UNIVERSITY OF CALIFORNIA

Santa Barbara

Real-time Storage Systems for Multimedia

by

Raju Rangaswami

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

Doctor of Philosophy

in

Computer Science

Committee in charge:

Professor Edward Chang, Chair
Professor Divyakant Agarwal,
Professor Elizabeth Belding-Royer,
Professor Klaus Schauser

September, 2004

September, 2004

VITA

February 28, 1977	–	Born in Mumbai, India
1995	–	Higher Secondaray School Certificate, D.G. Ruparel, Mumbai, India
1999	–	B.S. in Computer Science and Engineering, IIT Kharagpur, India
Summer 2001	–	Sony Digital Media Ventures, San Francisco, California
June 2003	–	M.S. in Computer Science, University of California, Santa Barbara
Summer 2004	–	Expertcity, Inc. Santa Barbara, California
1999-2004	–	Teaching and Research Assistant, Department of Computer Science, University of California, Santa Barbara
August 2004	–	Ph.D. in Computer Science, University of California, Santa Barbara

PUBLICATIONS

Raju Rangaswami, Zoran Dimitrijević, and Edward Chang. **Architectural Support for MEMS-disk Streaming Storage**. UCSB Technical Report, August 2004.

Raju Rangaswami, Zoran Dimitrijević, Kyle Kakligian, Edward Chang, and Yuan-Fang Wang. **The SfinX Video Surveillance System**. Proceedings of the IEEE International Conference on Multimedia and Expo, 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.

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **Architectural Support for Preemptive RAID Schedulers**. Usenix FAST WiP/Poster, March 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 and Raju Rangaswami. **Quality of Service Support for Real-time Storage Systems**. Proceedings of the International IPSI-2003 Conference, October 2003.

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **Design and Implementation of Semi-preemptible IO**. Proceedings of the Usenix File and Storage Technologies conference, March 2003.

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

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **Virtual IO: Preemptible Disk Access**. Proceedings of ACM Multimedia (poster), December 2002.

Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. **The XTREAM Multimedia System**. Proceedings of the IEEE International Conference on Multimedia and Expo, August 2002.

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

Raju Rangaswami, Edward Chang, Chen Li, and Milton Chen. **Data Placement for Multiuser Interactive Digital VCR**. Proceedings of the IEEE International Conference on Multimedia and Expo, August 2001.

Raju Rangaswami, Edward Chang and Gary Chan. **Managing Media Data for Enabling Interactive DTV**. UCSB Technical Report, November 2000.

Raju Rangaswami, Himyanshu Anand, and Indranil Sengupta. **Spoofers and Sniffers: Serious Security Threats**. Proceedings of the National Communications Conference, India, January 1999.

Abstract

Real-time Storage Systems for Multimedia

by

Raju Rangaswami

Over the last decade, storage has been playing catch-up with the meteoric improvement in computation and communications infrastructure. However, this trend requires a change if future systems must be able to support emerging applications that deal with massive amounts of data. These high data-volume applications require that storage systems support not just the traditional *quality-of-service* (QoS) metrics like reliability, availability, etc. but also new metrics like real-time data-delivery, and low response-time while maintaining high system throughput.

The development of conventional real-time systems has mainly focused on supporting the real-time paradigm for computational resources and usually assumes that all data resides in main memory. Real-time multimedia systems, however, manage large amounts of heterogeneous data that require to be placed on secondary storage. These include data that have real-time delivery constraints, traditional data that require best-effort service, as well as interactive data that require immediate service. In addition to satisfying computational real-time constraints, such systems must also support the real-time data storage and retrieval requirements of multimedia applications.

This dissertation focuses on storage systems that can support the varied requirements of heterogeneous multimedia data. For providing the IO guarantees required by such data as well as improving the overall cost-performance metric of the storage system, we propose and evaluate two principal approaches.

The first approach presents several methods for managing a traditional disk-only storage system to meet real-time delivery constraints as well as improving the response time of interactive operations. Unlike previous methods, these methods are developed based on accurate low-level profiling of storage device parameters.

The second approach proposes new architectures for storage systems using MEMS-based storage, an emerging technology, and evaluates its value for real-time multimedia applications. This approach is a specific-case solution for the more general problem of managing heterogeneous data and storage. We propose rules-of-thumb for partitioning data types across storage devices and propose architectures and mechanisms for building MEMS-disk storage systems that can support real-time streaming data.

Acknowledgements

This dissertation would not have been what it is without the advice, help, encouragement, and support, of many many people. I dedicate this acknowledgement to those who have influenced me in a variety of ways during the last five years.

I am gratefully indebted to my parents, Vijaya and Seshadri Rangaswami, my wife, Namitha, and my brother, Srinivas, for loving and caring about me and supporting me through thick and thin during this period. Without any one of you, this accomplishment would have been impossible. This dissertation is dedicated to you.

I wish to thank my advisor, Prof. Edward Chang, for agreeing to guide me through this journey. Ed, your enthusiasm and spirit for research is very unique and I hope that at least some of it has rubbed off on me. I immensely enjoyed and learnt from our numerous discussions, brainstorming sessions (especially after dinner!), and paper revisions. Thank you for finding time for me regardless of how busy your schedule was and providing me with the best work environment possible. Finally, I can never thank you enough for supporting me through lean periods in my work as well as personal life during my stay at UCSB.

I wish to thank Divy, Elizabeth, and Klaus, the other members in my Ph.D. committee for agreeing to lend their support to my dissertation. Divy, thank you for teaching me about distributed systems, for our discussions on storage, and for asking numerous thought-provoking questions. Elizabeth, thank you

for teaching me all about wireless networks. Klaus, thank you for teaching me about scalable systems. Your enthusiasm for teaching is unmatched in my experience. I really enjoyed working with you on MEMS-based storage.

I wish to thank Zoran Dimitrijević, without whom this dissertation would not be what it is. Zoran, I cannot say how much I have enjoyed your company and working with you. Working with you has been a great learning experience for me, not just academically. I enjoyed each of those "philosophical discussions" that we spent so much time on! I hope this friendship is one that will last a long time.

My experience in Santa Barbara would not have been as good as it was if it weren't for my numerous friends. I cannot possibly list all of you but I wish for you to know that I enjoyed interacting with each and every one of you. Thank you for making my Santa Barbara experience a one to cherish!

I thank the faculty of my department at UCSB for teaching me many important lessons in Computer Science. Thank you Divy, Kevin, Elizabeth, Pete, Amr, Teo, Alan, Ambuj, Subhash, Yuan-Fang, Tao, Michael, and Klaus for teaching me about various aspects of computer science. Finally, I would like to thank the wonderful people from the CS department office, Maryjane, Sandy, Amanda, and Juli for all their help.

Once again, Thank You All!

Funding Acknowledgement

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computation, Communication, and Storage – are the necessary components for building a computer system. While computation and communications infrastructure have enjoyed tremendous growth in the recent past, storage has been lagging behind significantly. Compared to the sustained improvement in computation power and communication bandwidth (60% [53] and 200% [28] per year respectively), improvements in storage technology, more specifically disk-drive technology [32, 83], are only in capacity and throughput (60% and 40% per year, respectively). The improvement in disk access time (10% per year) is significantly less. The long access times for disk drives necessitates accessing the disk drive in large chunks to achieve high throughput. Larger IOs imply that larger (expensive) buffers are required to temporarily store data. As a result, the storage system is usually the bottleneck in improving the overall performance of a multi-programmed computer system.

The nature of Internet content has undergone noticeable transformation over the last few of years. From text, to graphical images, to audio, and now video, the Internet is making the user experience richer. Consistent improvements in the capabilities of personal computers and the availability of high-bandwidth home Internet-access have influenced the current rapid increase in multime-

dia streaming content. With high-speed Internet access, rich media is gaining increasing popularity with Internet users. Recent surveys indicate that over 50% (close to 100 million) of internet users in the US access streaming media content [56].

Unlike traditional data, streaming data have quality of service requirements. Data must be retrieved or stored by a specific deadline otherwise the user experience suffers. Conventional real-time systems that guarantee quality of service have focused on supporting the real-time paradigm for computational resources. Real-time communication *or* networking support has also received reasonable attention with the development of networks like Firewire (IEEE 1394) and ATM that provide for bandwidth reservation. In the context of streaming data, storage is as important as computation or networking. In addition to guaranteeing real-time computation and communication constraints, such systems must also support quality of service requirements at the storage system.

With the emergence of applications that deal with massive amounts of data, system architects have begun considering storage as the main bottleneck resource and have been adapting solutions to account for this. Applications like video surveillance, large-scale sensor networks, storage-bound Web applications, and virtual reality, require that storage systems support new QoS primitives that can effectively manage the large data sizes as well as real-time data delivery constraints. To cite examples, video surveillance systems [13, 64] typically issue IOs of several megabytes to support streaming data to maintain high-throughput while supporting low-latency interactive scan operations simultaneously; the TinyDB project [47] at UC Berkeley proposes an ad-hoc sensor network of motes which are capable of transmitting continuous sensor readings of up to 50Kbps and which are required be stored in an online, real-time fashion; the GoogleFS [26] uses 64 MB IOs to access its data stores performing IOs to accomplish a variety of tasks of varying priority; the TerraFly project [12] at Florida International University integrates remote sensing image data obtained

from orbiting satellites into a real-time updating virtual-reality flight simulator used simultaneously by several users. Current disk-only storage systems that are designed to support such applications are limited by both cost and performance constraints.

Before we move on, we ask ourselves this question: What does the term *quality of service* imply for storage systems? When we refer to quality of service in storage systems, it could have multiple interpretations. One interpretation would be in terms of *reliability, availability* and *fault-tolerance metrics*. These metrics are of course important and are common to different types of applications and data. These are general quality of service metrics for all computer systems. The quality of service metrics used for storage in the context of streaming data are those of *guaranteed-rate IO, throughput* and *response time*. These performance metrics gain importance upon assuming that the underlying storage system is reliable, available and fault-tolerant. The fundamental question we ask in this thesis is: given a storage system, is it able to provide *quality of service* in terms of guaranteed-rate IO, throughput, and response time, for managing streaming data?

Surveying today's storage options for multimedia data, we note that 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 [83]. Today, disk drives are the preferred medium for storing multimedia data, since they can easily accommodate the large sizes of these data. Traditionally, operating systems have optimized disk drives for accessing non-real-time data. It is an interesting problem to support real-time storage and retrieval constraints on hard disk storage. We address this problem in this dissertation.

Although disk drives are extremely popular for high-volume storage, they

do have performance concerns. As mentioned earlier disk access times are have continued to lag behind the disk throughput increase of 40% and capacity increase of 60% [83]. 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 translates to large DRAM buffering cost. A large DRAM buffer is especially necessary for servers which stream to a large number of clients. Multimedia server architects have tried to cope with this performance gap by proposing solutions ranging from simple resource trade-offs [60, 98] to more complex ones that require substantial engineering effort [8, 41, 52].

Why is storage playing catch-up? In the past, system architects have been partially addressing the storage bottleneck problem by developing engineering solutions that try to maximize the available disk throughput. With the introduction of new storage technologies, there is now another approach to improving storage system performance by incorporating these technologies in part or full. We can broadly classify these principal approaches as:

- Super-engineer existing disk-only storage systems.

- Employ alternative competing and assisting storage technologies.

For several years now, system architects have improved the performance of the storage system for both real-time and non-real-time workloads using the first approach [60, 91, 52, 8, 41, 16, 69, 19]. These solutions are and will continue to be important for maintaining current and future disk-based storage systems. However, disk-only storage solutions are effective only to a finite extent. Beyond that the device mechanics are a limiting factor in improving its performance.

The latter approach is now becoming feasible with the introduction of new storage technologies which have relatively better access characteristics than disk storage. In fact, we see a trend toward building heterogeneous storage systems composed of multiple storage technologies. Micro-electro-mechanical-systems (MEMS) based storage is a relatively new technology that holds the promise

4

of improving the storage system performance upon integration into the existing storage hierarchy [71, 88, 95, 87, 63, 96, 70]. This research has served as the initial demonstration of the potential IO performance improvement due to these devices for a varied workload.

Emerging applications and storage technologies are necessitating the integration of real-time and non-real-time applications and data in a common platform. Applications compete for the same storage resources in terms of space as well as time (or bandwidth). Space partitioning is a significant problem because of the nature of heterogeneous storage systems which are composed of multiple components of varying cost and performance characteristics. Space partitioning must also take into account the performance and capacity requirements of the data itself. However, space may still be the more easily manageable of the two resources.

Bandwidth allocation and reservation is complex not just due to the reasons quoted above, but also because it is not an easily time-shared resource in storage systems. While serving multiple application-generated IOs simultaneously, a significant, non-uniform switching overhead exists for non-uniform memory access (NUMA) devices like disk drives and MEMS-based storage devices. Allocation and reservation schemes must take this additional overhead into account.

Applications generate a wide variety of IO, varying in size, throughput, and interactivity requirements. Some applications access the storage device in large chunks to maximize throughput. At the same time, others issue mission-critical interactive IOs for which response time must be kept short and which may not be able to wait for bulk transfers to complete. Application-generated IO traffic require IO primitives for supporting their QoS requirements that include, but are not limited to, the following:

- High throughput

- Guaranteed-rate IO

- Hard and soft timing guarantees

- Non-starvation and eventual completion

- Low Response-time for interactive IOs

In this dissertation, we present a new class of single-disk and RAID-based storage systems that also integrate the faster, albeit more expensive, MEMS-based storage devices. MEMS-based storage devices offer a unique cost-performance trade-off between those of DRAM and disk drives. We propose an analytical framework to evaluate the effective use of MEMS devices in a streaming media server.

The analytical framework that we have developed to analyze the use of MEMS storage in streaming servers is not restricted to MEMS storage alone. It is general enough to accommodate other devices with similar performance characteristics which are being developed by academia and industry including banks of NVRAM, holographic storage and atomic force microscopy (AFM) storage [58, 86].

Real-time storage support in OS

Current commodity operating systems offer little or no real-time storage support. The design of current operating systems are optimized for interactive or throughput-intensive workload. However, multimedia systems place unique, real-time demands on disk performance. If the deadline for data retrieval/storage is not met, end-user experience suffers or data is lost forever. Additionally, multimedia data is bulkier and bandwidth intensive. Thus, it presents the dual requirements of guaranteed delivery and high throughput to

file-system design. The next generation of storage systems are faced with the following requirements:

High-Throughput: To handle high-bandwidth multimedia data, these storage systems must be able to handle multiple requests simultaneously and deliver high throughput. To do so, the system software must have accurate knowledge of the performance characteristics of the underlying storage medium. One might argue that this problem could be solved by adding more hardware to achieve higher performance. However, we note that if the scale of the system is increased unnecessarily, manageability suffers and maintenance costs increase. In fact, building a high-performance system provides an orthogonal solution and can be used in conjunction with cluster-based solutions.

Quality of Service: Multimedia data need quality of service guarantees and have real-time requirements. The next generation of multimedia systems must therefore provide guarantees on data access. Guaranteeing real-time data delivery in combination with maintaining high throughput becomes a hard problem when file-systems are unaware of the exact operation of the storage device.

Low-level Device Knowledge: Multimedia file-systems must have accurate information of low-level storage device features. In the case of hard-disk storage, the file-system must have accurate knowledge of access overheads, transfer rates, caching and prefetching policies etc. in order to perform well. However, hard-disk manufacturers hide these features from the operating system by providing high-level interfaces like Small Computer Systems Interface (SCSI). The next generation multimedia file-system must be capable of circumventing such interface restrictions to obtain accurate device models and performance metrics.

Managing new storage: Emerging storage technology holds promise of changing existing storage architectures. Designing next generation storage

systems must take new architectural possibilities into consideration during design.

1.1   Thesis Statement

In this dissertation, we improve the state-of-the-art in real-time storage systems by pursuing two complementary research directions:

- Designing and implementing *disk-only storage* solutions using accurate disk-profiling, and

- Proposing new architectures and mechanisms for building *heterogeneous MEMS-disk storage* systems.

1.2   Thesis Contributions

This thesis focuses on storage systems that can support the varied requirements of heterogeneous multimedia data. The contributions of this thesis are as follows:

1. We propose placement, scheduling and admission control algorithms for managing both real-time and non-real-time data. Unlike prior approaches, the methods presented in this thesis are based on on accurate low-level profiling of storage performance features.

2. We present the implementation of a prototype multimedia streaming system, which uses the proposed data management policies. We also outline the architecture for a next-generation video surveillance system based on our real-time streaming storage system.

3. We explore the impact of integrating emerging storage technology into existing storage architectures on the class of streaming media applications.

We evaluate the use of MEMS-based storage in streaming media servers as a disk buffer and disk cache.

4. We introduce the problem of managing heterogeneous data on new storage architectures that include traditional as well as new storage components. We propose and evaluate architectures and mechanisms for building heterogeneous storage systems that can also support real-time streaming data.

Chapter 2

Background

In this chapter, we first provide an overview of multimedia application requirements which are significantly different from those of traditional applications. We then describe different storage technologies and their cost-performance characteristics that are important for storing and managing real-time multimedia data.

2.1   QoS requirements for multimedia

The different forms of multimedia today: *text*, *images*, *audio*, and *video*. Other multimedia forms are possible in the future (e.g., holograms). Each medium has unique requirements, possibly distinct from others. For instance, text might require low response time, while video might require guaranteed rate data retrieval and pose real-time requirements. Even within a single medium, the properties of different objects belonging to that medium can be different (See Table 2.1). The next generation storage solutions must take these diverse requirements into account. Here, we focus on the real-time requirements of streaming multimedia and also show how non-real-time data requirements within our framework.

Media Type	Sampling Rate	Bit-rate	Compressed
Voice	8 KHz / 8-bit	$64Kbps$	$16Kbps$
CD Audio	44.1 KHz / 16-bit / 2-ch	$1.4Mbps$	$353Kbps$
MP3 Audio	8 KHz / 8-bit	$64Kbps$	$64Kbps$
MPEG + CD Audio	320x480 / 25fps	$6.2Mbps$	$2.25Mbps$
NTSC Color	640x480 pixels / 24-bit	$216Mbps$	$8.8Mbps$
HDTV quality video	1280x760 pixels / 24-bit	$720Mbps$	$25.6Mbps$

Table 2.1: Multimedia Data Types.

Multimedia systems are generally characterized by the following requirements:

1. High Throughput

2. Low Initial Latency

3. Guaranteed Rate IO

4. Interactivity support

5. Support heterogeneous data

High throughput is required to support multiple simultaneous streams from the storage system. As the number of streams increases, the throughput of the storage system (for e.g. disk drives) degrades due to the extra overhead of accessing a large number of streams. Data placement and scheduling policies must be employed to ensure that the throughput of the storage system is maintained.

Multimedia streams are usually user-driven. In the user context, streams must be retrieved with *low initial latency*.

As much as throughput is important, the storage system must also provide for *guaranteed-rate IO* for individual streams. Streaming data has bit-rate requirements (either constant or variable) and the system must be designed so that streams are guaranteed their required IO rates.

Support for *interactivity* is another common requirement that multimedia systems. Users must be able to interact with video or audio streams by per-

forming fast-scan or pause operations.

Finally, any multimedia storage system must support *heterogeneous data*. By heterogeneous data, we mean both real-time as well as non-real-time data. The system must ensure that non-real-time data are not starved while trying to meet the quality of service requirements of real-time data. This requires that certain part of the storage bandwidth be reserved for non-real-time traffic, irrespective of the real-time load.

The above requirements are design goals for any multimedia system. However, achieving these goals is not straight-forward. To achieve high throughput, disk utilization must be maximized and memory buffer use must be minimized. Initial latency can be shown to be proportional to buffer use as well as disk access overhead. To guarantee IO rate, disk bandwidth and buffer reservation is required. Thus, the design goals translate to conflicting requirements of decreasing the buffer use and increasing disk utilization. The design of a high-performance multimedia storage system can face bottleneck in either the buffer subsystem or disk subsystem. Additionally, for current multimedia system requirements, trend charts indicate that bottlenecks at the disk subsystem are due to disk bandwidth rather than storage requirement. This calls for designing a multimedia system which can dynamically find the best engineering point in trading off buffer use with disk utilization.

2.1.1 Continuous Requirement

The bit-rate requirement for streaming data is referred to as *continuous requirement*, which implies that such data must be retrieved or stored in a continuous fashion. To satisfy the continuity requirements of such data, two classes of IO scheduling algorithms have been proposed in literature: *period-based scheduling* and *deadline-based scheduling*. *Quality Proportion Multi-Subscriber Servicing* (QPMS or *time-cycle* based scheduling) [60] and the *Earliest Deadline First*

(EDF) [14] are representative scheduling algorithms. Several improvements to these algorithms have been since investigated [8, 84, 52, 73].

