greater than zero will the system be able to fulfill all deadlines and support all streams in real-time.



(a) Overloaded server

(b) Under-utilized server

Figure 3.4: Available slack in each time cycle.

In order to achieve the two design objectives, Xtream must be able to predict the disk-throughput utilization accurately. This is a challenging problem because disk performance varies significantly depending on the disk access pattern and file-system data-placement policies. However, to remain independent of the underlying file-system, Xtream does not make any assumptions about file-system data layout on the disk, nor does it attempt to control the file placement. The only assumption made is that a single IO is sequential which is reasonable for multimedia files with a large ratio of file size to IO size. This feature of Xtream allows it to work with almost any file-system.

To perform good admission control under these restictions, Xtream relies on accurate modeling of disk-drive performance based on disk profiling. Equation 3.15 offers a simple model for disk utilization (U) which depends on the number of IO requests in one cycle (N). The transfer time ($T_{transfer}$) is the total time that the disk spends in data transfer from disk media in a time cycle. The access time (T_{access}) is the average access penalty for each IO request, which includes both the disk seek time and rotational delay.

$$U = \frac{T_{transfer}}{N \times T_{access} + T_{transfer}} \tag{3.15}$$

Since the disk utilization U depends only on the number of requests and the total amount of data transfered in a time cycle, it can be expressed as a function of just one parameter: the average IO request size (S_{avg}).

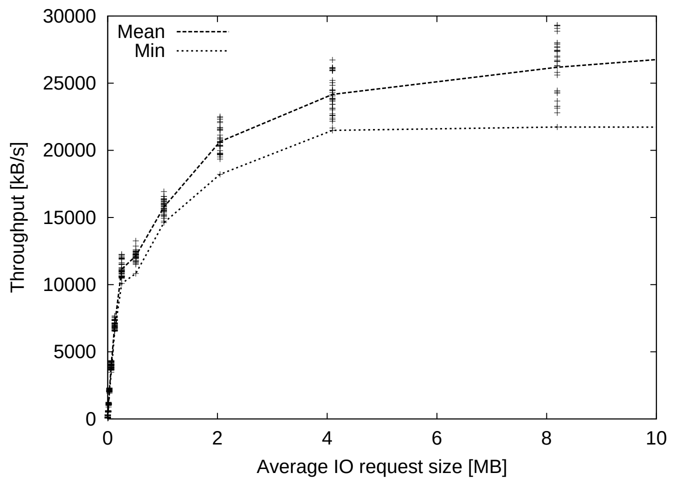


Figure 3.5: Disk throughput vs. average IO size.

We use our disk profiler tool to measure the disk-throughput utilization. The profiler performs sequential reads of the same size from random positions on the disk. Figure 3.5 shows the achieved disk throughput depending on the average IO request size. We propose and evaluate two classes of approaches for admission control: 1) *conservative* and 2) *aggressive*. The conservative class provides the best QoS level for all streams, while the aggressive class provides support for tunable QoS levels.

Let the bit-rate of each stream i in the system be denoted by BR_i. When a new request arrives (with required bit-rate BR_{new}), the admission controller first calculates the new average IO request size using Equation 3.16.

$$S_{avg} = \frac{\mathsf{T} \times (BR_{new} + \sum_{i=1}^{N} BR_i)}{N+1} \tag{3.16}$$

In the next step, we obtain the predicted disk utilization, $P(S_{avg})$, for an average request size of S_{avg} from the disk utilization curve (Figure 3.5). Then, if the condition in Equation 3.17 holds, the new request is accepted.

$$P(S_{avg}) > BR_{new} + \sum_{i=1}^{N} BR_i \tag{3.17}$$

3.4 Experimental Evaluation

In this section we present selected results from our disk profiling studies. We first present experimental results for profiling low-level disk features. We then present results for three high-level disk features: 1) rotational delay factors, 2) optimal chunk sizes for sequential access, and 3) disk admission-control curves.

3.4.1 Methodology

Diskbench consists of two separate tools: *Scsibench* and *Idextract*. *Scsibench* runs as a user-level process on Linux systems. It uses our custom user-level SCSI library to access the disk over the Linux SCSI generic interface [38, 79]. Using the command line interface, a user can specify features to extract, or traces to

	Disk	T_{rot}(in μs)	RPM	Interrog.
1.	ST39102LW	5972.56	10045.94	10045
2.	ST318437LW	8305.83	7223.84	7200

Table 3.1: Rotational time for two testbed disks.

execute. *Idextract* runs as a user-level process using Linux raw disk support for accessing any disk. All extraction methods in Idextract rely only on read and write disk commands.

We now present experimental results for each disk feature described in Section 3.3 on three testbeds. The first testbed is a dual Intel Pentium II 800MHz machine with 1GB of main memory and a 9GB Seagate ST39102LW 10000 RPM SCSI disk (12 disk heads). The second testbed is an Intel Pentium III 800MHz machine with 128MB of main memory and an 18GB Seagate ST318437LW 7200 RPM SCSI disk (2 disk heads). The third testbed is Intel Pentium 4 1500MHz with 512MB of main memory and an 40GB WD400BB-75AUA1 7200 RPM IDE disk.

The first configuration is a typical server system with a fast SCSI disk and a large number of tracks per cylinder. The second configuration is a typical workstation, with a large but slower hard disk (slower rotation speed). The third configuration is a slightly newer PC workstation with an IDE disk. We present results for the IDE disk only for methods which differ from SCSI disk methods.

3.4.2 Rotational Time

Using Equation 3.4 to calculate the time required for a single rotation of the disk, we obtained rotational times for the two testbed configurations. These are presented in Table 3.1.

3.4.3 Variations in OS Delay

Based on Equation 3.2, the variation in operating system delay for disk access is proportional to the variation in completion times for the same request. Using the trace execution described in Figure 3.6, we measured variations in request completion times (and thus, the distribution of operating system delay variations) for the two testbed configurations. These are presented in Figures 3.7 and 3.8.

```
B 0      ; Turn off all disk buffering
R 0 1    ; Read one sector starting from LBN 0
T 2      ; Store current time to register T_2

         ; repeat the following:
R 0 1    ; Read one sector starting from LBN 0
T 3      ; Store current time to register T_3
- 0 3 2  ; Print T_3 - T_2
- 2 3 0  ; T_2 = T_3 - 0
...
```

Figure 3.6: Sample trace file to find T_{OS_delay} variations.

Figure 3.7 shows the results for our first testbed configuration. We can see that the variations in OS delay are of the order of $10\mu s$. Figure 3.8 shows results for our second testbed configuration. Here the OS delay variations are of the order of $40\mu s$, with greater variations in OS delay as compared to the first testbed.

3.4.4 Mapping from Logical to Physical Block Address

We performed experiments to test both interrogative and empirical methods for the extraction of disk mappings.

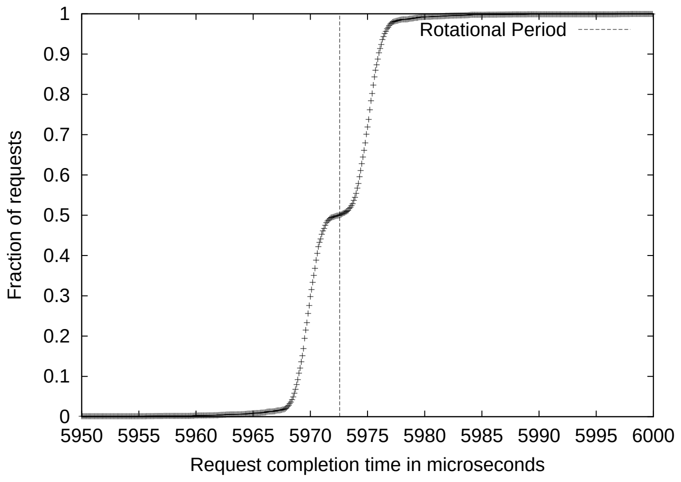


Figure 3.7: Distribution of request completion times for Seagate ST39102LW.

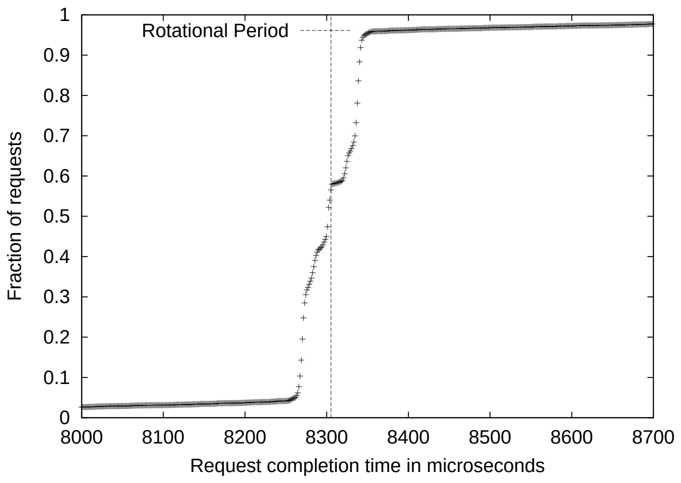


Figure 3.8: Distribution of request completion times for Seagate ST318437.

Interrogative Mapping

In Figure 3.9, we present sample results from the mapping extraction for our testbed 2 configuration.[6] In each line we print the mapping information for one cylinder. After C (representing the beginning of a new cylinder) we print the cylinder number, the starting LBA, and the cylinder size (in logical blocks). Then we print information for individual tracks after T, namely the track number, its starting LBA, and its size (in logical blocks). We print this information for all tracks.

Diskbench stores the starting LBA and the size of each disk track. If the number of bad sectors in a track is greater than the number of spare sectors allocated per track, then the track size is smaller than the size of a regular track in a particular disk zone (for example, cylinders $718 - 721$ in Figure 3.9).

Empirical Mapping

We now present results for the empirical extraction of mapping information for testbed 1 in Figures 3.10-3.13. This particular disk had 12 tracks per cylinder. In Figure 3.11 we present the access time $\Delta T(x, 0)$ (defined in Equation 3.6) between disk blocks 0 and x. The rotational period of the disk (T_{rot}) is approximately $6ms$. We detail our results in Figure 3.10, which is an enlargement of a small section of Figure 3.11.

We can see that for small values of x, the access time $\Delta T(x, 0)$ is larger than T_{rot}. When T_{OS_delay} (defined in Section 3.3.1) is larger than the rotational distance between blocks 0 and x, the second read request (the access to logical block x) has to be serviced during the next disk rotation. When $\Delta T(x, 0)$ is greater than T_{OS_delay}, an additional disk rotation is not needed. $\Delta T(x, 0)$ increases linearly for all blocks on the same track. When blocks $x - 1$ and x are located on different tracks, $\Delta T(x, 0)$ increases by the track (or cylinder) skew

[6]The results for testbed 1 do not provide any additional insights.

1. C 0 0 1500 T 0 0 750 1 750 750
2. C 1 1500 1500 T 0 2250 750 1 1500 750
3. C 2 3000 1500 T 0 3000 750 1 3750 750
4. C 3 4500 1500 T 0 5250 750 1 4500 750
5. C 4 6000 1500 T 0 6000 750 1 6750 750
6. C 5 7500 1500 T 0 8250 750 1 7500 750
7. C 6 9000 1500 T 0 9000 750 1 9750 750
8. ...
9. C 718 1077000 1499 T 0 1077000 750 1 1077750 749
10. C 719 1078499 1499 T 0 1079248 749 1 1078499 750
11. C 720 1079998 1499 T 0 1079998 750 1 1080748 749
12. C 721 1081497 1499 T 0 1082246 749 1 1081497 750
13. ...

Figure 3.9: Sample LBA-to-PBA mapping for Seagate ST318437LW.

time (after which $\Delta T(x, 0)$ continues to increase linearly). In our example this happens at logical block number 254.

When the access time to the block $x - 1$ requires a rotational delay of nearly T_{rot}, and the access to x does not require any rotational delay after seek, $\Delta T(x, 0)$ decreases by T_{rot}. This happens at block number 262 for the first time. At the next track boundary (508), a skew time increase and a T_{rot} decrease overlap. Figure 3.11 shows the $\Delta T(x, 0)$ curve for the distances up to 5000 logical blocks.

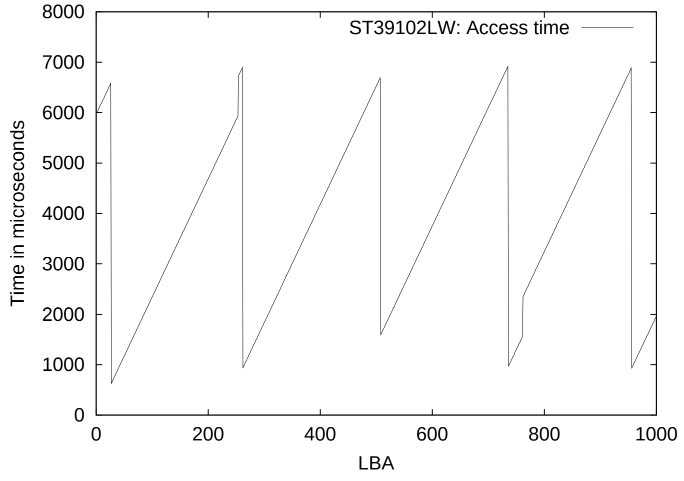


Figure 3.10: A sample of $\Delta T(x, 0)$ needed to read logical block x (on the X-axis) after reading LBA_0 for the ST39102LW.

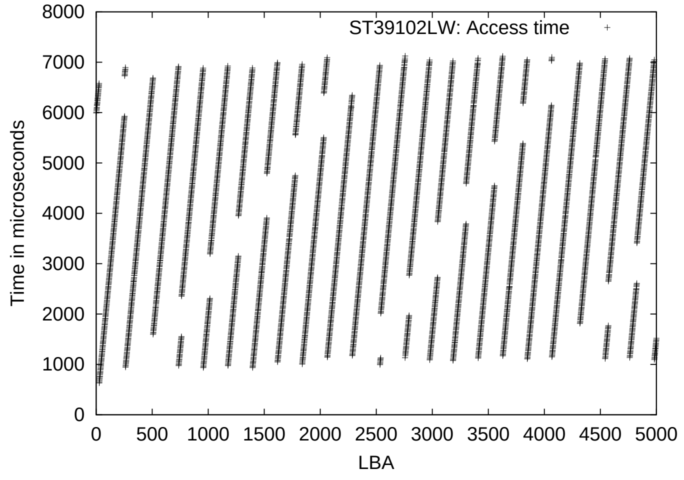


Figure 3.11: The time $\Delta T(x, 0)$ needed to read logical block x (on the X-axis) after reading LBA_0 for the ST39102LW.

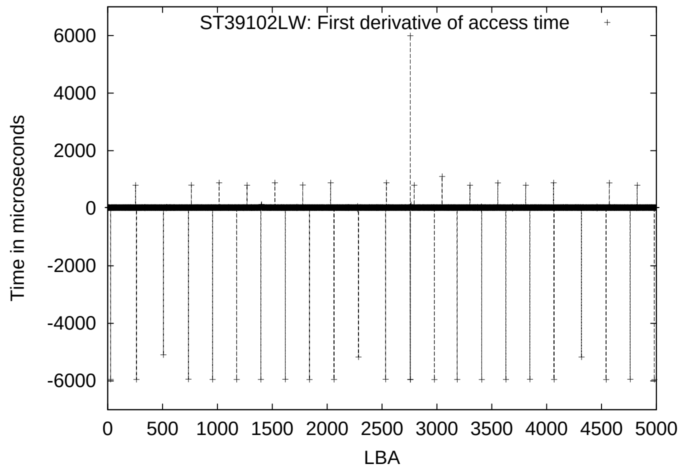


Figure 3.12: First derivative (Δ) of access time$(\Delta T(x, 0))$ for ST39102LW.

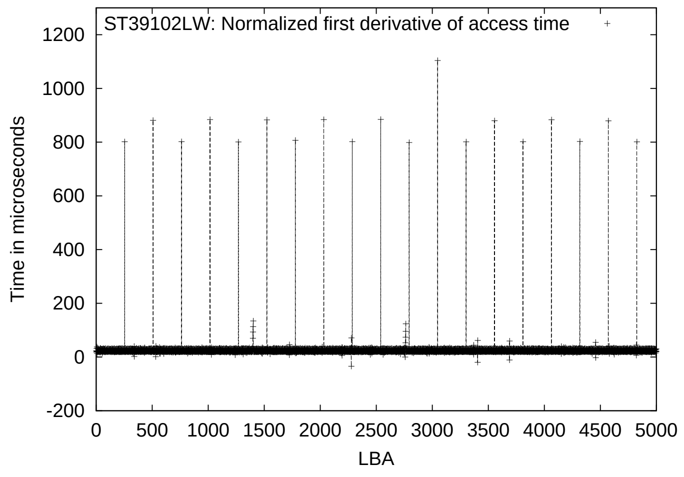


Figure 3.13: Normalized first derivative $(norm(\Delta))$ for ST39102LW.

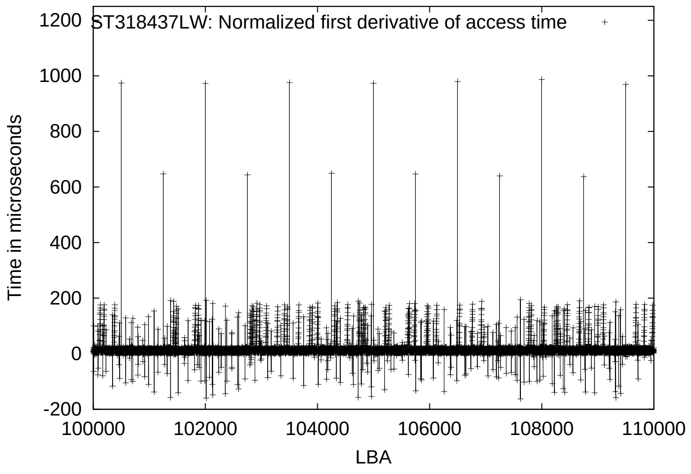


Figure 3.14: Normalized first derivative $norm(\Delta)$ for ST318437LW.

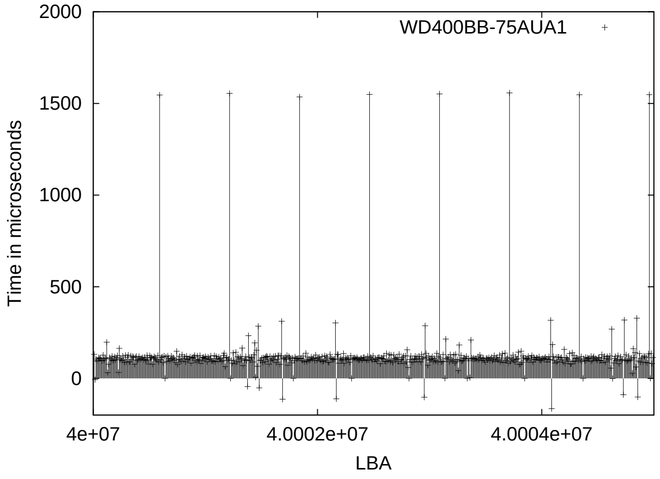


Figure 3.15: Normalized first derivative $norm(\Delta)$ for IDE WD400BB-75AUA1.

Figure 3.12 shows the first derivative of the $\Delta T(x, 0)$ curve (Δ) defined in Equation 3.7. We can see that, since T_{OS_delay} is a random variable, $\Delta T(x, 0)$ can also incur a sudden increase of T_{rot} in successive measurements. In this experiment, it occurs at $x = 2758$. This happens when the difference between T_{OS_delay} in successive measurements is substantial, so that $\Delta T(x, 0)$ incurs one disk rotation more than $\Delta T(x - 1, 0)$. A positive T_{rot} increase in $\Delta T(x, 0)$ is always followed by a negative T_{rot} decrease in the next few accesses.

In order to perform the empirical mapping automatically, we use several heuristics to find the normalized value for Δ ($norm(\Delta)$) defined in Equation 3.8. Figure 3.13 presents $norm(\Delta)$ for our testbed 1, where we capture only the track and cylinder skew times. The positions of the track and cylinder skew times on the x-axis are exact positions of the track and cylinder boundaries (occurring every 254 blocks in Figure 3.13). The track and cylinder skew times for this disk are approximately $880\mu s$ and $1100\mu s$ respectively. The skew times to switch from an odd to an even track, and from an even to an odd track, are also slightly different ($880\mu s$ and $800\mu s$ respectively).

Figure 3.14 presents the empirical mapping results for testbed 2. The disk used in this configuration has two tracks per cylinder. Results from Section 3.4.3 show that variations in operation system delay, and hence the noise in the measured $norm(\Delta)$, are much higher than for the first testbed configuration. However, since T_{OS_delay} is a random variable, we can repeat the experiment to limit the noise level and extract $norm(\Delta)$ accurately.

Figure 3.15 presents the empirical mapping results for testbed 3. We can see that Idextract results are similar to Scsibench ones. For this particular disk we are not able to find out the number of tracks per cylinder since the track skew is nearly identical to the cylinder skew time.

Zone	*Cylinders*	t_{size}	TR	TR_{max}	$\gamma(1)$	H
1	0-847	254	18.85	21.77	1108	884
2	848-1644	245	18.02	21.01	1108	885
3	1645- 2393	238	17.51	20.40	1098	876
4	2394- 3097	227	16.70	19.45	1114	891
5	3098- 3758	217	15.99	18.60	1115	890
6	3759- 4380	209	15.43	17.91	1105	885
7	4381- 4965	201	14.84	17.23	1100	876
8	4966- 5515	189	13.98	16.20	1123	901
9	5516- 6031	181	13.39	15.52	1125	903
10	6032- 6517	174	12.89	14.92	1107	885
11	6518- 6961	167	12.38	14.31	1118	899

Table 3.2: Disk zone features for ST39102LW.

Zone	*Cylinders*	t_{size}	TR	TR_{max}	$\gamma(1)$	H
1	0- 4553	750	31.07	46.24	977	652
2	4554- 6582	687	28.94	42.35	985	654
3	6583- 8247	678	28.48	41.80	987	648
4	8248-11554	666	27.92	41.06	1134	651
5	11555-14597	625	27.13	38.53	981	646
6	14598-17370	600	26.00	36.62	983	652
7	17371-19908	583	25.44	35.94	987	657
8	19909-22226	550	24.49	33.91	982	648
9	22227-26338	500	22.74	30.82	982	650
10	26339-28170	458	21.04	28.23	1159	654
11	28171-29850	437	20.24	26.94	990	663

Table 3.3: Disk zone features for ST318437LW.

3.4.5 Seek Curves

In Figure 3.16 we present the seek curve for our first testbed (ST39102LW). We can see that the difference between the rotational period and the maximum seek time is less than a factor of two. Since the variations in the seek curve are negligible, we can also deduce that the seek time depends mainly on the seek distance in cylinders, and not on starting or destination cylinder positions. Figure 3.17 presents the seek curve for the second testbed.

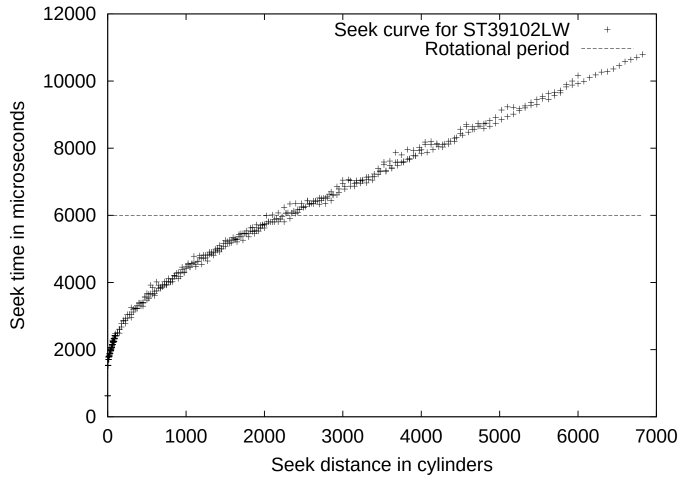


Figure 3.16: Complete seek curve for ST39102LW.

3.4.6 Read Cache

Figures 3.18 and 3.19 depict results for the extraction of the cache segment size, using the method explained in Section 3.3.1. Both disks stopped prefetching when they filled up the first cache segment. The extracted size for testbeds 1 and 2 was 561 and 204 blocks, respectively. Both disks continued prefetching into the next available cache segment whenever a long sequential access was detected.