In the *period-based scheduling* paradigm, stream servicing is broken into multiple IOs. IOs for each stream must be performed in time to ensure the continuity of the stream. In the period-based scheduling model, time is split into basic units called *time-cycles* or *periods*. In each time-cycle, exactly one IO operation is performed for the each of the streams serviced by the system. Sufficient data is retrieved or stored for each stream so that there is no jitter in the stream and the data buffers do not overflow or underflow. Within each time-cycle, IOs can be serviced either in *first-come-first-serve* (FCFS) order or the *elevator* order. Figures 2.1 and 2.2 depict time-cycle scheduling based on FCFS and elevator order respectively. In each approach, part of the time-cycle can be reserved to service non-real-time jobs, so that they are not starved.
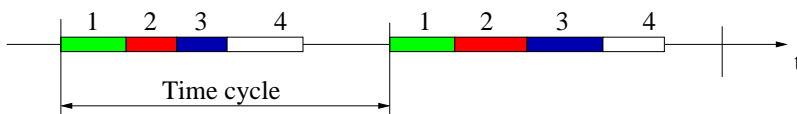
Figure 2.1: FCFS Period-based scheduling.
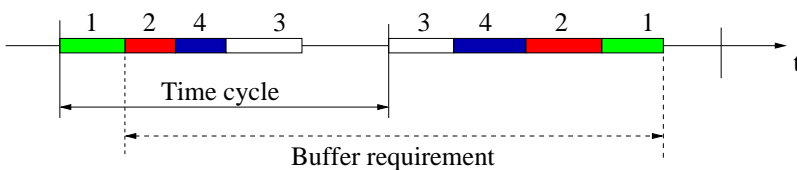
Figure 2.2: Elevator Period-based scheduling.

In the deadline-based scheduling paradigm, stream IOs are represented as individual *tasks*. This paradigm was originally developed for real-time CPU scheduling [45] and eventually adapted to disk scheduling [14, 52]. In this paradigm, each task is represented as the 3-tuple of $< deadline, priority, phase >$. The notion of periodicity of the tasks is imposed by deadlines for the tasks.

13

In the class of deadline-based scheduling algorithms, the rate-monotonic algorithm [45] is proved to be the optimal fixed priority algorithm. According to this algorithm, tasks with shorter deadlines automatically assume higher priority and ties are broken arbitrarily.

To compare the two scheduling paradigms, we note that the deadline-based approaches have well-established CPU scheduling theory as the basic framework. However, they do not have the ease of visualization of the period-based approach. Moreover the memory requirement for streams scheduled using the deadline-based approach is difficult to model. Non-real-time jobs could potentially starve under the deadline-based approach. The period-based approach provides a very natural way to reserve disk-bandwidth for non-real-time or high-priority kernel IOs. For these reasons, in this thesis, unless otherwise mentioned, we use the period-based IO scheduling paradigm.

2.1.2 Data Buffering
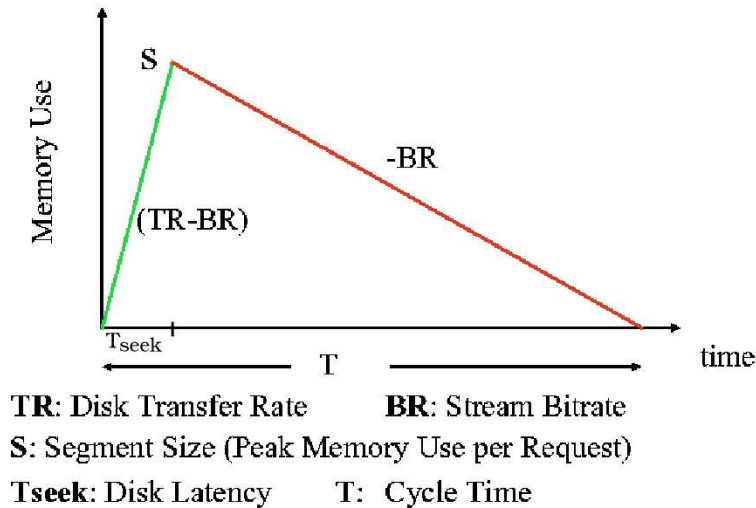
Figure 2.3: Memory model (From [8]).

To account for IO rate-variability, most buffer management schemes use the worst-case bit-rate to reserve buffer space. The amount of buffer space required is proportional to the bit-rate of the stream as well as the duration between successive IOs. If the IO scheduler uses FCFS scheduling, careful management of data buffers [8] can reduce the amount of buffer space required to the size of a single IO request or the duration of a single time-cycle. Figure 2.3 shows how the buffer is filled up and consumed over the duration of a single time-cycle (T). Initially the buffer is filled up at the rate TR. The buffer is also consumed at the rate BR throughout the duration of the time-cycle. Instead of FCFS, if the IO scheduler uses elevator scheduling, due to the variability in IO positions of individual streams in successive cycles (Figure 2.2), the amount of buffer required increases equivalent to two time-cycles.

2.1.3 Interactivity

User interactions with a multimedia system translate into IO operations which must be serviced with the minimum latency. However, the IO scheduler must ensure that the real-time guarantees for existing streams are maintained while servicing the new request. To handle interactive operations with minimum response time, the study of [7] proposes using a reserved IO slot within each time-cycle. This slot is called the *bubble slot*. The bubble slot is always maintained after the current IO slot. When a new interactive request arrives, it is serviced immediately following the servicing of the current IO request, in the bubble slot. The response time for interactive requests is thus reduced to the total time required to service the ongoing IO request and the new IO request.

2.2 Storage Devices and Architectures

Traditional file-system design was focused on a storage system which employed the server-attached disk architecture, in which storage devices, such as disks, are locally attached to servers. Clients access data on disks via the server over the network. However, new storage and delivery architectures are becoming popular in recent years. The Content Delivery Network Architecture [2] tries to solve the network bandwidth problem by replicating content across the Internet in order to reduce the average number of hops to access content as also to solve the scalability problems of traditional server design. Single node servers are being replaced by Cluster Systems [4] since they provide a low-cost solution for scaling. Another emerging trend is that of Storage Area Networks [79, 27]. This architecture proposes the separation of storage devices from servers. It consists of a Storage Area Network (SAN) to which storage devices such as disks are attached; servers access these devices via the storage area network. This architecture allows clients to directly access data from disks, thus excluding the server from the data path.

2.2.1 Primary Storage

We can categorize primary storage as volatile random access memory, which loses its data upon power loss, and nonvolatile memory, such as flash and read-only memories, in which data persists [67].

Random Access Memory

Random access memory can be classified into static and dynamic RAM. Although both are volatile, DRAM memory requires to be refreshed thousands of times per second. SRAM does not require refreshing to protect from data corruption. However, SRAM is physically larger than DRAM and is not practical

for main memory. Main memories are composed of DRAM.

Flash Memory

Flash memory is a popular form of non-volatile storage, i.e. storage that does not lose its contents when power is lost. Some common places where we can find Flash memory being used are: BIOS chip in computers, compact flash (most often found in digital cameras), smart media (most often found in digital cameras), disk drives, PCMCIA cards (used in laptops), and as memory cards for video game consoles. Although flash memory has the size advantage in solid-state storage, it is relatively slow to write, and hence has limited applicability.

2.2.2 Secondary Storage

Hard Disk

Hard disk arrived on the storage scene almost 50 years ago, when introduced by IBM to replace punch cards. In 1973, IBM introduced the first Winchester disk drive, featuring two spindles of 30 MBytes each. Since then, disk drives have maintained themselves as the most popular medium for permanent storage. Nearly every desktop computer and server in use today contains one or more hard-disk drives.

Hard disks have a hard platter that holds the magnetic medium, as opposed to the flexible plastic film found in tapes and floppies. The hard disk drive has a solid place in the random access storage area for cost, reliability, and performance, but its portability and shelf life are concerns. Nevertheless, the seemingly endless innovations in this area indicate that designers will find a cost-effective solution to these concerns. For example, the micro-drive is a 1-inch disk drive that can store a gigabyte of data and could possibly expand the role of disk drives in certain applications.

MEMS Storage

Although hard disks have a solid place in random access storage, new storage technology promises to replace or at least offload part of the disk storage. MEMS-based storage is one such technology which promises to fill the growing performance gap between DRAM and disk drives in today's storage hierarchy.

IBM has developed Millipede [88], a prototype micro-electromechanical system that provides a terabyte of storage on a single chip the size of a postage stamp. The storage medium is a thin polymer film on the chip surface that stores bits as 10-nanometer-diameter holes. Millipede, which features high-density nonvolatile erasable memory suitable for portable digital systems, targets the flash memory market. The tips of the Millipede device used to read and write the media are sculpted into the silicon and write by heating to 400C. They read by heating the film to 300C, which does not melt the polymer, and detecting the cooling effect of dropping into a hole.

Another prototype that has been developed at the Carnegie Mellon University [5] employs magnetic storage media much like those used by disk drives. Data is read/written using MEMS probe tips. The media surface moves linearly in the X and Y directions to seek appropriate data.

MEMS-based storage is a whole new storage technology promising low entry cost, access time, volume, mass, power dissipation, failure rate, and shock sensitivity [5]. Moreover, these devices can integrate computation with storage and thus create complete system-on-chip solutions.

2.2.3 Archival Storage

Tape

Tape was introduced by IBM 50 years ago and stored 1.4 MB of data at that time. Today's high-end tapes store 1 GB of data. Although hard disk prices

have dropped and capacities increased in recent years to make them compete with tapes for long-term storage, hard disks do not match the portability and shelf-life of disk drives.

Recent price decreases and capacity increases in disk drives have made them competitive with tape for long-term storage. However, hard disks do not match tape's portability and shelf life.

Tapes are now being used widely for backup purposes over wide and local area networks. Currently tapes are available as 50 GB cartridges and with 1GB prototypes demonstrated by IBM. Although DVDs will remain competition for low-end tape and archival media, the high-end applications will need tapes with larger capacities and faster access speeds.

Optical Drives: CD and DVD

Optical drives are available in CD and DVD formats. CDs and DVDs are used mostly for backing up data (archiving) or for distributing software. CDs provide 700MB data capacity on inexpensive media and operated by inexpensive drives. CDs are now available as read-only CDs, write-once CD-R, and the newer rewritable CD-RWs.

DVD discs have the same physical dimensions as CDs and come in a range of different physical formats with capacities from 4.7 GB to 17.1 GB. There are numerous DVD formats, however, the three most popular are: DVD-5 (single side, single layer), DVD-9 (single-side, dual layer) and DVD-10 (double side, single layer).

Chapter 3

Modeling Disk Drives and MEMS storage

3.1   Disk Architecture

Before we get into the details of disk features that are of interest when designing high performance systems, we provide a brief overview of the 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

20

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 the 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 the magnetic surfaces (media) used in disks today. Since the outer 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.

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 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. The cache parameters of interest are the segment size, the number of cache segments, the segment replacement policy, prefetching algorithms, and the write buffer organization. Prefetching is used to improve the performance of sequential reads. The write buffer is used to delay the actual writing of data to the disk media and enable the disk to re-schedule writes to optimize its throughput.

3.2   Disk Modeling

In this section, we present methods for extracting certain disk features using Diskbench [21], a disk profiling tool. Diskbench uses a combination of interrogative and empirical methods. Interrogative methods use interrogative SCSI commands to inquire the disk about its features. Empirical methods measure

completion times for various disk access patterns, and extract disk features based on timing measurements.

In some of our extraction methods we assume the ability to force access to the disk media for read or write requests. 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.

3.2.1   Rotational Time

Since modern disks have a constant rotational speed, if 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 to obtain T_{rot}: First, we ensure that read (or write) commands access the disk media. 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 need to wait for one full disk rotation for each successive disk block access, we can write the following equations:

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

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

The rotational period for current disks is much greater than OS delays and other IO overheads (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 negligible.

3.2.2 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 on disk in successive disk rotations, as in the empirical method for extracting T_{rot}. We measure completion times for each read request. For current disks, variations in the rotational period T_{rot} are negligible. Because of this, variations in $T_i - T_{i-1}$ from Equation 3.2 give us 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.

3.2.3 Mapping from Logical to Physical Block Address

Most current SCSI disks implement SCSI commands for address translation (Send/Receive Diagnostic Command [72]) 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 (e.g. ATA disks), empirical methods are necessary.

Interrogative Mapping

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. Additionally, most disks use the skipping method to skip bad sectors (instead of re-mapping them) during the low-level format. Due to this, logical blocks on a track are generally placed sequentially. Thus, we can extract highly accurate mapping information by translating just one address per track, except when we detect anomalies (tracks with bad blocks).

Empirical Mapping

Empirical methods for the extraction of mapping information are needed for disks that do not support address translation. The empirical extraction method used in Diskbench follows. 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.

Disk Zones

Using the extracted disk mapping, Diskbench implements methods for the extraction of zoning information, including precise zone boundaries, the track and cylinder skew factors for each zone, the track size in logical blocks, and the sequential 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 may 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.

Zone	Cylinders	T_{size}	R	R_{max}	$\gamma(1)$	H
1	0-847	254	18.56	21.77	1104	883
2	848-1644	245	18.02	21.00	1108	885
3	1645-2393	238	17.49	20.40	1108	876
4	2394-3097	227	16.70	19.46	1115	890
5	3098-3758	217	15.99	18.60	1115	890
6	3759-4380	209	15.70	17.92	1105	884
7	4381-4965	201	14.83	17.23	1099	875
8	4966-5515	189	13.98	16.20	1124	901
9	5516-6031	181	13.39	15.51	1124	903
10	6032-6517	174	12.89	14.92	1109	886
11	6518-6960	167	12.38	14.32	1119	887

Table 3.1: Seagate ST39102LW disk zone features.

3.2.4 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 also implement an efficient binary-search method in order to find the minimum

access time or seek time. $T_{seek}(x, y)$ returns seek time (in μs) between logical blocks x and y. This function is symmetrical, i.e., $T_{seek}(x, y) = T_{seek}(y, x)$. This seek time also includes the disk head settling time.

3.2.5 Predicting Rotational Delay

In order to optimize disk scheduling, the OS may use both seek and rotational delay characteristics of a disk [37, 93, 46, 51]. We can predict rotational distance between two LBAs using the following:

- mapping information extracted in Section 3.2.3, 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

$$s_{c_i, t_j} = [T(LBA_{start}(c_i, t_j)) - T(LBA_0)] \bmod T_{rot}. \tag{3.5}$$

If the number of spare and bad sectors is small, we can accurately predict the rotational distance $(T_{rot\_del}(y, x))$ between two LBAs (x and y) [21]. Using the seek time $T_{seek}(y, x)$ defined in Section 3.2.4 and the rotational delay prediction, we can predict the access time to a disk block y after access to a block x, $T_a(y, x)$ as

$$T_a(y, x) = T_{rot\_del}(y, x) + T_{rot} \times \left\lceil \frac{T_{seek}(y, x) - T_{rot\_del}(y, x)}{T_{rot}} \right\rceil. \tag{3.6}$$

Apart from these parameters, we also extract read cache and write buffer parameters, like the size of cache segment and number of read cache and write buffer segments.

3.3   MEMS-based Storage

Although disk drives have managed to survive as the most cost-effective medium for storing large amounts of data, new storage technologies are appearing on the horizon. Among new storage technology, MEMS-based storage is one of the most promising. MicroElectroMechanical Systems (MEMS)-based storage devices are very small scale mechanical structures formed by the integration of sensors, actuators, and electronics [88, 31]. These are created using photolithographic processes similar to semiconductor devices. These devices can be made to slide, bend and deflect in response to electrostatic or electromagnetic forces from actuators or from external forces in the environment. Practical MEMS-based storage devices are being developed in several research labs including IBM Zurich Research Laboratory [88], Carnegie Mellon University [5], and Hewlett-Packard Laboratories [34].

Researchers at the Carnegie Mellon University have proposed one possible architecture for a MEMS-based storage device [31, 71] depicted in Figure 3.1. They propose MEMS devices which would be fabricated on-chip, but would use a spring-mounted magnetic media sled as a storage medium. The media sled is placed above a two-dimensional array of micro-electro-mechanical read/write heads (tips). Actuators move the media sled above the array of fixed tips along both the X and Y dimensions. Moving along the Y dimension at a constant velocity enables the tips to concurrently access data stored on the media sled. Using a large number of tips (of the order of thousands) concurrently, such a device can deliver high data throughput. The light-weight media sled of the MEMS device can be moved and positioned much faster than bulkier disk servomechanisms, thus cutting down access times by an order of magnitude.

Table 3.2 summarizes important characteristics of different storage media for the year 2002 and the predicted values for the year 2007. The MEMS device projections are borrowed from [71]; the disk drive projections are based on [83];
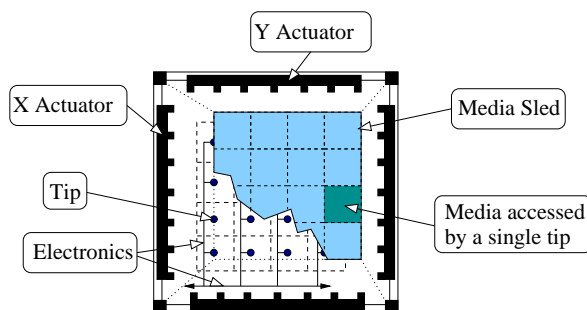
Figure 3.1: MEMS-based storage architecture.

Year		DRAM	MEMS	Disk
	Capacity [GB]	0.5	n/a	100
	Access time [ms]	0.05	n/a	$1 - 11$
2002	Bandwidth [MB/s]	$2,000$	n/a	$30 - 55$
	Cost/GB	\$200	n/a	\$2
	Cost/device	\$50- \$200	n/a	\$100- \$300
	Capacity [GB]	5	10	$1,000$
	Access time [ms]	0.03	$0.4 - 1$	$0.75 - 7$
2007	Bandwidth [MB/s]	$10,000$	320	$170 - 300$
	Cost/GB	\$20	\$1	\$0.2
	Cost/device	\$50- \$200	\$10	\$100- \$300

Table 3.2: Storage media characteristics.

and DRAM predictions are based on [59].

Typically, most storage media are optimized for sequential access. For instance, the maximum DRAM throughput is achieved when data is accessed in sequential chunks, about the size of the largest cache block. These are typically tens to hundreds of bytes. However, both magnetic disks and MEMS-based storage devices (MEMS) have much longer access times than DRAM. These devices have to be accessed in much larger chunks to mask the access overheads, the MEMS device being the faster of the two by an order of magnitude. Figure 3.2 shows the effective throughput of the disk drive and the MEMS device depending on the average IO sizes on these devices. In Figure 3.2 we use the

maximum access times for servicing MEMS IO requests, and the average access times for disk IO requests.
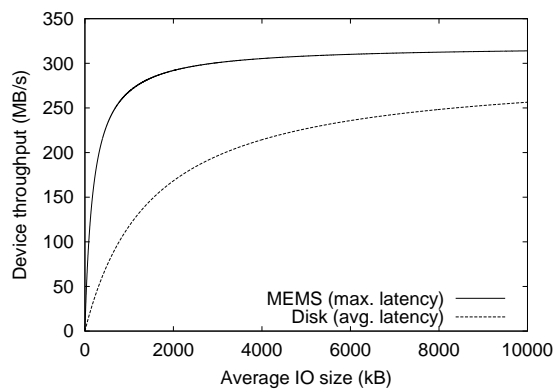
Figure 3.2: Effective device throughputs.

Chapter 4

Quality of Service: Disk Drives

Due to the proliferation of video content, it has become increasingly important to manage video data effectively to facilitate efficient retrieval. On-demand interactive video streaming for education or entertainment over the Internet or broadband networks is becoming popular[8, 38, 89, 44]. In true *video-on-demand* (VoD), a video server allocates a dedicated stream to each user so that the user can freely interact with the video by means of VCR controls (such as pause, fast-forward, and instant replay). But such a system becomes expensive in both network and server bandwidth when tens of thousands of concurrent users have to be accommodated. A more scalable solution is to serve multiple requests for the same video with a single broadcast or a few multicast streams [3, 6, 36].

In this chapter, we present three prototype systems that we have built, the Interactive Media Proxy (IMP, for short), the XTREAM real-time streaming system, and SFinX, a next-generation video-surveillance system. These systems provide methods for data placement, IO scheduling, and admission control for multimedia data. Both these systems are based on accurate modeling and profiling of low-level disk drive features.