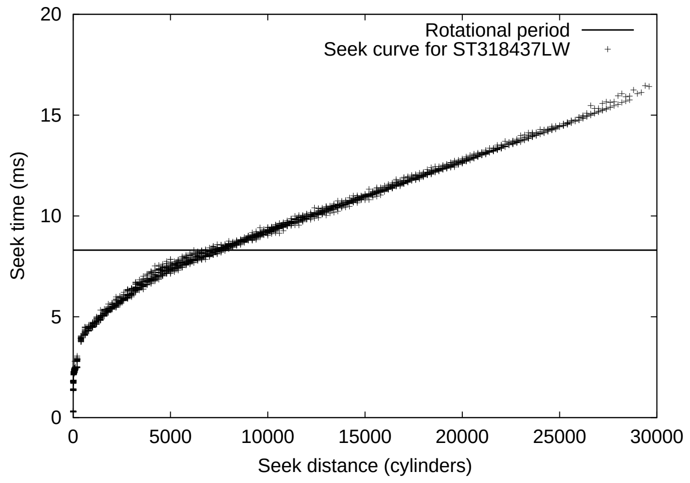


Figure 3.17: Complete seek curve for ST318437LW.

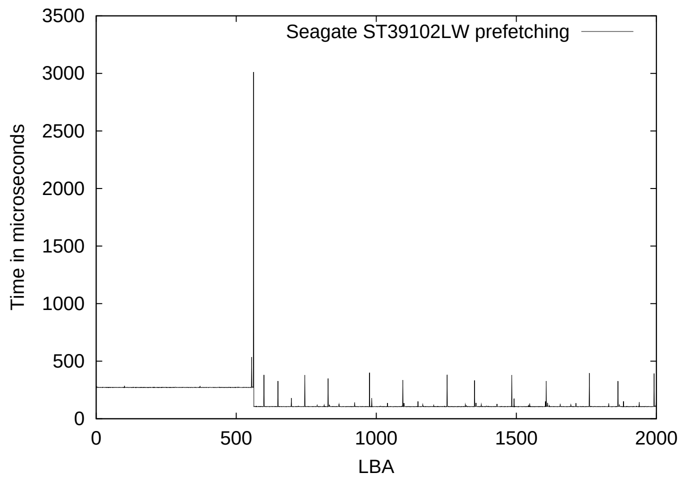


Figure 3.18: Read request completion times for ST39102LW.

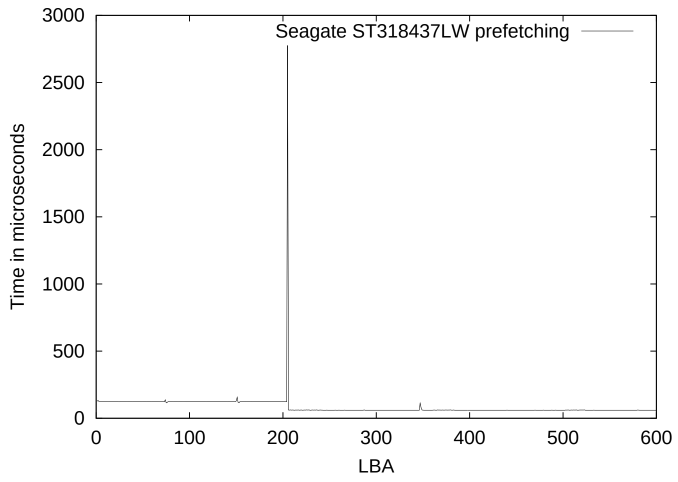


Figure 3.19: Read request completion times for ST318437LW.

Using the extracted cache segment size, we can find out the number of cache segments in the disk read cache, as explained in Section 3.3.1. Using this method, we have detected three cache segments for testbed 1, and 16 segments for testbed 2.

3.4.7 Write Buffer

Figures 3.20 and 3.21 present the results for write buffer size extraction using the method introduced in Section 3.3.1.

The write buffer size for testbed 1 and testbed 2 was measured to be 561 and 204 blocks respectively. When comparing these to the cache segment-size extractions presented in Section 3.4.6, we can see that both disks use exactly one cache segment as a write buffer (for each write sequential stream of data).

Using these measurements we can also measure the write throughput, both to the disk write buffer (slope of the curve for write request sizes that fit into the write buffer), and to the disk platter (slope for request sizes greater than the write buffer size). In the future we plan to extract the number of cache

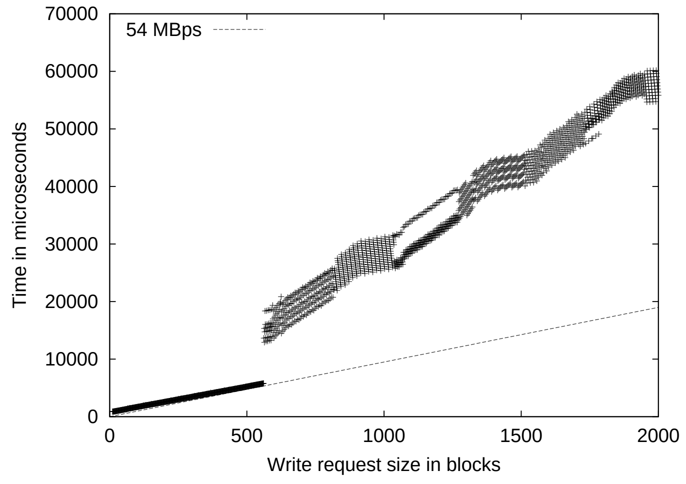


Figure 3.20: Write request completion times for ST39102LW.

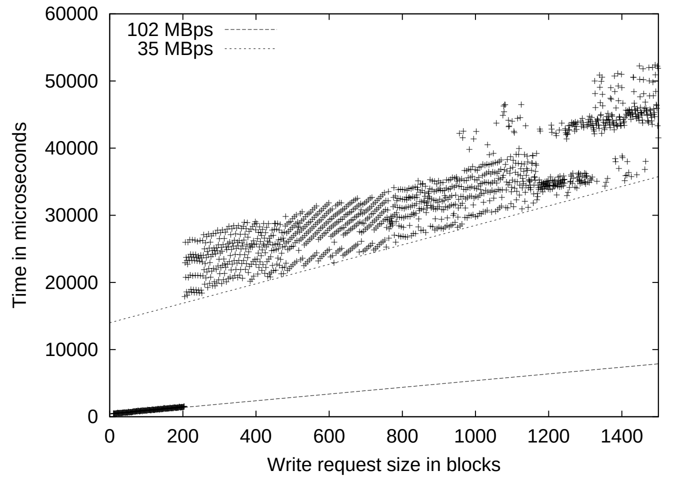


Figure 3.21: Write request completion times for ST318437LW.

segments that can be used as write buffers. We believe that we can use a method similar to the method we used for detecting the number of read cache segments presented in Section 3.3.1.

3.4.8 Disk Zones

Tables 3.2 and 3.3 present the zoning information extracted for the Seagate ST39102LW and Seagate ST318437LW disks respectively. t_{size} denotes the track size in logical blocks. TR is the transfer rate measured for long sequential reads that span multiple cylinders. TR_{max} is the calculated theoretical maximum transfer rate for read requests which incurs no seek, rotation or switching overheads. H and $\gamma(1)$ are the track and cylinder switch times in microseconds.

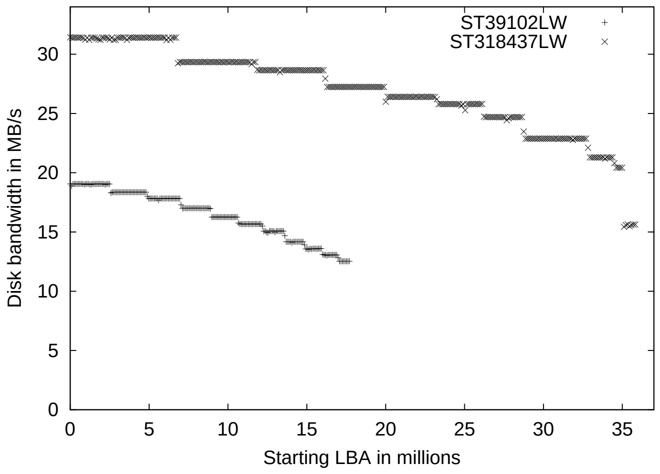


Figure 3.22: Disk bandwidth depending on data location for two SCSI disks.

Figure 3.22 depicts the disk bandwidth for large sequential accesses depending on the starting LBA for testbed 1 and 2. For modern disks, the difference between the maximum and minimum sequential disk bandwidth is usually a factor of two. Figure 3.23 presents the disk zone bandwidths for testbed 3.

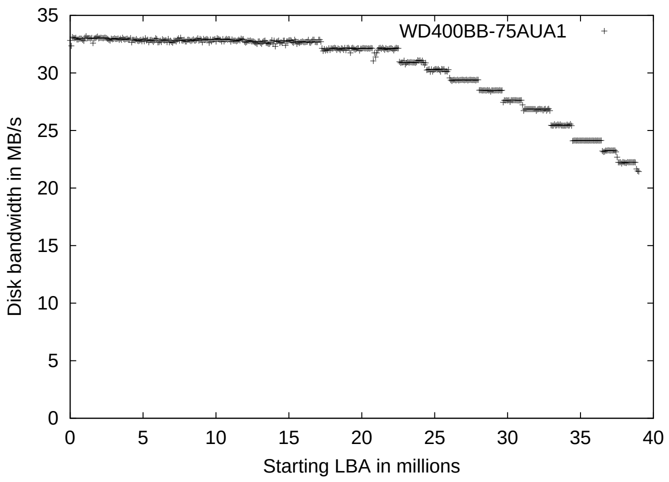


Figure 3.23: Disk bandwidth depending on data location for an IDE disk.

3.4.9 Rotational Delay Prediction

Based on Equation 3.12, in order to predict the rotational delay accurately, we need to extract the skew factor (defined in Equation 3.9) for each track. Sample results for the extracted skew factors (in μs) are presented in Table 3.4, wherein we also present the LBAs for blocks residing on the same disk radius (LBA_{rot0}).

In Figure 3.24, we plot the skew times against track numbers. We notice a distinct trend in skew times, a property which enables us to compress this information effectively and to reduce its space requirement. In Figure 3.24, we also notice a slight deviation from the normal trend for tracks 12, 24, and 36. This is due to cylinder skew, which occurs when the next track falls on an adjacent cylinder instead of the same cylinder. The experimental disk had exactly 12 surfaces. Hence, we expect a trend deviation on tracks that are multiples of 12 to account for an increased switching overhead.

Based on Equation 3.12 and the compressed information about skew times above, we were able to predict the rotational delay between two disk accesses. In

Cyl	$Track$	$Skew(\mu s)$	LBA_{rot0}
0	0	0	0
0	1	755.15	475
0	2	1595.02	694
0	3	2352.35	915
0	4	3191.09	1134
0	5	3949.03	1356
0	6	4788.61	1574
0	7	5546.36	1796
0	8	414.31	2268
0	9	1170.86	2490
0	10	2009.76	2708
0	11	2767.37	2930
1	0	3829.42	3139
...	...	...	...

Table 3.4: Rotational delay modeling for ST39102LW (the track size for the first zone is 254 blocks). Disk blocks with LBA_{rot0} are on the same disk radius.

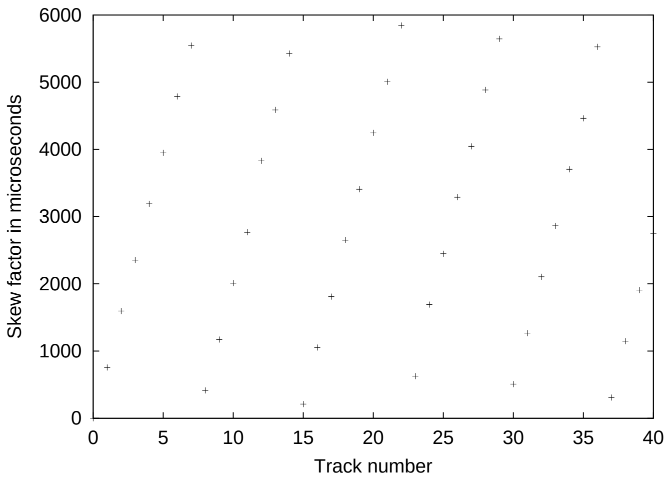


Figure 3.24: Skew factor s from Table 3.4 for ST39102LW.

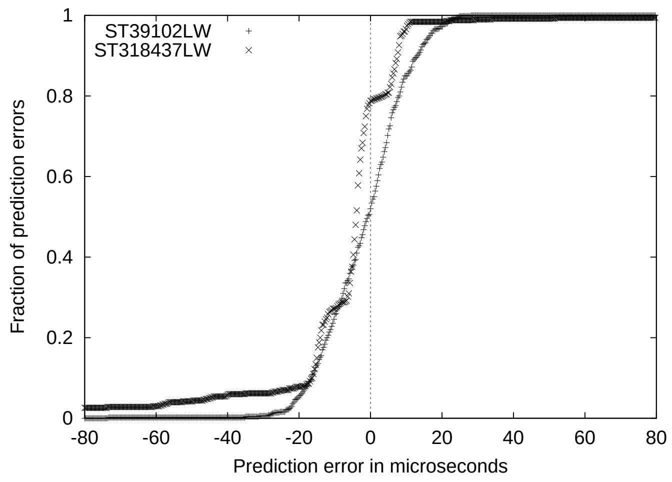


Figure 3.25: Rotational delay prediction accuracy for ST39102LW and ST318-437LW.

Figure 3.25, we present the error distribution of rotational delay predictions for a large number of random request-pairs. We note that for the SMP-like testbed (testbed 1), which has a very predictable distribution of OS delay variations (Figure 3.7), our prediction is accurate within 25 μs for 99% of the requests. Even for the workstation-like testbed (testbed 2), which has less predictable OS delay variations (Figure 3.8), our prediction is accurate within 80 μs for 99% of the requests. These errors are negligible compared to variations in seek time, which are of the order of a millisecond. We thus conclude that with detailed disk parameters, systems can implement very accurate mechanisms for predicting rotational delays. We used seek time and rotational delay predictions from Diskbench to predict disk access times in the implementation of Semi-preemptible IO [20, 22], which is presented in Chapter 4.

3.4.10 Sequential Throughput and Chunking

As regards chunking, the disk profiler provides the optimal range for the chunk size. Figure 3.26 depicts the effect of chunk size on the read throughput performance for one SCSI and one IDE disk drive. Figure 3.27 shows the same for the write case. Clearly, the optimal range for chunk size (between points a and b illustrated previously in Figure 3.3 in Section 3.3.2) can be automatically extracted from these figures.

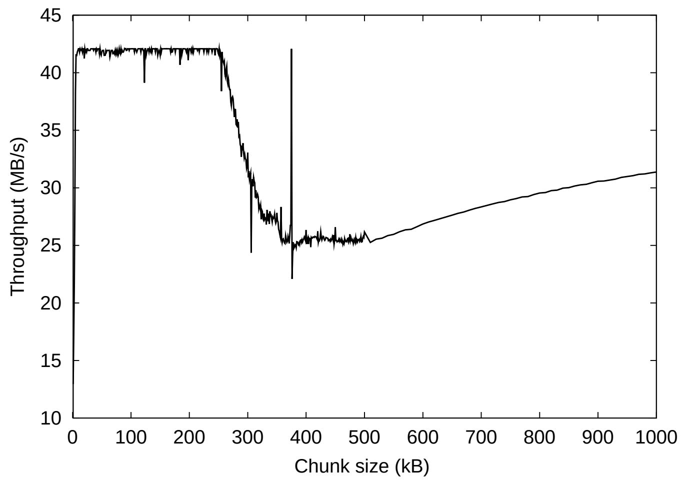
3.4.11 Disk Admission Control

In this section, we evaluate the Xtream system using the following metrics: 1) maximum system throughput, 2) initial latency, and 3) accuracy of admission control methods. We used an Intel Pentium 4 1.5 GHz Linux based PC, with 512 MB of main memory and a WD400BB 40 GB hard drive. The maximum sequential disk throughput was 31 MBps in the fastest zone and 21 MBps in the slowest zone. The LAN was 100 Mbps ethernet, which enables streaming several mpeg2 and a large number of mpeg4 encoded videos.

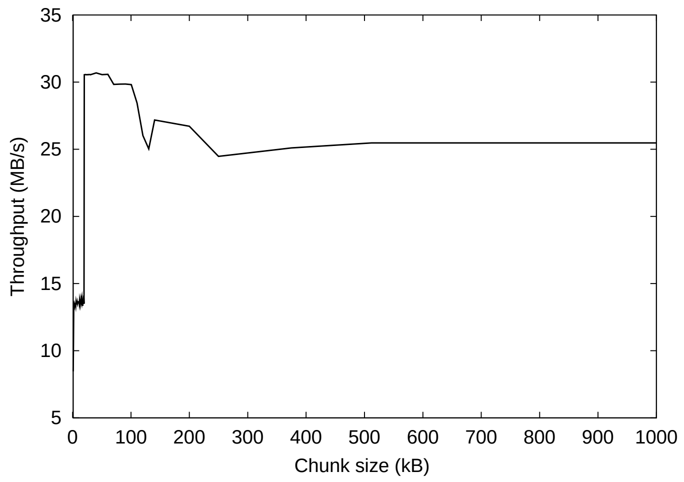
In order to evaluate the hard disk scheduler, a client can require "dummy" streaming with constant (CBR) or variable bit-rate (VBR). Dummy streams were not streamed over the network. The client specified whether the dummy stream was read or write, and whether the bit-rate was constant or variable.

Throughput

Figure 3.28 depicts the percentage of missed cycle deadlines depending on the total bit-rate of serviced streams. In each of the three experiments (denoted by the three lines), all serviced streams had the same bit-rate. Larger bit-rates result in larger disk IOs, and consequently higher disk utilization (See Figure 3.5). In each experiment, we used a time cycle of one second. Since one of the main goals of the system is to maintain real-time guarantees, the maximum throughput of the system equals the maximum value on the x-axis when the system does not miss any deadlines. Depending on the required QoS, the admission control module can choose an appropriate maximum through-

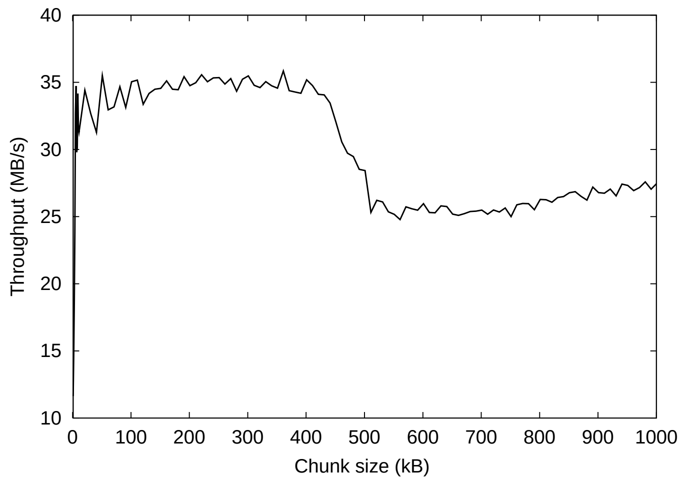


(a) SCSI ST318437LW

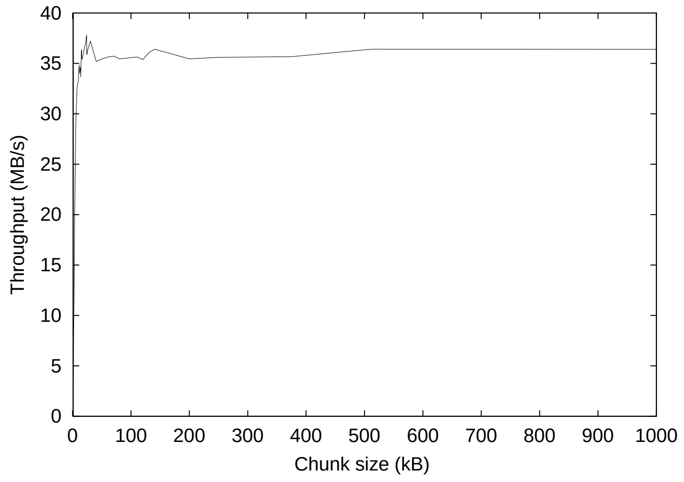


(b) IDE WD400BB

Figure 3.26: Sequential read throughput vs. chunk size.



(a) SCSI ST318437LW



(b) IDE WD400BB

Figure 3.27: Sequential write throughput vs. chunk size.

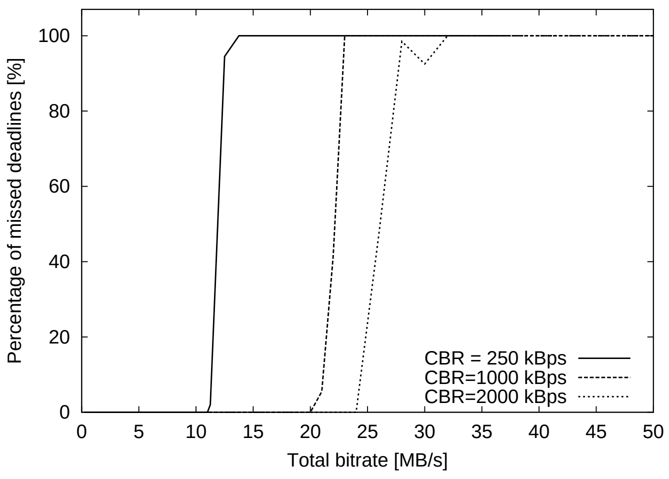


Figure 3.28: Percentage of missed cycle deadlines.

put (total bit-rate) for the disk. Thus, the trade-off between QoS and system throughput decides the admission control policy. Table 3.5 depicts the accuracy for one conservative and two aggressive Xtream admission control policies.

Initial Latency

For this experiment, we defined *initial latency* as the delay between the moment the request handler received a client request, and the moment when the initial buffer for the new stream was filled. Data on the measured initial latency, depending on the number of streams in the system, is presented in Figure 3.29. The initial latency did not depend on the load of the system but only on the size of IO requests (which depends on stream bit-rates). The experiment did not consider or evaluate network or client-side latency.

Guaranteed IO

For a streaming multimedia system to guarantee QoS for IO, its admission control policy must be accurate. We evaluated the three different admission

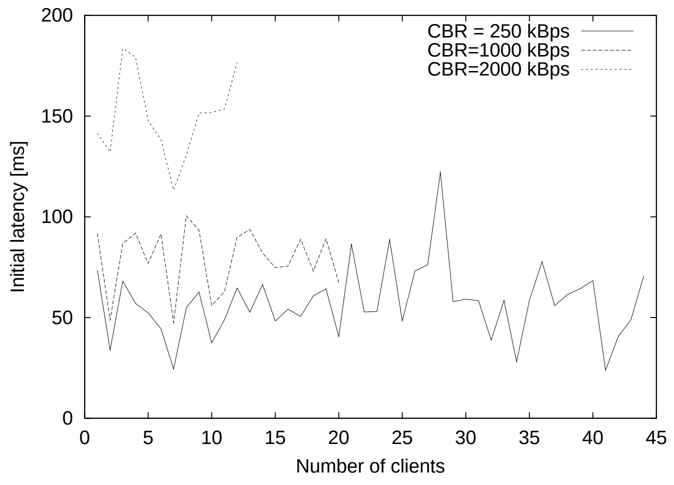


Figure 3.29: Initial latency.

control configurations: conservative admission control (*Adm*1) and two aggressive admission controls (*Adm*2 and *Adm*3). *Adm*1 uses the min curve from Figure 3.5 to perform admission control. *Adm*2 uses the mean curve from Figure 3.5 and maximum stream bit-rates. *Adm*3 uses the mean curve from Figure 3.5 and average stream bit-rates.

Avg. BR	Type	$MaxN$	$Adm1$	$Adm2$	$Adm3$
250 kBps	C	44	39	43	n/a
1000 kBps	C	20	14	15	n/a
2000 kBps	C	12	9	10	n/a
250 kBps	V	44	30	34	43
1000 kBps	V	23	11	12	15
2000 kBps	V	11	7	8	10
250 kBps	H	44	39	43	n/a
1000 kBps	H	16	14	15	n/a
2000 kBps	H	10	9	10	n/a

Table 3.5: Admission control accuracy.

To determine the accuracy of the three admission control configurations, we performed experiments for the following two scenarios: homogeneous CBR (type C) and VBR (type V) streams where all serviced streams have the same bit-rate; and heterogenous CBR streams (type H) where each stream has a constant bit-rate, but the bit-rate for individual streams in the system varies between 1/10 and 10 times the average value. $MaxN$ denotes the maximum number of streams that the system can support without missing deadlines. $MaxN$ was calculated manually within each experiment using trial and error. The results presented in Table 3.5 show that Xtream provided guaranteed disk IO performance, while not significantly under-utilizing the system.

3.4.12 Summary of Results

We evaluated Diskbench using two testbeds, a typical server configuration (an SMP multiprocessor machine with a fast 10K RPM SCSI disk) and a typical workstation configuration (a single CPU machine with a large 7200 RPM SCSI disk). For profiling high-level disk features, we performed additional experiments using several IDE disks.