4.1 The IMP System

By the nature of broadcast and multicast, end users cannot interact with a TV program or a video using VCR-like controls. We propose an Interactive Media Proxy (IMP) server architecture which acts as a dual client/server system to enable interactivity. As a client of broadcasters (or multicasters), the proxy reduces network traffic to support interactivity. Additionally, it functions as a server by managing streams to enable interactivity for a large number of end users. With interactive capability, students in a virtual classroom or at a library can watch a live lecture at their own pace. A hotel can turn televised programs or movies into interactive ones in its guest rooms. A cable service provider can provide interactive video services to thousands of subscribers. We believe that a proxy architecture is attractive because it not only solves the scalability problem of the traditional server-based VoD model, but also provides a cost-effective solution to user interactivity.

With precise disk information extracted from the disk, data can be placed intelligently, and IO can be scheduled efficiently for improving disk throughput. We now describe how IMP, an Interactive Media Proxy server, takes advantage of an accurate disk profile to perform fine-grained device management for improving system performance. The design of IMP consists of three complementary device management strategies: *high-level data organization*, *low-level disk placement*, and *IO scheduling*. These components of IMP work together to improve disk throughput by minimizing both intra-stream and inter-stream seek delay, and by improving the effective data transfer rate.

In this section, we describe each component of IMP in detail. First, we describe the *adaptive tree scheme*, a high-level data organization scheme for reducing intra-stream disk latency and supporting interactive operations efficiently. This organization scheme forms the basis of our low-level disk placement policy. Next, we present *zoning placement* and *cylinder placement*, which are two

complementary low-level disk placement strategies for placing stream data on disk. These schemes improve effective data transfer rate and reduce the inter-stream disk latency. Finally, we present *step-sweep*, an IO scheduling policy, which takes advantage of the above two components and improves the overall disk throughput.

4.1.1 High-level Data Organization

Without loss of generality, we can assume that an MPEG stream[1] consists of m frame-sequences; each sequence has δ frames on average, led by an I frame and followed by a number of P and B frames.

To support a K-times speed-up fast-scan, the system displays one out of every K frames. Allowing K to be any positive integer, however, can result in the system requiring high IO and network bandwidth, as well as incurring significant memory and CPU overhead. This is because a frame that is to be displayed (e.g., a B frame) may depend on other frames that are to be skipped (e.g., an I and a P frame). The client/server dual system ends up having to read, stage (in main memory), and transmit to a client many more frames than the client displays causing poor IO resolution. (*IO resolution* is the ratio of the useful data read to the total data read from the hard disk.) We intend to remedy this poor IO resolution problem with a high-level data organization scheme, the *adaptive tree* scheme.

To avoid processing the frames that are to be skipped, we do not include any B frames in a fast-scan, and the playback system displays a P frame only if the corresponding I frame is also included in the fast-scan. This restriction will not allow a user to request a fast-scan of any speed-up. Since VCR or DVD players support only three to five fast-scan speeds, we support only a few

---

[1]The Advanced Television System Committee (ATSC) has adopted MPEG2 as the encoding standard of DTV and HDTV.

33

selected speeds of fast-scans.
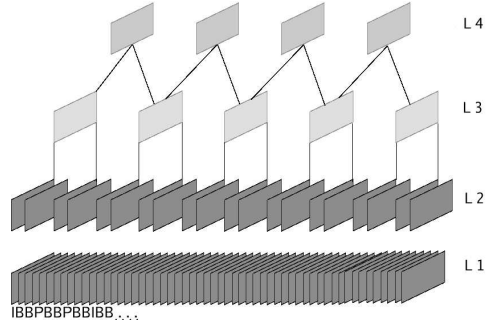
Figure 4.1: The Truncated Binary Tree (TBT) formation.

We now describe the adaptive tree data organization scheme in detail. In the adaptive tree approach for high-level MPEG data organization, we use a basic truncated binary tree structure to store MPEG frames. To provide good IO resolution for all playback speeds, we organize MPEG frames in a truncated binary tree structure, as shown in Figure 4.1. The levels (Li) of the adaptive tree can be described as follows:

• *Level 1 (the leaf level).* The original MPEG stream.

• *Level 2.* All I and P frames stored in their playback sequence.

• *Level 3 to* $(\kappa + 2)$. Containing only sampled I-frames. κ is the number of sub-streams containing only I-frames. The higher the level, the lower the sampling rate.

Each level of the tree forms a sub-stream of the original video stream and is stored as a sequential file on the disk. The placement of the different tree levels on the disk will be addressed in the low-level placement (Section 4.1.2). To service a request, only one level of the tree is read from the disk with good IO resolution.

Let S_i denote the set of frames that belong to the ith level. An example of a five-level organization is:

$$S_1 = \{I_1, B_1, B_2, P_1, B_3, B_4, P_2, B_5, B_6, I_2 \cdots\}$$

34

$$S_2 = \{I_1, P_1, P_2, I_2, P_3, P_4, I_3 \cdots \}$$
$$S_3 = \{I_1, I_2, I_3, I_4 \cdots \}$$
$$S_4 = \{I_3, I_6, I_9, I_{12} \cdots \}$$
$$S_5 = \{I_6, I_{12}, I_{18}, I_{24} \cdots \}$$

When a nine-times fast-scan is requested, we can read in S_3 to achieve perfect resolution. (We assume that an I-frame leads a nine-frame sequence.) When a 54-times fast-scan is requested, we read in S_5 to achieve perfect IO resolution.

This adaptive tree approach trades storage space for IO efficiency. The precise trade-off can be controlled by fine-tuning the following two parameters:

- *Height (h).* The height parameter describes the number of levels (or files) in which the data is organized. Height is a static parameter and it controls the number of fast-scan streams supported. It can be expressed as: $h = \kappa + 2$.

- *Density (η).* Density is a tunable system parameter whose value can range from zero to one. A smaller η value eliminates some tree levels and decreases the tree density. For example, at $\eta = 1/3$, only one out of every three levels of the tree (from the leaf level up) are retained. The higher the density, the higher the IO resolution (favorable) and storage cost (not favorable).

Another factor that affects the disk bandwidth requirement of a fast-scan request is the playback frame rate. A typical DVD player plays a fast-scan stream at 3 to 8 fps (frames per second), instead of 24 to 30 fps, the regular playback speed. A fast-scan stream needs to be displayed at a lower rate so that the viewer can comprehend the content and react in time. Due to its high IO resolution and lesser frame rate, a fast-scan stream under the adaptive tree scheme typically requires lesser disk bandwidth than that of a regular playback stream.

4.1.2 Low-level Disk Placement

Now, we describe how we physically place each stream S_i on disk. Here, our goals are to maximize the effective data transfer rate and minimize disk latency. Our low-level data placement scheme achieves these goals by employing two independent solutions. We utilize *zoning* information to perform *zoning placement* to achieve higher transfer rates, and we perform *cylinder placement* in order to minimize disk latency.

Zoning Placement

In the previous section, we described the adaptive tree scheme for MPEG data organization, which splits a stream into high and low bandwidth sub-streams. Higher bandwidth streams require higher data transfer rates to reduce overall data transfer time. From our experiments in disk feature extraction, we realize that inter-zone bandwidth variations can be as great as 50% (see Table 3.1). Zoning placement performs bit-rate matching of streams to zones.

In order to map streams to disk zones, we first create a set of logical disk zones, Z_l, from the physical zones of the disk. Let these logical zones be numbered from 1 to $|Z_l|$, from the outermost zone to the innermost. We map the physical zones of the disk to one of the ($|Z_l| = \lceil h \cdot \eta \rceil$) *logical* zones, where the η and h are the density and height parameters of the adaptive tree. Each sub-stream is mapped to one of these logical zones in which it is stored.

First, we assume that all streams (and sub-streams) have equal popularity to be accessed. (We will relax this assumption shortly.) Using adaptive tree levels, we divide each video stream into $|Z_l|$ sub-streams of differing bit-rates as described earlier. We group the similar bit-rate sub-streams from different video streams and place them in the same stream group S_i to obtain $S_1, S_2, \cdots S_{|Z_l|}$ such stream groups. Then the objective function to place each stream group S_i

in a unique logical zone Z_j, can be written as:

$$O = \min \sum_{i=1}^{|Z_l|} \frac{\bar{B}_i}{R_j}, \qquad (4.1)$$

where $\bar{B}_i$ denotes the common display rate of all sub-streams grouped in set S_i, and R_j denotes the transfer rate of zone Z_j in which the sub-stream group S_i is placed. However, since all streams do not enjoy the same popularity at all times, *is this the best objective function for optimal placement?*

If we can predict the future with high accuracy, we can possibly rewrite the objective function as:

$$O = \min \sum_{i=1}^{|Z_l|} \frac{\bar{B}_i}{R_j} P(i), \qquad (4.2)$$

where $P(i)$ denotes the access probability of the ith sub-stream set. Unfortunately, predicting $P(i)$ is difficult. For instance, a sports highlight enjoys only a burst of interest.

However, if we look at the problem from a different perspective, zoning placement can provide a bound for the worst-case IO cycle time. A lower worst-case bound is useful because it conserves memory space for staging the streams.

[**Lemma 4.1.1**] *(Zoning Placement)*

Using $i = j$ in objective function (4.1) to place streams achieves the lowest worst-case bound for the total IO time for servicing any N requests.

The formal proof appears in [61]. Here, we illustrate the idea using an example.

[**Example 4.1.2**] *(Zoning Placement)*

Suppose an MPEG file is organized into three sub-streams, and their bit-rates are:

- S_1 (regular speed stream): 6 Mbps.

- S_2 (10 times fast-scan stream): 4 Mbps.

- S_3 (30 times fast-scan stream): 3 Mbps.

37

Suppose a disk has three zones (Z_1, Z_2, and Z_3), and their transfer rates are 150, 100, and 75 Mbps, respectively. Suppose each zone can store only one stream, and the system services three requests in one IO cycle. Placing S_1 in Z_1, S_2 in Z_2, and S_3 in Z_3 gives the system the lowest worst-case data transfer time. To sustain one second of playback for each stream, the total worst-case data transfer time is $3 \times (6/150) = 120$ ms. It is easy to see that if we place S_1 in either Z_2 or Z_3, the total worst-case data transfer time increases.

Although we introduce zoning in the context of storing MPEG video data, we can see that the above principles can be applied to general purpose multimedia file servers that support multiple data types such as images, audio, and video as well as traditional non real-time data. For instance, a text file can be stored in a low bandwidth disk zone, a 256 Kbps *mp3* file can be stored in a medium bandwidth zone, and a 19.2 Mbps HDTV stream can be stored in the high bandwidth zone to maximize disk throughput.

Cylinder Placement

IMP is characterized by N_w broadcast streams which need to be stored on disk and N_r interactive user streams. Careful stream placement on disk tracks can minimize seek and rotational overheads. Using zoning placement, we combine similar bit-rate streams in the same logical zone. Cylinder placement describes how to place data for different streams which share a zone.

In the IMP system, write streams are deterministic in terms of start time, duration, and average data-rate. Read streams depend on user interactions and are inherently non-deterministic and unpredictable. The key idea of the *cylinder placement* strategy is to exploit the deterministic nature of write streams and use a best-effort approach for reads. For each stream, we allocate a group of adjacent cylinders of size c on the disk. Each consecutive write stream is allocated the next c cylinders on disk adjacent to the c allocated cylinders for

the previous stream. When any write stream uses up its allocated c cylinders, a new set of $c \cdot N_w$ free cylinders within the same zone and adjacent to the previous cylinder set is allocated. The write streams are stored in the newly allocated cylinders starting from the next IO cycle. Cylinder placement maintains the same relative cylinder distance between the stream pairs so that the scheduling order can be preserved across IO cycles. This minimizes *IO variability*. Minimizing IO variability is crucial for minimizing memory requirements [8]. Cylinder placement might lead to some fragmentation of disk space. However, we observe that for high bandwidth applications, the disk is bandwidth-bound rather than storage-bound. Hence some storage can be sacrificed for the sake of improving disk throughput. Placing write streams in adjacent cylinder groups has the following advantages:

1. The seek overhead for switching from one write stream to the next write stream requires the disk to seek c cylinders, typically a number less than 50. From our experiments in calculating disk seek time, we note that this overhead is almost equal to the minimum seek time for a single cylinder.

2. The strategy of reserving cylinder groups for individual streams greatly reduces the probability that a single read operation may be non-sequential.

4.1.3 IO Scheduling

For a system to perform optimally under a given disk placement scheme, a closely coupled scheduling algorithm should be in place. Our IO scheduling algorithm, *step-sweep*, is designed to work with the other two components of IMP.
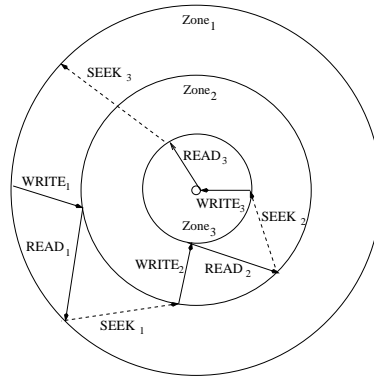
Figure 4.2: Step-sweep IO scheduling.

Procedure: *Step-sweep*

- **Variables:**

  i : Logical zone variable

  Z_l : Set of logical disk zones

- **Execution:**

  1. Initialize $i = 1$

  2. Service write requests in zone Z_i by sweeping in the direction toward the center of the disk.

  3. Service read requests in zone Z_i. These read requests are serviced in a predetermined order to reduce IO variability.

  4. Set $i = (i + 1) \% |Z_l|$

  5. Go to step 2

Given a constant amount of memory and disk bandwidth, step-sweep is designed to: 1) *maximize throughput*, and 2) *minimize response time*. The step-sweep algorithm operates as shown in Figure 4.2. The disk-arm services IO requests

zone by zone, starting from the outermost zone. In each zone, it first services the write requests by sweeping in the direction toward the center of the disk. Read requests in the zone are then scheduled in a pre-determined order which does not vary from one IO cycle to the next. The largest seek overhead in the case of read requests is restricted to the size (in tracks) of a single zone. The disk arm then moves to the adjacent inner zone and repeats this sequence. When the disk arm has serviced all the requests in the innermost zone, it gets re-positioned to the outermost zone of the disk. This completes one cycle of IO requests. This cycle is repeated over and over again. A formal algorithm for the step-sweep scheduling policy is given below.

The design of step-sweep is based on the following considerations. First, we know that the write streams are deterministic in nature. The system has control on where it can place the write streams on disk. Using *Cylinder Placement*, the seek overheads for write streams can be minimized without suffering from IO variability at the expense of some fragmentation of disk space. Thus, step-sweep schedules write streams optimally. Read streams, on the other hand, are unpredictable. Using simple sweep scheduling for read streams might result in large IO variability. Using step-sweep, the service order for read streams in a zone is pre-determined to minimize IO variability [8, 98]. Step-sweep achieves its design objectives in the following manner:

1. To maximize throughput, we need to minimize the IO cycle time as well as the memory use per stream. We define *IO cycle time* as the time required to complete a single round of IO for each stream serviced by the system. Reducing IO cycle time allows the system to service more users in a given period of time. Step-sweep reduces IO cycle time by minimizing seek overhead. When seek overhead is reduced, the system needs to pre-fetch less data to fulfill the real-time playback requirement for each stream, and this reduces data transfer time. Step-sweep minimizes memory use per stream by minimizing IO variability.

2. To minimize response time, we use a reservation time-slot, T_s, within each IO cycle time T. We use this reservation slot to schedule "unexpected" new requests with minimum delay. These unexpected requests might also include requests for non-real-time data. Each new request can be serviced as soon as the current non-preemptible IO operation is finished. Thus the reservation time slot T_s can be used up in small chunks throughout the IO time cycle.

4.1.4   Quantitative Analysis

In this section, we present results obtained from our quantitative analysis. Our quantitative model is developed assuming cylinder placement and step-sweep IO scheduling. In Table 4.1, we enumerate disk parameters along with other parameters of the system.

Let the IO cycle time be denoted by T. This is the time required to complete a single round of IO for each stream serviced by the system. This IO cycle is repeated over and over again in the system. Let the disk support N_w broadcast channels, which perform simultaneous writes, and N_r time-shift streams, which perform reads. Suppose a fraction f of the time-shift streams are fast-scans and the ratio of write to read requests is ρ. Let C be the maximum throughput of the disk. Also, let T_s denote the worst case reservation slot time

If T_{Z_i} is the total time required to service all requests in zone i, and T_s is the reservation time slot, we can express T, the cycle time, as:

$$T = \sum_{i=1}^{p} T_{Z_i} + T_s \tag{4.3}$$

where, $p = |Z_l|$ is the number of logical zones used. In each zone, the scheduler first schedules the write requests, then the read requests. The disk arm is then moved to the beginning of the next zone by seeking to it. If the worst-case zone seek requires T_{zone} time, then:

$$T_{Z_i} = T_{W_i} + T_{R_i} + T_{zone} \tag{4.4}$$

42

Parameter	Description
N_w	Number of broadcast channels (write streams)
N_r	Number of interactive requests (read streams)
ρ	Write-read ratio
f	Fraction of interactive streams (i.e., fast-scans)
R	Minimum disk data-transfer rate
$\gamma(d)$	Worst-case latency function to seek d cylinders
H	Head-switch time
$\bar{B}$	Average display rate of MPEG stream
$\bar{B}_f$	Average display rate of a fast-scan stream
C	Throughput of the disk (number of requests serviced)
T	IO cycle time
T_s	Reservation time-slot
M	Available system memory
Z_l	Logical zone set
α , β	Adaptive tree parameters
κ	Number of exclusive I-frame sub-streams
h	Height of the adaptive tree
η	Density of the adaptive tree
Φ	IO resolution

Table 4.1: IMP system parameters.

Now, we proceed to quantify the write and read times in each logical zone. In each logical zone, Z_i, the disk performs N_w write requests. Thus, the time required to complete write operations in zone Z_i is:

$$T_{W_i} = N_w \cdot \left[\gamma(d_w) + \frac{T \times B(x)}{R_i} \right] \tag{4.5}$$

where d_w is the average seek distance for write operations, R_i is the average transfer rate of logical zone i. $B(x) = \bar{B}$ for writing regular playback streams and $B(x) = \bar{B}_f$ for fast-scan streams. The total read time in the entire cycle is given by:

$$\sum_{i=1}^{p} T_{R_i} = N_r \cdot \left[\gamma(d_r) + T \cdot \left(\frac{(1-f) \cdot \bar{B}}{R_1} + \frac{f \cdot \bar{B}_f}{avg_{i=2}^{p}(R_i)} \right) \right] \tag{4.6}$$

Substituting equations 4.4, 4.5, and 4.6 in equation 4.3, we can obtain a closed form solution for cycle time T.

To fulfill real-time data requirements, each stream, read or write, allocates two buffers. The size of one buffer must be large enough to sustain the playback before another buffer is replenished. The memory requirement for a fast-scan stream is given by $2 \cdot T \cdot \bar{B}_f$ whereas that for write streams and regular-speed read streams is given by $2 \cdot T \cdot \bar{B}$. The total memory usage cannot exceed the available memory M. We can thus quantify the memory requirement as:

$$M \geq 2 \cdot T \cdot \bar{B} \times \left[N_w + (1 - f) \cdot N_r + f \cdot N_r \cdot \frac{\bar{B}_f}{\bar{B}} \right] \qquad (4.7)$$

Given M, $\bar{B}$, $\bar{B}_f$, R, $\gamma(d)$, f, Φ and the write-read request ratio, ρ, we can use Equations 4.3 and 4.7 to estimate the throughput of the disk.

4.1.5 IMP System Evaluation

In this section, we evaluate the performance of the IMP system. First, we define the performance metric. Given a system with fixed resources, we would like to maximize the number of video streams that can be supported simultaneously. Hence we measure the system performance in terms of the number of streams supported by the disk. We refer to this performance metric as *disk throughput*. We performed the following experiments to evaluate the disk throughput.

System Configuration Evaluation. We performed a case study on three sample configurations of the *adaptive tree* data organization scheme (Section 4.1). We studied the effect of the following three system parameters on the throughput (N):

- Available system memory (M),
- Write-read ratio (ρ), and
- Fraction of interactive streams (f).

Keeping two of the above parameters fixed, we examined the effect of changing the third.

Data Management Strategy Evaluation. We examined the individual as well cumulative effect of the following fine-grained device management strategies on the throughput (N):

- Zoning placement,

- Cylinder placement, and

- Step-sweep IO scheduling.