First, we presented experimental results for profiling the following low-level disk features: 1) rotational time, 2) OS delay variations, 3) disk mappings, 4) seek curves, 5) read cache, 6) write cache, and disk zoning.

Second, we presented experimental results for the folowing high-level disk features: 1) rotational delay factors, 2) optimal chunk size for sequential access, and 3) disk admission-control curves.

3.5 Summary

In this chapter, we have presented Diskbench, a user-level tool for disk feature extraction. Diskbench uses both interrogative and empirical methods to extract disk features. The empirical methods extract accurate low-level disk features like track and cylinder boundaries, track and cylinder skew times, the number of tracks per cylinder, track size (in logical blocks), as well as read and write buffer parameters. Diskbench also extracts high-level disk features necessary for advanced scheduling methods like our Semi-preemptible IO [22] or rotationally-aware schedulers [42, 45, 53, 102, 52, 93].

We believe that this work can be used by system and application programmers to improve and guarantee real-time disk performance. Using knowledge about disk features provided by Diskbench, system or application programmers can fine-tune disk accesses to match application requirement and can predict disk performance, which is necessary for real-time disk scheduling.

Chapter 4

Preemptibility of Disk Access

In this chapter, we investigate the preemptibility of disk access and present the design and implementation of *Semi-preemptible IO* [20, 22].

4.1 Introduction

Traditionally, disk IOs have been thought of as non-preemptible operations. Once initiated, they cannot be stopped until completed. Over the years, operating system designers have learned to live with this restriction. However, non-preemptible IOs can be a stumbling block for applications that require a short response time. We propose methods to make disk IOs semi-preemptible, thus providing the operating system a finer level of control over the disk-drive.

Preemptible disk access is desirable in certain settings. One such domain is that of real-time disk scheduling. Real-time scheduling theoreticians have developed schedulability tests (tests of whether a task set is schedulable such that all deadlines are met) in various settings [46, 48, 51]. In real-time scheduling theory, *blocking*[1], or priority inversion, is defined as the time spent when a higher-priority task is prevented from running due to the non-preemptibility

[1]We refer to blocking as the *waiting time.*

of a low-priority task. Blocking degrades schedulability of real-time tasks and is thus undesirable. Making disk IOs preemptible would reduce blocking and improve the schedulability of real-time disk IOs.

Another domain where preemptible disk access is essential is that of interactive multimedia such as video, audio, and interactive virtual reality. Because of the large amount of memory required by these media data, they are stored on disks and are retrieved into main memory only when needed. For interactive multimedia applications that require a short response time, a disk IO request must be serviced promptly. For example, in an immersive virtual world, the latency tolerance between a head movement and the rendering of the next scene (which may involve a disk IO to retrieve relevant media data) is around 15 milliseconds [4]. Such interactive IOs can be modeled as higher-priority IO requests. However, due to the typically large IO size and the non-preemptible nature of ongoing disk commands, even such higher-priority IO requests can be kept waiting for tens, if not hundreds, of milliseconds before being serviced by the disk.

To reduce the response time for a higher-priority request, its waiting time must be reduced. The *waiting time* for an IO request is the amount of time it must wait, due to the non-preemptibility of the ongoing IO request, before being serviced by the disk. The response time for the higher-priority request is then the sum of its waiting time and service time. The *service time* is the sum of the seek time, rotational delay, and data transfer time for an IO request. (Service time can be reduced by intelligent data placement [104] and scheduling policies [102]. However, our focus is on reducing the waiting time by increasing the preemptibility of disk access.)

In this study, we explore *Semi-preemptible IO* (previously called Virtual IO [20]), an abstraction for the disk IO, which provides highly preemptible disk access (average preemptibility of the order of one millisecond) with little loss in disk throughput. *Semi-preemptible IO* breaks the components of an IO job into fine-grained physical disk-commands and enables IO preemption between them. It thus separates the preemptibility from the size and duration of the operating system's IO requests.

Semi-preemptible IO maps each IO request into multiple fast-executing disk commands using three methods. Each method addresses the reduction of one of the possible components of the waiting time — the ongoing IO's transfer time ($T_{transfer}$), rotational delay (T_{rot}), and seek time (T_{seek}).

- **Chunking $\mathbf{T_{transfer}}$.** A large IO transfer is divided into a number of small chunk transfers, and preemption is made possible between the small transfers. If the IO is not preempted between the chunk transfers, chunking does not incur any overhead. This is due to the prefetching mechanism in current disk drives (Section 4.2.1).

- **Preempting $\mathbf{T_{rot}}$.** By performing just-in-time (JIT) seek for servicing an IO request, the rotational delay at the destination track is virtually eliminated. The pre-seek slack time thus obtained is preemptible. This slack can also be used to perform prefetching for the ongoing IO request, and/or to perform seek splitting (Section 4.2.2).

- **Splitting $\mathbf{T_{seek}}$.** *Semi-preemptible IO* can split a long seek into sub-seeks, permitting a preemption between two sub-seeks (Section 4.2.3).

The following example illustrates how *Semi-preemptible IO* can reduce the waiting time for higher-priority IOs (and hence improve the preemptibility of disk access).

4.1.1 Illustrative Example

Suppose a 500 kB read-request has to seek $20,000$ cylinders requiring T_{seek} of 14 ms, must wait for a T_{rot} of 7 ms, and requires $T_{transfer}$ of 25 ms at a transfer rate of 20 MBps. The expected waiting time, $E(T_{waiting})$, for a higher-priority request arriving during the execution of this request, is 23 ms, while the maximum waiting time is 46 ms (please refer to Section 4.2 for equations). *Semi-preemptible IO* can reduce the waiting time by performing the following operations.

It first predicts both the seek time and rotational delay. Since the predicted seek time is long ($T_{seek} = 14$ ms), it decides to split the seek operation into two sub-seeks, each of $10,000$ cylinders, requiring $T'_{seek} = 9$ ms each. This seek splitting does not cause extra overhead in this case because the $T_{rot} = 7$ can mask the 4 ms increased total seek time ($2 \times T'_{seek} - T_{seek} = 2 \times 9 - 14 = 4$). The rotational delay is now $T'_{rot} = T_{rot} - (2 \times T'_{seek} - T_{seek}) = 3$ ms.

With this knowledge, the disk driver waits for 3 ms before performing a JIT-seek. This JIT-seek method makes T'_{rot} preemptible, since no disk operation is being performed. The disk then performs the two sub-seek disk commands, and then 25 successive read commands, each 20 kB in size, requiring 1 ms each. A higher-priority IO request could be serviced immediately after each disk-command. *Semi-preemptible IO* thus enables preemption of an originally non-preemptible read IO request. Now, during the service of this IO, we have two scenarios:

- **No higher-priority IO arrives.** In this case, the disk does not incur additional overhead for transferring data due to disk prefetching (discussed in Sections 4.2.1 and 4.2.4). (If T_{rot} cannot mask seek-splitting, the system can also choose not to perform seek-splitting.)

- **A higher-priority IO arrives.** In this case, the maximum waiting time for the higher-priority request is now a mere 9 ms, if it arrives during one of the two seek disk commands. However, if the ongoing request is at the stage of transferring data, the longest stall for the higher-priority request is just 1 ms. The expected value for waiting time is only $\frac{1}{2}\,\frac{2\times 9^2+25\times 1^2}{2\times 9+25\times 1+3} = 2.03$ ms, a significant reduction from 23 ms (see Section 4.2 for details).

This example shows that *Semi-preemptible IO* substantially reduces the expected waiting time and hence increases the preemptibility of disk access. However, if an IO request is preempted to service a higher-priority request, an extra seek operation may be required to resume service for the preempted IO. The distinction between *IO preemptibility* and *IO preemption* is an important one. Preemptibility enables preemption, but incurs little overhead itself. Preemption will always incur overhead, but it will reduce the service time for higher-priority requests. Preemptibility provides the system with the choice of trading throughput for short response time when such a tradeoff is desirable. We explore the effects of IO preemption further, in Section 4.3.3.

4.1.2 Contributions

In summary, the contributions of Semi-preemptible IO are as follows:

- We introduce *Semi-preemptible IO*, which abstracts both read and write IO requests so as to make them preemptible. As a result, the system can substantially reduce the waiting time for a higher-priority request at little or no extra cost.

- We show that making write IOs preemptible is not as straightforward as it is for read IOs. We propose one possible solution for making them preemptible.

- We present a feasible path to implement *Semi-preemptible IO*. We explain how the implementation is made possible through use of a detailed disk profiling tool.

4.2 Semi-preemptible IO

Before introducing the concept of *Semi-preemptible IO*, let us first summarize some terms which we will use throughout the rest of this chapter. Then, we present the Semi-preemptible IO, a preemptible abstraction for disk IOs.

Definitions:

- A *logical disk block* is the smallest unit of data that can be accessed on a disk drive (typically 512 B). Each logical block resides at a physical disk location, depicted by a physical address (cylinder, track, sector).
- A *disk command* is a non-preemptible request issued to the disk over the IO bus (e.g., the read, write, seek, and interrogative commands).
- An *IO request* is a request for read or write access to a sequential set of logical disk blocks.
- The *waiting time* is the time between the arrival of a higher-priority IO request and the moment the disk starts servicing it.

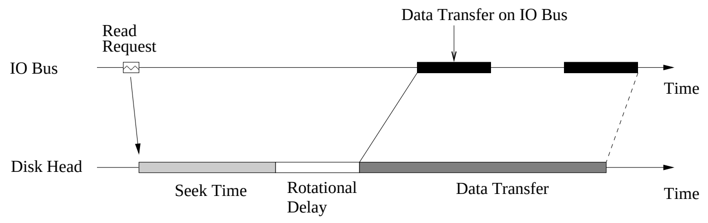


Figure 4.1: Timing diagram for a disk read request.

In order to understand the magnitude of the waiting time, let us consider a typical read IO request, depicted in Figure 4.1. The disk first performs a seek to the destination cylinder requiring T_{seek} time. Then, the disk must wait for a rotational delay, denoted by T_{rot}, so that the target disk block comes under the disk arm. The final stage is the data transfer stage, requiring a time of

$T_{transfer}$, when the data is read from the disk media to the disk buffer. This data is simultaneously transferred over the IO bus to the system memory.

For a typical commodity system, once a disk command is issued on the IO bus, it cannot be stopped. Traditionally, an IO request is serviced using a single disk command. Consequently, the operating system must wait until the ongoing IO is completed before it can service the next IO request on the same disk. Let us assume that a higher-priority request may arrive at any time during the execution of an ongoing IO request with equal probability. The waiting time for the higher-priority request can be as long as the duration of the ongoing IO. The expected waiting time of a higher-priority IO request can then be expressed in terms of seek time, rotational delay, and data transfer time required for the ongoing IO request as

$$E(T_{waiting}) = \frac{1}{2}(T_{seek} + T_{rot} + T_{transfer}). \tag{4.1}$$

Let V_i be the sequence of fine-grained disk commands we use to service an IO request. Let the time required to execute disk-command V_i be T_i. Let T_{idle} be the duration of time during the servicing of the IO request, when the disk is idle (i.e., no disk command is issued). Using the above assumption that the higher-priority request can arrive at any time with equal probability, the probability that it will arrive during the execution of the i^{th} command V_i can be expressed as $p_i = \frac{T_i}{\sum T_i + T_{idle}}$. Finally, the expected waiting time of a higher-priority request in *Semi-preemptible IO* can be expressed as

$$E(T'_{waiting}) = \frac{1}{2}\sum(p_i T_i) = \frac{1}{2}\frac{\sum T_i^2}{(\sum T_i + T_{idle})}. \tag{4.2}$$

In the remainder of this section, we present 1) *chunking*, which divides $T_{transfer}$ (Section 4.2.1); 2) *just-in-time seek*, which enables T_{rot} preemption (Section 4.2.2); and 3) *seek splitting*, which divides T_{seek} (Section 4.2.3). In addition, we present our disk profiler, Diskbench, and summarize all the disk parameters required for the implementation of *Semi-preemptible IO* (Section 4.2.4).

4.2.1 Chunking: Preempting $\mathbf{T_{transfer}}$

The data transfer component ($T_{transfer}$) in disk IOs can be large. For example, the current maximum disk IO size used by Linux and FreeBSD is 128 kB, and it can be larger for some specialized video-on-demand systems[2]. To make the $T_{transfer}$ component preemptible, *Semi-preemptible IO* uses *chunking.*

Definition 4.2.1: *Chunking* is a method for splitting the data transfer component of an IO request into multiple smaller *chunk* transfers. The chunk transfers are serviced using separate disk commands, issued sequentially.

Benefits: Chunking reduces the transfer component of $T_{waiting}$. A higher-priority request can be serviced after a chunk transfer is completed instead of after the entire IO is completed. For example, suppose a 500 kB IO request requires a $T_{transfer}$ of 25 ms at a transfer rate of 20 MBps. Using a chunk size of 20 kB, the expected waiting time for a higher-priority request is reduced from 12.5 ms to 0.5 ms.

Overhead: For small chunk sizes, the IO bus can become a performance bottleneck due to the overhead of issuing a large number of disk commands. As a result, the disk throughput degrades. Issuing multiple disk commands instead of a single one also increases the CPU overhead for performing IO. However, for the range of chunk sizes, the disk throughput using chunking is optimal with negligible CPU overhead.

The Method

To perform chunking, the system must decide on the chunk size. *Semi-preemptible IO* chooses the minimum chunk size for which the disk throughput is optimal and the CPU overhead acceptable. Surprisingly, large chunk sizes can also suffer from throughput degradation due to the sub-optimal implementation

[2]These values are likely to vary in the future. *Semi-preemptible IO* provides a technique that does not deter disk preemptibility with the increased IO sizes.

of disk firmware (Section 4.2.4). Consequently, *Semi-preemptible IO* may achieve even better disk throughput than the traditional method where an IO request is serviced using a single disk command.

In order to perform chunking efficiently, *Semi-preemptible IO* relies on the existence of a read cache and a write buffer on the disk. It uses disk profiling to find the optimal chunk size. We now present the chunking for read and write IO requests separately.

The Read Case

Disk drives are optimized for sequential access, and they continue prefetching data into the disk cache even after a read operation is completed [74]. Chunking for read IO requests is illustrated in Figure 4.2. The x-axis shows time, and the two horizontal time lines depict the activity on the IO bus and the disk head, respectively. Employing chunking, a large $T_{transfer}$ is divided into smaller chunk transfers issued in succession. The first read command issued on the IO bus is for the first chunk. Due to the prefetching mechanism, all chunk transfers following the first one are serviced from the disk cache rather than the disk media. Thus, the data transfers on the IO bus (the small dark bars shown on the IO bus line in the figure) and the data transfer into the disk cache (the dark shaded bar on the disk-head line in the figure) occur concurrently. The disk head continuously transfers data after the first read command, thereby fully utilizing the disk throughput.

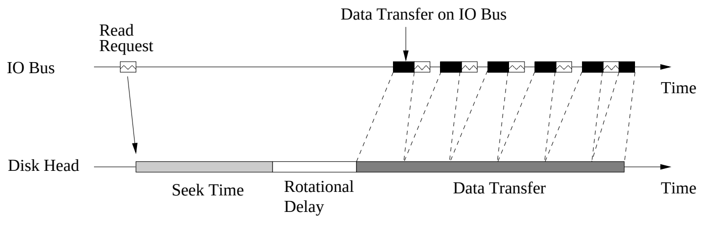


Figure 4.2: Virtual preemption of the data transfer.

Figure 4.3 illustrates the effect of the chunk size on the disk throughput using a mock disk. The optimal chunk size lies between a and b. A smaller chunk size reduces the waiting time for a higher-priority request. Hence, *Semi-preemptible IO* uses a chunk size close to but larger than a. For chunk sizes smaller than a, due to the overhead associated with issuing a disk command, the IO bus is a bottleneck. Point b in Figure 4.3 denotes the point beyond which the performance of the cache may be sub-optimal[3].

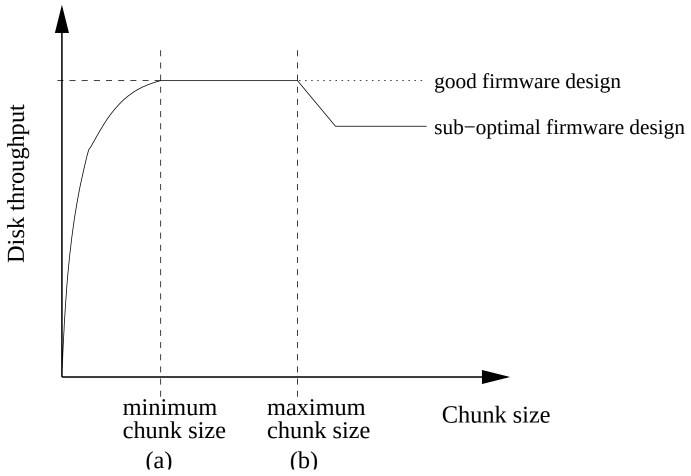


Figure 4.3: Effect of chunk size on disk throughput.

The Write Case

Semi-preemptible IO performs chunking for write IOs similarly to chunking for read requests. However, the implications of chunking in the write case are different. When a write IO is performed, the disk command can complete as soon as all the data is transferred to the disk write buffer[4]. As soon as the write

[3]We have not fully investigated the reasons for sub-optimal disk performance and it is the subject of our future work.

[4]If the size of the write IO is larger than the size of the write buffer, then the disk signals the end of the IO as soon as the excess amount of data (which cannot be fitted into the disk buffer) has been written to the disk media.

command is completed, the operating system can issue a disk command to service a higher-priority IO. However, the disk may choose to schedule a write-back operation for disk write buffers before servicing a new disk command.We refer to this delay as the *external waiting time*. Since the disk can buffer multiple write requests, the write-back operation can include multiple disk seeks. Consequently, the waiting time for a higher-priority request can be substantially increased when the disk services write IOs.

In order to increase preemptibility of write requests, we must take into consideration the external waiting time for write IO requests. External waiting can be reduced to zero by disabling write buffering. However, in the absence of write buffering, chunking would severely degrade disk performance. The disk would suffer from an overhead of one disk rotation after performing an IO for each chunk. To remedy external waiting, our prototype forces the disk to write only the last chunk of the write IO to disk media by setting force-unit-access flag in SCSI write command. Using this simple technique, it triggers the write-back operation at the end of each write IO. Consequently, the external waiting time is reduced since the write-back operation does not include multiple disk seeks.

4.2.2 JIT-seek: Preempting $\mathbf{T_{rot}}$

After the reduction of the $T_{transfer}$ component of the waiting time, the rotational delay and seek time components become significant. The rotational period (T_P) can be as much as 10 ms in current-day disk drives. To reduce the rotational delay component (T_{rot}) of the waiting time, we propose a *just-in-time seek* (*JIT-seek*) technique for IO operations.

Definition 4.2.2: The *JIT-seek* technique delays the servicing of the next IO request in such a way that the rotational delay to be incurred is minimized. We refer to the delay between two IO requests, due to JIT-seek, as *slack time*.

Benefits:

1. The slack time between two IO requests is fully preemptible. For example, suppose that an IO request must incur a T_{rot} of 5 ms, and JIT-seek delays the issuing of the disk command by 4 ms. The disk is thus idle for $T_{idle} = 4$ ms. Then, the expected waiting time is reduced from 2.5 ms to $\frac{1}{2}\frac{1 \times 1}{1+4} = 0.1$ ms. In effect, the rotational delay for the next IO operation is made preemptible.

2. The slack obtained due to JIT-seek can also be used to perform data prefetching for the previous IO or to service a background request, and hence potentially increase the disk throughput.

Overhead: *Semi-preemptible IO* predicts the rotational delay and seek time between two IO operations in order to perform JIT-seek. If there is an error in prediction, then the penalty for JIT-seek is at most one extra disk rotation and some wasted cache space for unused prefetched data.

The Method

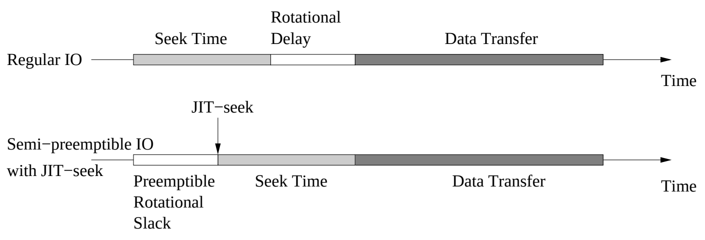


Figure 4.4: JIT-seek.

The JIT-seek method is illustrated in Figure 4.4. The x-axis depicts time, and the two horizontal lines depict a regular IO and an IO with JIT-seek, respectively. With JIT-seek, the read command for an IO operation is delayed and issued just-in-time so that the seek operation takes the disk head directly to the destination block, without incurring any rotational delay at the destination track. Hence, data transfer immediately follows the seek operation. The available rotational slack, before issuing the JIT-seek command, is now preemptible. We can make two key observations about the JIT-seek method. First, an accurate JIT-seek operation reduces the T_{rot} component of the waiting time without any loss in performance. Second, and perhaps more significantly, the ongoing IO request can be serviced as much as possible, or even completely, if sufficient slack is available before the JIT-seek operation for a higher-priority request.

The pre-seek slack made available due to the JIT-seek operation can be used in three possible ways:

- The pre-seek slack can be simply left unused. In this case, a higher-priority request arriving during the slack time can be serviced immediately.
- The slack can be used to perform additional data transfers. Operating systems can perform data prefetching for the current IO beyond the necessary data transfer. We refer to this as *free prefetching* [53]. Chunking is used for the prefetched data, to reduce the waiting time of a higher-priority request. Free prefetching can increase the disk throughput. We must point out, however, that free prefetching is useful only for sequential data streams where the prefetched data will be consumed within a short time. Operating systems can also perform another background request as proposed elsewhere [53, 71].
- The slack can be used to mask the overhead incurred in performing *seek-splitting*, which we shall discuss next.

4.2.3 Seek Splitting: Preempting $\mathbf{T_{seek}}$

The seek delay (T_{seek}) becomes the dominant component when the $T_{transfer}$ and T_{rot} components are reduced drastically. A full stroke of the disk arm may require as much as 20 ms in current-day disk drives. It may then be necessary to reduce the T_{seek} component to further reduce the waiting time.

Definition 4.2.3: *Seek-splitting* breaks a long, non-preemptible seek of the disk arm into multiple smaller sub-seeks.

Benefits: The *seek-splitting* method reduces the T_{seek} component of the waiting time. A long non-preemptible seek can be transformed into multiple shorter sub-seeks. A higher-priority request can now be serviced at the end of a sub-seek, instead of being delayed until the entire seek operation is finished. For example, suppose an IO request involves a seek of $20,000$ cylinders, requiring a T_{seek} of 14 ms. Using seek-splitting, this seek operation can be divided into two 9 ms sub-seeks of $10,000$ cylinders each. Then the expected waiting time for a higher-priority request is reduced from 7 ms to 4.5 ms.

Overhead:

1. Due to the mechanics of the disk arm, the total time required to perform multiple sub-seeks is greater than that for a single seek of a given seek distance. Thus, the seek-splitting method can degrade disk throughput. Later in this section, we discuss this issue further.

2. Splitting the seek into multiple sub-seeks increases the number of disk head accelerations and decelerations, consequently increasing power usage and noise.

The Method

To split seek operations, *Semi-preemptible IO* uses a tunable parameter, the maximum sub-seek distance. The *maximum sub-seek distance* decides whether to split a seek operation. For seek distances smaller than the maximum sub-seek distance, seek-splitting is not employed. A smaller value for the maximum sub-seek distance provides higher responsiveness at the cost of possible throughput degradation.

Unlike the previous two methods, seek-splitting may degrade disk performance. However, we note that the overhead due to seek-splitting can, in some cases, be masked. If the pre-seek slack obtained due to JIT-seek is greater than the seek overhead, then the slack can be used to mask this overhead. A specific example of this phenomenon was presented in Section 4.1. If the slack is insufficient to mask the overhead, seek-splitting can be aborted to avoid throughput degradation. Making such a tradeoff, of course, depends on the requirements of the application.