For experimental evaluation of the IMP system, we used disk trace support provided by our Diskbench tool [21]. Diskbench supports the execution of disk traces using primitive disk commands like *seek*, *write*, and *read*. It also provides for accurate timing measurement. Using Diskbench, we performed trace executions on the Seagate ST39102LW disk presented in Table 3.1. In some cases, where trace executions were not possible, we evaluate the system using analytical estimation.

System Configuration Evaluation

In this section, we report a case study of three sample configurations of the *adaptive tree* data-organization scheme (Section 4.1), each of which is designed to optimize on a subset of the design parameters so as to perform optimally for a specific class of target applications, introduced in Section 4.1.

1. *Truncated Binary Tree or TBT ($\eta = 1$).* This is the normal configuration in which all levels of the truncated binary tree are stored. The advantage of this scheme is that we have prepared streams for supporting each fast-scan speed and hence can achieve 100% IO resolution and minimum disk latency at the same time. However, due to replication of data, storage cost increases. A *local-area interactive service* supporting a large number of clients could use

this configuration.

2. *Partial TBT or PTBT ($\eta = 0.5$).* In the PTBT configuration, the tree is partially dense. With $\eta = 0.5$, we store only alternate levels of the tree. Thus, in this configuration, there are prepared streams only for some of the fast-scan speeds. A fast-scan stream that is not directly accessible is created by selectively sampling the frames in another fast-scan stream, which serves a lower speed. Such streams suffer from a degraded IO resolution. The number of files to be written into (i.e., the number of seeks by the disk-arm) for each stream is $\lceil \eta \cdot h \rceil$. This scheme serves as a middle-ground between the SEQ and TBT configurations and could be used by *@HOME* type applications.

3. *Sequential or SEQ ($\eta = \frac{1}{h}$).* In the sequential configuration, only one out of h levels is stored to obtain a tree with density $\frac{1}{h}$. The sequential configuration stores only Level 1 of the tree. Higher levels of the tree are simply not stored. The goal of this scheme is to reduce seek overhead for writes, thus conserving memory use in write-intensive applications. However, this scheme suffers from very poor IO resolution for fast-scans. This scheme would be practical for a *video surveillance* type application.

Thus, each scheme aims to optimize a different subset of the design parameters. In Tables 4.2 and 4.3, we summarize each configuration's pros (with positive signs) and cons (with negative signs). IOR represents the IO resolution.

Scheme	Seek Overhead	IOR	Storage
Sequential	++	−−	N/A
TBT	++	++	N/A
PTBT	++	+	N/A

Table 4.2: Scheme summary for Read operations.

Available System Memory

In this section, we compare the memory requirement for each of the three sample

Scheme	Seek Overhead	IOR	Storage
Sequential	++	N/A	++
TBT	−−	N/A	−−
PTBT	−	N/A	−

Table 4.3: Scheme Summary for Write operations.

configurations of the adaptive tree data-organization scheme using disk traces. These disk traces were generated so as to mimic the IO load of a media server. The trace executions were performed using the data management strategies of the IMP system.

Assuming a given amount of main memory, first we calculated the maximum IO cycle time expendable to support N users. Next, we obtained the actual IO cycle time from disk runs. If the disk runs were shorter than the analytically computed IO time cycle, we could conclude that the disk can support N users. If not, we would repeat the above experiment with $(N-1)$ users. We continued this iterative process till we obtain a feasible N, so that all the user streams are hiccup-free. We repeat the same experiment by assuming different memory sizes. For the following traces conducted using the Diskbench tool, we assumed that the fraction of user requests that are for fast-scan streams is 0.2. We assumed that the peak data consumption and input rates are 6.4 Mbps (the Standard DTV broadcast rate). Further, we assumed that the frame-rate for fast-scans was 5 fps (please refer to Section 4.1 for why we chose a lower frame-rate for fast-scan streams). We will use the same numbers for all subsequent experiments unless others are specified explicitly.

Figure 4.3(a), 4.3(b), and 4.3(c) present the disk throughput for the SEQ, PTBT, and TBT configurations respectively against varying memory size (M) and for different values of the write-read ratio ($\rho = 0.25, 0.5, 1, 2, 4$). We note that for the SEQ scheme, we need as little as 32 MBytes of memory in order to maximize disk throughput. For the PTBT and TBT schemes the corresponding

(a) SEQ

(b) PTBT

(c) TBT
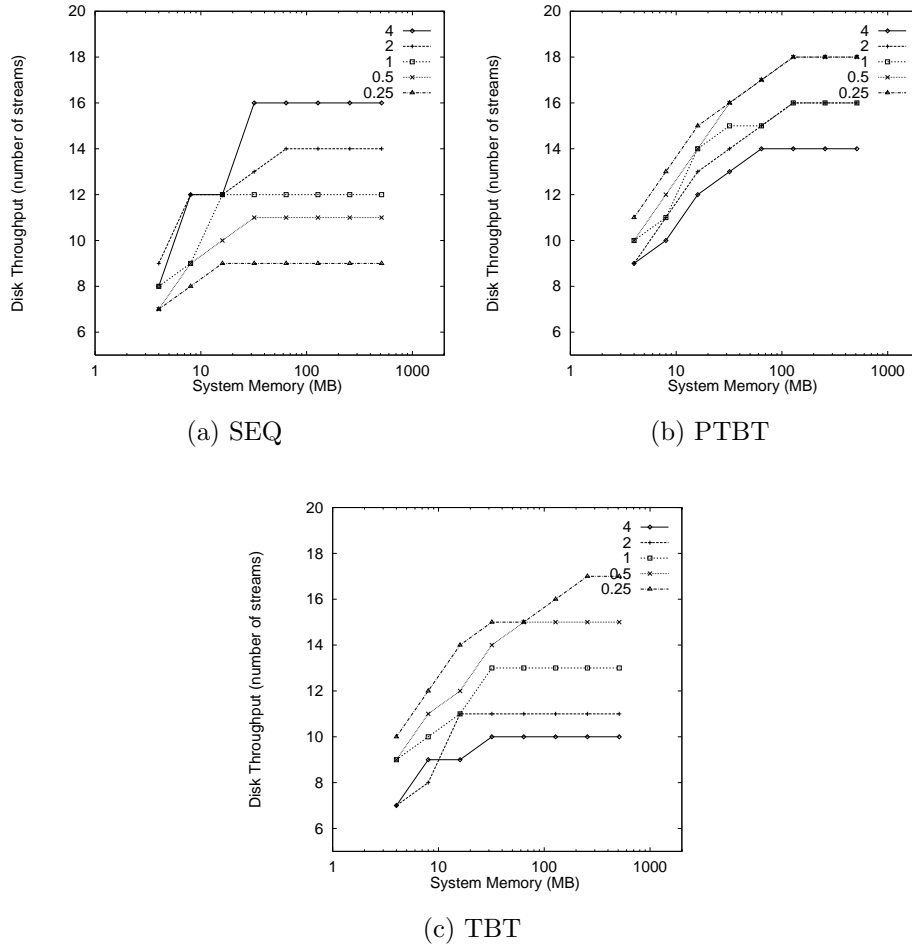
Figure 4.3: Disk throughput for adaptive tree configurations.

numbers are 128 MBytes and 64 MBytes respectively. Looking at these figures from a different perspective, a mere 32 MBytes of main memory is sufficient to drive the disk throughput to almost 90% of the maximum achievable value. This is the case because for such an amount of memory, the system can buffer enough data so that all disk accesses are in large chunks. When the disk is accessed in large chunks, disk latency is much less in comparison to the time

48

spent in data transfer. At this point, the bottleneck is the raw data transfer rate of the hard drive, which can support only a fixed maximum number of high bandwidth streams.

Write-Read Ratio

In this section, find out the variation in disk throughput under different write-read ratios using disk traces. We also compare the IMP system against the traditional UNIX-like file-system that tries to store file data sequentially on disk and present the improvement in performance. Also, we present analytical results for wider ranges of the write-read ratio.

(a)   (b)
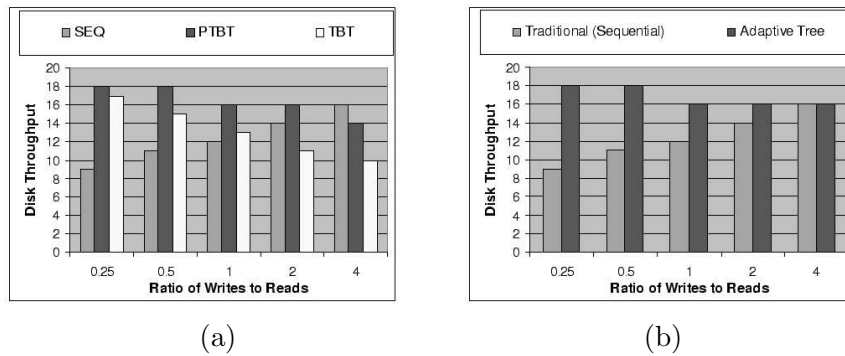
Figure 4.4: Throughput comparison for (a) Different configurations of the adaptive tree scheme, and (b) Improvement over traditional sequential placement of data.

Figure 4.4(a) compares the relative performance of the IMP system in each of the sample configurations: SEQ, PTBT, and TBT. Since the SEQ scheme is optimized for a large number of writes, it achieves a high throughput for write-intensive loads. PTBT performs optimally in mid-ranges and TBT performs well for read-intensive loads. For different values of ρ, different configurations achieve the highest throughput, thus defining a distinct *ideal* region.

Next, we compare performance improvement of the IMP system over the traditional approach to storing and retrieving data. A traditional operating

49

system like Unix is optimized for sequential access. However, for an interactive video application, access to data is not always sequential. For instance, a fast-scan stream needs to access only key frames from the entire file. In addition to providing interactive capability, the IMP system provides "fine-grained" device management strategies for adapting to changing request workload. Figure 4.4(b) shows that for different workload configurations, IMP either outperforms or equals the performance of a traditional file-system. Throughput gains can be as much as 100% using IMP.
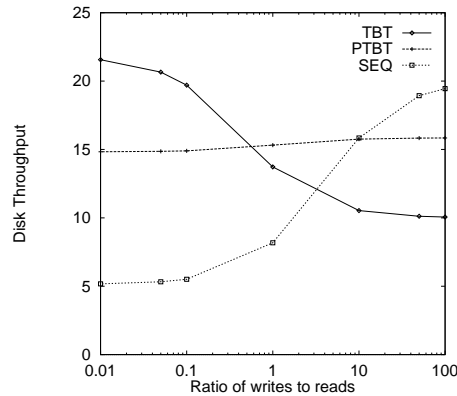
Figure 4.5: Throughput for the three sample configurations.

Since trace driven executions restrict the available parameter space for the write-read ratio, we perform further evaluation of the three sample configurations by analytical estimation. We use parameters for the Seagate ST39102LW disk, presented in Table 3.1, and seek-curves obtained using our disk profiler, to perform our analysis. In Figure 4.5, the x-axis represents the write-read ratio, and the y-axis shows the disk throughput achievable by the configurations. We can see clearly that for different values of ρ, different configurations achieve the highest throughput, thus defining a distinct *ideal* region. This means that the density of the adaptive tree has to be configured dynamically for optimal performance.

Data Management Strategy Evaluation

In this section, we perform an evaluation of our three fine-grained data management strategies: *zoning placement*, *cylinder placement*, and *step-sweep IO scheduling*. We compare the IMP system to a traditional UNIX-like file-system that uses *sweep* scheduling. Most modern operating systems use sweep algorithm for disk IO scheduling, in which the disk arm moves from the outermost cylinder to the innermost, servicing requests along the way. Then, we add our device management strategies, one at a time, to examine the marginal improvement in throughput due to each strategy. Finally, we present the cumulative effect of these disk management strategies, in increasing the throughput of the system.

In order to make a fair comparison, we give the traditional system the benefit of using the adaptive tree data organization scheme to achieve good IO resolution. In addition, for each of the following evaluations, we assume that the system is dynamically tuned to the "ideal" adaptive tree configuration under changing write-read ratios. In essence we try to capture the effect of fine-grained device management of the IMP system in the form of *zoning placement*, *cylinder placement*, and *step-sweep* IO scheduling on throughput improvement.

Zoning placement

Figure 4.6(a) compares the throughput of the baseline sweep with sweep using zoning placement. Zoning placement improves throughput for read-intensive loads by as much as 65%. For write-intensive loads, our adaptive tree scheme adopts the ideal sequential (SEQ) configuration and hence zoning has no effect.

Cylinder placement

Figure 4.6(b) compares the the throughput of the baseline sweep with and without cylinder placement. Cylinder placement improves write performance by reducing the seek overhead for write operations. It increases the throughput by about 10% for write-intensive loads.

51

(a) Zoning placement   (b) Cylinder placement

(c) Step-sweep scheduling   (d) Total improvement
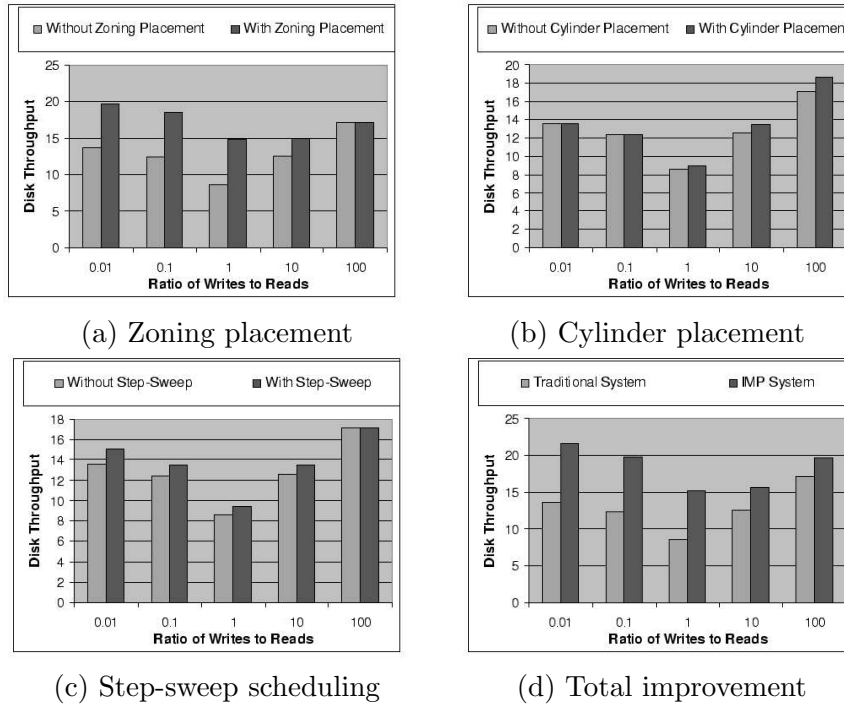
Figure 4.6: Throughput improvement due to device management strategies.

Step-sweep IO scheduling

Next, we examine the marginal effect of step-sweep algorithm in comparison with traditional sweep. Step-sweep increases throughput by decreasing memory use. It reduces memory use by minimizing IO variability and seek overhead. Figure 4.6(c) shows that without step-sweep scheduling, there is a 5 to 10% degradation in throughput.

Cumulative Effect

Finally, Figure 4.6(d) shows the cumulative effect of our fine-grained device management strategies. The IMP system offers a performance gain of as much as 75% over traditional systems. This highlights the importance of performing fine-grained storage management using the IMP system.

As regards response time, a detailed evaluation is beyond the scope of this paper and will be left to future work. However, it is clear that in case of step-sweep, which serves new requests immediately in a reservation time-slot, the worst-case response time is bounded by the time required to complete the current non-preemptible IO. This time is typically in the order of tens of milliseconds. Traditional IO schedulers like sweep as well as GSS [98], have worst-case response times of the order of minutes under heavy load, and at least of the order of seconds under average load conditions.

Observations

We conclude our evaluation section by making the following key observations:

- High-level data organization in the form of the adaptive tree scheme is crucial to maintain disk throughput. The effect of the system configuration on the performance of the IMP system is summarized as follows:

  1. Increasing the system memory beyond a certain threshold does not significantly increase system throughput. At this point, the disk throughput becomes the bottleneck. Beyond this threshold, the disk throughput can only be increased by increasing IO efficiency using the adaptive tree scheme for data organization.

  2. Using the adaptive tree MPEG data organization scheme, the IMP system can optimize for read-intensive as well as write-intensive loads. The achieved throughput is as much as 100% better than that of a traditional file-system for a large range of write-read ratios.

  3. Each sample configuration [SEQ, PTBT, and TBT], of the adaptive tree scheme operates efficiently for a unique sub-configuration defined by the values of write-read ratio (ρ) and interactive stream fraction (f).

- We evaluated the performance gain due to our data management strategies: *zoning placement*, *cylinder placement*, and *step-sweep IO scheduling*. In summary,

  1. Zoning placement matches stream bit-rates to disk zone transfer rates so as to maximize data throughput of the disk for serving continuous media streams. It improves throughput by as much as 65%.

  2. Cylinder placement improves write performance by reducing the seek overhead for write IOs. It increases the throughput by about 10% for write-intensive loads.

  3. Step-sweep increases throughput by decreasing memory use. Throughput gains range from 5 to 10%.

  The cumulative effect of the above strategies makes it possible for the IMP system to offer performance gains of as much as 75% over traditional systems.

In this section, we present the design and implementation of XTREAM, a real-time streaming storage system. Traditional file systems are optimized for supporting good interactive performance and high IO throughput. Multimedia systems place additional real-time requirements on disk performance. In these systems, data must be retrieved from disk and played back by a specific deadline, or else end users experience unacceptable video jitters or audio pops. A multimedia user also expects fast access to content, which translates to a low initial latency requirement for storage access. Thus, streaming multimedia presents the often conflicting requirements of real-time delivery, high throughput, and short initial latency. For example, reducing the size of disk requests reduces the initial latency and the required memory buffer, but this may degrade disk throughput. In this paper, we present the implementation of XTREAM, a streaming multimedia system that can achieve the three performance requirements—high throughput, low initial latency, and guaranteed

IO—at the same time.

To guarantee high throughput, XTREAM uses an *IO Scheduler* module for servicing disk IOs. To offer low initial latency, a *Request Scheduler* component services new requests with high priority. To guarantee quality of service (QoS) to multimedia streams, XTREAM employs an *Admission Controller* which ensures that all IO requests can be completed in time and that the system is not under-utilized. The design of XTREAM is based on accurate disk drive modeling, using our disk profiling tool [21].

XTREAM runs in the user mode. It supports heterogeneous streaming media types (with different bit-rates) as well as non-real-time data retrieval. It supports guaranteed-rate IO for both write (e.g., recording by a surveillance camera) and read streams (e.g., mp3 or video playback). The XTREAM scheduler supports servicing non-real-time data like text or HTML while meeting all real-time streaming requirements.

4.1.6   XTREAM Design

We now provide an overview of the design of the XTREAM system. 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 server and clients.

As shown in Figure 4.7, 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.
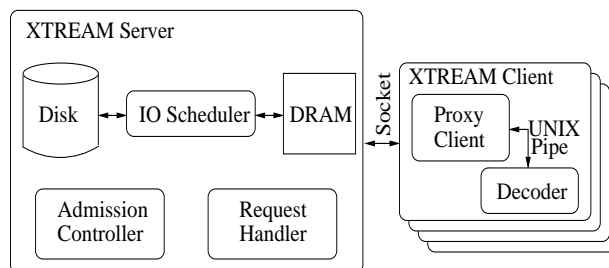
Figure 4.7: XTREAM system architecture.

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 [65] 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. In addition, XTREAM uses a *Disk Profiler* [21] 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 [65], 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 4.1.6). 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.

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:

1. 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 [18].

2. The double buffering scheme also frees IO scheduler from using *fixed-stretch* [8], 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.

Using this scheme, the request handler can service a new IO request stream as soon as the current IO request is completed by the disk drive. To further decrease the response time, the notion of *semi-preemptible IO* [18] can be used. Semi-preemptible IO is an abstraction for 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.