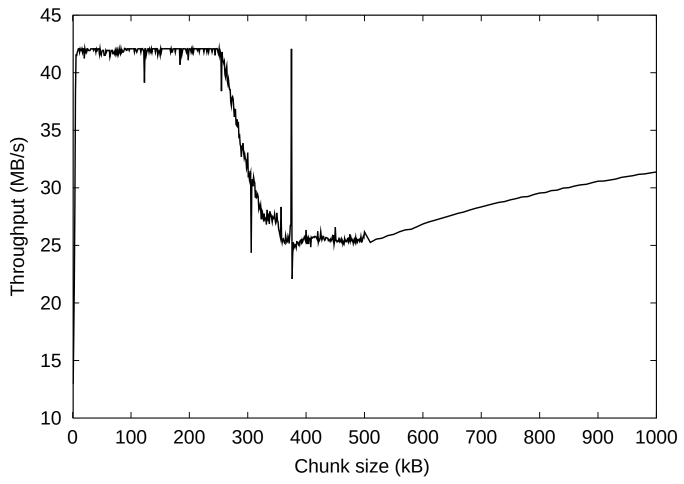
4.2.4 Disk Profiling

As mentioned in the beginning of this section, *Semi-preemptible IO* greatly relies on disk profiling to obtain accurate disk parameters. The disk profiler obtains the following required disk parameters:

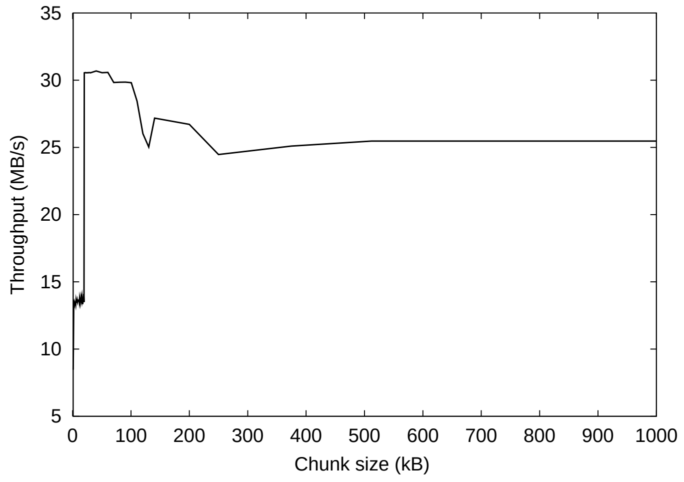
- **Disk block mappings.** The system uses disk mappings for both logical-to-physical and physical-to-logical disk block address transformation.
- **Optimal chunk size.** In order to efficiently perform chunking, *Semi-preemptible IO* chooses optimal chunk size from the optimal range extracted using the disk profiler.
- **Disk rotational factors.** In order to perform JIT-seek, the system requires an accurate rotational delay prediction, which relies on the extracted disk rotation period and rotational skew factors for disk tracks.
- **Seek curve.** JIT-seek and seek-splitting methods rely on accurate seek time prediction.

The extraction of these disk parameters is described in Chapter 3.

As regards chunking, the disk profiler provides the optimal range for the chunk size. Figure 4.5 depicts the effect of chunk size on the read throughput performance for one SCSI and one IDE disk drive. Figure 4.6 shows the same for the write case. Clearly, the optimal range for chunk size (between points a and

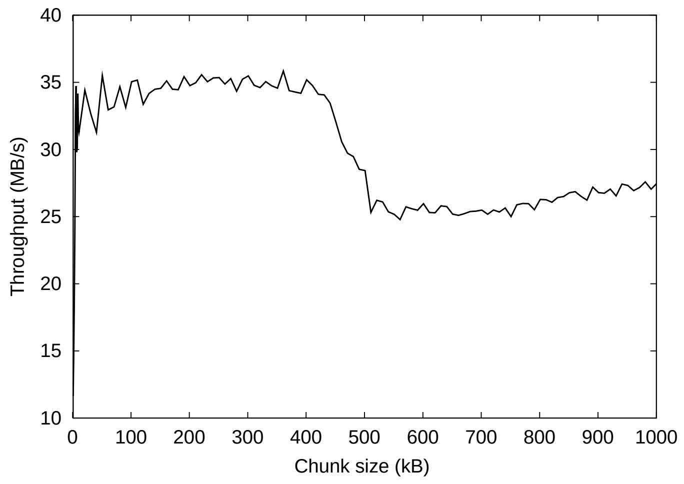


(a) SCSI ST318437LW

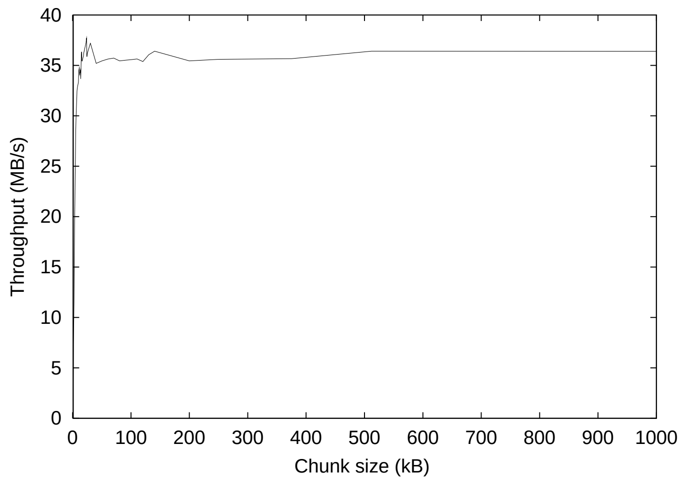


(b) IDE WD400BB

Figure 4.5: Sequential read throughput vs. chunk size.



(a) SCSI ST318437LW



(b) IDE WD400BB

Figure 4.6: Sequential write throughput vs. chunk size.

b illustrated previously in Figure 4.3) can be automatically extracted from these figures. Disk profiler implementation was successful in extracting optimal chunk size for several SCSI and IDE disk drives with which we experimented. For those who might also be interested in CPU overhead for performing chunking, we present CPU utilization when transferring a large data segment from the disk, using different chunk sizes in Figure 4.7 for an IDE disk. CPU utilization decreases rapidly as chunk size increases. Beyond a chunk size of 50 kB, the CPU utilization remains relatively constant. This figure shows that chunking, using even small chunk size (50 kB), is feasible for an IDE disk without incurring the significant CPU overhead. For SCSI disks, the CPU overhead of chunking is even less than that for IDE disks, since the bulk of the processing is done by the SCSI controller.

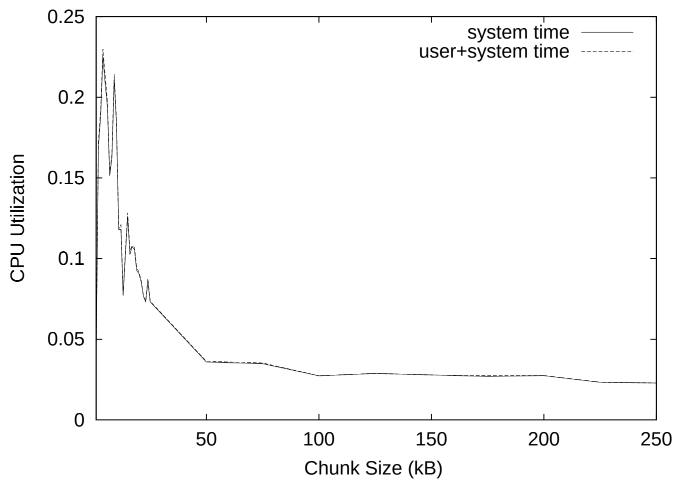


Figure 4.7: CPU utilization vs. chunk size for IDE WD400BB.

To perform JIT-seek, the system needs an accurate estimate of the seek delay between two disk blocks. The disk profiler provides the seek-time curve as well as variations in seek time. A sample seek-time curve is presented in Figure 4.8. The disk profiler also obtains the required parameters for rotational delay prediction between accessing two disk blocks in succession with near-

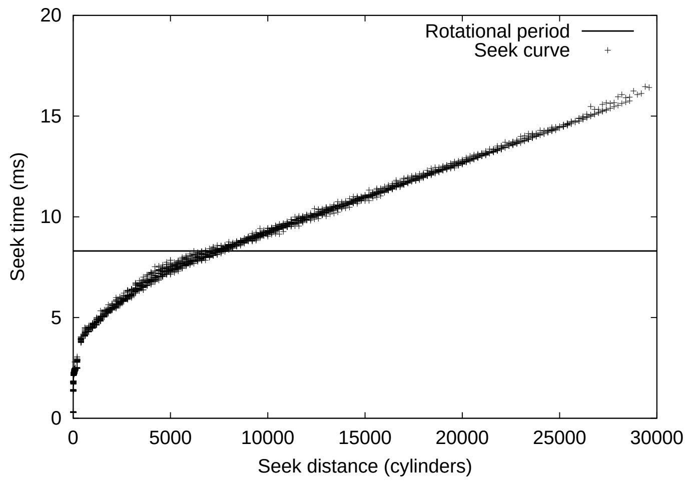


Figure 4.8: Seek curve for SCSI ST318437LW.

microsecond-level precision. However, the variations in seek time can be of the order of one millisecond, which restricts the possible accuracy of prediction. Finally, to perform JIT-seek, the system combines seek time and rotational delay prediction to predict T_{rot}. Detailed information about the prediction of T_{rot} and the disk profiling is presented in Chapter 3.

4.3 Experimental Evaluation

We now present the performance results for our implementation of *Semi-preemptible IO*. Our experiments aimed to answer the following questions:

- What is the level of *preemptibility* of *Semi-preemptible IO* and how does it influence the disk throughput?
- What are the *individual contributions* of the three components of *Semi-preemptible IO*?
- What is the effect of IO *preemption* on the average response time for higher-priority requests and disk throughput?

4.3.1 Methodology

In order to answer these questions, we have implemented a prototype system which can service IO requests using either the traditional non-preemptible method (*non-preemptible IO*) or *Semi-preemptible IO*. Our prototype runs as a user-level process in Linux and talks directly to a SCSI disk using the Linux SCSI-generic interface. The prototype uses logical-to-physical block mapping of the disk, seek curves, and rotational skew times, all of which are automatically generated by Diskbench [25]. All experiments were performed on a Pentium III 800 MHz machine with a Seagate ST318437LW SCSI disk. This SCSI disk has two tracks per cylinder, with 437 to 750 blocks per track, depending on the disk zone. The total disk capacity is 18.4 GB. The rotational speed of the disk is 7200 RPM. The maximum sequential disk throughput is between 24.3 and 41.7 MBps.

For performance benchmarking, we performed two sets of experiments. First, we tested the preemptibility of the system using simulated IO workload. For the simulated workload, we used equal-sized IO requests within each experiment. The low-priority IOs are for data located at random positions on the disk. In the experiments where we actually performed preemption, the higher-priority IO requests were also at random positions. However, their size was set to only one block in order to provide the lower estimate for preemption overhead. We tested the preemptibility under *first-come-first-serve (FCFS)* and *elevator* disk scheduling policies. In the second set of experiments we used a trace workload obtained on the tested Linux system. We acquired the traces from the instrumented Linux-kernel disk-driver. In the simulated experiments, non-preemptible IOs are serviced using chunk sizes of 128 kB. This is the size used by Linux and FreeBSD for breaking up large IOs. We assume that a large IO cannot be preempted between chunks, since such is the case for current operating systems. On the other hand, our prototype services larger IOs using multiple disk commands and preemption is possible after each disk command is completed. Based on disk profiling, our prototype used the following parameters for *Semi-preemptible IO*. Chunking divided the data transfer into chunks of 50 disk blocks each, except for the last chunk, which can be smaller. JIT-seek used an offset of 1 ms to reduce the probability of prediction errors. Seeks for more than a half of the disk size in cylinders were split into two equal-sized, smaller seeks. We used the SCSI *seek* command to perform sub-seeks.

4.3.2 Preemptibility

The experiments relating to the preemptibility of disk access measure the duration of (non-preemptible) disk commands, in both non-preemptible IO and *Semi-preemptible IO*, in the absence of higher-priority IO requests. The results include both detailed distribution of disk commands durations (and hence maximum possible waiting time) and the expected waiting time calculated using Equations 4.1 and 4.2, as explained in Section 4.2.

Generated Workload

Figure 4.9 depicts the difference in the expected waiting time between non-preemptible IO and *Semi-preemptible IO*. In this experiment, IOs were serviced for data situated at random locations on the disk, using FCFS scheduling policy. The expected waiting time for non-preemptible IOs increases linearly with IO size due to increased data-transfer time. However, the expected waiting time for *Semi-preemptible IO* actually decreases with IO size, since the disk spends more time in data transfer, which is more preemptible.

Figure 4.10 depicts improvements in the expected waiting time when the system uses an elevator-based scheduling policy. (The figure shows the results of randomly generated IO requests serviced in batches of 40.) The results are better than those of FCFS access since the elevator scheduler reduces the seek

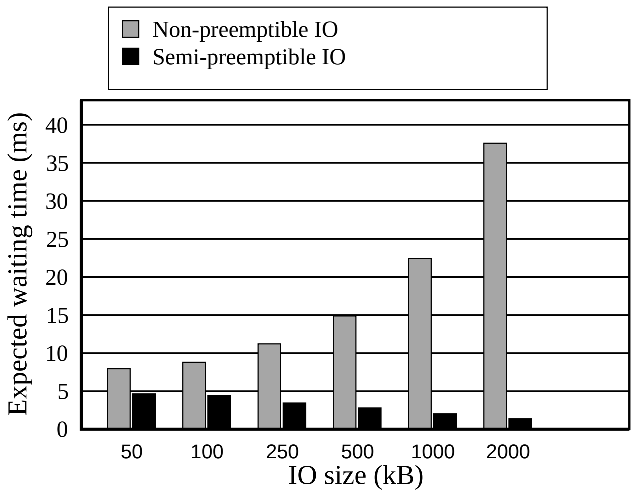


Figure 4.9: Improvements in the expected waiting time (FCFS).

component that is the least-preemptible.

Figures 4.11 and 4.12 show the effect of improving IO preemptibility on the achieved disk throughput when an FCFS scheduling policy is used. There is a noticeable reduction in disk throughput using *Semi-preemptible IO* (less than 15%). This reduction is due to the overhead of seek-splitting and mis-prediction of seek and rotational delay. More details on the accuracy of rotational delay predictions can be found in Chapter 3. Another point worth mentioning is that the reduction in disk throughput in *Semi-preemptible IO* is smaller for large IOs than for small IOs due to the reduced number of seeks and hence the smaller overhead.

Since disk commands are non-preemptible (even in *Semi-preemptible IO*), we can use the duration of disk commands to measure the expected waiting time. A smaller value implies a more preemptible system. Figure 4.13 shows the distribution of the durations of disk commands for both non-preemptible IO and *Semi-preemptible IO* (for exactly the same sequence of IO requests). In the case of non-preemptible IO (Figure 4.13 (a)), one IO request is serviced using

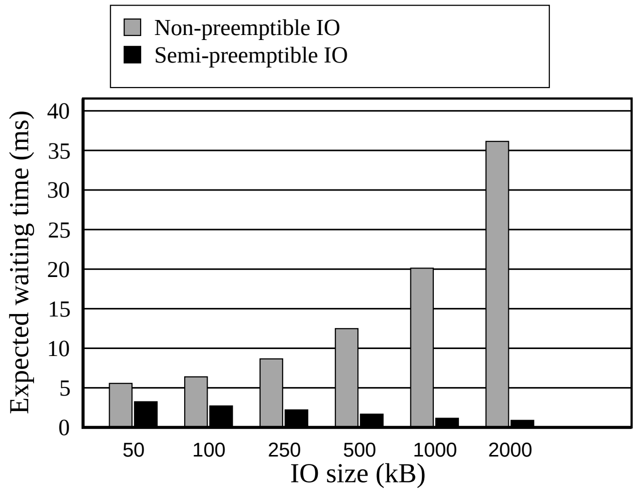


Figure 4.10: Improvements in the expected waiting time (Elevator).

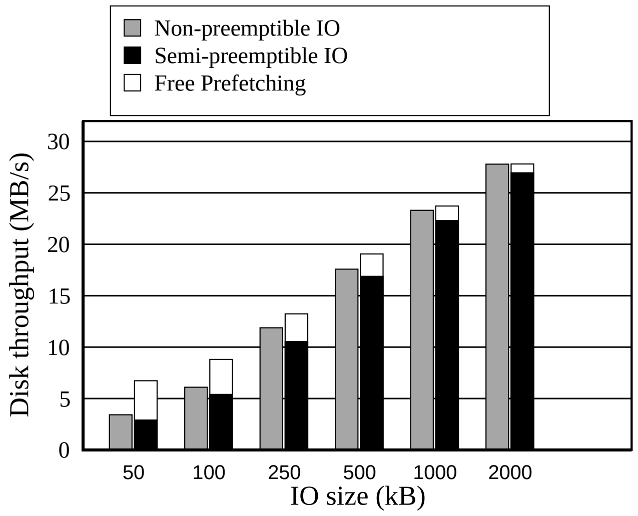


Figure 4.11: Effect on achieved disk throughput (FCFS).

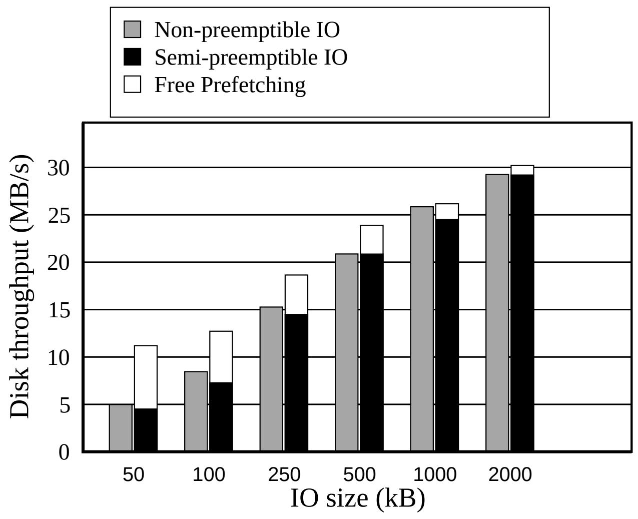


Figure 4.12: Effect on achieved disk throughput (Elevator).

a single disk command. Hence, the disk access can be preempted only when the current IO request is completed. The distribution is dense near the sum of the average seek time, rotational delay, and transfer time required to service an entire IO request. The distribution is wider when the IO requests are larger, because the duration of data transfer depends not only on the size of the IO request, but also on the throughput of the disk zone where the data resides.

In the case of *Semi-preemptible IO*, the distribution of the durations of disk commands does not directly depend on the IO request size, but on individual disk commands used to perform an IO request. (We plot the distribution for the *Semi-preemptible IO* case in logarithmic scale, so that the probability density of longer disk commands can be better visualized.) In Figure 4.13 (b), we see that for *Semi-preemptible IO*, the largest probability density is around the time required to transfer a single chunk of data. If the chunk includes the track or cylinder skew, the duration of the command will be slightly longer. (The two peaks immediately to the right of the highest peak, at approximately 2 ms, have the same probability because the disk used in our experiments has two tracks per cylinder.) The part of the distribution between 3 ms and 16 ms in the figure is due to the combined effect of JIT-seek and seek-splitting on the seek and rotational delays. The probability for this range is small, approximately 0.168, 0.056, and 0.017 for 50 kB, 500 kB, and 2 MB IO requests, respectively.

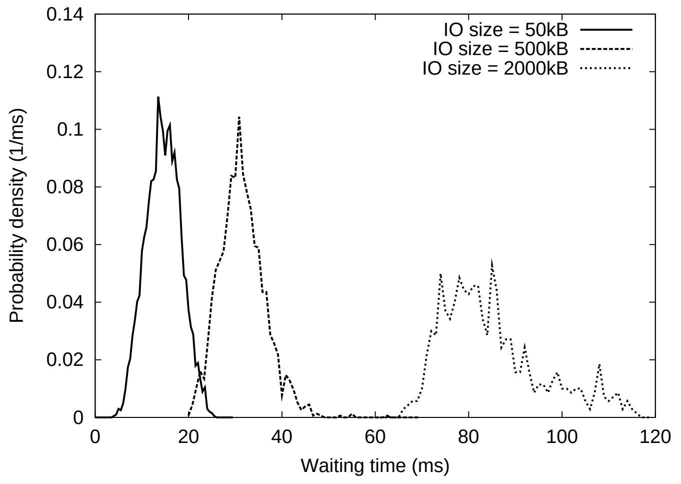
Trace Workload

Trace	Number of requests	Avg. req. size [*blocks*]	Max. block number
DV15	10800	128.7	28442272
Elevator15	10180	127.6	28429968
TPC	1376482	126.5	8005312

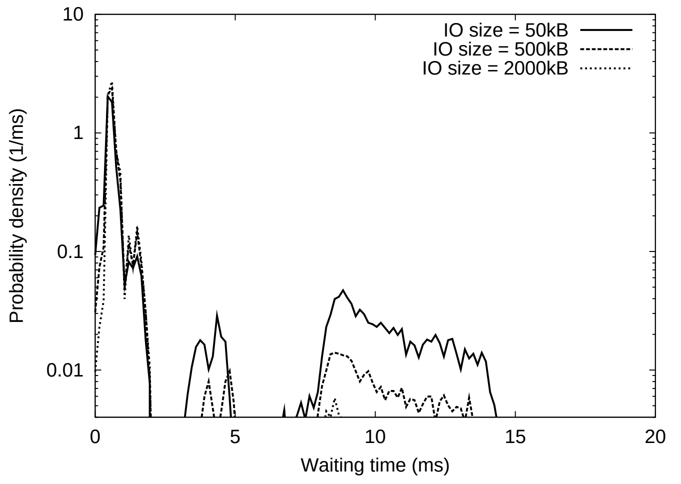
Table 4.1: Trace summary.

We now present preemptibility results using IO traces obtained from a Linux system. IO traces were obtained from three applications. The first trace (DV15) was obtained when the Xtream multimedia system [21] was servicing 15 simultaneous video clients using the FCFS disk scheduler. The second trace (Elevator15) was obtained using similar setup where Xtream lets Linux elevator scheduler to handle concurrent disk IOs. The third was a disk trace of the TPC-C database benchmark with 20 warehouses obtained from [61]. Trace summary is presented in Table 4.1.

Figures 4.14 and 4.15 show the expected waiting time and disk throughput for these trace experiments. The expected waiting time was reduced by as much as 65% (Figure 4.14) with less than 10% (Figure 4.15) loss in disk throughput for all traces. (Elevator15 had smaller throughput than DV15 because several processes were accessing the disk concurrently, which increased the total number of seeks.)



(a) Non-preemptible IO (linear scale)



(b) *Semi-preemptible IO* (logarithmic scale)

Figure 4.13: Distribution of the disk command duration (FCFS). Smaller values imply a higher preemptibility.

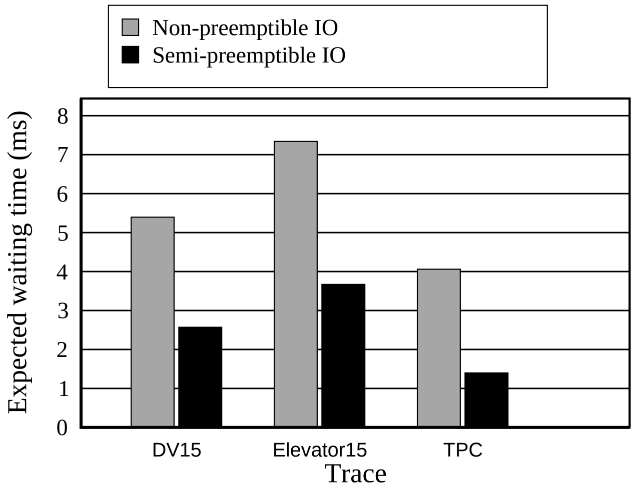


Figure 4.14: Improvement in the expected waiting time (using disk traces).

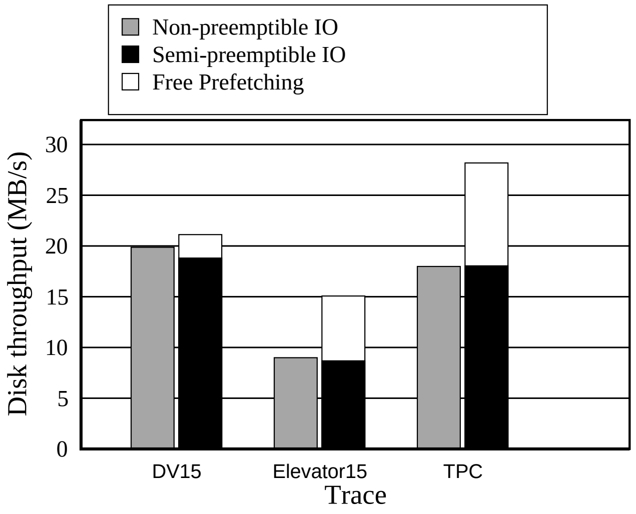


Figure 4.15: Effect on the achieved disk throughput (using disk traces).