Admission Controller

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 4.8 depicts the available *slack* in each time cycle for two scenarios. Figures 4.8(a) and 4.8(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
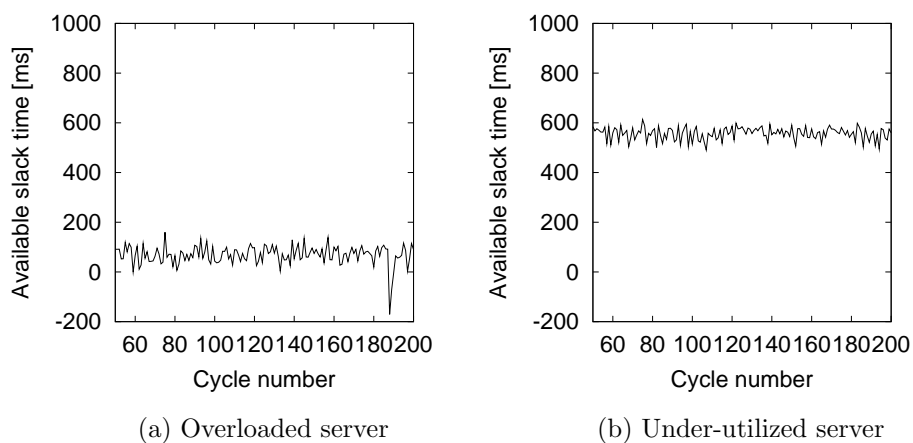
Figure 4.8: Available slack in each time cycle.

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 the disk performance varies significantly depending on the disk access pattern and the file-system data placement policies. However, to remain independent of the underlying file-system, XTREAM does not make any as-

sumptions 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 4.8 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.

$$U = \frac{T_{transfer}}{N \times T_{access} + T_{transfer}} \tag{4.8}$$

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}).
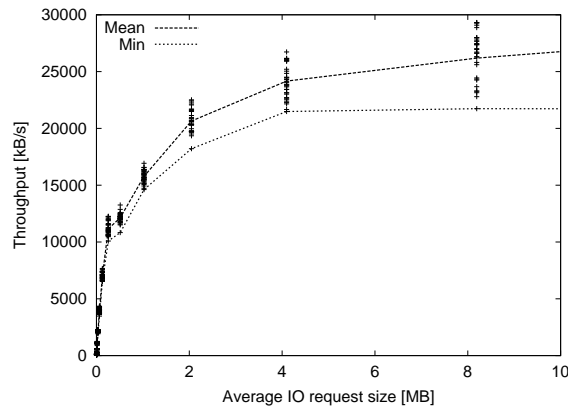
Figure 4.9: Disk throughput vs. average IO size.

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 4.9 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 B_i. When a new request arrives (with required bit-rate B_{new}), the admission controller first calculates the new average IO request size using Equation 4.9.

$$S_{avg} = \frac{T \times (B_{new} + \sum_{i=1}^{N} B_i)}{N + 1} \qquad (4.9)$$

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 4.9). Then, if the condition in Equation 4.10 holds, the new request is accepted.

$$P(S_{avg}) > B_{new} + \sum_{i=1}^{N} B_i \qquad (4.10)$$

4.1.7 XTREAM Results

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 use 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 is 31 MBps in the fastest zone and 21 MBps in the slowest zone. The LAN is 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 are not streamed over the network. The client can specify whether the dummy stream is read or write, and whether the bit-rate is constant or variable.
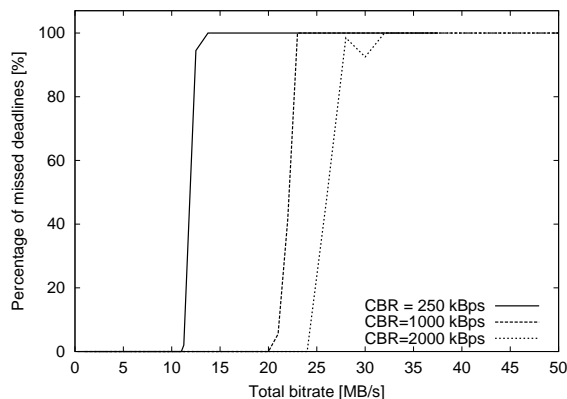
Figure 4.10: Percentage of missed cycle deadlines.

Throughput

Figure 4.10 shows 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 have the same bit-rate. Larger bit-rates result in larger disk IOs, and consequently higher disk utilization (See Figure 4.9). In each experiment, we use 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 is the maximum value on x-axis when the system does not miss any deadlines. Depending on the required QoS, the admission control module can choose an appropriate maximum throughput (total bit-rate) for the disk. Thus, the trade-off between QoS and system throughput decides the admission control policy. Table 4.4 shows Xtream accuracy for one conservative and two aggressive admission control policies.

Initial Latency

In this paper we define *initial latency* as the delay between the moment the request handler receives a client request, and the moment when the initial buffer for the new stream is filled. Data on the initial latency, depending on the number of streams in the system, are presented in Figure 4.11. The initial

latency does not depend on the load of the system but only on the size of IO requests (which depends on the stream bit-rates). The following results do not consider or evaluate network or client-side latency.
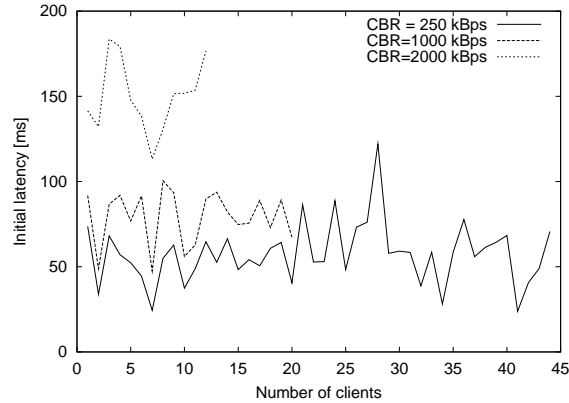
Figure 4.11: Initial latency.

Guaranteed IO

For a streaming multimedia system to guarantee QoS for IO, its admission control policy must be accurate. In this section, we evaluate the three different admission control configurations introduced in Section 4.1.6: conservative admission control ($Adm1$) and two aggressive admission controls ($Adm2$ and $Adm3$). $Adm1$ uses the min curve from Figure 4.9 to perform admission control. $Adm2$ uses the mean curve from Figure 4.9 and maximum stream bit-rates. $Adm3$ uses mean curve from Figure 4.9 and average stream bit-rates.

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 heterogeneous 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. $MaxN$ is calculated manually within each experiment using trial and error. The results

63

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 4.4: Admission control accuracy.

presented in Table 4.4 show that XTREAM provides QoS for disk IO while not significantly under-utilizing the system.

4.2 The SFinX Video Surveillance System

Video surveillance has been a key component in ensuring security at airports, banks, casinos, and correctional institutions. More recently, government agencies, businesses, and even schools are turning toward video surveillance as a means to increase public security. With the proliferation of inexpensive cameras and the availability of high-speed, broad-band wired/wireless networks, deploying a large number of cameras for security surveillance has become economically and technically feasible. However, several important research questions remain to be addressed before we can rely upon video surveillance as an effective tool for crime prevention, crime resolution, and crime prosecution. SfinX (multi-Sensor Fusion and mINing Xystem) aims to develop several core components to process, transmit, and fuse video signals from multiple cameras, to mine unusual activities from the collected trajectories, and to index and store video information for effective viewing [22].

The current state-of-the-art in commercial video surveillance equipment typ-

ically consists of analog cameras and tape-based VCRs which are functionally very limited. For instance, these systems do not support simultaneous recording and reviewing of camera data. Analog data on tape must be first converted to digital format before it can be subjected to further analysis. Moreover, retrieval of archived videos is manual and therefore time-consuming. All these issues make current commercial systems obsolete. Current and future surveillance systems must be all digital, capable of handling multiple simultaneous viewing and recording sessions, automatically detect suspicious activity, and most of all, be affordable. To this end, we propose to use cheap off-the-shelf digital video cameras and desktop computers to store, retrieve, analyze, and query the captured videos. Our architecture requires only one high-end camera (with zoom and motion capabilities) per physical location for tracking objects or humans in close-up.

The target application that we intend to support would not only be able to support viewing video streams in real-time, but also support 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 propose the architecture of a next-generation video-surveillance system which not only supports real-time monitoring and storage of all the video streams, but also performs video analysis and answers semantic database queries. We analyze each component of the proposed architecture and present the research problems that need to be solved in order to build a successful video-surveillance system. We present preliminary results of the performance of certain components of the system.

In recent times, there has been a renewed interest in designing all digital

video surveillance systems [13, 22, 25, 78, 94, 99]. However, a number of research problems remain to be solved before we can build efficient and reliable surveillance systems. We outline the major components within SfinX and associated research problems in Section 4.2.2.

4.2.1 System Architecture

In this section, we introduce the hardware and software architecture of the SfinX system.
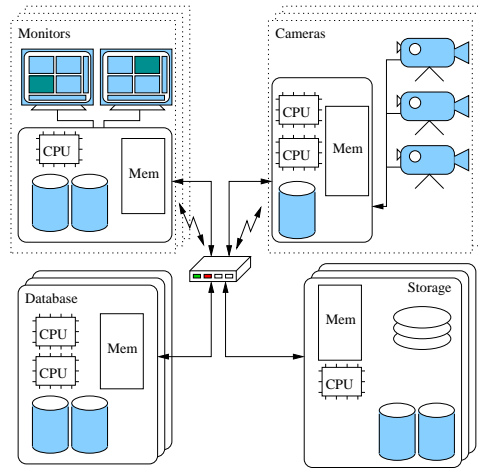
Figure 4.12: Hardware architecture.

Figure 4.12 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 activities are 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

66

retrieve and inspect video-clips of interest.
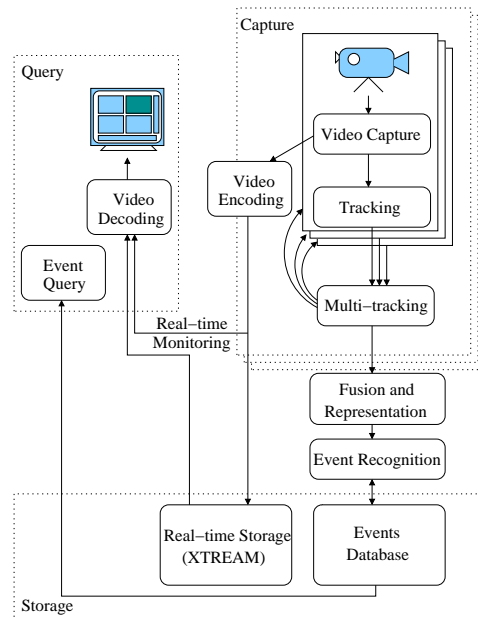
Figure 4.13: Software architecture.

Figure 4.13 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 *encoded* and sent off to be stored onto Xtream [17], 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. There exist multiple multi-trackers, which track objects in physically disjoint 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

67

module represents the trajectory of each object using *sequence data representation* [94]. This information is stored in the *events database* for future reference.

The user-interface consists of two distinct components. First, the *real-time monitoring* component using which a user can view live camera feeds as well as interact with live feeds to scan through the stream. This helps the user to immediately track objects by moving through the stream at will. Second, the viewer can also analyze the stored video streams by performing database queries. An example of such a query was presented in Section 4.2. Controlling the query semantics, the user can get detailed information from the database.

4.2.2   System Components

In this section, we 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. Similar to a previous study [99], we use a single high-end camera per location with 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 calibration [25]. Ideally, this must be an automatic process, that maps the camera coordinates to coordinates in the physical location. In addition, the close-tracking high-end camera must be perfectly calibrated 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 (e.g., H.263, MPEG1, or MPEG4). Each stream is then stored using a real-time storage system like Xtream [17] for future viewing purposes. The storage system provides real-time stream retrieval and supports scan operations (e.g., 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 (e.g., for each database update) or long-term (e.g., 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 [60] 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 [18].

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 the 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 [13, 78]. 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 which monitor 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 in 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* [94]. To arrive at a reasonable representation, the trajectory of each object is smoothed using Kalman filters [39] 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 (e.g., 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 combinations 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.

4.2.3 Results

In this section, we present results obtained while measuring camera performance, video compression efficiency, network capability, and storage performance.

Camera Performance

Table 4.5 presents the performance characteristics of the high-end camera that we currently use to track objects in close-up.

Parameter	Value
Camera model	Sony EVI-D30
Output resolution	460x350 NTSC tv lines
Pan range	193.75 degrees (specs say 200 degrees)
Maximum pan speed	80.7 degrees/sec (specs say 80)
Pan accuracy	±0.45 degrees approx.
Tilt range	47.36 degrees
Maximum tilt speed	52.6 degrees/sec (specs say 50)
Tilt accuracy	±0.22 degrees approx.
Zoom ("tele" and "wide")	1x to 12x at 6 speed settings

Table 4.5: Measured performance parameters for the Sony EVI-D30 high-end camera.

Compression results

Of the four compression methods we tested (H.263, MPEG4, MSMPEG4, and MPEG1), all were within approximately 8% CPU usage of each other. MSMPEG4 was the slowest, though it did exhibit the best quality. Here we present the compression results using MPEG1 encoding. Experiments were carried out on an Intel P4 2.66GHz using the open source video encoder *ffmpeg*.

We notice an interesting trend in Figure 4.14, which depicts the CPU utilization to compress video MPEG1 at 15 fps at different resolutions. A larger resolution video naturally requires more CPU. Also, the larger the target bit-rate, the larger is the CPU usage since the parameter space for the compression algorithm increases with the target bit-rate. In addition to this chart, we found that capturing at 15fps uses a little over half of the CPU as capturing at 30fps.
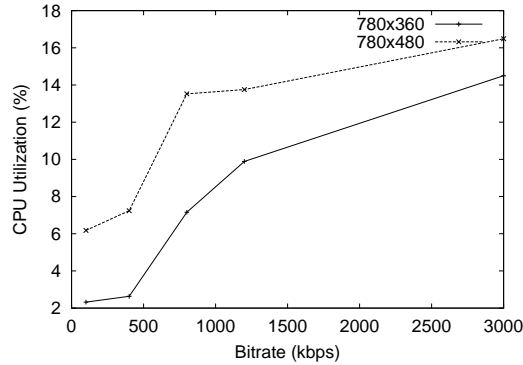
Figure 4.14: CPU utilization for compression.

Bit-rate (kbps)	# Streams
100	919
400	229
800	114
1200	76
3000	30

Table 4.6: Throughput of 100 Mbps network.

Network streaming results

Table 4.6 gives an estimate about the number of streams that can be supported at various bit-rates over a 100Mbps local area network. The network is switched Ethernet and the switch is a HP 2324 Pro-curve box.

4.3 Summary

In this chapter, we presented three prototype real-time storage systems that we have designed and implemented based on an accurate low-level profiling of hard-disk performance parameters obtained from Diskbench [21], our disk profiling tool. These systems incur little overhead and offer significantly superior

real-time support than traditional file systems and make the case for designing future storage systems with real-time capability.

Chapter 5

Quality of Service: MEMS-based storage

5.1 MEMS-based Streaming

Streaming data is characterized by the dual requirements of guaranteed-rate IO and high throughput. To service multiple streams, the disk-drive bandwidth is time-shared among the streams. However, this time-sharing degrades disk throughput. Any system designed for servicing streaming data must address the inherent trade-off between disk throughput and data buffering requirements.

In this paper, we adopt the time-cycle based scheduling model [60] for scheduling continuous media streams. In this model, time is split into basic units called IO cycles. In each IO cycle, the disk performs exactly one IO operation for each media stream. Given the stream bit-rates, the *IO cycle time* is the amount of time required to transfer sufficient amount of data for each stream so as to sustain jitter-free playback. The IO cycle time depends on the system configuration as well as the number and type of streams requested. To service multiple streams, the IO scheduler services the streams in the same order in each time-cycle. Careful management of data buffers and precise scheduling [8]

can reduce the total amount of buffering required to the amount of data read in one time-cycle.
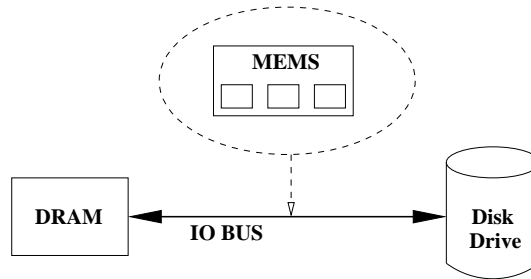
Figure 5.1: The MEMS Architecture.

In traditional multimedia servers, the buffering requirement is addressed using the system memory (DRAM). MEMS-based storage devices can be used to offload part of the DRAM buffering requirement. They can also be used as caches to provide faster access to multimedia content. Figure 5.1 illustrates the new system architecture that includes the MEMS device in the storage hierarchy. The MEMS storage module can consist of multiple MEMS devices to provide greater storage capacity and throughput. The MEMS device is accessed through the IO bus. It can be envisioned as part of the disk drive or as an independent device. In either case, IOs can be scheduled on the MEMS device as well as on the disk drive independently. Similar to disk caches found on current-day disk drives, we assume that MEMS storage devices would also include on-device caches. In what follows, we present two possible scenarios in which such an architecture can be used to improve the performance of a multimedia system.

5.1.1 MEMS Multimedia Buffer

Using MEMS storage as an intermediate buffer between the disk and DRAM enables the disk drive to be better utilized. At a fraction of the cost of DRAM, MEMS storage can provide a large amount of buffering required for achieving

high disk utilization (see Figure 3.2). Although DRAM buffering cannot be completely eliminated, the low access latency of MEMS storage provides high throughput with significantly lesser DRAM buffering requirement. The MEMS device can thus act as a speed-matching buffer between the disk drive and the system memory, in effect addressing the disk utilization and data buffering trade-off.

Using MEMS storage as an intermediate buffer implies that the MEMS-based device must handle both disk and DRAM data traffic simultaneously. To understand the service model, let us assume that the multimedia streams being serviced are all read streams, so that stream data read from the disk drive is buffered temporarily in the MEMS device before it is read into the DRAM. This model can be easily extended to address write streams.

To service buffered data from the MEMS device, we use the time-cycle-based service model previously proposed for disk drives. Data is retrieved in cycles into the DRAM such that no individual stream experiences data underflow at the DRAM. At the same time, the data read from the disk drive must be written to the MEMS device. The disk IO scheduler controls the read operations at the disk drive. The MEMS IO scheduler controls the write operations for data read from the disk as well as read operations into the DRAM. In the steady state, the amount of data being written to the MEMS device is equal to the amount read from it. The MEMS bandwidth is thus split equally among read and write operations. Thus, although the MEMS device can help improve disk utilization, we must realize that to do so, it must operate at twice the throughput of the disk drive. In order to minimize buffering requirements between the disk drive and the MEMS storage, the disk and the MEMS IO schedulers must therefore co-operate.

The MEMS buffer could consist of multiple physical MEMS devices to provide greater buffering capacity and throughput. As we shall see in Section 5.3, a bank of MEMS devices may be required to buffer IOs for a single disk. The (pre-

dicted) low entry-cost of these devices makes such configurations practical. We now present a feasible IO schedule that maintains real-time guarantees, when a single MEMS device is used for buffering. We then extend our methodology to work with a bank of MEMS devices.

IO Scheduling: Single MEMS

We now present one possible IO scheduler which guarantees real-time data retrieval from the disk using a single-device MEMS buffer. The MEMS IO scheduler services IOs on the MEMS device in rounds or *IO cycles*. In each IO cycle, the MEMS device services exactly one DRAM transfer for each of the N streams serviced by the system. The amount of data read for each stream is sufficient to sustain playback before the next IO for the stream is performed. Further, the MEMS device also services M transfers ($M < N$) from the disk in each IO cycle. In the steady state, the total amount of data transferred from the disk to the MEMS buffer is equal to that transferred from the MEMS buffer to DRAM. Thus, there exist two distinct IO cycles, one for the disk (the *disk IO cycle*, during which N IOs are performed on the disk-drive) and the other for the MEMS buffer (the *MEMS IO cycle*, during which N IO transfers occur from the MEMS buffer to the DRAM).
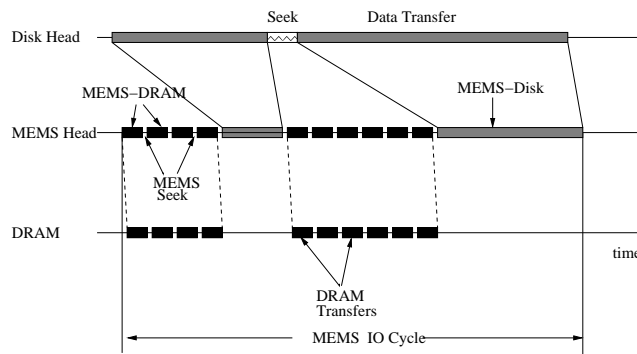

Figure 5.2: MEMS IO Scheduling.