Individual Contributions

Figure 4.16 shows the individual contributions of the three strategies with respect to expected waiting time for the random workload with the elevator scheduling policy. In Section 4.3.2, we showed that the expected waiting time can be significantly smaller in *Semi-preemptible IO* than in non-preemptible IO. Here we compare only contributions within *Semi-preemptible IO* to show the importance of each strategy. Since the time to transfer a single chunk of data is small compared to the seek time (typically less than 1 ms for a chunk transfer and 10 ms for a seek), the expected waiting time decreases as the data transfer time becomes more dominant. When the data transfer time dominates the seek and rotational delays, chunking is the most useful method for reducing the expected waiting time. When the seek and rotational delays are dominant, JIT-seek and seek-splitting become more effective in reducing the expected waiting time.

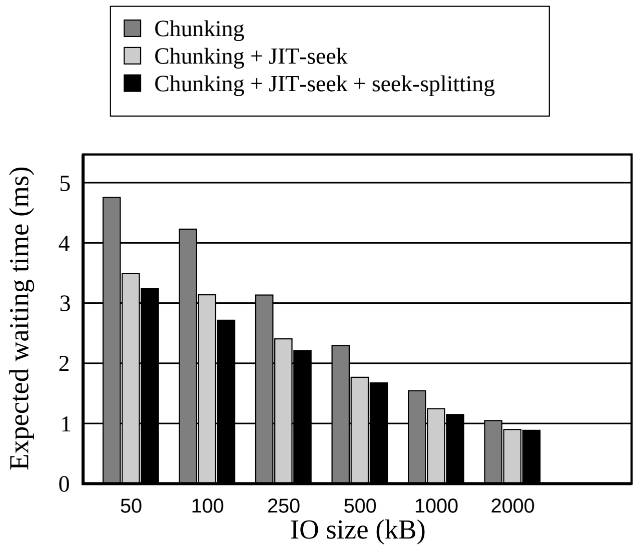


Figure 4.16: Individual contributions of *Semi-preemptible IO* components on the expected waiting time (Elevator).

Figure 4.17 summarizes the individual contributions of the three strategies

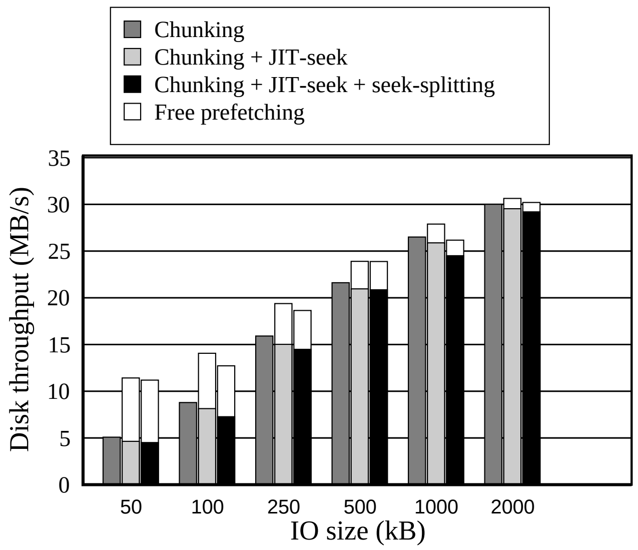


Figure 4.17: Individual effects of *Semi-preemptible IO* strategies on disk throughput (Elevator).

with respect to the achieved disk throughput. Seek-splitting can degrade disk throughput, since whenever a long seek is split, the disk requires more time to perform multiple sub-seeks. JIT-seek requires accurate prediction of the seek time and rotational delay. It introduces overhead in the case of mis-prediction. However, when the data transfer is dominant, benefits of chunking can mask both seek-splitting and JIT-seek overheads. JIT-seek aids the throughput with free prefetching. The potential free disk throughput acquired using free prefetching depends on the rate of JIT-seeks, which decreases with IO size. We believe that the free prefetching is a useful strategy for multimedia systems that often access data sequentially and hence can use most of the potential free throughput.

4.3.3 Preemptions

To estimate the response time for higher-priority IO requests, we conducted experiments wherein higher-priority requests were inserted into the IO queue at a constant rate (ν). While the constant arrival rate may seem unrealistic, the main purpose of this set of experiments is only to "estimate" the benefits and overhead associated with preempting an ongoing *Semi-preemptible IO* (in order to service a higher-priority IO request).

Preempting Read IOs

Table 4.2 presents the response time for a higher-priority request when using *Semi-preemptible IO* in two possible scenarios: (1) when the higher-priority request is serviced after the ongoing IO is completed (non-preemptible IO), and (2) when the ongoing IO is preempted to service the higher-priority IO request (*Semi-preemptible IO*). If the ongoing IO request is not preempted, then all higher-priority requests that arrive while it is being serviced must wait until the IO is completed. The results in Table 4.2 illustrate the case when the ongoing request is a read request. The results for the write case are presented in Table 4.5.

IO	ν	*Avg. Resp.* $[ms]$		*Throughput* $[MB/s]$	
$[kB]$	$[req/s]$	*npIO*	*spIO*	*npIO*	*spIO*
50	*0.5*	19.2	19.4	3.39	2.83
50	*1*	21.8	16.0	3.36	2.89
50	*2*	20.8	17.6	3.32	2.82
50	*5*	21.0	18.2	3.18	2.62
50	*10*	21.2	18.3	2.95	2.30
50	*20*	21.1	18.4	2.49	1.68
500	*0.5*	29.2	15.7	16.25	16.40
500	*1*	28.1	15.5	16.15	16.20
500	*2*	28.2	16.7	15.94	15.77
500	*5*	28.6	16.0	15.28	14.58
500	*10*	28.9	16.3	14.24	12.48
500	*20*	29.4	16.8	11.96	8.57

Table 4.2: The average response time and disk throughput for non-preemptible IO (*npIO*) and *Semi-preemptible IO* (*spIO*).

Each time a higher-priority request preempts a low-priority IO request for disk access, an extra seek is required to resume servicing the preempted IO after the higher-priority request has been completed. Table 4.2 presents the average response time and the disk throughput for different arrival rates of higher-priority requests. For the same size of low-priority IO requests, the average response time does not increase significantly with the increase in the arrival rate of higher-priority requests. However, the disk throughput does decrease with an increase in the arrival rate of higher-priority requests. As explained earlier, this reduction is expected since the overhead of IO preemption is an extra seek operation per preemption.

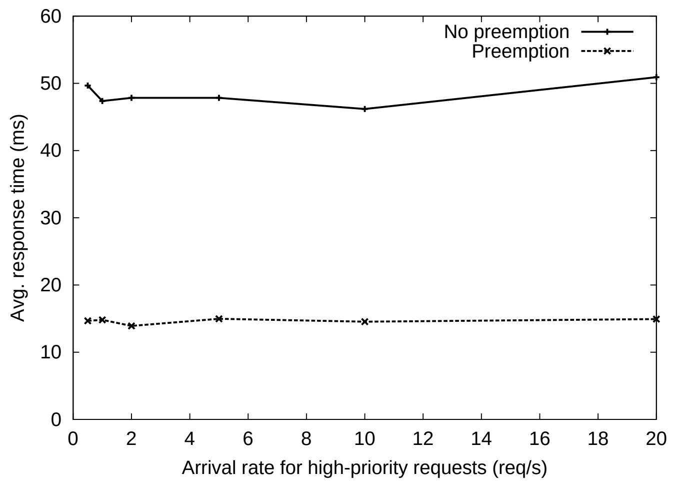


Figure 4.18: The average response time for higher-priority requests depending on their arrival rate (ν). Ongoing IO requests are $2,000$ kB each.

Figure 4.18 depicts the average response time for higher-priority requests depending on their arrival rate (ν). The low-priority requests are $2,000$ kB each. By preempting the ongoing semi-preemptible IOs, the response time for a high priority request was reduces by a factor of four. The maximum response times (not shown) for *Semi-preemptible IO* with and without preemption were measured as 34.6 ms and 150.1 ms respectively. More extensive experimental results are presented in Table 4.2.

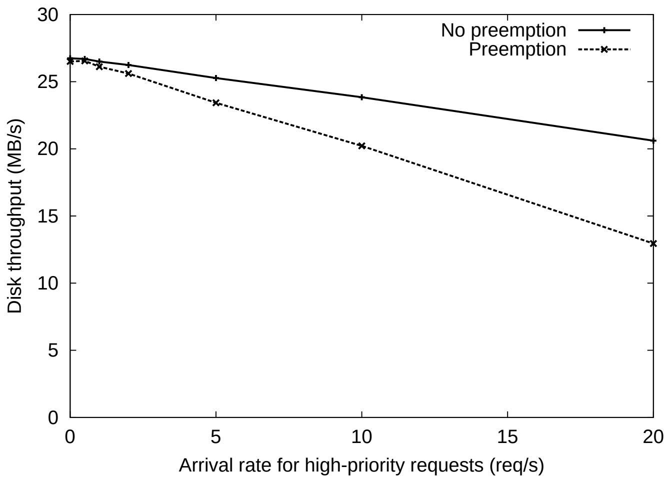


Figure 4.19: Disk throughput for $2,000$ kB IOs depending on the arrival rate of higher-priority requests (ν).

Preemption of IO requests does not come without costs. Each time a higher-priority request preempts a low-priority semi-preemptible IO, an extra seek is required to resume the preempted IO after the high-priority request completes. Figure 4.19 presents the disk throughput depending on the arrival rate of the higher-priority requests. Table 4.3 summarizes the results from Figures 4.18 and 4.19.

ν	*npIO*			*spIO*		
$[Hz]$	mean	σ	max	mean	σ	max
0.5	49.7	74.6	100.1	14.7	21.2	23.1
1	47.4	71.1	115.1	14.8	21.4	25.2
2	47.8	71.7	96.6	13.9	20.1	24.6
5	47.8	71.7	117.2	15.0	21.7	29.8
10	46.2	69.5	110.9	14.5	21.1	33.9
20	50.9	75.8	150.1	14.9	21.6	34.6

Table 4.3: Response time with and without preemption for $2,000$ kB IOs (mean, standard deviation (σ), and maximum).

Preempting Write IOs

	Exp. Waiting [ms]				*Avg. Response* [ms]			
IO	*npIO*		*spIO*		*npIO*		*spIO*	
[kB]	RD	WR	RD	WR	RD	WR	RD	WR
50	8.2	11.4	3.9	9.5	21.8	105.8	16.0	24.6
250	11.8	12.9	3.1	5.6	25.5	27.2	16.1	21.2
500	16.4	18.7	2.5	4.7	28.1	36.0	15.5	20.3
$1,000$	25.9	33.3	1.9	3.7	36.8	45.7	14.4	19.5
$2,000$	45.4	60.9	1.4	2.9	58.3	70.0	14.7	18.3

Table 4.4: The expected waiting time and average response time for non-preemptible and *Semi-preemptible IO* ($\nu = 1$ req/s).

In Section 4.2.1, we explained the difference in the preemptibility of read and write IO requests and introduced the notion of external waiting time. Table 4.4 summarizes the effect of external waiting time on the preemption of write IO requests. The arrival rate of higher-priority requests is set to $\nu = 1$ req/s. As shown in Table 4.4, the average response time for higher-priority requests for write experiments is several times longer than for read experiments. Since the higher-priority requests have the same arrival pattern in both experiments, the average seek time and rotational delay are the same for both read and write experiments. The large and often unpredictable external waiting time in the write case explains these results.

Table 4.5 presents the results of our experiments, aimed at finding out the effect of write IO preemption on the average response time for higher-priority requests and disk write throughput. For example, in the case of 50 kB write IO requests, the disk can buffer multiple requests, and the write-back operation can include multiple seek operations. *Semi-preemptible IO* succeeds in reducing external waiting time and provides substantial improvement in the response time. However, since the disk is able to efficiently reorder the buffered write requests in the case of non-preemptible IO, it achieves better disk throughput. For large IO requests, *Semi-preemptible IO* achieves write throughput comparable to that

IO	ν	*Avg. Response* $[ms]$		*Throughput* $[MB/s]$	
$[kB]$	$[req/s]$	*npIO*	*spIO*	*npIO*	*spIO*
50	*0.5*	93.1	26.9	4.85	1.98
50	*1*	105.8	24.6	4.75	1.96
50	*2*	91.1	22.7	4.68	1.94
50	*5*	102.2	24.4	4.40	1.84
50	*10*	87.5	23.7	3.95	1.70
50	*20*	81.3	23.3	3.09	1.42
500	*0.5*	32.4	20.3	13.71	11.41
500	*1*	36.0	20.3	13.64	11.24
500	*2*	35.0	20.8	13.45	11.02
500	*5*	34.9	20.5	12.82	10.36
500	*10*	36.6	20.3	11.67	9.13
500	*20*	34.6	20.7	9.64	6.92

Table 4.5: The average response time and disk write throughput for non-preemptible and *Semi-preemptible IO*.

of non-preemptible IO. We suggest that write preemption can be disabled when it is essential to maintain the high system throughput, and the disk reordering is useful (reordering could also be done in the operating system scheduler using low-level disk knowledge).

4.3.4 Summary of Results

First, we showed that *Semi-preemptible IO* can reduce the expected waiting time (hence, improve disk preemptibility) to about $3ms$ for small IOs (of the order of 50 kB) and to about 1 ms for larger IOs (of the order of 1 MB and more), with little loss in disk throughput. The expected waiting time for non-preemptible access is about 5 ms and 20 ms, respectively.

Second, we found out that read and write disk access experiences different levels of preemptibility. *Semi-preemptible IO* must consider additional external waiting time when servicing write IOs.

Third, we found out that current disk drives do not necessarily provide the optimal sequential throughput for all chunk sizes (Section 4.2.4). Disk schedulers must first profile the disk and then ensure that they use the optimal chunk size for their sequential accesses.

4.4 Summary

In this chapter, we have presented the design of *Semi-preemptible IO*, and proposed three techniques for reducing IO waiting-time — data transfer chunking, just-in-time seek, and seek-splitting. These techniques enable the preemption of a disk IO request, and thus substantially reduce the waiting time for a competing higher-priority IO request. Using both synthetic and trace workloads, we have shown that these techniques can be efficiently implemented, given detailed disk parameters. Our empirical studies have shown that *Semi-preemptible IO* can reduce the waiting time for both read and write requests significantly when compared with non-preemptible IOs.

We believe that preemptible IO can especially benefit multimedia and real-time systems, which are delay-sensitive and which issue large-size IOs for meeting real-time constraints. We are currently implementing *Semi-preemptible IO* in Linux kernel. We plan to further study its performance impact on traditional and real-time disk-scheduling algorithms.

Chapter 5

Preemptive RAID Scheduling

In this chapter we investigate the effectiveness of preemptive disk-scheduling algorithms to achieve better quality of service (QoS). We present an architecture for QoS-aware RAID systems based on Semi-preemptible IO [20, 22]. We show *when* and *how* to preempt IOs to improve the overall performance of the RAID. Using our simulator for preemptible RAID systems, we evaluate the benefits and estimate the overhead of the proposed approach.

5.1 Introduction

Emerging applications such as video surveillance, large-scale sensor networks, storage-bound Web applications, and virtual reality require high-capacity, high-bandwidth RAID storage to support high-volume IOs. All these applications typically access large sequential data-segments to achieve high disk throughput. In addition to high-throughput non-interactive traffic, these applications also service a large number of interactive requests, requiring a short response time. The deployment of high-bandwidth networks promised by research projects such as OptIPuter[87] will further magnify the access-time bottleneck of a remote RAID store, inevitably making the access-time reduction increasingly important.

What is the worst-case disk-access time, and how can it be mitigated? On an idle disk, the access time is composed of a seek and a rotational delay. However, when the disk is servicing an IO, a new interactive IO, requiring a short response time, must wait at least until after the ongoing IO has been completed. For the applications mentioned earlier, the typical IO sizes are of the order of a few megabytes. For example, while concurrently servicing interactive queries, the Google File System [35] stores data in 64 MB chunks and video surveillance systems [17, 70] record video segments of several megabytes each. Another example is a virtual-reality flight simulator from the TerraFly project [16], which continuously streams the image data for multiple users from their database of satellite images. Simultaneously, the system must support interactive user operations.

In this chapter, we introduce preemptive RAID scheduling, or Praid. In Semi-preemptible IO [22] we investigated the preemptibility of disk access. In addition to Semi-preemptible IO, Praid provides 1) *preemption* mechanisms to allow the ongoing IOs to be preempted, and 2) *resumption* mechanisms to resume preempted IOs on same or different disks. We also propose scheduling policies to decide whether and when to preempt, for maximizing the *yield*, or the total value, of the schedule. Since the yield of an IO is application- and user-defined, our scheduler maps external value propositions to internal yields, producing a schedule that can maximize total external value for all IOs, pending and current.

5.1.1 Illustrative Example

We now present an example to show how preemptive scheduling works, and why it can outperform a traditional priority-based scheduling policy. Suppose that the disk is servicing a long sequential write when a higher priority read IO arrives. The new IO can arrive at either time t_1 or t_2, as depicted in Figure 5.1. If the write IO has been buffered in a non-volatile RAID buffer[1], the IO can be preempted to service the new request. The preempted write IO is delayed, to be serviced at a later time. When the write IO is resumed, additional disk overhead is incurred. We refer to this overhead as a *preemption overhead.*

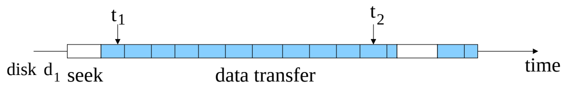


Figure 5.1: Sequential disk access.

Now, a simple priority-based scheduler will always preempt the long sequential write access (and incur a preemption overhead) regardless of whether the read IO arrives at time t_1 or t_2. However, preempting the write access at t_2 may not be profitable, since the write is nearly completed. Such a preemption is likely to be counter-productive — not gaining much in response time, but incurring preemption overhead. Our Praid scheme is able to discern whether and when a preemption should take place.

The above example shows just one simple scenario where additional mechanisms can lead to performance gains for RAID systems. In the rest of this chapter, we will detail our preemption mechanisms and scheduling policies.

[1]Most current RAID systems are equipped with a large non-volatile buffer. Write IOs are reported to the operating system as serviced, as soon as the IO data is copied into this buffer.

5.1.2 Contributions

In addition to the overall approach, the specific contributions of this chapter can be summarized as follows:

- *Preemption mechanisms.* We introduce two methods to preempt disk IOs in RAID systems — JIT-preemption and JIT-migration. Both methods are used by the preemptive schedulers (presented in this chapter) to simplify preemption decisions.

- *Preemptible RAID policies.* We propose scheduling methods which aim to maximize the total QoS value (each IO is tagged with a yield function) and use this metric to decide whether IO preemption is beneficial or not.

- *System architecture for preemptible RAID systems.* We introduce an architecture for QoS-aware RAID systems based on the preemptible framework. We implement a simulator for these systems (PraidSim) that is used in evaluating our approach.

The rest of this chapter is organized as follows: Section 5.2 introduces the preemption methods used for preemptive RAID scheduling. Section 5.3 presents the preemptible-RAID system architecture and the scheduling framework. In Section 5.4, we present our experimental environment and evaluate different scheduling approaches using simulation. We summarize and suggest directions for future work in Section 5.5.

5.2 Mechanisms

In this section we introduce methods for IO preemption and resumption. We first summarize the Semi-preemptible IO introduced in Chapter 4. We then explain the following three mechanisms for IO preemption: 1) JIT-preemption with IO resumption at the same disk, 2) JIT-preemption with migration of the ongoing IO to a different disk (favoring the newly arrived IO), and 3) preemption with JIT-migration of the ongoing IO (favoring the ongoing IO).

Semi-preemptible IO

Semi-preemptible IO [22] maps each IO request into multiple fast-executing (and hence short-duration) disk commands using three methods. (The ongoing IO request can be preempted between these disk commands.) Each of these three methods addresses the reduction of one of the following IO components: $T_{transfer}$ (denoting transfer time), T_{rot} (denoting rotational delay), and T_{seek} (denoting seek time).

1. *Chunking* $T_{transfer}$. A large IO transfer is divided into a number of small chunk transfers, and preemption is made possible between the small transfers. If the IO is not preempted between chunk transfers, chunking does not incur any overhead. This is due to the prefetching mechanism in current disk drives.

2. *Preempting* T_{rot}. By performing just-in-time (JIT) seek for servicing an IO request, the rotational delay at the destination track is virtually eliminated. The pre-seek slack time thus obtained is preemptible. This slack can also be used to perform prefetching for the ongoing IO request, and/or to perform seek splitting.

3. *Splitting* T_{seek}. Semi-preemptible IO splits a long seek into sub-seeks, and permits preemption between two sub-seeks.

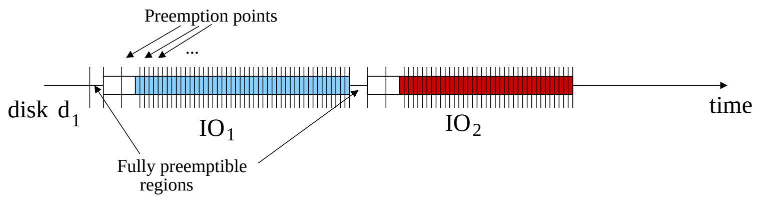


Figure 5.2: Possible preemption points for semi-preemptible IO.

Figure 5.2 shows the possible preemption points while servicing a semi-preemptible IO. Preemption is possible only after completion of any disk command or during disk idle time. The regions before the JIT-seek operation are fully preemptible (since no disk command is issued). The seek operations are the least preemptible, and the data transfer phase is highly preemptible (preemption is possible after servicing each chunk, which is about 0.5 ms).[2]

5.2.1 JIT-preemption

When the disk scheduler decides that preempting and delaying an ongoing IO would yield a better overall schedule, the IO should be preempted using *JIT-preemption*. This is a local decision, meaning that a request for the remaining portion of the preempted IO is placed back in the local queue, and resumed later on the same disk (or dropped completely[3]).

Definition 5.2.1: *JIT-preemption* is a method for preempting an ongoing semi-preemptible IO at points that minimize the rotational delay at the destination track (for the higher-priority IO which is serviced next). The scheduler decides when to preempt the ongoing IO using knowledge about available JIT-preemption points. These points are roughly one disk rotation apart.

Preemption: This method relies on JIT-seek (described in Semi-preemptible

[2]If we know in advance when to preempt the ongoing IO, we can choose the size for the last data-transfer chunk before preemption, and further tune the desired preemption point.

[3]For example, the scheduler may drop unsuccessful speculative reads, cache-prefetch operations, or preempted IOs whose deadlines have expired.

IO [22]), which requires rotational delay prediction (also used in other disk schedulers [45, 53]). JIT-preemption is similar to free-prefetching [53]. However, if the preempted IO will be completed later, then the JIT-preemption always yields useful data transfer (prefetching may or may not be useful).[4]

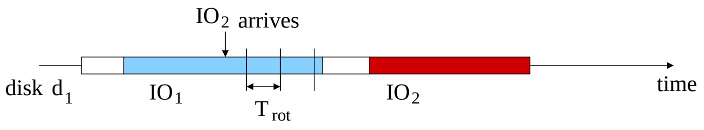


Figure 5.3: Possible JIT-preemption points.

Figure 5.3 depicts the positions of possible JIT-preemption points. If IO_1 is preempted anywhere between two such points that are adjacent, the resulting service time for IO_2 would be exactly the same as in a situation where the preemption is delayed until the next possible JIT-preemption point. This is because the rotational delay at the destination track varies depending on when the seek operation starts. The rotational delay is minimal at the JIT-preemption points, which are roughly one disk rotation apart.

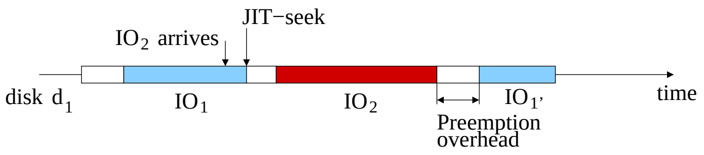


Figure 5.4: JIT-preemption during data transfer.

Figure 5.4 depicts the case where the ongoing IO_1 is preempted during its data transfer phase in order to service IO_2. In this case, the first available JIT-preemption point is chosen. The white regions represent the access-time overhead (seek time and rotational delay for an IO). Since JIT-seek minimizes

[4]Another difference is that JIT-preemption can also be used for write IOs, although its implementation outside of disk firmware is more difficult for write IOs than it is for read IOs [22].

rotational delay for IO_2, its access-time overhead is reduced for the case with JIT-preemption (compared to the no-preemption case depicted in Figure 5.3).
Resumption: The preempted IO is resumed later on the same disk. Preemption overhead (depicted in Figure 5.4) is the additional seek time and rotational delay required to resume the preempted IO_1. Depending on the scheduling decision, IO_1 may be resumed immediately after IO_2 completes, at some later time, or never (it is dropped and does not complete). We explain scheduling decisions in detail later in Section 5.3.3.

5.2.2 JIT-preemption with Migration

RAID systems duplicate data for deliberate redundancy. If an ongoing IO can also be serviced at some other disk that holds a copy of the data, the scheduler has the option of preempting the IO and migrating its remaining portion to the other disk. In the traditional static RAIDs, this situation can happen in RAID levels 1 and 0/1 [14] (mirrored or mirrored/striped configuration). It might also happen in reconfigurable RAID systems (for example, HP AutoRAID [100]), in object-based RAID storage [58], or in non-traditional large-scale software RAIDs [35].
Definition 5.2.2: *JIT-preemption-with-migration* is a method for preempting the ongoing IO and migrating it to a different disk in a fashion that minimizes service time for newly arrived IO.
Preemption: For preemption, this method relies on the previously described JIT-preemption. Figure 5.5 depicts the case when it is possible to use JIT-preemption to promptly service IO_2, while migrating IO_1 to another disk. Preemption overhead is in the form of additional seek time and rotational delay required for the completion of IO_1 at the replica disk.
Resumption: The preempted IO is resumed later on the disk to which it was migrated. The preempted IO enters the scheduling queue of the mirror disk and

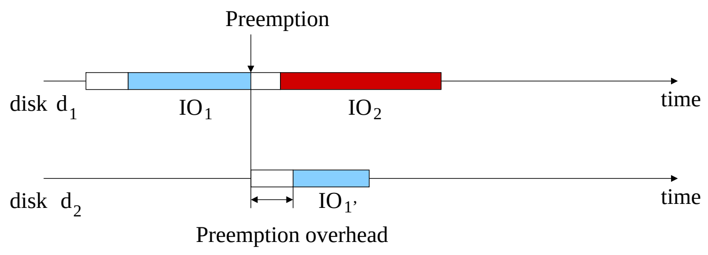


Figure 5.5: JIT-preemption with migration.

is serviced according to the single-disk scheduling policy. Preemption overhead exists only at the mirror disk. This suggests that this method may be able to improve the schedule when load balance is hard to achieve.

5.2.3 JIT-migration

When a scheduler decides to migrate the preempted IO to another disk with a copy of the data, it can choose to favor the newly arrived IO or the ongoing IO. The former uses JIT-preemption introduced earlier, but migrates the remaining portion of the preempted IO to the queue of some other disk holding the data. The latter uses *JIT-migration*.

Definition 5.2.3: *JIT-migration* is a method for preempting and migrating an ongoing IO in a fashion that minimizes its service time. The ongoing IO is preempted at the moment when the destination disk starts performing data-transfer for the remaining portion of the IO. The original IO is then preempted, but its completion time is not delayed.

Preemption: JIT-migration also relies on JIT-seek and is used to preempt and migrate the ongoing IO only if it does not increase its service time thereby favoring the ongoing IO.

Figure 5.6 depicts the case when the ongoing IO (IO_1) is more important than the newly arrived IO (IO_2). However, if the disk with the replica is idle or servicing less important IOs, we can still reduce the service time for IO_2. As

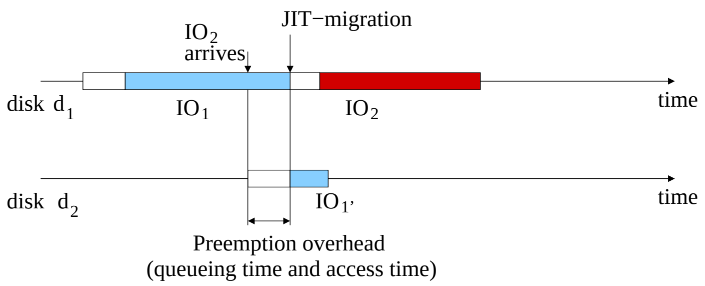


Figure 5.6: Preemption with JIT-migration.

soon as IO_2 arrives, the scheduler can issue a speculative migration to another disk with a copy of the data. When the data transfer is ready to begin at the other disk, the scheduler can migrate the remaining portion of IO_1 at the desired moment. Since the disks are not necessarily rotating in unison, the IO_1 can be serviced only at approximately the same time when compared with the no-preemption case. The preemption delay for IO_1 depends on the queue at the disk with the replica. If the disk with the replica is idle, the delay will be of the order of 10 ms (equivalent to the access-time overhead).

Resumption: In the case of JIT-migration, IO_1 is not preempted until the disk with the mirror is ready to continue its data transfer. Again, the preemption overhead exists only at the mirror disk signifying the possibility of improvement in the presence of a load-imbalance.

5.3 Architecture

In this section, we first present a high-level system architecture for RAID systems with support for preemptive disk scheduling. We then explain global (RAID) and local (single-disk) scheduling approaches. All scheduling methods presented within this framework are designed to be implemented in the firmware for hardware RAID controllers or in the OS driver for software RAIDs.

5.3.1 PRAID System Architecture

Figure 5.7 depicts a simplified architecture of preemptible RAID systems. The main system components are the external IO interface, the RAID controller, and the attached disks. The components of the RAID controller are the RAID scheduler, the single-disk schedulers (one for each disk in the array), the RAID cache (both the volatile read cache and the non-volatile write buffer), and the RAID reconfiguration manager.

External IOs are issued by the IO scheduler external to the RAID system (for example, the operating system's disk scheduler). These IOs are tagged with their QoS requirements, so that the RAID scheduler can optimize their scheduling. The external IOs may also be left untagged, making them best-effort IOs. We have extended a Linux kernel to enable such an IO interface [27].

The *RAID scheduler* maps *external IOs* to *internal IOs* and dispatches them to appropriate single-disk scheduling queues. Internal IOs are also generated by the RAID reconfiguration manager for various maintenance, reconfiguration, or failure-recovery procedures.

Internal IOs are IOs which reside in the scheduling queues of individual disks. They are tagged with internally generated yield functions, and serviced using *Semi-preemptible IO*. The RAID scheduler and the local single-disk schedulers reside on the same RAID controller, and communication between them is fast

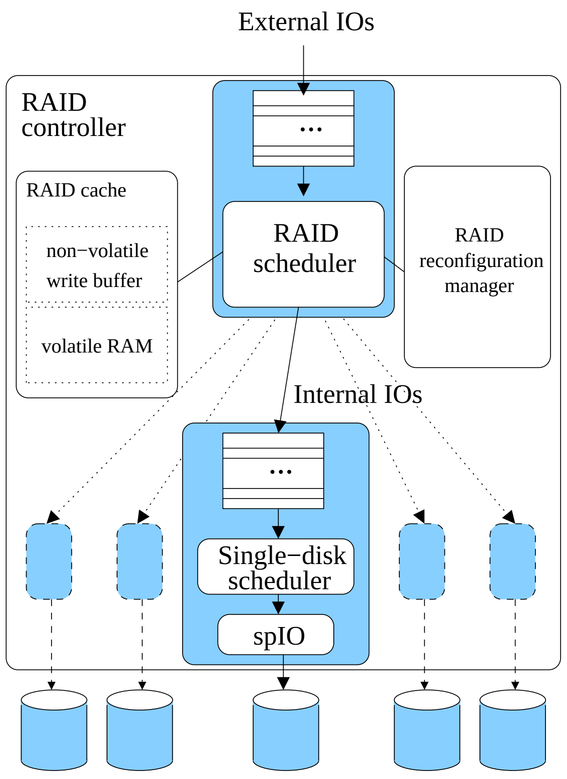


Figure 5.7: A simplified Preemptible RAID architecture.

and cheap.[5]

Single-disk schedulers make local scheduling decisions for internal IOs waiting to be serviced at a disk. Internal IOs are semi-preemptible, and single-disk schedulers can decide to preempt ongoing internal IOs. Since the communication between individual disk schedulers is efficient, single-disk schedulers in the same RAID group cooperate to improve the overall QoS-value for the entire system.

The *RAID cache* consists of both volatile memory for caching read IO data

[5]The assumption of efficient communication between single-disk schedulers holds for most RAID systems implemented as a single box, which is typically the case for current RAID systems. We use this assumption for efficient migration of internal IOs from one disk to another.

and non-volatile memory for buffering write IO data. The non-volatile memory is typically implemented as battery-backed RAM in most currently used RAIDs. The *RAID reconfiguration manager* controls and optimizes internal data organization within the RAID system. For instance, in HP AutoRAID systems [100], the reconfiguration manager can dynamically reconfigure the data to optimize for the performance (between RAID 0/1 and RAID 5 configurations) or migrate the data to hot-swap disks (in case of disk failures). These operations create additional internal IOs within the RAID system.

5.3.2 Global RAID Scheduling

The global RAID scheduler is responsible for mapping external IOs to internal IOs and for dispatching internal IOs to appropriate single-disk scheduling queues.

External IOs

In this chapter we refer to IO requests generated by a file system outside of the RAID system as external IOs. They can be tagged with the application-specified QoS class or can be left as regular, best-effort requests.[6]

Our approach for providing QoS hints to the disk scheduler is to enable applications to specify desired QoS parameters per each file descriptor. Internally, we pass the pointer to these QoS parameters along with each IO request in the disk queue. After the *open()* system call, file accesses get assigned the default best-effort QoS class. We introduce several new *ioctl()* commands which enable an application to set up different QoS parameters for its open files. These additional *ioctl()* commands are summarized in Table 5.1.

[6]Most commodity operating systems still do not provide such an interface. However, several research prototypes have implemented QoS extensions for commodity operating systems [59, 90, 93, 27]

Ioctl command	Argument	Description
IO_GET_QOS	**struct ucsb_io ***	Get file's QoS
IO_BESTEFFORT		Set best-effort class
IO_QOS_CLASS	**int *class**	Set IO's QoS class
IO_PRIORITY	**int *priority**	Set IO's priority
IO_DEADLINE	**int *deadline**	Set IO's deadline

Table 5.1: Additional *ioctl()* commands.

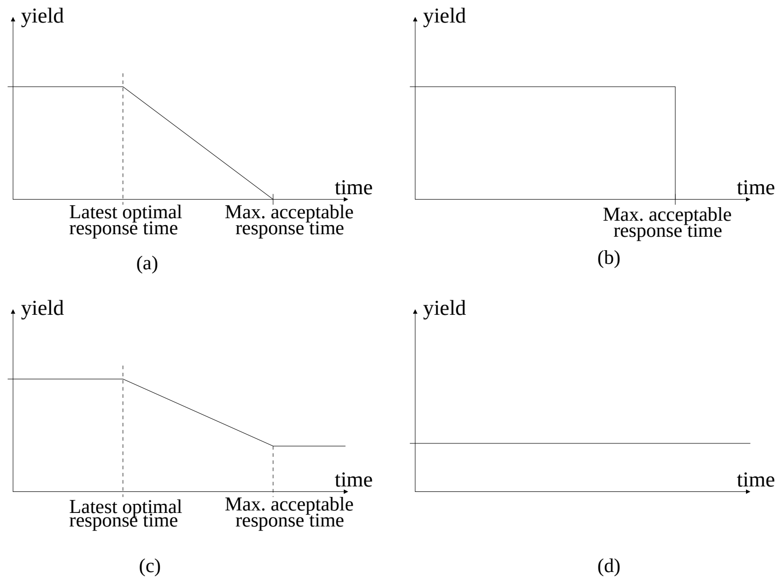


Figure 5.8: Yield functions: (a) interactive real-time IO, (b) hard real-time IO, (c) interactive best-effort IO, and (d) best-effort IO. (The exact values depend on the actual implementation.)

The yield function attached to an external IO determines the QoS value added to the system upon its completion. Figure 5.8 depicts four possible yield functions that we use in this study. Functions (a) and (b) represent the case when a hard deadline is associated with servicing the IO. If the deadline is missed, the IO should be dropped since its completion does not yield any value.[7]

[7]The option of dropping an IO request at the storage level is not widely used in today's systems. Additional handling might be needed at the user level. However, the current interface

Servicing best-effort IOs always yields some QoS value, and these IOs should not be dropped. We must point out that the yield functions presented here are not the only ones possible. The framework enables specifying one "user-defined" yield function for each QoS class, which is part of our future work.

To customize the yield ($y_{ext}(t)$) function for each external IO, we use a generic yield function for each QoS class ($yield(t)$ from Figure 5.8) and four additional parameters. The additional parameters are: time when the external IO is submitted (t_{start}), IO size ($size$), IO priority (p), and IO deadline (T_{deadln}). We assign more value to a larger and higher-priority IO using a linear approach. Our system provides an option for the OS or user-level applications to customize yield functions according to the following equation (P_{def} denotes the default priority):

$$y_{ext}(t) = size \times \frac{p}{P_{def}} \times yield\left(\frac{t - t_{start}}{T_{deadln}}\right). \tag{5.1}$$

For example, if the OS wants to give more QoS value to particular IOs, it would then assign a priority that is greater than the default one. If the OS wants to stretch the yield function (from Figure 5.8), it would then assign a longer deadline. Finally, if the OS wants to specify the same yield function for all IOs independent of their size, it would then assign different priorities (higher priority for shorter IOs and lower priority for longer IOs).[8]

RAID Scheduler

The most important task that the RAID scheduler performs is mapping external IOs to internal IOs. Internal IOs are also generated by the RAID reconfiguration manager, and scheduled to appropriate local-disk queues by the RAID scheduler. Each external IO (*parent IO*) is mapped to a set of internal

need not be changed, since systems can use the existing error-handling mechanisms.

[8]In real systems, additional QoS classes for same-importance IOs may be favorable.

IOs (*child IOs*). To perform this mapping, the RAID scheduler has to be aware of the low-level placement of data on the RAID system.

The RAID scheduler has a global view of the load on each of the disks in the array. For read IOs, the internal IO can be scheduled to any disk containing a copy of the data. The scheduler can choose the least-loaded disk or use a round-robin strategy. For write IOs, the internal IOs are dispatched to all disks where duplicate copies are located. To maintain a consistent view, the segment in the non-volatile RAID buffer is not freed until all its internal IOs finish.

The RAID scheduler makes the following scheduling decisions to dispatch internal IOs to corresponding local-disk scheduling queues:

- *Read splitting.* To further reduce response time for interactive read requests, the RAID scheduler may split the read request into as many parts as there are disks with copies of the data, issuing each part to a different disk. The read request might be completed faster by utilizing all possible disks. However, this involves more disk-seek overhead. The advantage of having QoS values over the traditional RAIDs enables preemptible RAIDs to split only interactive IOs (when additional seek overhead leads to better QoS).

- *Speculative scheduling.* Apart from dispatching read requests to the least-loaded disk, the RAID scheduler might also dispatch the same request with best-effort priority to other disks which hold a copy of the data requested. This is done in the hope that if a more loaded disk manages to clear its load earlier, then the read request can be serviced sooner.

5.3.3 Local Disk Scheduling

Using a local disk scheduling algorithm, single-disk schedulers dispatch internal (semi-preemptible) IOs and decide about IO preemptions.

Internal IOs

We refer to IO requests generated within the RAID system as internal IOs. These IOs are generated by the RAID firmware and managed by the RAID system itself. Usually, multiple internal IO requests (for several disks) must be issued to service each external IO. Requests related to data parity management, RAID auto reconfiguration, data migration, and data recovery are independently generated by the RAID reconfiguration manager, and they are not associated with any external IO. Each internal IO is tagged with its own descriptor. The internal IO descriptor is summarized in Table 5.2. The deadline and the yield function for the parent IO are used to (1) give more local-scheduling priority to earlier deadlines and (2) drop the internal IO after its hard deadline expires.

Attribute	Description
Starting block	Logical number for 1^{st} data block
IO Size	The internal IO size in disk blocks
Parent's IO value	The external IO value (from Eq. 5.1)
Parent's deadline	The external IO deadline
Parent's IO size	The remaining external IO size

Table 5.2: Internal IO descriptor.

Single-disk Scheduler

For external IOs whose value deteriorates rapidly with time, a disk scheduler may benefit if it preempts less urgent IOs. In traditional systems this is usually accomplished by bounding the size of disk IOs to relatively small values and using non-preemptive priority scheduling (for example, Linux 2.4 and 2.6 kernels use 128 kB as maximum IO size). However, this approach has two disadvantages. First, it greatly increases the number of disk IOs[9] in the scheduling queue, which might complicate the implementation of sophisticated QoS-aware schedulers and increase their computational requirements. Second, the schedulers rarely account for the overhead of disrupting sequential disk access, since they do not actually *preempt* the low-level disk IOs.

In this chapter, we present a scheduler that uses an explicitly preemptible approach, which does not need to bound maximum size for low-level disk IOs (for example, a single 8 MB IO does not need to be split into eighty 128 kB low-level disk IOs). The scheduler explicitly considers the overhead of disrupting sequential accesses and whenever it chooses to preempt the ongoing IO, the expected waiting time is substantially shorter than in the case of traditional non-preemptible IOs [22].

The single-disk scheduler maintains a queue of all internal IOs for a particular disk. The components of the internal IO response time are *waiting time* and *service time*. The waiting time is the amount of time that the request spends in the scheduling queue. The service time is the time required by the disk to complete the scheduled request, consisting of access latency (seek time and rotational delay) and data transfer time.

Internal scheduling values: The completion of an internal IO yields some QoS value for the RAID system. However, it is hard to estimate this value. First, external QoS value is generated only after the completion of the last internal IO due for a parent external IO. Second, when performing write-back operations for buffered write IOs, their external QoS value has already been harvested. However, not servicing these internal IOs implies that servicing future write IOs will suffer when the write buffer gets filled up. Third, internally generated IOs (for example, due to the RAID reconfiguration manager) must be serviced although their completion does not yield any immediate external QoS value.

[9]The number of low-level IOs for each application-generated IO might be one or two orders of magnitude greater for systems that bound the maximum IO size.

Although we do not always know the QoS value generated due to the completion of an internal read IO, we can estimate it using the following approach. When the scheduler decides to schedule an internal IO, it predicts the service time for the IO ($T_{service}$).[10] Let $y_{ext}(t)$ be the value function for the parent IO, as defined in Equation 5.1. Let $size_{int}$ denote the size of the internal IO, and $size_{remain}$ denote the remaining size of the parent IO. We estimate the scheduling value for the internal read IO (y_{int_read}) using the following heuristic:

$$y_{int\_read} = y_{ext}(t + T_{service}) \times \frac{size_{int}}{size_{remain}} \ . \tag{5.2}$$

The reasoning behind Equation 5.2 is to give more scheduling value (and hence higher priority) to internal IOs for soon-to-complete external IOs. This is necessary since we do not gain any external value from servicing internal IOs until we service the whole parent external IO. Servicing a small internal IO for a large external IO should have low priority. However, servicing a small internal IO as the last fragment for a large, nearly-completed external IO should have high priority. This is achieved by giving more internal yields for IOs whose $size_{remain}$ diminishes faster.[11]

Figure 5.9 depicts the dynamic nature of the scheduling value for internal write IOs. Unlike internal read IOs, the scheduling value of internal write IOs does not depend directly on the value of the corresponding external IOs. The idea is to drain a *nearly-full* write buffer at a faster rate, and to drain a *nearly-empty* write buffer at a slower rate. Additionally, if the buffer is full, we need to increase the draining rate depending on the value of pending IO requests. Whenever the RAID system services a new external write IO, the non-volatile write buffer space decreases, and performing write-back operations gain more

[10]Performing this prediction does not incur additional overhead since it is already required by Semi-preemptible IO [22].

[11]This is just one of several possible heuristics to address the problem. More detailed study in this regard is part of our future work.

importance. Hence, we increase the scheduling value for write IOs. Whenever the last internal write IO for a particular external IO completes, its data is flushed from the non-volatile buffer, making more space available. This reduces the importance of write-back operations, and thereby decreases the scheduling value for internal write IOs.

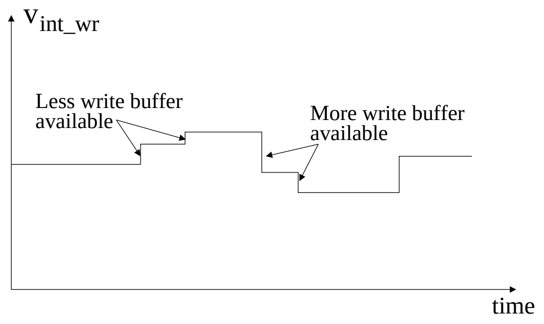


Figure 5.9: Scheduling value for internal write IOs.

In estimating the scheduling value for internal write IOs, we need to consider both the available non-volatile buffer space and all the pending external write IOs when the buffer is full. Let $I_{wr}(space)$ denote the value of freeing space in the non-volatile buffer (it is a function of the buffer utilization). Let $y_{ext}^{wr_i}(t)$ denote the value of the i^{th} external write IO waiting to be buffered. Let $size_{remain\_wr}$ denote the remaining size of all of the internal IO's siblings that need to be completed to flush parent data from the non-volatile buffer. We use the following heuristic to estimate the scheduling value of internal write IOs:

$$y_{int\_wr} = \frac{(size_{int})^2}{size_{remain\_wr}} \times (I_{wr}(space) + Max\{y_{ext}^{wr_i}(t)\}) \ . \tag{5.3}$$

$I_{wr}(space)$ should assign a low value to write IOs when the buffer is nearly empty, giving higher priority to read IOs. When the buffer is nearly full, $I_{wr}(space)$ should give high value to write IOs, giving higher priority to write-back operations. We use the maximum value of all pending external write IOs to further increase the priority of internal write IOs when the non-volatile buffer is full. The design and implementation of a *good* I_{wr} function is application specific, and it is critical for gracefully servicing both read and write IOs. As in the read case, we give more value to large IOs and the soon-to-complete IOs (which is the reason for $(size_{int})^2$ factor).

Scheduling: Scheduling IOs whose service yields various values and incurs differing kinds of overhead is a hard problem. In this dissertation we do not intend to ascertain which scheduling method is the best. We use a simple greedy approach which chooses the IO with the maximum predicted *average yield* to schedule next. We define the average yield of an IO (y_{avg}) as

$$y_{avg} = \frac{y_{int\_\{read/wr\}}}{T_{service}} \ . \tag{5.4}$$

Thus, the average yield takes into consideration the predicted time required to service the internal IO (including its access delay and transfer time). Equations 5.2 and 5.3 estimate the value of internal IOs. The single-disk scheduler selects the internal IO with currently highest average yield, with the goal of maximizing the sum of all external yields. If more than one IO has the same y_{avg}, then we choose the one with the shortest deadline to break the tie.

Figure 5.10 depicts the average yield (solid line) for two internal IOs serviced by the same disk. The dotted line denotes the yield for the same IOs when distributed over the useful data transfer periods latency. When the scheduler must choose an IO to service next from the queue, it services the IO with the maximum average yield. Our initial design goal was to have the scheduler effectively mimic the behaviour of frequently used disk schedulers like the shortest-access-time-first (SATF) scheduler [45] (when preemptions do not happen).

Preemptions: We now present two preemption approaches *conservative preemption* and *aggressive preemption* that aim to optimize for the long-term and the short-term respectively.

Whenever a new IO arrives, the scheduler checks whether preempting the

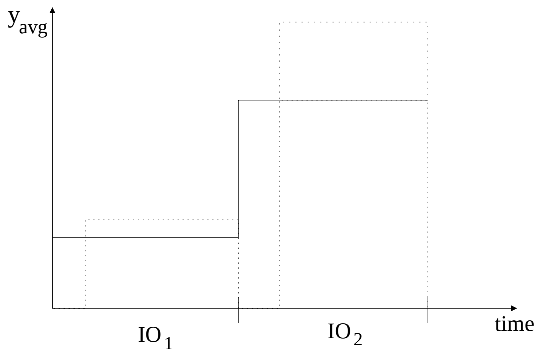


Figure 5.10: Average yield.

ongoing IO (using the preemption methods introduced in Section 5.2), servicing the new IO, and immediately resuming the preempted IO, offers a better average yield than would be obtained without preemption. To calculate the average yield in either case, we must consider the yields due to both IOs. Let the ongoing IO be denoted as IO_1 and the newly arrived IO as IO_2. Let $T^1_{service-remain}$ denote the time required to service the remaining portion of IO_1 irrespective of whether it is preempted or not.[12] In either case, we use the following formulation to calculate the average yield due to both IOs:

$$y_{avg} = \frac{y^1_{int} + y^2_{int}}{T^1_{service-remain} + T^2_{service}} \ . \tag{5.5}$$

Notice that although we consider only the remaining time left to service the ongoing IO, we still include its entire yield, as opposed to including only the yield corresponding to the remaining portion of the IO. Indeed, the ongoing IO yields any value *only if* it is serviced entirely.

Conservative Preemption: The conservative approach makes a decision based on a long-term optimization criterion. Only if the preemption of the ongoing IO yields an overall average yield in the long term (given by Equation 5.5)

[12]The value of $T^1_{service-remain}$ will be different depending on which case gets instantiated. It will include preemption overhead in case the IO is preempted.

which is greater than the no preemption case, the ongoing IO is preempted. Figure 5.11 depicts the case when even though the newly arrived IO (IO_2) offers a greater average yield than that of the remaining portion of the ongoing IO (IO_1), the conservative approach chooses not to preempt the ongoing IO. By not preempting the ongoing IO, an overall greater yield is obtained after both IOs have been serviced.