78

Figure 5.2 describes the data transfer operations occurring during a single MEMS IO cycle. The X-axis is the time axis; the three horizontal lines depict the activity at the disk head, the MEMS tips, and the DRAM, respectively. In this example, the system services 10 streams ($N = 10$). The lightly shaded regions depict data-transfer from the disk drive into the MEMS device. The dark regions depict data-transfer between the MEMS device and the DRAM. The MEMS device performs N small IO transfers between MEMS and DRAM, and M large disk transfers in each MEMS IO cycle.

IO Scheduling: Multiple MEMS

According to certain predictions [71, 88], a single MEMS device might not be able to support twice the bandwidth of future disk drives. In such cases, a bank of k MEMS devices would provide a higher aggregate bandwidth. Using k MEMS devices for buffering disk IOs raises interesting questions. *How should stream data be split across these devices? What constitutes an IO cycle at the MEMS buffer? To what uses can we put any spare storage or bandwidth at the MEMS devices?* We answer each question in turn.

Placing stream data: Buffered data can be placed in one of two ways: stripe the buffered data for each stream across the MEMS bank or buffer each stream on a single MEMS device. Striping data for each stream across the k MEMS devices can be accomplished by splitting each disk IO into k parts and routing each part to a different MEMS device. The size of disk-side IOs performed on the MEMS device is reduced by a factor of k. Since a smaller average IO size decreases the MEMS device throughput, striping can be undesirable.

Instead of striping the data, the set of streams could be split across the MEMS bank. Each stream would thus be buffered on a single MEMS device. This would preserve the size of disk-side IO transfers. To achieve such a split, the disk IOs are routed to the MEMS devices in a round-robin fashion. Every

k^{th} disk IO is routed to the same MEMS device. Routing each IO to a single MEMS device improves the MEMS throughput and is thus preferable to striping each disk IO across the MEMS bank.
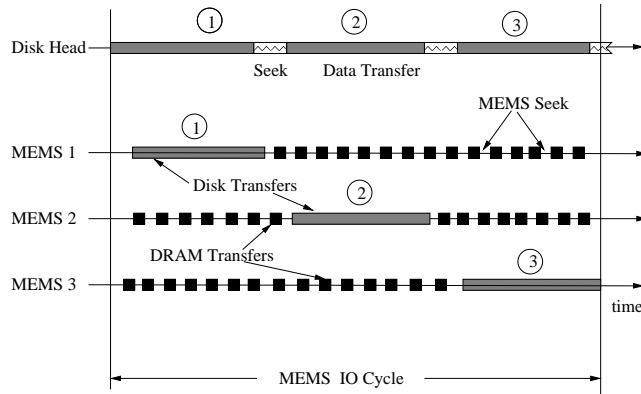
Figure 5.3: IO Scheduling for a MEMS bank.

IO Cycle for a k-device MEMS bank: Routing each disk IO to a single MEMS device splits the set of streams across the k MEMS devices in the bank. The notion of *IO cycle* for the MEMS bank can be defined in the same manner as that for a single device MEMS buffer. It is the time required to perform exactly one DRAM transfer for each of the N streams serviced by the system. For the sake of simplicity, let us assume that each MEMS device services $\frac{N}{k}$ streams. Figure 5.3 depicts the operations during an IO cycle at each MEMS device. The number of streams, N, is 45 and the number of MEMS devices in the bank is $k = 3$. For each disk IO, 15 DRAM transfers take place. The amount of data read from and written into the MEMS device is the same in the steady state.

Using the spare MEMS storage and bandwidth: Depending on the number and type of streams serviced and the capacity of the MEMS device bank, spare storage and/or bandwidth might be available at the MEMS device. If additional storage is available at the MEMS device, the operating system

could use it for other non-real-time data: as a persistent write buffer, as a cache for read data with temporal or spatial locality, or as a disk prefetch buffer. The MEMS storage can also be used to cache entire streams, as we shall explore next. Spare bandwidth, if available, can be used for non-real-time traffic.

5.1.2 MEMS Multimedia Cache

Multimedia content usually has a well-defined popularity distribution, and some content are accessed more frequently than others. Besides using MEMS storage as a buffer for streaming multimedia data from the disk drive, we can also use a MEMS storage device as a cache for popular multimedia content. Since the MEMS device offers low latency data access at throughput levels similar to those of the disk-drive, storing popular content on a MEMS cache reduces the buffering requirement for streaming data and hence DRAM cost. By using k MEMS devices, we can also use the aggregate bandwidth of the MEMS cache for improving the server throughput.

One significant difference between using a MEMS device as a cache and using it as a buffer is that with caching, the MEMS cache behaves primarily as a read-only device. The MEMS cache is updated only to account for changes in stream popularity. This can be accomplished off-line, during service down-time. To service streams from the cache, we use time-cycle-based IO scheduling. Again, there exist two distinct IO cycles, one for the streams serviced from the disk-drive and the other for those serviced from the MEMS cache. The performance of either device depends on the available DRAM buffering and the number of streams serviced from it.

When more than one MEMS device is used for caching, the performance of the MEMS cache depends on the data management policy used for the MEMS bank. With multiple devices, we must ensure that the load on the MEMS devices is balanced. In this regard, we can draw on research from data management

policies for disk arrays [11]. We now investigate two cache-management policies, representing two classes of load balancing strategies, which ensure total load-balance across the MEMS bank. These approaches make different trade-offs to optimize for a sub-set of system configurations. More sophisticated load-balancing strategies, including hybrid approaches of the above, have been proposed in literature [11, 66, 90, 91]. We investigate two simple, representative approaches as a first step.

Striped Cache-management

Using *striped cache-management*, each stream is bit- or byte-striped across all the k MEMS devices. There is no redundancy, and data for each stream is distributed in round-robin fashion across the MEMS devices. To perform an IO on the MEMS cache, all the devices access exactly the same relative data location, in a lock-step fashion. The load is thus perfectly balanced across the MEMS cache. The effective data transfer rate of the MEMS cache is k times that of a single device. The effective access latency of the MEMS cache is the same as that of a single device. If N_m streams are serviced from the MEMS cache, the total number of seek operations in an IO cycle is $k \cdot N_m$. Using striped cache-management, perfect load-balancing is achieved at the cost of reduced access parallelism of the devices.

Replicated Cache-management

Using *replicated cache-management*, the cached streams are replicated on the k MEMS devices. All the devices store exactly the same content. To perform an IO on the MEMS cache, any of the k devices can be accessed. If the number of streams serviced from the MEMS cache is N_m, then each MEMS device services exactly $\frac{N_m}{k}$ of the streams. Since each MEMS device stores all the cached streams, such a division is possible. The effective data transfer rate of the MEMS

cache is k times that of a single device. Operating the k devices independently improves parallelism of access. The total number of seek operations in an IO cycle is only N_m as opposed to $k \cdot N_m$ in striped cache-management. Using replicated cache-management, perfect load balancing is achieved at the cost of reduced cache size due to data redundancy. In Section 5.2.2, we further analyze the trade-offs involved in each policy.

5.2 Quantitative Analysis

In this section we present a quantitative model to analyze buffering requirements for systems supporting real-time streaming applications using MEMS devices. We discuss two system configurations:

- *MEMS buffer.* All data retrieved from the disk to DRAM are first retrieved into the MEMS buffer and then transferred to DRAM.

- *MEMS cache.* Selected data are cached on the MEMS device. Requested data can be serviced either from the disk-drive or the MEMS cache.

To evaluate the effectiveness of MEMS-based buffering and caching, we compare the system cost with and without MEMS storage. Let C_{dram} and C_{mems} denote the unit cost (\$/B) of DRAM and MEMS buffer, respectively. Furthermore, we use a per-device cost model for MEMS storage. The k MEMS devices cost $k \times C_{mems} \times Size_{mems}$ even if the system does not utilize all the available MEMS storage. The proofs for the results presented in this section can be found in [62].

5.2.1 MEMS Multimedia Buffer

Let $S_{disk-dram}$ denote the per-stream IO size from disk to DRAM, $S_{disk-mems}$ from disk to MEMS, and $S_{mems-dram}$ from MEMS to DRAM. Let k denote the

Parameter	Description
N	Number of continuous media streams
$\bar{B}$	Average bit-rate of the streams serviced $[B/s]$
k	Number of MEMS devices in system
R_{disk}	Data transfer rate from disk media $[B/s]$
R_{mems}	Data transfer rate from MEMS media $[B/s]$
$\bar{L}_{disk}$	Average latency for disk IO operations $[s]$
$\bar{L}_{mems}$	Average latency for MEMS IO operations $[s]$
C_{dram}	Unit DRAM cost $[\$/B]$
C_{mems}	Unit MEMS cost $[\$/B]$
$Size_{mems}$	MEMS capacity per device $[B]$
$Size_{disk}$	Disk capacity $[B]$
$S_{disk-dram}$	Average IO size from disk to DRAM $[B]$
$S_{disk-mems}$	Average IO size from disk to MEMS $[B]$
$S_{mems-dram}$	Average IO size from MEMS to DRAM $[B]$
T_{disk}	Disk IO cycle $[s]$
T_{mems}	MEMS IO cycle $[s]$

Table 5.1: Parameter definitions.

number of MEMS devices in the system. Let N denote the number of streams in the system. The buffer cost with and without the MEMS buffer is

$$COST_{without\_mems} = N \times C_{dram} \times S_{disk-dram} \qquad (5.1)$$

$$
\begin{aligned}
COST_{with\_mems} \;=\; & k \times C_{mems} \times Size_{mems} \;+\; \\
& N \times C_{dram} \times S_{mems-dram} \qquad (5.2)
\end{aligned}
$$

where $k \times Size_{mems} \geq N \times S_{disk-mems}$. Using MEMS devices in a streaming system is cost effective only if $COST_{with\_mems} < COST_{without\_mems}$.

In order to calculate the system cost, we first calculate IO sizes that guarantee the real-time streaming requirements. We next compute disk and MEMS IO sizes given the following four input parameters:

- N: The number of streams that the server supports.

- $\bar{B}$: The average bit-rate of the N streams.

- R_d: The data transfer rate of device d. R_d is substituted by R_{disk} (disk transfer rate) or R_{mems} (MEMS transfer rate) depending on where the IO takes place.

- $\bar{L}_d$: The average latency of device d in a time-cycle. $\bar{L}_d$ is substituted by $\bar{L}_{disk}$ (disk latency) or $\bar{L}_{mems}$ (MEMS latency) depending on where the IO takes place. $\bar{L}_d$ also depends on the scheduling policy employed to manage device d.

In computing IO size, we make two assumptions that are commonly used in modeling a media server. First, we use time-cycle-based IO scheduling (Section 5.1). Second, to simplify the analytical model, we assume all streams to be in constant bit-rate (CBR).[1] We summarize the parameters used in this paper in Table 5.1.

Theorem 1. For a system which streams directly from the disk to DRAM, the minimum size of per-stream DRAM buffer required to satisfy real-time requirements is

$$S_{disk-dram} \quad = \quad \frac{N \times \bar{L}_{disk} \times R_{disk} \times \bar{B}}{R_{disk} - N \times \bar{B}} \quad , \tag{5.3}$$
$$where \quad R_{disk} > N \times \bar{B}.$$

Corollary 1. To stream directly from the MEMS device to the DRAM, the minimum size of per-stream DRAM buffer required to satisfy real-time requirements is

$$S_{mems-dram} \quad = \quad \frac{N \times \bar{L}_{mems} \times R_{mems} \times \bar{B}}{R_{mems} - N \times \bar{B}} \quad , \tag{5.4}$$
$$where \quad R_{mems} > N \times \bar{B}.$$

Although Theorem 1 is well established [60], calculating IO sizes is more complex

---

[1]VBR can be modeled by CBR plus some memory cushion for handling bit-rate variability [48].

in a system that uses MEMS as an intermediate buffer between the disk and DRAM because we must consider the real-time requirements between the disk and MEMS as well as between the MEMS and DRAM.

Theorem 2. For a system which uses k MEMS devices as a disk buffer, the minimum size of per-stream DRAM buffer required to satisfy real-time requirements is

$$
\begin{aligned}
S_{mems-dram} &= \bar{B} \times \frac{C \times (1 + \frac{2k-2}{N}) \times T_{disk}}{T_{disk} - C}, \qquad (5.5) \\
where\ C &= \frac{N \times \bar{L}_{mems} \times R_{mems}}{k \times R_{mems} - 2 \times (N + k - 1) \times \bar{B}}.
\end{aligned}
$$

T_{disk} is the largest value such that the following three conditions (real-time requirement, storage requirement, and scheduling requirement) are satisfied:

$$
T_{disk} \geq \frac{N \times \bar{L}_{disk} \times R_{disk}}{R_{disk} - N \times \bar{B}} \qquad (5.6)
$$

$$
2 \times N \times T_{disk} \times \bar{B} \leq k \times Size_{mems} \qquad (5.7)
$$

$$
\frac{T_{mems}}{T_{disk}} = \frac{M}{N}, \quad M < N, \quad M \in Integer. \qquad (5.8)
$$

Corollary 2. When N and M are divisible by k (or are relatively large compared to k), k MEMS devices behave as a single MEMS device with both k times smaller average latency and k times larger throughput.

5.2.2 MEMS Multimedia Cache

Although streaming data do not have temporal locality, they are often characterized by a non-uniform popularity distribution. Caching popular content in MEMS storage can decrease the DRAM buffering requirement so that the system can support more streams. Let h denote the hit-rate for the MEMS cache. Given N streams to service, $n = N \times h$ of them are serviced from the MEMS cache, and $N \times (1 - h)$ streams are serviced from the disk. We can express the

cost of DRAM buffer and MEMS cache as

$$
\begin{aligned}
COST_{with\_mems-cache} = {} & k \times C_{mems} \times Size_{mems} \; + \\
& h \times N \times C_{dram} \times S_{mems-dram} \; + \\
& (1-h) \times N \times C_{dram} \times S_{disk-dram}. \quad (5.9)
\end{aligned}
$$

Using Theorem 1 we can calculate $S_{disk-dram}$ as

$$
S_{disk-dram} = \frac{(1-h) \times N \times \bar{L}_{disk} \times R_{disk} \times \bar{B}}{R_{disk} - (1-h) \times N \times \bar{B}} \;, \quad (5.10)
$$
$$
where \;\; R_{disk} > (1-h) \times N \times \bar{B}.
$$

In order to calculate h and $S_{mems-dram}$, let us assume that the popularity distribution of content is specified by $X : Y$, where $X\%$ of the streams are accessed $Y\%$ of the time. Let us assume that both popular ($X\%$) and non-popular streams ($100\% - X$) are accessed uniformly in their class. The capacity of the disk, $Size_{disk}$, is the total storage required for all the streams serviced by the system. Let the capacity of a single MEMS device be denoted by $Size_{mems}$. Let the percentage of movies cached be denoted by p. Then the cache hit ratio, h, can be expressed as

$$
h = \begin{cases}
\frac{p}{X} \times \frac{Y}{100\%} & \text{if } X \geq p, \\
\frac{Y}{100\%} + \frac{p-X}{100\%-X} \times \frac{100\%-Y}{100\%} & \text{otherwise.}
\end{cases} \quad (5.11)
$$

If the MEMS cache contains a single MEMS device, then p and $S_{mems-dram}$ (using Equation 5.4) are

$$
\begin{aligned}
p & = \frac{SIZE_{mems}}{SIZE_{disk}} \; ; \\
S_{mems-dram} & = \frac{n \times \bar{L}_{mems} \times R_{mems} \times \bar{B}}{R_{mems} - n \times \bar{B}}
\end{aligned}
$$

However, if the cache consists of more than one MEMS device, both p and $S_{mems-dram}$ depend on the cache management policy used for accessing the MEMS cache. We explore the two policies, namely *striped* and *replicated*, introduced in Section 5.1.2.

Striped Cache Management

Theorem 3. For a server that employs the striped cache-management policy across an array of k MEMS devices for serving n streams, the minimum size of per-stream DRAM buffer required to satisfy real-time streaming requirements is

$$S_{mems-dram} = \frac{n \times \bar{L}_{mems} \times (k \times R_{mems}) \times \bar{B}}{(k \times R_{mems}) - n \times \bar{B}}, \tag{5.12}$$
$$where \ \ k \times R_{mems} > n \times \bar{B}.$$

Corollary 3. A striped cache, consisting of k MEMS devices, behaves as a single MEMS cache with k times larger throughput and unchanged access latency.

Replicated Cache Management

Theorem 4. For a server that employs the replicated cache-management policy across an array of k MEMS devices for serving n streams, the minimum size of per-stream DRAM buffer required to satisfy real-time streaming requirements is

$$S_{mems-dram} = \frac{(n + k - 1)\frac{\bar{L}_{mems}}{k}(k \cdot R_{mems}) \times \bar{B}}{(k \cdot R_{mems}) - (n + k - 1) \times \bar{B}}$$
$$where \ k \cdot R_{mems} > (n + k - 1) \times \bar{B}. \tag{5.13}$$

Corollary 4. When N is divisible by k (or is relatively large compared to k), k replicated cache behaves as a single MEMS device with k times larger throughput as well as k times smaller average latency.

5.3 MEMS Evaluation

This section presents an experimental evaluation of a streaming multimedia system equipped with MEMS storage. We compare its performance to that of

a system without MEMS storage. We evaluate separately the performance of the MEMS device when it is used to buffer streaming data stored on the disk drive, and when it is used to cache popular streams. The evaluation presented in this section is based on the analytical model presented in Section 5.2.

Parameter	FutureDisk	G3 MEMS	DRAM
RPM	20,000	–	–
Max. bandwidth [MB/s]	300	320	10,000
Average seek [ms]	2.8	–	–
Full stroke seek [ms]	7.0	0.45	–
X settle time [ms]	–	0.14	–
Capacity per device [GB]	1,000	10	5
Cost/GB [\$]	0.2	1	20
Cost/device [\$]	100–300	10	50–200

Table 5.2: Performance characteristics of storage devices in the year 2007.
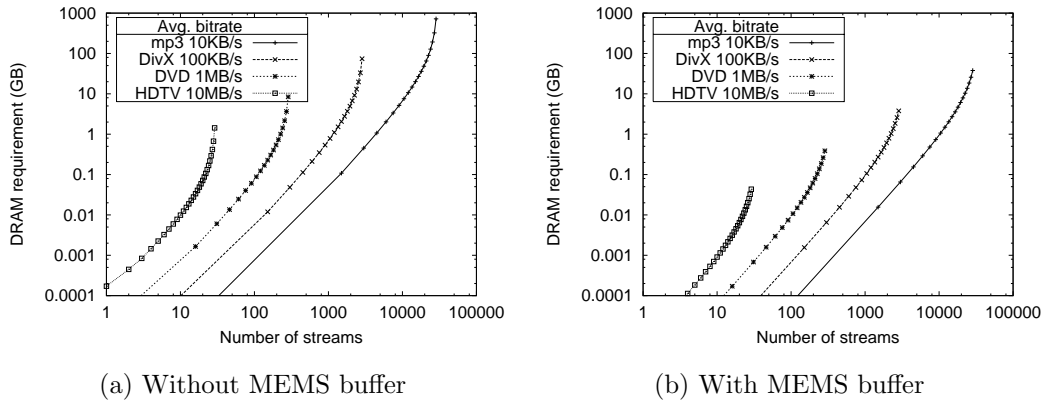

(a) Without MEMS buffer   (b) With MEMS buffer

Figure 5.4: DRAM requirement for various media types.

To model the performance of the MEMS device, we closely followed one such model proposed by researchers at Carnegie Mellon University. The work in [71] describes their model comprehensively. For our experiments, we use the "3rd

Generation" (G3) MEMS device model proposed in [71]. We obtained predictions for disk-drive and DRAM performance by using projections on current-day devices produced by Maxtor [50] and Rambus [59], respectively. These are summarized in Table 6.1.

In our experiments the average bit-rate of streams, $\bar{B}$, was varied within the range of 10KB/s to 10MB/s. Since the maximum bandwidth of the FutureDisk, R_{disk}, is 300MB/s, it can support tens of high-definition streams at a few megabytes per second each, more than a hundred compressed MPEG2 (DVD quality) streams at 1MB/s, or a thousand DivX (MPEG4) streams at 100KB/s, or even tens of thousands of MP3 audio at a bit-rate of 10KB/s. To minimize the mis-prediction of seek-access characteristics for the MEMS device, we assume that MEMS accesses, $\bar{L}_{mems}$, always experience the maximum device latency. We use scheduler-determined latency values, $\bar{L}_{disk}$, for disk accesses. The disk IO scheduler uses elevator scheduling to optimize for disk utilization.