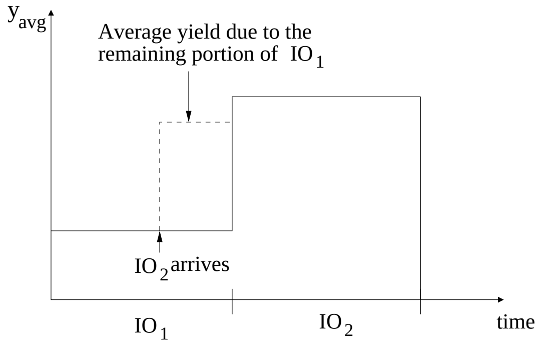


Figure 5.11: Conservative preemption.

Aggressive Preemption: Although the current IO offers a lesser average yield than the newly arrived IO, the conservative approach might conceivably choose not to preempt it. This happens because the conservative approach considers the overall average yield for servicing both IOs before making a decision, taking into consideration the preemption overhead. When the preemption overhead is considered within the framework of Equation 5.5, by not preempting the current IO (and thus eliminating preemption overhead) we obtain an overall better yield on the completion of the two IOs.

However, it is also conceivable that additional IO requests arrive in this period with higher priority than the ongoing IO. In this case, the best schedule might be simply to service all the higher priority IOs in the queue before finally servicing the ongoing IO. The aggressive preemption approach preempts the ongoing IO as soon as another IO with a higher average yield arrives. Figure 5.12

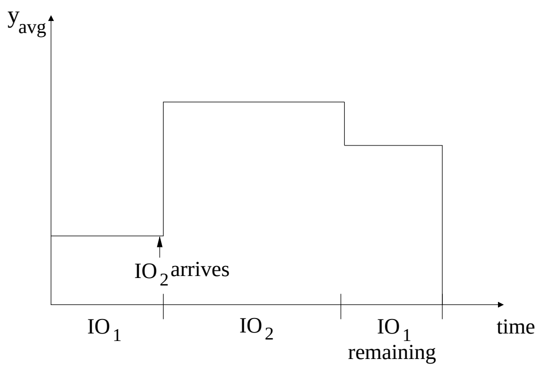


Figure 5.12: Aggressive preemption.

depicts the case when the aggressive approach preempts the ongoing IO in a greedy manner to immediately increase the average yield.

Finally, to support cascading preemptions (preempting an IO which already caused the preemption of another IO), we simply return the preempted IO to the scheduling queue. According to Equation 5.4, the predicted average yield increases for the remaining portions of preempted IOs (because parts of their data have already been transfered). This is necessary in order to maintain the feasibility of the greedy approach — actual QoS value is generated only after the whole IO completes. Hence, we have to control the number of preemptions. Our approach also prevents thrashing due to cascading preemptions. Cascading preemptions occur only when the average yield for all IOs in the cascade is maximum.[13]

5.4 Experimental Evaluation

In this study we have relied on simulation to validate our preemptive scheduling methods. *Semi-preemptible IO* [22] shows that it is feasible to implement

[13]Since we use a greedy approach, starvation is possible. To handle starvation, we can add a simple modification to our internal scheduling value, forcing it to increase with time.

preemption methods necessary for preemptive RAID scheduling outside of disk firmware. In this study we used the previous work in disk modeling and profiling [22, 33, 52] to build an accurate simulator for preemptible RAID systems (PraidSim). We evaluate the PRAID system using several micro-benchmarks and for two simulated real-time streaming applications.

5.4.1 Methodology

We use PraidSim to evaluate preemptive RAID scheduling algorithms. PraidSim is implemented in C++ and uses Disksim [33] to simulate disk accesses. We do not use the RAID simulator implemented in Disksim, but write our own simulator for QoS-aware RAID systems based on the architecture presented in Section 5.3. PraidSim can either execute a simulated workload for external IOs or perform a trace-driven simulation. We have chosen to simulate only the chunking and JIT-seek methods from *Semi-preemptible IO*. The seek-splitting method only helps in reducing the maximum IO waiting time and adds noticeable overhead. The chunking method relies only on optimal chunk size for a particular disk, which is easy to profile for both IDE and SCSI disks [22]. JIT-seek, which has been previously implemented in several schedulers [22, 52], is used here for JIT-preemption.

Table 5.3 summarizes the configurable parameters in PraidSim. The internal RAID configuration is chosen by specifying the RAID level, number of disks in the array, number of mirror replicas, stripe size, and the name of the simulated disk for Disksim. For the experiments in this chapter we used the Quantum Atlas 10K disk model. The IO arrival rate is specified with the arrival rate and random distribution for write IOs, deadline read IOs, and interactive read IOs; or by specifying a trace file. The next set of parameters is used to specify the PraidSim scheduling algorithm for non-interactive read and write IOs, the preemption decisions, methods for scheduling interactive reads, and the dynamic

Parameter name	Description
RAID level	RAID 0, RAID 0/1, or RAID 5
Number of disks	Number of disks in the disk array
Mirrors	Number of mirror disks
Disksim model	Name of the parameter file for Disksim disks
Striping unit	Size of the striping unit in disk blocks (512 B)
Write IOs	Write IO arrival rate and random distribution
Read IOs	Read IO deadlines, arrival rate and rand. dist.
Interactive IOs	Interactive IO arrival rate and rand. dist.
Scheduling	SCAN or FIFO for each IO class
Preemption	Preempt writes, reads, or no preemption
Interactivity	Preemption criteria for interactive IOs
Write priority	Buffer size and dynamic QoS value for writes
Chunk size	Chunk size for *Semi-preemptible IO*

Table 5.3: Summary of PraidSim parameters.

value for internal write IOs. The chunk size parameter specifies the chunk size used to schedule semi-preemptible IOs. For all experiments in this chapter we used a chunk size of 20 kB. We varied the simulated workloads to cover a large parameter space and then performed experiments using parameters that approximate the behavior of interactive video streaming applications (write-intensive video surveillance and read-intensive interactive video streaming applications).

5.4.2 Micro-benchmarks

Our micro-benchmarks aimed to answer the following questions:

- Does *preempting* non-interactive IOs always improve the quality of service?
- How does *preemption* help when interactive operations consist of several IOs in a cascade?
- What is the overhead of preempting and delaying *write* IOs to promptly service read requests?

Preemption Decisions

In order to show that decisions about preempting sequential disk accesses are not trivial for all applications, we performed the following experiment. We varied the size of non-interactive IOs and measured both the response time for interactive IOs and the throughput for non-interactive IOs. We fixed the arrival rate for interactive IOs to 10 req/s, and kept the disk fully utilized with non-interactive IOs. The size of the interactive requests was 100 kB.

Figure 5.13 depicts the average response time for interactive IOs for preempt-never and preempt-always approaches. For small IO sizes the benefit of preemption is of the order of $5 - 10$ ms. However, for large non-interactive sequential IOs, the preemption yields improvements of the order of 100 ms. The preemptive approach also provides less variation in response times, which is very important for interactive systems. Figure 5.14 shows the difference in throughput between the preempt-never and preempt-always approaches. The main question is whether the trade-off between improved response time and reduced throughput yields better quality of service.

Figure 5.15 depicts the improvements in aggregate interactive value (for all external interactive IOs) of the preempt-always over the preempt-never approach. We use a yield function for interactive real-time IOs from Figure 5.8(a)

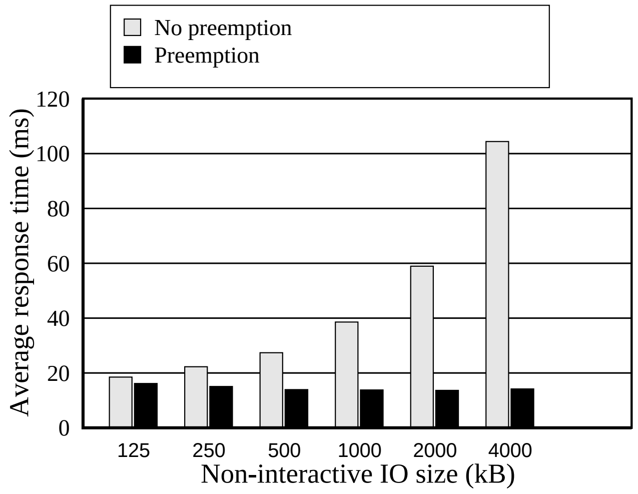


Figure 5.13: Average response time for interactive IOs vs. non-interactive IO size.

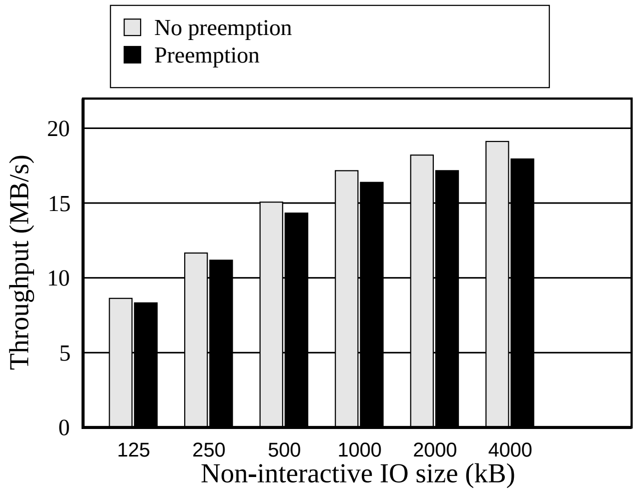


Figure 5.14: Disk throughput vs. non-interactive IO size.

in Section 5.3.2. If non-interactive IOs are small, the preempt-always approach does not offer any improvement, since all interactive IOs can be serviced before their deadlines even without preemptions. For large sizes of non-interactive IOs and short (100 ms) deadlines, preempt-always yielded up to 2.8 times the value of the non-preemptive approach (180% improvement). For applications with shorter deadlines the improvements are substantially higher. However, even for large non-interactive IOs, if the deadlines are of the order of 200 ms, then the preempt-always approach makes only marginal improvements over the preempt-never approach.

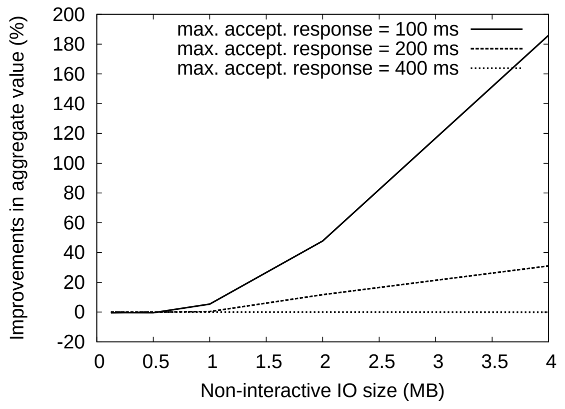
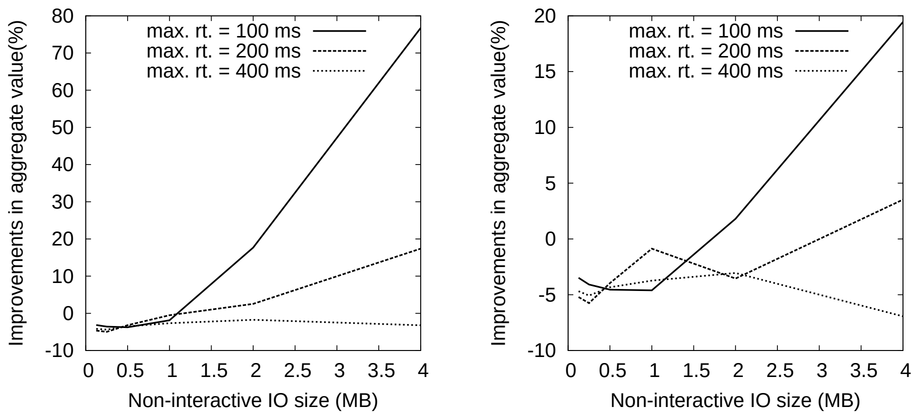


Figure 5.15: Improvements in aggregate interactive value.

Figure 5.16 shows the difference between the aggregate values for all serviced IOs for the preempt-always and the preempt-never approaches. For the case when the non-interactive requests yield the same as or greater value than the interactive IOs, the preempt-always approach degrades the aggregate value when a disk services small non-interactive IOs (up to approximately 2 MB in Figure 5.16). For cases when interactive requests are substantially more important than the non-interactive ones, the difference in aggregate value for all IOs converges to the curve presented in Figure 5.15. Simple priority-based scheduling cannot easily handle both cases.



(a) Equal IO values (b) Less interactive value

Figure 5.16: Differences in aggregate values for all IOs between the preempt-always and preempt-never approaches: (a) non-interactive and interactive IOs are equally important and (b) non-interactive IOs are more important (their value is five times greater).

Response Time for Cascading IOs

Interactive operations often require multiple IOs for their completion. For example, a video-on-demand system has to first fetch meta-data containing information about the position of a requested frame in a file. For large systems, meta-data cannot always reside in the memory cache, and requires an additional disk IO. Another example is a video surveillance system that supports complex interactive queries with data dependences [30, 70].

In order to show how preemptions help when an interactive operation consists of issuing multiple IO requests in a cascade, we performed the following experiment. The background, non-interactive workload consists of both read and write IOs (external), each being 2 MB long. We use the RAID 0/1 configuration with 8 disks. The sizes of internal IOs are between 0 and 2 MB and the interactive IOs are 100 kB each. As soon as one interactive IO completes, we issue the next IO in the cascade, measuring the time required to complete all cascading IOs. Figure 5.17 depicts the effect of cascading interactive IOs on the average response time for the whole operation. If the maximum acceptable response time for interactive operations is around 100 ms, the preemptive approach can service six cascading IOs, whereas the non-preemptive approach can service only two.

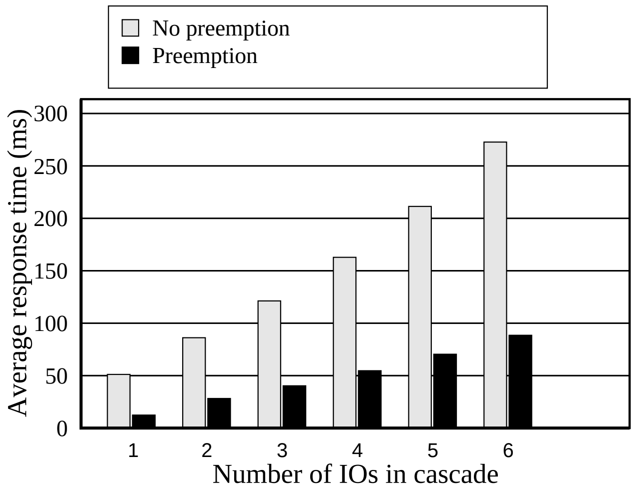


Figure 5.17: Response time for cascading interactive IOs.

Overhead of Delaying Write IOs

In order to show the overhead of preempting and delaying write IOs, we performed the following experiment. We varied the arrival rate for read requests and plotted the overhead in terms of increased buffering requirements and reduced idle time. We compared the following three scheduling policies: (1) SCAN scheduling without priorities, (2) SCAN scheduling with priorities for reads but without preemptions, and (3) SCAN scheduling with write preemptions.

Figure 5.18 depicts the RAID write-buffer requirements for different read arrival rates. In this case, we used RAID level 0/1, 4+4 disks, each external

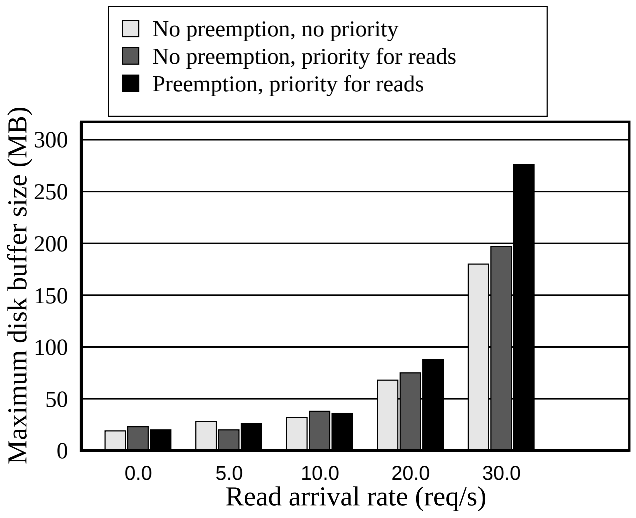


Figure 5.18: RAID write-buffer requirements.

read IO was 1 MB, and the external write rate was 50 MB/s (100 MB/s internally). Results show that independent of the scheduling criteria, whenever the available disk idle time is small, the required buffer size increases exponentially. The additional write-buffer requirement is acceptable for a range of read arrival rates in the system with preemptions. A real system must control the number of preemptions as well as the read/write priorities depending on available RAID idle time. Figure 5.19 depicts the average disk idle-time for different read arrival rates. The results showed that for arrival rates of up to around 10 req/s, preemption only marginally increases the write-buffer requirement and reduces the RAID idle-time, with noticeable improvements in interactive performance.

5.4.3 Write-intensive Real-time Applications

In this section we discuss the benefits of using preemptive RAID scheduling for write-intensive real-time streaming applications. We generated a workload similar to that of a video surveillance system that services read and write streams with real-time deadlines. In addition to IOs for real-time streams, we

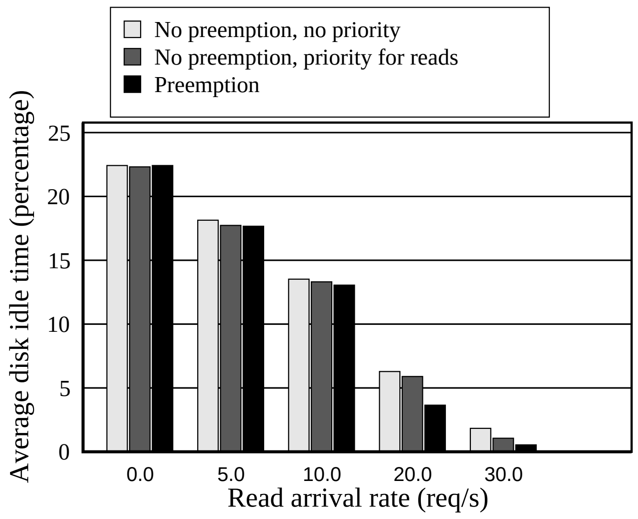


Figure 5.19: Average RAID idle-time.

also generate interactive read IOs. We present results for a typical RAID 0/1 (4+4 disks) configuration with a real-time write-rate of 50 MB/s (internally 100 MB/s) and a real-time read rate of 10 MB/s. The arrival rate for interactive IOs is 10 req/s. The external non-interactive IOs are 2 MB each, and interactive IOs are 1 MB each. The workload corresponds to a video surveillance system with 50 DVD-quality write video streams, 20 real-time read streams, and 10 interactive operations performed each second.

Figure 5.20 depicts the improvements in the response times for interactive IOs and the overhead in reduced RAID idle time. The system was able to satisfy all real-time streaming requirements in all three cases. Using the JIT-preemption method, our system decreased the interactive response time from 110 ms to 60 ms, by reducing RAID idle-time from 7.2% to 6.5%. The read-splitting method from Section 5.3.2 further decreased the response time (by reducing the data-transfer component on a single disk) with the substantially larger effect on reduced average disk idle time.

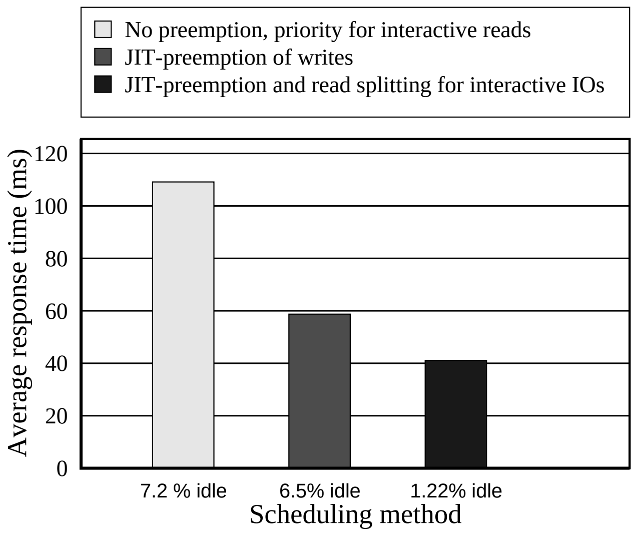


Figure 5.20: Average interactive read response times.

5.4.4 Read-intensive Applications

Figure 5.21 depicts the average response times for interactive read requests for read-intensive real-time streaming applications. The setup is the same as for write-intensive applications in the previous section, but the system services only read IOs. The streaming rate for non-interactive reads is 129 MB/s. The interactive IOs are 1 MB each, and their arrival rate is 10 req/s. The improvements in average response times were similar to those in our write-intensive experiment. The JIT-preemption with migration didn't substantially improve the average response for interactive IOs, but the better load-balancing compensated for the reduction in idle time due to JIT-preemption.

5.4.5 Summary of Results

First, we found that it is not always desirable to preempt non-interactive IOs. The decision depends on the application and the relative importance of user requests. Whenever we preempt nearly-completed IOs, we introduce additional

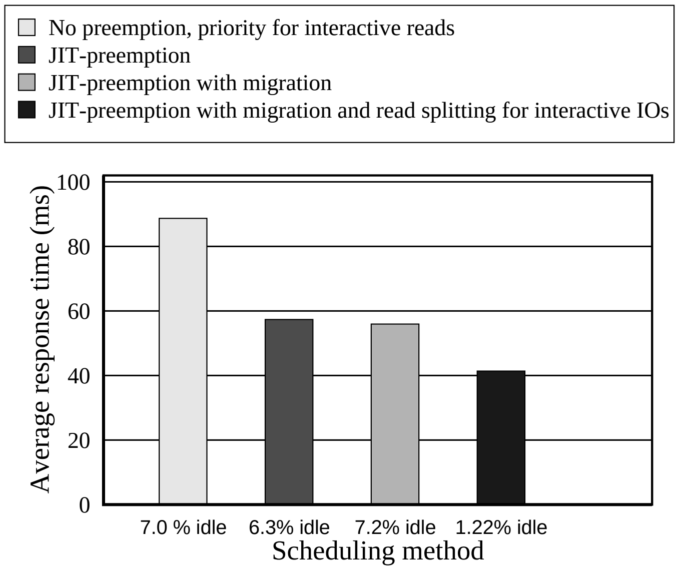


Figure 5.21: Average interactive read response times.

seek overhead without obtaining any additional value for servicing interactive IOs faster.

Second, we discovered that preemption can lead to substantial QoS improvements for interactive IOs consisting of several cascading IOs where each subsequent IO request depends on the completition of the previous one. Our system was able to service six cascading IOs in less than 100 ms, compared to only two in the case of the non-preemptible approach. This is important for large-scale commercial systems servicing interactive users [35] and emerging video surveillance systems [30, 70].

Third, we discovered that the increased write-buffer requirements and reduced disk idle-time are acceptable for a range of interactive and non-interactive streaming rates. We performed experiments on the range of read- and write-intensive streaming workloads (simulating typical video streaming systems). In summary, a preemptible system can reduce the interactive response time by nearly half (for example, from 110 ms to 60 ms) while reducing disk idle-time by only 0.7 % (for the same size of write buffer).

5.5 Summary

In this chapter we have investigated the effectiveness of IO preemptions to provide better disk scheduling for RAID-based storage systems. We first introduced methods for preemptions and resumptions of disk IOs — JIT-preemption and JIT-migration. We then proposed an architecture for QoS-aware RAID systems and a framework for preemptive RAID scheduling. We implemented a simulator for such systems (PraidSim). Using the simulator, we evaluated benefits and estimated the overhead associated with preemptive scheduling decisions. Our evaluation showed that using IO preemptions can lead to a better overall system QoS for applications with large sequential accesses and interactive user requests.

We plan to further this work in the following two directions. First, based on the existing Linux QoS extensions, we plan to implement a preemptive scheduler for software RAIDs. Second, we plan to investigate the effectiveness of preemptive scheduling in cluster-based storage systems.

Chapter 6

Related Work

In this chapter, we survey the representative work on disk management. We first survey related work in disk profiling, followed by previous work related to preemptible disk access and the implementation of *Semi-preemptible IO*. Finally, we survey related work relevant to preemptive RAID scheduling.

6.1 Disk Modeling and Profiling

Several previous studies have focused on the problem of disk feature extraction. Worthington, Ganger, et al. [76, 101] extract disk features in order to model disk drives accurately. Both studies rely on interrogative SCSI commands to extract LBA-to-PBA mapping from the disk. Patterson et al. [91] present several methods for empirical feature extraction, including an approximate mapping extraction method. Aboutabl et al. [1] propose methods for obtaining detailed temporal characteristics of disk drives, including methods for predicting rotational delay.

Scheduling algorithms in [42, 45, 53, 102] assume the ability to predict the rotational delay between successive requests to the disk. Lumb et al. [52] implemented their scheduler outside of disk firmware relying on their disk profiler.

Our study includes both interrogative and empirical methods for the accurate extraction of disk mapping information. Empirical methods can be used for disks that do not support interrogative commands in order to extract disk mapping information. Using empirical extraction, we can obtain accurate disk mapping information, including precise positions of track and cylinder boundaries. However, empirical methods are slower than interrogative ones. To predict rotational delay between disk IOs, Diskbench [28] uses an approach that is similar to the approach presented in [1]. Methods in [1] are continuously keeping track of the disk head position. We concentrate on predicting rotational distance between two LBAs and do not require knowledge of the head position, as explained in Section 3.3.2 and Section 3.4.9. Such prediction capability can be used for scheduling requests in real systems, where requests are known to arrive in a bursty fashion [73]. Diskbench can extract all the disk features that are required in order to predict disk access times with high accuracy (we have used the predictions to implement Semi-preemptible IO [20, 22]).

6.2 Preemptible Disk Access

Before the pioneering work of Daigle and Strosnider [18], Molano et al. [59], and Thomasian [94], it was assumed that the nature of disk IOs was inherently non-preemptible. Daigle and Strosnide [18] proposed breaking up a large IO into multiple smaller chunks to reduce the data transfer component of the *waiting time* for higher-priority requests. They proposed a minimum chunk size of one disk track. In this dissertation, we improve upon the conceptual model of [18] in three respects: 1) in addition to enabling preemption of the data transfer component, we show how to enable preemption of T_{rot} and T_{seek} components; 2) we improve upon the bounds for zero-overhead preemptibility; and 3) we show that making write IOs preemptible is not as straightforward as it is for read IOs, but we propose one possible solution.

Semi-preemptible IO [22] uses a *just-in-time seek* (JIT-seek) technique to make the rotational delay preemptible. JIT-seek can also be used to mask the rotational delay with useful data prefetching. In order to implement both methods, our system relies on accurate disk profiling [1, 25, 76, 91, 101]. Rotational delay masking has been proposed in multiple forms. Chiueh et al. [42] and Worthington et al. [102] present rotational-latency-sensitive schedulers, which consider the rotational position of the disk arm to make better scheduling decisions. Ganger et al. [52, 53, 71] presented *freeblock scheduling*, wherein the disk arm services background jobs using the rotational delay between foreground jobs. Seagate uses a variant of just-in-time seek [80] in some of its disk drives to reduce power consumption and noise. *Semi-preemptible IO* uses similar techniques for a different goal — to make rotational delays preemptible.

There is a large body of literature proposing IO scheduling policies for multimedia and real-time systems that improve disk response time [9, 82, 84, 92]. *Semi-preemptible IO* is orthogonal to these contributions. We believe that the existing methods can benefit from using preemptible IO to improve schedulability and further decrease response time for higher-priority requests. For instance, to model real-time disk IOs, one can draw from real-time CPU scheduling theory. Molano et al. [59] adapt the *Earliest Deadline First* (EDF) algorithm from CPU scheduling to disk IO scheduling. Since EDF is a preemptive scheduling algorithm, a higher-priority request must be able to preempt a lower-priority request. However, an ongoing disk request cannot be preempted instantaneously. Applying such classical real-time CPU scheduling theory is simplified if the preemption granularity is independent of system variables like IO sizes. *Semi-preemptible IO* provides such an ability. However, further study is necessary to address a non-linear preemption overhead occuring in preemptible disk scheduling (we present two possible greedy preemptive scheduling approaches in Chapter 5.3).

6.3 Preemptive RAID Scheduling

In the late eighties, the emerging market for personal computers changed the basic model for building high-performance commercial storage systems. The redundant arrays of inexpensive disks (RAID) provided better price-performance ratio over specialized disks [13, 15, 60]. After the initial single-server RAID systems, the next step was to provide an interface for direct-disk access over the network [31, 37]. RAID systems provide logical-disk abstraction on which a standard file-system stores data. After the introduction of cluster-based computing models [3], totally distributed cluster-based file-systems are introduced [2, 54, 55, 56]. Motivated by emerging multimedia applications, several authors investigated streaming support for these systems [96, 41].

Preemptive RAID scheduling is based on detailed knowledge of low-level disk characteristics [28, 76, 101]. A number of scheduling approaches rely on these low-level characteristics [22, 44, 52, 63]. RAID storage was the focus of a number of important studies including [15, 31, 75, 94, 96, 100]. John Wilkes et al. [98, 99] stressed the importance of providing quality-of-service scheduling in storage systems.

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 [59, 85, 90]. Molano et al. [59] 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. [85] and Sundaram et al. [90] presented their QoS extensions for the Linux operating system (QLinux). Our goals are similar to QLinux since we want to add QoS support for Linux disk access. However, in this dissertation we investigated an approach with minimal changes in Linux kernel space, which is sufficient for an efficient implementation of QoS disk scheduling.

Multimedia real-time systems were the focus of several relevant survey and trend studies [34, 36, 62, 89]. Plagemann et al. [62] survey related work in operating system architectures, CPU scheduling, disk management, memory management, and low-level bus, cache, and device management. Gemmell et al. [34] concentrate on disk file systems and scheduling for continuous media applications. In this dissertation, we also focus on disk QoS management, but we are interested in heterogenous media applications, which manage interactive, continuous, and best-effort data. We use the explicitly preemptive approach to improve QoS disk scheduling.

Chapter 7

Concluding Remarks

In this chapter, we first present a brief summary of the dissertation. We then discuss the impact of the proposed techniques, particularly *Semi-preemptible IO* and preemptive RAID scheduling. Finally, we make concluding remarks and suggest directions for future work.

7.1 Dissertation Summary

With the emergence of ubiquitous computing, we have witnessed an increased number of data-intensive real-time applications. For example, video surveillance, environmental monitoring, sensor networks, digital libraries, and large scientific setups often generate multiple gigabyte- or terabyte-per-day of real-time data that requires efficient, reliable storage and processing. Currently, the most cost-effective technology for non-volatile storage is based on magnetic disks.

First, we present an open-source low-level disk profiling tool (*Diskbench*). *Diskbench* extracts necessary features for realistic disk modeling and accurate performance predictions, which are required for real-time disk schedulers. We consider the following low-level disk features: rotational time, mapping from logical to physical block addresses, track boundaries, disk-zone information, and buffer parameters. We also consider high-level disk features including optimal chunk-size ranges (for both sequential reads and writes) and admission control curves for guaranteed-rate disk schedulers.

Second, we investigate preemptibility of disk IO requests and propose a class of preemptible disk scheduling algorithms. Previously, researchers were considering the scheduling of non-preemptible disk IOs. Thinking of disk IOs as preemptible IOs enables real-time schedulers to separate access latency from IO size. Hence, schedulers can operate with large IO requests and still provide short, guaranteed response times. We present the design and implementation of *Semi-preemptible IO* [22] prototype, which shows significant improvements over traditional, non-preemptible disk IOs. Essentially, *Semi-preemptible IO* maps each IO request into multiple fast-executing (and hence short-duration) disk commands using three methods. (The ongoing IO request can be preempted between these disk commands.) Each of these three methods addresses the reduction of one of the following IO components: $T_{transfer}$ (denoting transfer time), T_{rot} (denoting rotational delay), and T_{seek} (denoting seek time).

- *Chunking* $T_{transfer}$. A large IO transfer is divided into a number of small chunk transfers, and preemption is made possible between the small transfers. If the IO is not preempted between the chunk transfers, chunking does not incur any overhead. This is due to the prefetching mechanism in current disk drives.

- *Preempting* T_{rot}. By performing just-in-time (JIT) seek for servicing an IO request, the rotational delay at the destination track is virtually eliminated. The pre-seek slack time thus obtained is preemptible. This slack can also be used to perform prefetching for the ongoing IO request, and/or to perform seek splitting.

- *Splitting T_{seek}. Semi-preemptible IO* splits a long seek into sub-seeks, and permits preemption between two sub-seeks.

Using both synthetic and trace workloads, we have shown that these techniques can be efficiently implemented, given detailed disk parameters. Our empirical studies showed that *Semi-preemptible IO* can reduce the waiting time for both read and write requests significantly when compared with non-preemptible IOs.

Third, we investigate the effectiveness of preemptive disk-scheduling algorithms to achieve better QoS. We present an architecture for QoS-aware RAID systems based on Semi-preemptible IO [22]. We show *when* and *how* to preempt IOs to improve the overall QoS of the RAID. In order to decide when to preempt an IO, we propose preemptive scheduling methods which aim to maximize the total RAID QoS value. In order to decide how to preempt an IO, we introduced two mechanisms for IO preemptions in RAID systems — JIT-preemption and JIT-migration. We evaluated the benefits and estimated the overhead of the proposed approach using our simulator for preemptible RAID systems.

7.2 Impact

Preemptive CPU scheduling is a norm in the design of real-time systems. We feel that disk schedulers for data-intensive real-time applications can benefit from experience in the area of preemptive CPU scheduling. This dissertation presents the framework for implementing preemptive disk schedulers. Using the semi-preemptible IO abstraction, real-time disk schedulers can take the explicitly preemptive aproach and make intelligent decisions whether to preempt their sequential disk accesses or not.

7.3 Future Directions

This work can be extended in many ways. We list just the following three directions.

- Most current disk schedulers assume that disk IOs are non-preemptible. Based on the preemptible framework presented in this dissertation, additional work is needed to design preemptive disk schedulers for various real-time applications. The main difficulty in employing existing preemptive CPU schedulers is two-fold. First, preemption overhead is non-linear. Nevertheless, we can still accurately estimate the preemption overhead using disk profiling. Existing preemptive CPU schedulers are not designed to consider this predictable, but complex, preemption overhead. Second, optimal preemption cannot be done at any point, but only at well defined, JIT-preemption points. Furthermore, these points differ depending on the position of the next-to-be-scheduled IO. Preemptive schedulers need to take into account both of these issues.

- Most current operating systems do not implement sophisticated QoS disk scheduling. Preemptive disk schedulers can be employed to improve the QoS support for commodity operating systems, especially for open-source software RAID implementations.

- Several large storage-bound applications implement their storage as cluster-based software RAID on top of unreliable commodity hardware. The Google File System [35] is an example of such a system. The preemptive disk scheduling approach might help in providing better QoS scheduling for these cluster-based storage systems.

Bibliography

[1] M. Aboutabl, A. Agrawala, and J.-D. Decotignie, *Temporally determinate disk access: An experimental approach*, Univ. of Maryland Technical Report CS-TR-3752, (1997).

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, *Serverless network file systems*, Proceedings of the ACM SOSP, (1995).

[3] T. E. Anderson, D. E. Culler, and D. A. Patterson, *A case for NOW (network of workstations)*, IEEE Micro, (1995).

[4] R. T. Azuma, *Tracking requirements for augmented reality*, Communications of the ACM, 36 (1993).

[5] P. Bosch and S. J. Mullender, *Real-time disk scheduling in a mixed-media file system*, Proceedings of the IEEE RTAS, (2000).

[6] M. M. Buddhikot, *Project Mars: Scalable, high performance, web based multimedia-on-demand services and servers*, University of Washington Ph.D. Dissertation, (1998).

[7] L. R. Carley, G. R. Ganger, and D. Nagle, *MEMS-based Integrated-Circuit Mass-Storage systems*, Communications of the ACM, 43 (2000).

[8] CERN, *CERN*, http://www.cern.ch, (2004).

[9] E. Chang and H. Garcia-Molina, *Bubbleup - Low latency fast-scan for media servers*, Proceedings of the 5th ACM Multimedia, (1997).

[10] ——, *Effective memory use in a media server*, Proceedings of the 23rd VLDB, (1997).

[11] E. Chang, H. Garcia-Molina, and C. Li, *2d BubbleUp - Managing parallel disks for media servers*, Proceedings of the 5th Foundations on Data Organization, (1998).

[12] M.-S. Chen, D. D. Kandlur, and P. S. Yu, *Optimization of the grouped sweeping scheduling (GSS) with heterogeneous multimedia streams*, Proceedings of the ACM Multimedia, (1993).

[13] P. Chen and D. Patterson, *Maximizing performance in a striped disk array*, in Proceedings of the 17th IEEE ISCA, May 1990.

[14] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, *RAID: High-performance, reliable secondary storage*, ACM Computing Surveys, 26 (1994).

[15] ——, *RAID: High-performance, reliable secondary storage*, ACM Computing Surveys, 26 (1994).

[16] T. Clarke, *The TerraFly project*, Nature Electronic Magazine, (2001).

[17] R. Collins, A. Lipton, T. Kanade, H. Fujiyoshi, D. Duggins, Y. Tsin, D. Tolliver, N. Enomoto, and O. Hasegawa, *A system for video surveillance and monitoring*, Robotics Institute, Carnegie Mellon University Technical Report, (2000).

[18] S. J. Daigle and J. K. Strosnider, *Disk scheduling for multimedia data streams*, Proceedings of the IS&T/SPIE, (1994).

[19] Z. Dimitrijevic and R. Rangaswami, *Quality of service support for real-time storage systems*, Proceedings of the International IPSI-2003 Conference, (2003).

[20] Z. Dimitrijevic, R. Rangaswami, and E. Chang, *Virtual IO: Preemptible disk access*, Proceedings of the 10th ACM Multimedia, (2002).

[21] ——, *The XTREAM multimedia system*, Proceedings of the IEEE ICME, (2002).

[22] ——, *Design and implementation of Semi-preemptible IO*, Proceeding of the 2nd Usenix FAST, (2003).

[23] ——, *Architectural support for preemptive RAID schedulers*, WiP Report/Poster at Usenix FAST 2004, (2004).

[24] ——, *Preemptive RAID scheduling*, UCSB Technical Report, (2004).

[25] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya, *Diskbench*, http://www.cs.ucsb.edu/∼zoran/papers/db01.pdf, (2001).

[26] ——, *Diskbench*, http://www.cs.ucsb.edu/∼zoran/diskbench/, (2002).

[27] 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).

[28] Z. Dimitrijevic, R. Rangaswami, D. Watson, and A. Acharya, *Diskbench: User-level disk feature extraction tool*, UCSB Technical Report, (2004).

[29] Z. Dimitrijevic, D. Watson, and A. Acharya, *Scsibench*, http://www.cs.ucsb.edu/∼zoran/scsibench/, (2000).

[30] Z. Dimitrijevic, G. Wu, and E. Chang, *SFINX: A multi-sensor fusion and mining system*, Proceedings of the IEEE Pacific-rim Conference on Multimedia, (2003).

[31] A. L. Drapeau, K. Shirriff, E. K. Lee, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, K. Lutz, and D. A. Patterson, *RAID-II: A high-bandwidth network file server*, Proceedings of the ACM ISCA, (1994).

[32] T. Gandhi and M. Trivedi, *Motion analysis of omni-directional video streams for a mobile sentry*, ACM International Workshop on Video Surveillance, (2003).

[33] G. R. Ganger, B. L. Worthington, and Y. N. Patt, *The DiskSim simulation environment version 2.0 reference manual*, Reference Manual, (1999).

[34] D. J. Gemmell, H. M. Vin, D. D. Kandlur, and P. V. Rangan, *Multimedia storage servers: A tutorial and survey*, IEEE Computer, (1995).

[35] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system*, Proceedings of the ACM SOSP, (2003).

[36] G. Gibson, J. Vitter, and J. Wilkes, *Strategic directions in storage io issues in large-scale computing*, ACM Computing Survey, 28 (1996).

[37] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Ze-

lenka, *A cost-effective, high-bandwidth storage architecture*, Proceedings of the ACM ASPLOS, (1998).

[38] D. Gilbert, *The Linux SCSI generic (sg) howto*, http://tldp.org/HOWTO/SCSI-Generic-HOWTO/, (2002).

[39] J. Gray and P. Shenoy, *Rules of thumb in data engineering*, Proceedings of the IEEE ICDE, (1999).

[40] J. Grin, S. Schlosser, G. Ganger, and D. Nagle, *Operating systems management of MEMS-based storage devices*, Proceedings of the 4th Usenix OSDI, (2000).

[41] R. Haskin and F. Schmuck, *The tiger shark filesystem*, Proceedings of the IEEE COMPCON, (1996).

[42] L. Huang and T. Chiueh, *Implementation of a rotation-latency-sensitive disk scheduler*, SUNY at Stony Brook Technical Report, (2000).

[43] A. N. S. I., *SCSI-2 specification X3T9.2/375R revision 10L*, January 1995.

[44] S. Iyer and P. Druschel, *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O*, Proceedings of the 18th ACM SOSP, (2001).

[45] D. M. Jacobson and J. Wilkes, *Disk scheduling algorithms based on rotational position*, HPL Technical Report, (1991).

[46] K. Jeffay, D. F. Stanat, and C. U. Martel, *On non-preemptive scheduling of periodic and sporadic tasks*, Proceedings of the 12th IEEE RTSS, (1991).

[47] E. Kalman, Rudolph, *A new approach to linear filtering and prediction problems*, Transactions of the ASME–Journal of Basic Engineering, 82 (1960).

[48] D. I. Katcher, H. Arakawa, and J. K. Strosnider, *Engineering and analysis of fixed priority schedulers*, Software Engineering, 19 (1993).

[49] J. Korst, *Random duplication assignment: An alternative to striping in video servers*, Proceedings of the 5th ACM Multimedia Conference, (1997).

[50] S.-H. Lee, K.-Y. Whang, Y.-S. Moon, and I.-Y. Song, *Dynamic buffer allocation in Video-on-Demand systems*, Proceedings of the ACM SIGMOD, (2001).

[51] C. Liu and J. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment*, ACM Journal, (1973).

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

[53] 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 Usenix OSDI, (2000).

[54] K. Magoutis, *The optimistic direct access file system: Design and network interface support*, Proceedings of Workshop on Novel Uses of System Area Networks (SAN-1), (2002).

[55] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer, *Making the most out of direct-access network attached storage*, Proceedings of the 2nd Usenix FAST, (2003).

[56] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. G. Wickremesinghe, and E. Gabber, *Structure and performance of the direct access file system*, Proceedings of the Usenix Annual Technical Conference, (2002).

[57] M. McKusick, W. Joy, S. Leffler, and R. Fabry, *A fast file system for UNIX*, ACM Transactions on Computer Systems 2, 3 (1984).

[58] M. Mesnier, G. R. Ganger, and E. Riedel, *Object-based storage*, IEEE Communications Magazine, (2003).

[59] A. Molano, K. Juvva, and R. Rajkumar, *Guaranteeing timing constraints for disk accesses in RT-Mach*, Proceedings of the IEEE RTSS, (1997).

[60] D. Patterson, G. Gibson, and R. Katz, *A case for redundant arrays of inexpensive disks (RAID)*, in Proceedings of the ACM SIGMOD, 1988.

[61] Performance Evaluation Laboratory, Brigham Young University, *Trace distribution center*, http://tds.cs.byu.edu/tds/, (2002).

[62] T. Plageman, V. Goebel, P. Halvorsen, and O. J. Anshus, *Operating system support for multimedia system*, Computer Communications, (2000).

[63] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *Robust, portable I/O scheduling with the disk mimic*, Proceedings of the Usenix Annual Technical Conference, (2003).

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

[65] P. V. Rangan and H. M. Vin, *Efficient storage techniques for digital continuous multimedia*, IEEE Transactions on Knowledge and Data Engineering, 5 (1993).

[66] P. V. Rangan, H. M. Vin, and S. Ramanathan, *Designing and on-demand multimedia service*, IEEE Communications Magazine, 30 (1992).

[67] ——, *Designing and on-demand multimedia service*, IEEE Communications Magazine, 30 (1992).

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

[69] R. Rangaswami, Z. Dimitrijevic, E. Chang, and K. E. Schauser, *Mems-based disk buffer for streaming media servers*, Proceedings of the IEEE ICDE, (2003).

[70] R. Rangaswami, Z. Dimitrijevic, K. Kakligian, E. Chang, and Y.-F. Wang, *The SfinX video surveillance system*, Proceedings of the IEEE ICME, (2004).

[71] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle, *Data mining on an OLTP system (nearly) for free*, Proceedings of the ACM SIGMOD, (2000).

[72] M. Rosenblum and J. Ousterhout, *The design and implementation of a log-structured file system*, Proceedings of the 13th SOSP, (1991).

[73] C. Ruemmler and J. Wilkes, *UNIX disk access patterns*, Proceedings of the Usenix Conference, (1993).

[74] C. Ruemmler and J. Wilkes, *An introduction to disk drive modeling*, IEEE Computer, 2 (1994).

[75] J. R. Santos, R. R. Muntz, and B. A. Ribeiro-Neto, *Comparing random data allocation and data striping in multimedia servers*, Measurement and Modeling of Computer Systems, (2000).

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

[77] 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, (2002).

[78] S. W. Schlosser, J. L. Griffin, D. Nagle, and G. R. Ganger, *Designing computer systems with MEMS-based storage*, Proceedings of the ACM ASPLOS, (2000).

[79] Seagate Technology, *SCSI interface, product manual 2*, http://www.seagate.com/support/disc/manuals/scsi/38479j.pdf, (1999).

[80] ——, *Seagate's sound barrier technology*, http://www.seagate.com/docs/pdf/whitepaper/sound_barrier.pdf, (2000).

[81] SETI, *SETI Institute*, http://www.seti.org, (2004).

[82] C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri, *On scheduling atomic and composite multimedia objects*, IEEE Transactions on Knowledge and Data Engineering, 14 (2002).

[83] 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).

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

[85] ——, *Cello: A disk scheduling framework for next generation operating systems*, Proceedings of the ACM Sigmetrics, (1998).

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

[87] L. L. Smarr, A. A. Chien, T. DeFanti, J. Leigh, and P. M. Papadopoulos, *The OptIPuter*, Communications of the ACM, (2003).

[88] C. Stauffer and E. Grimson, *Learning patterns of activity using real-time tracking*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 22 (2000).

[89] R. Steinmetz, *Multimedia file systems survey: approaches for continuous media disk scheduling*, Computer Communications, (1995).

[90] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin, *Application performance in the QLinux multimedia operating system*, Proceedings of the 8th ACM Multimedia, (2000).

[91] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson, *Microbenchmark-based extraction of local and global disk characteristics*, UC Berkeley Technical Report, (1999).

[92] W. Tavanapong, K. Hua, and J. Wang, *A framework for supporting previewing and vcr operations in a low bandwidth environment*, Proceedings of the 5th ACM Multimedia Conference, (1997).

[93] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger, *A framework for building unobtrusive disk maintenance applications*, Proceedings of the 3rd Usenix FAST, (2004).

[94] A. Thomasian, *Priority queueing in RAID5 disk arrays with an NVS cache*, Proceedings of MASCOTS, (1995).

[95] D. A. Thompson and J. S. Best, *The future of magnetic data storage technology*, IBM Journal of Research and Development, 44 (2000).

[96] F. Tobagi, J. Pang, R. Baird, and M. Gang, *Streaming RAID-a disk array management system for video files*, Proceedings of the 1st ACM Multimedia, (1993).

[97] P. Vettiger, M. Despont, U. Drechsler, U. Durig, W. Haberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binning, *The "Millipede" - More than one thousand tips for future AFM data storage*, IBM Journal of Research and Development, 44 (2000).

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

[99] ——, *Data services – from data to containers*, Keynote speech at the 2nd Usenix FAST, (2003).

[100] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, *The HP AutoRAID hierarchical storage system*, ACM Transactions on Computer Systems, 14 (1996).

[101] B. L. Worthington, G. Ganger, Y. N. Patt, and J. Wilkes,

Online extraction of SCSI disk drive parameters, Proceedings of the ACM Sigmetrics, (1995).

[102] B. L. Worthington, G. R. Ganger, and Y. N. Patt, *Scheduling algorithms for modern disk drives*, Proceedings of the ACM Sigmetrics, (1994).

[103] G. Wu, Y. Wu, L. Jiao, Y.-F. Wang, and E. Chang, *Multi-camera spatio-temporal fusion and biased sequence-data learning for security surveillance*, Proceedings of the 11th ACM Multimedia, (2003).

[104] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson, *Trading capacity for performance in a disk array*, Proceedings of the Usenix OSDI, (2000).

[105] X. Zhou, R. Collins, T. Kanade, and P. Metes, *A master-slave system to acquire biometric imagery of humans at distance*, ACM International Workshop on Video Surveillance, (2003).