5.3.1 MEMS Multimedia Buffer

As mentioned in Section 5.1, using MEMS storage to buffer streaming data requires that it supports twice the streaming bandwidth of the disk-drive. In our experiments, we used at least two G3 MEMS devices for buffering the streaming data, which provided a maximum aggregate MEMS throughput of 640 MB/s. To evaluate the performance of the MEMS multimedia buffer, our experiments aimed to determine the reduction in DRAM requirement as well as overall system cost due to the addition of MEMS storage. In addition, we also determined the sensitivity of the above metrics to variations in MEMS device characteristics.

Theorems 1 and 2 presented in Section 5.2 define the relationship between the system parameters. To study the sensitivity of our evaluation to MEMS device characteristics, we introduce the *latency ratio*, $\frac{\bar{L}_{disk}}{\bar{L}_{mems}}$, as a tunable pa-

90

rameter.

Latency ratio: We define the *latency ratio* as the ratio of the average disk access latency to the maximum MEMS access latency. We varied the latency ratio within the range 1 to 10. The value for this parameter is around 5 for the FutureDisk and the G3 MEMS device listed in Table 6.1.

For performance evaluation, we conducted three experiments. In the first two experiments, we assumed that the maximum amount of DRAM and MEMS storage was unlimited. We also used a cost-per-byte price model for MEMS storage. These relaxations allowed us to observe the relationship between the system parameters. In the third experiment, we performed a case-study using an "off-the-shelf" system which could be developed by the year 2007. The available buffering on this system is limited, and its size is based on current trends in server system configurations.

Reduction in DRAM requirement

In Figure 5.4, we vary the number of streams, N, and the average stream bit-rate, $\bar{B}$. We plot the DRAM requirement on the Y-axis. The X and Y axes are drawn to logarithmic scale. The total buffering requirement increases rapidly with the number of streams (according to Equations 5.1 and 5.4). For a fixed system throughput, the buffering requirement is thus much larger for smaller bit-rates than for larger ones. In the absence of a MEMS buffer, the DRAM requirement for a fully utilized disk ranges from 1GB for 10MB/s streams to 1TB for 10KB/s streams. With a MEMS buffer, the DRAM requirement is reduced by an order of magnitude to support a given system throughput.

Reduction in Cost

Addition of a MEMS buffer reduces DRAM requirement. However, we must take the total system cost into account before reaching a conclusion about the

(a) Cost-reduction curves   (b) Cost-reduction regions
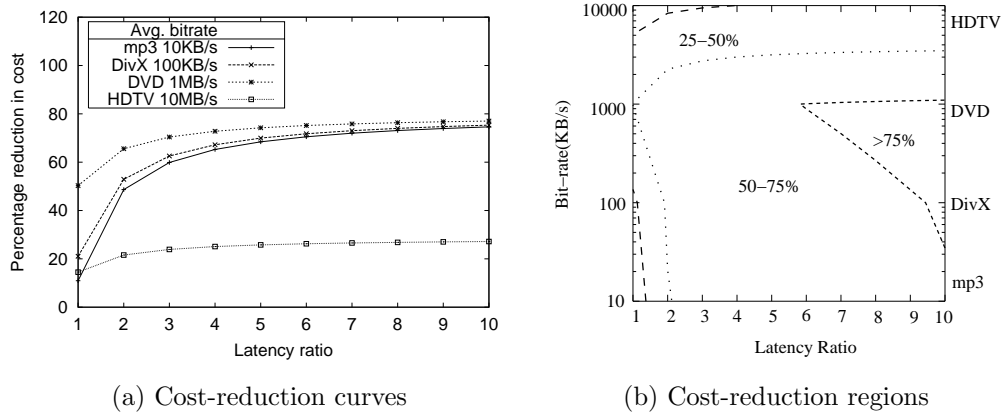
Figure 5.5: Percentage cost reduction.

benefits. To calculate the cost of buffering, we use cost predictions as presented in Table 6.1. According to the predictions, MEMS buffering is 20 times cheaper than DRAM buffering per-byte.
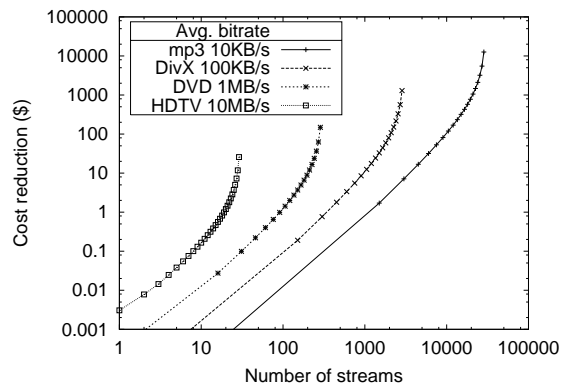
Figure 5.6: Reduction in the total buffering cost.

Figure 5.6 shows the reduction in total buffering cost, including the additional cost of the MEMS buffer. In spite of the addition of the MEMS buffer, the total cost of buffering is reduced for all media types. Cost savings range from tens of dollars for high bit-rate streams to tens of thousands of dollars for lower bit-rates. These cost savings are almost directly proportional to the DRAM reductions presented in Figure 5.4, since the fractional cost-addition

due to the MEMS storage is negligible. This curve also shows that using a MEMS buffer for streaming large bit-rates does not offer a significant reduction in the buffering cost. Indeed, for large bit-rates, even DRAM buffering is sufficient to achieve high disk utilization. Upon calculation, we found that the cost-reduction ranges between 80% and 90% depending on the number of streams, N, and their average bit-rates, $\bar{B}$.

A Parameter Sensitivity Study

We now present a hypothetical off-the-shelf system which could be developed in the future. We determine the sensitivity of cost-reductions to unpredictable trend changes in device characteristics by varying the *latency-ratio*.[2]

In our earlier experiments, we assumed that the system could use an unlimited amount of DRAM and/or MEMS storage. However, a system with terabytes of DRAM and/or buffering would be prohibitively expensive. In the following experiment, we restricted the system configuration to an off-the-shelf system projected for the year 2007 (Table 6.1). The maximum DRAM size is restricted to $5GB$ and $20GB$ of MEMS buffering is used by adding two MEMS devices. The cost of the MEMS buffer is $20, thereby restricting the cost-savings to a maximum of 80%.

Figure 5.5(a) depicts the percentage reduction in the buffering cost for different average stream bit-rates when the latency ratio is varied. As the latency-ratio is increased, the MEMS device can deliver higher throughputs with less buffering. As a result, the buffering cost is reduced. When the average bit-rate is increased, the performance of both the disk drive and the MEMS device improves. The differential improvement in throughput for the MEMS device is

---

[2]We also investigated the sensitivity of the MEMS buffer performance to the cost and bandwidth values. Our conclusion (that MEMS buffering is effective for low and medium bit-rate traffic) holds true as long as the MEMS device is an order of magnitude cheaper than DRAM and provides streaming bandwidths comparable to or greater than those of disk-drives.

greater. However, beyond a certain average bit-rate (1MB/s in this case), the DRAM requirement in the absence of the MEMS buffer is small, and the available 5GB DRAM buffer is under-utilized even without the MEMS buffer. In the absence of a MEMS buffer, the DRAM requirement for the 10MB/s bit-rate range is approximately 1.5GB, thereby limiting the cost-reduction to only 30%.

In Figure 5.5(b), contour lines on the XY-plane depict the regions in which the percentages of reduction in total buffering costs are 25%, 50%, and 75%. The MEMS buffer is cost-effective almost over the entire parameter space, with at least a 50% to 75% reduction in the total buffering cost.

5.3.2 MEMS Multimedia Cache

When streaming content has a non-uniform access probability, then a MEMS cache for popular streams can improve the performance of a multimedia server as analyzed in Section 5.2. To measure the performance of the MEMS cache, we calculated the improvement in the server throughput (number of additional streams serviced) using a MEMS cache by fixing the total cost of the system. The additional cost of using a MEMS cache is reflected by a reduced amount of available DRAM buffering. We evaluated the performance of the MEMS cache using the two cache-management policies described in Section 5.1: *striped* and *replicated*.

The number of streams serviced from the MEMS device depends on the number of cache hits (it follows Equation 5.9). In our experiments, we assumed that if a stream was found in the cache, it would be serviced only from the cache[3].

The performance of the MEMS cache depends on four parameters: the popularity distribution, the total system cost, the average stream bit-rate, and the

---

[3]It can be shown that the performance of the cache can be further enhanced by off-loading the servicing of some popular streams to the disk drive if the disk is under-utilized. We leave this study for future work.
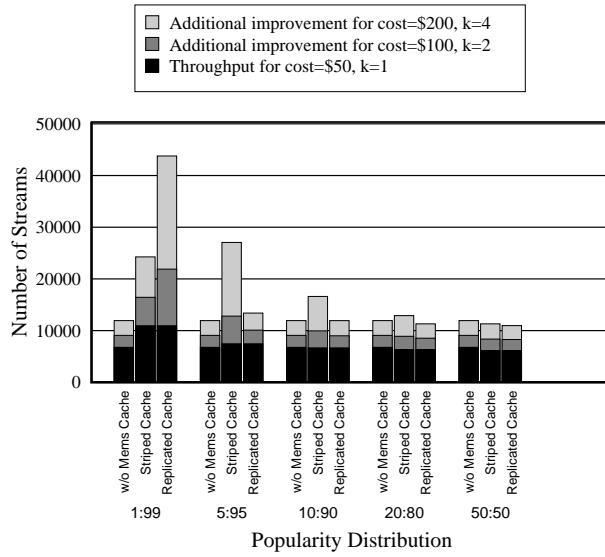
(a) Average bit-rate 10KB/s
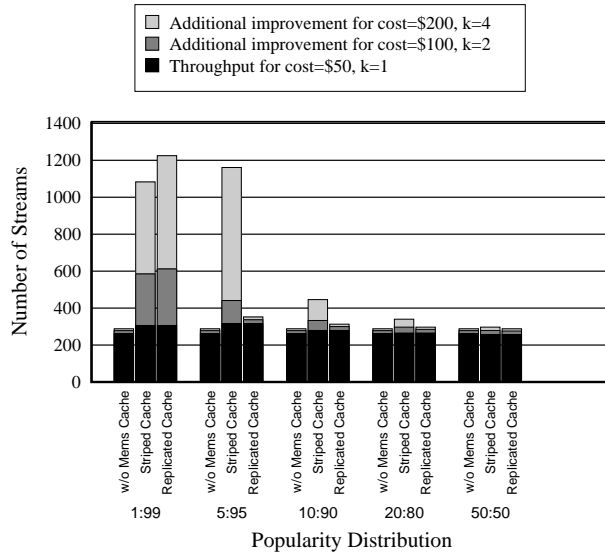
(b) Average bit-rate 1MB/s

Figure 5.7: MEMS cache performance.

size of the MEMS bank. To evaluate the impact of these parameters, we conducted the following experiment. We fixed the cost of the system and varied the popularity distribution to determine the server throughput with and without a MEMS cache. Each MEMS caching device that we added reduced the amount of DRAM buffering available by 500MB in order to keep the cost invariant. The popularity distribution follows X:Y, where $X\%$ of the streams are accessed $Y\%$ of the time. We also assumed that all $X\%$ of the streams were equally popular. Similarly, we assumed that the remaining $(100\text{-}X)\%$ of the streams were equally unpopular. We conducted the same experiment for three different values of the total buffering cost: \$50, \$100, and \$200. At these costs, the numbers of MEMS devices, k, used for caching are chosen as 1, 2, and 4 respectively. The entire experiment was performed for two different average stream bit-rates: 10KB/s, and 1MB/s.

Effect of popularity distribution

Figure 5.7(a) compares the performance of the two cache-management policies: *striped* and *replicated*, against a system without a MEMS cache. The X-axis represents the popularity distribution and the Y-axis represents the server throughput in terms of the number of streams serviced. The average stream bit-rate is fixed at 10KB/s. A uniform popularity distribution is denoted by the point 50:50 along the X-axis. When $k = 1$, the replicated and striped caching is equivalent. For skewed popularity distributions of 1:99, 5:95, and 10:90, both cache management policies outperform the system without the MEMS cache. However, when the popularity distribution leans toward the uniform side, the MEMS caching is not cost-effective. The MEMS cache is able to store only a small fraction of the popular content, thereby failing to offset its cost.

We can also compare the relative performance of the two cache-management policies. When the popularity distribution is greatly skewed (1:99), the repli-

cated cache-management policy supports the maximum number of streams. Since all the popular streams are available on the cache regardless of which policy is used, replication outperforms striping by offering much lower average access latencies to the MEMS cache. This is not the case for more uniform popularity distributions, wherein striping can cache more popular content than replication can.

Effect of varying the total cost

If the cost of the system is flexible, a system designer must know the size of the MEMS cache and DRAM buffer which provides the best server performance. Caching improves system performance when the number of MEMS devices is large enough to cache a majority of the popular movies. More MEMS devices can be used for caching when the available budget for buffering and caching is sufficient. The greater the available budget, the bigger the range of popularity distributions wherein the MEMS cache is cost-effective. For instance, in Figure 5.7(a), with a $50 budget, the MEMS cache (using one MEMS device) is cost-effective only within the popularity range of 1:99 to 5:95. With a higher budget ($200), the MEMS cache with four devices is cost-effective over a greater popularity range, 1:99 to 20:80.

Effect of varying the average stream bit-rate

Unlike the MEMS buffer, the performance improvement due to the MEMS cache is almost independent of the bit-rate of the streams serviced. Figures 5.7(a) and (b) show this behavior. Regardless of the bit-rate, addition of one or more MEMS devices increases the total streaming capacity of the server. An interesting behavior that depends on the bit-rate occurs in the case without the MEMS cache. In Figure 5.7(b), the additional improvement for buffering costs of $100 and $200 are negligible. For large bit-rates, even a small amount of buffering is

sufficient for utilizing the disk throughput.

Effect of varying the number of MEMS devices
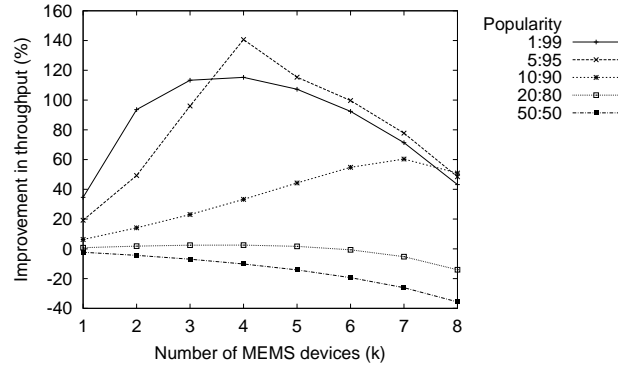
Figure 5.8: Varying the size of the MEMS cache.

Figure 5.8 shows the effect of varying the size of the MEMS cache, k, on the improvement in the server throughput. Striped cache-management policy is used for this experiment. The total cost of buffering and caching with and without the MEMS cache is fixed at \$100 and the system serves an average bit-rate of 100KB/s. Each MEMS device can cache 1% of the content. As k is increased, the size and throughput of the MEMS cache also increase. However, to keep the total cost invariant, the available DRAM buffer size decreases. Hence, for each popularity distribution, there exists a unique optimal size for the MEMS bank. When the popularity distribution is uniform, represented by 50:50, the MEMS cache always degrades performance. For skewed popularity distributions (1:99, 5:95, and 10:90), the MEMS cache improves the server throughput by as much as $2.4X$.

5.4　Summary

In this chapter, we investigated the possibility of using MEMS storage for buffering and caching streaming multimedia content. 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. We proposed using MEMS storage in two possible configurations: *MEMS as a multimedia buffer* and *MEMS as a multimedia cache*. We developed an analytical framework for evaluating each of these configurations while using either a single MEMS device or a k-device MEMS bank. Based on our analytical study we have provided the following design guidelines: (i) for low and medium bit-rate streams, using MEMS storage as buffer reduces the cost of designing server systems by as much as an order of magnitude, (ii) when the popularity distribution is non-uniform, the server throughput can be improved by caching data in the MEMS device. The MEMS device improves access to popular content and also provides additional streaming bandwidth, regardless of the stream bit-rates.

Although this is a somewhat speculative study, several research and industry efforts [5, 40, 54, 9, 88] suggest that commercially mass-produced MEMS storage devices are only a few years away. Thus, it is necessary to start thinking now about the architectural impact of these devices on current applications. We hope that this study can provide guidelines for designing next-generation media servers.

Chapter 6

Heterogeneous MEMS-disk Storage Systems

In this chapter, we extend our single-disk work in Chapter 5 to a RAID system. RAID systems are typically used in todays high data-volume servers. The introduction of MEMS storage into a RAID-based storage architecture is more complex than a single-disk system since it raises issues of IO bus contention and sharing of the MEMS buffer/cache space between the disk drives.

In this chapter, we present a new class of single-disk and RAID-based storage systems that also integrate the faster, albeit more expensive (in terms of cost-per-byte), MEMS-based storage devices. MEMS-based storage devices offer a unique cost-performance trade-off between those of DRAM and disk drives and promise to bridge the increasing performance gap in the storage hierarchy.

We introduce the problem of building a heterogeneous MEMS-disk storage system for real-time streaming data. We propose various ways of integrating MEMS-based storage into existing storage architectures. In particular, we make the following contributions:

1. Mapping between data types and storage device types.

2. Hardware storage architecture and logical hardware abstractions.

3. Data placement and IO scheduling for MEMS-disk streaming storage.

The rest of this chapter is organized as follows: In Section 6.1, we introduce key storage technologies and propose an architecture for a heterogeneous MEMS-disk storage system. In Section 6.2, we address the problem of managing heterogeneous storage and data, with a focus on time-sensitive real-time data. In Section 6.3, we conduct a performance evaluation of the MEMS-disk streaming storage system based on our analytical model. In Section 6.4, we summarize the chapter contributions.

6.1 Heterogeneous Streaming Storage Architecture

Storage systems are becoming increasingly complex. Today, we see storage farms consisting of several RAID modules with redundancy and fail-over mechanisms built in. Taking this paradigm into the future and incorporating new storage technologies, we envision inherently heterogeneous storage systems built using multiple storage technologies.

6.1.1 Overview of Storage Devices

To give order to the complex nature of storage systems, they are organized in a hierarchy ranging from the on-board L1 and L2 caches, the main-memory, the RAID-controller cache, the disk cache, and the permanent disk storage. To the existing mix, MEMS-based storage adds a new level, and based on its cost-performance characteristics, naturally places itself between the storage system cache and the disk store. To simplify the discussion, we focus on key storage levels in the hierarchy (in bold font below) that gain significance when dealing with high-volume data.

- L1/L2 cache: Caching code and data.

- **DRAM**: Buffering, Caching.

- RAID-controller cache and disk cache: Caching IO data.

- **MEMS-based storage**: Intermediate buffering, Caching.

- **Disk drives**: Permanent store for all data.

In understanding the role of these devices in a heterogeneous storage system, we must first identify their performance characteristics. We now look at each device in turn to decipher their role in a modern storage system.

Disk Drives

Typically, most storage media are optimized for sequential access. However, since magnetic disks have much longer access times than MEMS-based storage devices and DRAM, the effective throughput is governed by the amortized average of the disk access overhead. These devices have to be accessed in much larger chunks to mask the access overheads. Due to its large access latency and low cost-per-byte of storage, the disk drive is resource constrained primarily in bandwidth and not space. System architects therefore cope with this problem by issuing large IO requests (to amortize the access overhead) and by employing intelligent disk scheduling algorithms. To deal with real-time streaming data, scheduling gains additional importance to ensure timeliness of data delivery.

MEMS-based Storage

Researchers at the Carnegie Mellon University have proposed one possible architecture for a MEMS-based storage device [31, 71] which was introduced in Chapter 5 and depicted in Figure 3.1.

Figure 3.2 shows the effective throughput of the MEMS device depending on the average IO sizes when compared to disk drives. Due to its moderate access latency and moderate cost-per-byte, MEMS-based storage devices are both bandwidth and space constrained. Data allocation and placement as well as IO scheduling are equally important in managing these devices. Since the amount of buffer or cache space in the MEMS devices severely impacts the performance of the disk drives, it must be managed carefully. While dealing with real-time streaming data, the buffering bandwidth requirement is twice the actual bit-rate of each stream (as we shall see in Section 6.2). This additional bandwidth requirement as well as the real-time delivery constraints requires more complex IO scheduling algorithms at the MEMS devices.

DRAM

The achievable throughput performance from a DRAM device is about two orders of magnitude greater than both disk drives and MEMS-based storage. The maximum DRAM throughput is achieved when data is accessed in sequential chunks, about the size of the largest cache block. These are typically tens to hundreds of bytes. In such a scenario, it is very unlikely that the DRAM bandwidth would become a bottleneck. Rather than its bandwidth, the DRAM is resource-constrained in space. The available DRAM buffer and cache space impacts the performance of the devices which are placed below in the memory hierarchy, namely the MEMS-based storage and disk drives.

6.1.2   Heterogeneous Storage Hardware Architecture

Commodity RAID systems today consist of several disks connected to a RAID controller over an IO bus. The RAID controller issues both operating system generated IOs as well as control IOs required for RAID reconfiguration and management. Typically, the IO bus can support several disk-drives without

becoming a bottleneck itself. Data read or written on the storage system is transferred over the PCI bus to the PCI interface of the DRAM and finally to the DRAM itself. Designing new storage architectures requires that none of the data-path components, including the data buses, become a performance bottleneck.

Low-level Architecture

Due to their performance characteristics, it is a widely held belief that MEMS-based storage devices will be placed between the disk and DRAM components of the storage hierarchy. Figure 6.1 presents one possible hardware storage architecture which integrates MEMS devices. The RAID controller also has an on-board MEMS bank which it can use to temporarily buffer or cache IO data. This architecture does not increase IO traffic on any of the buses on the data-path, but may require additional on-board bus bandwidth on the controller to transfer data to/from the MEMS bank.
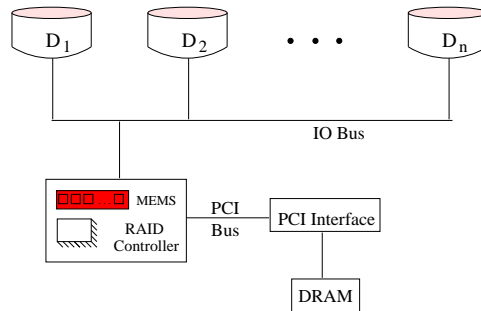

Figure 6.1: Integrated Storage Hardware Architecture.

Depending on how MEMS-based storage technology is packaged in the future and the cost-cutting nature of storage industry, it may or may not be feasible to integrate them into the RAID controller. If MEMS-based storage devices are packaged with a SCSI or IDE -like interface, they may have to be

104

independently connected to and accessed on the IO bus. Future RAID controllers may be equipped to manage this additional device, or an independent MEMS controller may be required. Although such an architecture is possible (even possibly unavoidable), it generates additional traffic on the IO bus and hence is an undesirable configuration.

Another future scenario arises if disk manufacturers immediately embrace MEMS storage devices, considering it an assisting technology rather than a competing one. In such a situation, they could integrate them into disk drive units to be used as a large disk cache. Such a cache can improve disk performance tremendously by allowing large amounts of data to be prefetched into the cache. In addition, if an interface is provided to the operating system or the RAID controller to independently access these devices, they could also be used to selectively buffer or cache file data.

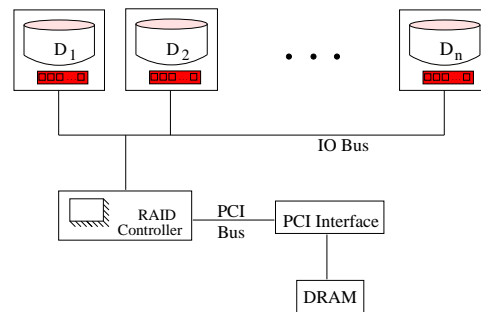Figure 6.2: Partitioned Storage Hardware Architecture.

Logical Hardware Abstractions

Based on the hardware architecture scenarios outlined above, two logical abstractions of the hardware architecture aid us in designing higher level data placement and IO scheduling schemes. These are the the *integrated* and *partitioned* MEMS buffer-cache abstractions. Figure 6.3 present these abstractions.
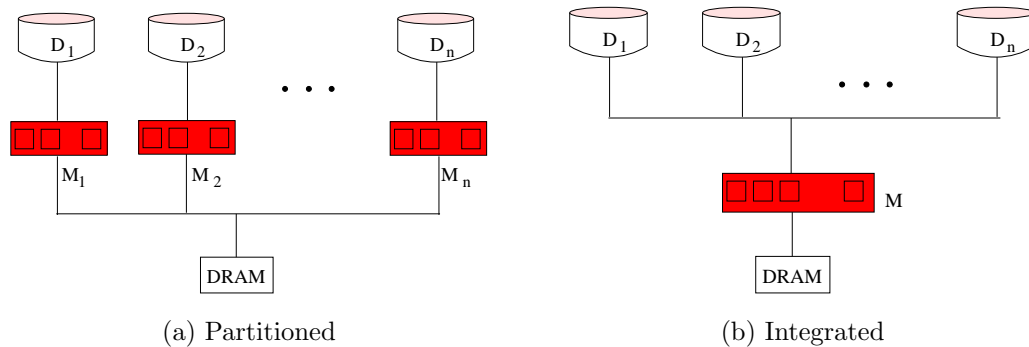
Figure 6.3: Logical Hardware Abstractions.

Although the partitioned hardware architecture can only allow the partitioned logical abstraction, the integrated hardware architecture allows for both logical abstractions and can be dynamically configured and tailored to the workload.

6.2 Managing Heterogeneous Streaming Storage

We now address the question of managing heterogeneous MEMS-disk systems that store a heterogeneous mix of data types. We start out by introducing the problem of managing heterogeneous storage systems and data, and propose a direction. We then look at the more specific question of constructing a heterogeneous storage architecture for a system that integrates real-time and non-real-time data.

6.2.1 Managing heterogeneous storage and data

Data present itself in numerous forms and they can be organized into a few categories. These data types include, but are not limited to,

106

- **Urgent:** Most IOs issued by the operating system are for urgent data, which must be serviced as early as possible. Examples of such data include file-system meta-data and virtual memory.

- **Real-time:** Streaming reads and writes are examples of real-time data. These data usually have guaranteed-rate requirements for IO requiring IO bandwidth reservation.

- **Interactive real-time:** Multimedia streams are expected to be inter-active. Interactive operations may include REW, FWD, PAUSE, ZOOM, etc. with the stream. These operations require quick response from the system and data must be retrieved with low access latency.

- **Other:** Logging, browsing, editing, etc. are examples of other types of data that must be supported by the system. These are typically non-real-time, best-effort data. The requirement for such data is that IOs must not experience starvation.

The heterogeneous nature of data is matched by the heterogeneous nature of today's storage systems. From the discussion in Section 6.1, we narrow down our discussion on storage systems and focus on the following three devices, each of which plays a role in the storage system hierarchy:

- **DRAM**: Buffering, Caching.

- **MEMS-based storage**: Intermediate buffering, Caching.

- **Disk drives**: Permanent store for all data.

Based on their characteristics, certain data types are naturally suited to specific storage types. The characteristics of data include size, type, distribution, bit-rate, popularity, etc. Figure 6.4 presents one possible distribution of data to different storage types based on a few characteristics, namely, file-size, bit-rate,

and popularity. The shaded region depicts a permissible placement of the files.
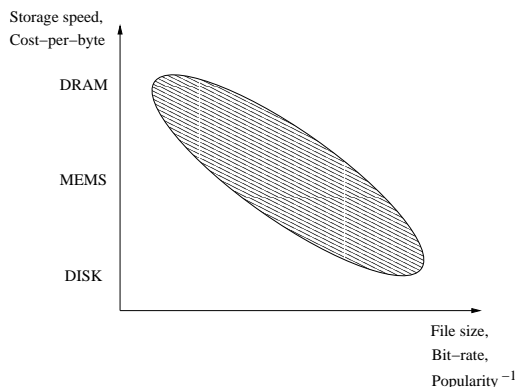
Figure 6.4: Data type v/s Storage type.

For cost-efficiency, large files must be cached on disk drives. However, smaller files may be cached even entirely in DRAM. Typically, continuous streams with high bit-rate are accessed sequentially in large chunks. They are typically stored on disk drives not just because they tend to be large, but more because the disk drives can efficiently service large IOs. Medium bit-rate streams may be partially buffered or cached entirely on MEMS-based storage to improve access to them. (We address the buffering and caching issues in detail later in this section.) Finally, since the cost-per-byte of DRAM and MEMS-based storage are significantly greater than those of disk drives, only popular streams must be cached on them. These simple guidelines assist in designing cost-effective and efficient heterogeneous storage systems.

6.2.2 Streaming Data

We now look at the more specific problem of managing streaming data. Streaming data is different from other data in two main respects. First, they are read or written on the storage device at a certain rate. This brings up the

problem of real-time IO scheduling in the storage system. Second, segments of a streaming data file must be temporarily buffered before they are consumed to obtain acceptable performance from the storage device. This brings up the problem of determining which storage component must be chosen to temporarily buffer or cache streaming data.

Based on the nature of streaming data, we can build rules of thumb for placing them on storage devices. Streaming data files are usually large and hence must be placed on disk drives for cost-effective storage. They may be temporarily cached on a MEMS device if they are popular and accessed frequently. Caching and buffer streaming files on a MEMS device can significantly improve performance and lower cost. Accessing them as an intermediate layer (rather than disk drives directly) greatly reduces the DRAM requirement for buffering a large number of streams simultaneous serviced from the storage system. Since DRAM is the most expensive device in the storage hierarchy, minimizing the DRAM requirement is critical for designing cost-effective streaming storage systems.

The problem of IO scheduling goes hand-in-hand with data placement and must be addressed together for a complete solution to managing streaming data. One of the principal requirements in streaming applications is to maintain high data throughput to be able to service as many simultaneous streams as possible. Achieving high disk throughput in streaming applications necessitates accessing the disk drive in larger chunks. This translates to a rapidly-increasing DRAM buffering cost. A large DRAM buffer is especially necessary for servers which stream to a large number of clients. Filling this buffering requirement is not economically feasible using DRAM. Using a MEMS-bank as an intermediate buffering device may alleviate most of the buffering requirement. Segments of streams served from the disk drives are temporarily buffered in the MEMS buffer at a much lower buffering cost compared to DRAM.

To service disk IO for streaming data, we use the notion of *time-cycles* [60].

In each time-cycle, given the consumption rates of the individual streams, sufficient data is retrieved for each stream so that the data buffers do not underflow and introduce hiccups in stream playback. There exists a trade-off between the amount of data buffered and the achieved disk utilization for servicing multimedia streams. Operating the disk drive at a higher utilization requires accessing the disk using larger I/Os and consequently requiring more data buffering for the individual streams. The disk scheduling algorithm employed decides the trade-off point in this design space.

Using MEMS storage as an intermediate buffer between the disk and DRAM enables the disk drive to be better utilized. At a fraction of the cost of DRAM, MEMS storage can provide a large amount of buffering required for achieving high disk utilization (see Figure 3.2). Although DRAM buffering cannot be completely eliminated, the low access latency of MEMS storage provides high throughput with significantly lesser DRAM buffering requirement. The MEMS device can thus act as a speed-matching buffer between the disk drive and the system memory, in effect addressing the disk utilization and data buffering trade-off.

Using MEMS storage as an intermediate buffer implies that the MEMS-based device must handle both disk and DRAM data traffic simultaneously. To understand the service model, let us assume that the multimedia streams being serviced are all read streams, so that stream data read from the disk drive is buffered temporarily in the MEMS device before it is read into the DRAM. This model can be easily extended to address write streams.

To service buffered data from the MEMS device, we use the time-cycle-based service model previously proposed for disk drives. Data is retrieved in cycles into the DRAM such that no individual stream experiences data underflow at the DRAM. At the same time, the data read from the disk drive must be written to the MEMS device. The disk IO scheduler controls the read operations at the disk drive. The MEMS IO scheduler controls the write operations for data read

from the disk as well as read operations into the DRAM. In the steady state, the amount of data being written to the MEMS device is equal to the amount read from it. Thus, although the MEMS device can help improve disk utilization, we must realize that to do so, it must operate at twice the throughput of the disk drive.

We analyze how the two logical hardware abstractions (*partitioned* and *integrated*) can be managed to cache and buffer streaming data. Each of these abstractions has its advantages and disadvantages when used to store streaming data. We propose two MEMS buffer/cache management schemes that work with these abstractions.

Partitioned MEMS buffer

The *partitioned* MEMS buffer scheme can work with both logical hardware abstractions. It assumes that it is possible to allocate a fixed and exclusive MEMS buffer/cache to each disk. Each disk only has access to its exclusive buffer/cache and cannot access those of the other disks in the RAID. Such an exclusion, naturally available in the partitioned abstraction, can also be enforced for an integrated hardware abstraction.

In the partitioned configuration, streams stored on the disk-drive are cached and buffered on its exclusive MEMS buffer/cache. Figure 6.5(a) depicts the stream flows in this configuration for the integrated hardware architecture. Strict partitioning of the space is enforced within the MEMS buffer/cache and there is no contention between the disks for buffer or cache space. In this configuration, the solutions presented earlier for the single-disk case are directly applicable. The exclusive nature of the buffer cache makes the data placement and IO scheduling solutions for the partitioned configuration relatively easier to design and implement. The only difference from the single-disk case is that the amount of DRAM required is the sum of those required for the individual

111

disks in the array.

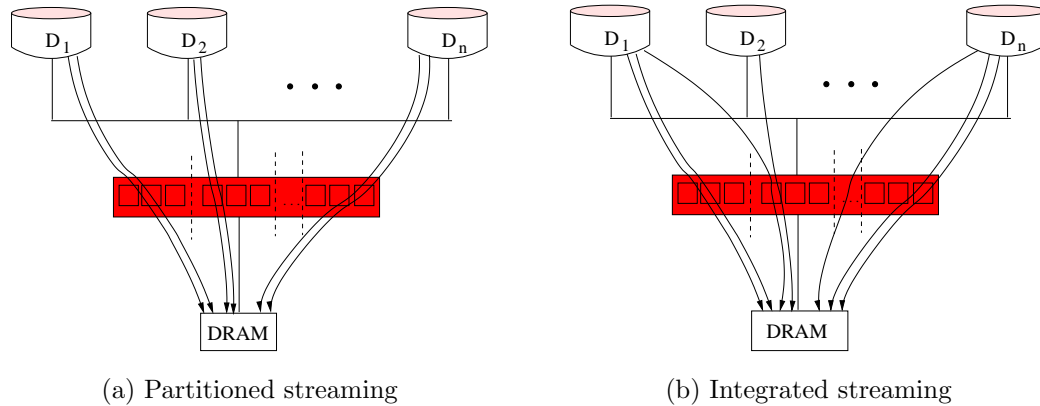(a) Partitioned streaming   (b) Integrated streaming

Figure 6.5: The partitioned and integrated MEMS buffer/cache schemes.

Although the partitioned solution is easy to implement, it may not be the optimal under certain conditions:

Load imbalance: *Load-imbalance* in a RAID system is defined as the ratio of the number of streams serviced by the most-loaded disk to the number of streams served by the least-loaded disk, assuming that all streams share a common bit-rate. In the case of load-imbalance, some disk drives may be serving more data streams than others. In such a case, exclusive access to buffer or cache space may lead to under-utilization and overload. Disk drives that serve more streams may under-perform because the MEMS cache or buffer space is insufficient. At the same time, MEMS buffers for other disks may be under-utilized.

Bit-rate heterogeneity: *Bit-rate heterogeneity* in a RAID system is defined as the ratio of the maximum average-bit-rate to the minimum average-bit-rate of the streams served by any single disk in the RAID, assuming that the all the disks stream the same total bandwidth. Lower bit-rate streams typically require more buffer space to sustain higher throughput levels. If a disk drive
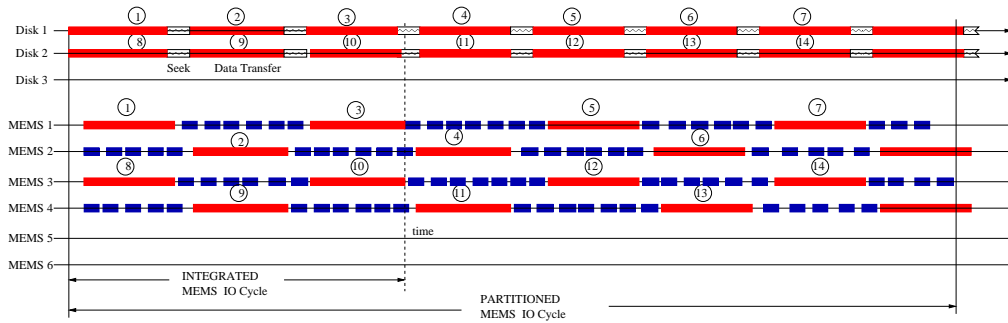
Figure 6.6: Partitioned Scheduling.

serves a large fraction of low bit-rate streams, it may find buffer space to be inadequate to sustain high throughput levels. Another disk drive serving exclusively high bit-rate streams may waste buffer space.

Under-utilized MEMS bandwidth: The portion of the MEMS buffer/cache that is unused also implies unused MEMS buffering bandwidth. The unusable bandwidth on the MEMS buffer/cache drives up the DRAM buffering requirement for streams by requiring larger accesses on the MEMS device to deliver the throughput required.

Figure 6.6 depicts the data transfer operations for a partitioned MEMS buffer for a three-disk RAID system with two MEMS devices per disk. Of these, Disk 3 has zero load and hence its corresponding MEMS devices (#5&6) are completely unutilized. In a real system, such load imbalance scenarios may arise when stream popularity changes or when disks are added or removed dynamically. Given a RAID system where load is imbalanced, the partitioned solution fractures the MEMS buffer space as well as bandwidth and is unsuitable under the conditions outlined above. The *integrated* solution that we propose next overcomes these shortcomings.

113

Integrated MEMS buffer

The *integrated* MEMS buffer configuration applies only to the integrated hardware abstraction. It overcomes the shortcomings of the partitioned solution by sharing a common MEMS buffer/cache pool between all the RAID drives. However, single-disk solutions cannot be directly applied to this configuration. Since the buffer/cache resources are now common rather than being completely exclusive, new sharing and protection schemes need to be introduced.

Figure 6.5(b) depicts the stream flows in this configuration for the integrated hardware architecture. Disk drives that are more loaded than others use up more of the MEMS buffer and cache space as well as bandwidth. Even in the case of load imbalance or bit rate heterogeneity, all MEMS resources are utilized. Since all the available MEMS buffer/cache bandwidth is utilized, DRAM requirement is also minimized.

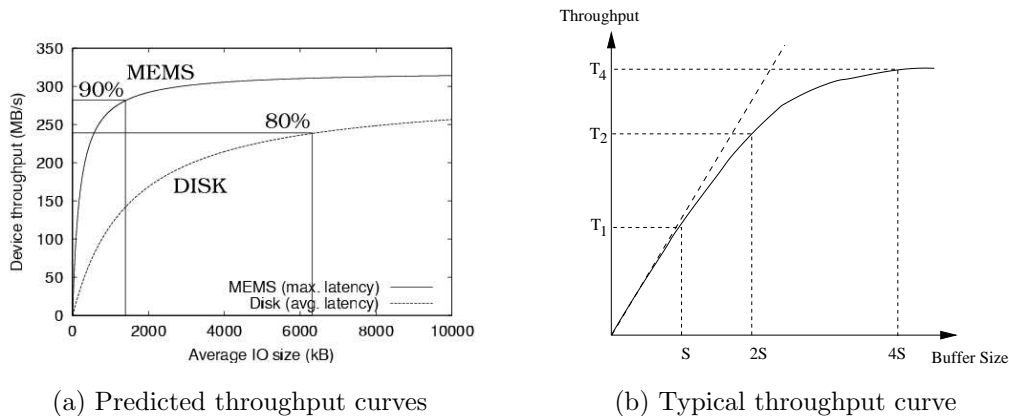(a) Predicted throughput curves   (b) Typical throughput curve

Figure 6.7: Throughput characteristics of NUMA storage.

The potential improvement due to an integrated MEMS buffer over the partitioned can be best explained using throughput curves. Figure 6.7 represents the predicted and typical throughput curves for non-uniform access storage devices. Figure 6.7(a) shows the superior access characteristics of the MEMS

device compared to the disk drive for hypothetical year 2007 devices. Since the MEMS device is able to achieve higher throughput utilizations at a lower average IO size, it is able to reduce DRAM buffering cost by an order of magnitude for streaming workloads while enabling higher disk throughputs as a low-cost buffer [63]. In the context of multiple disk RAID systems, a partitioned MEMS buffer may limit disk throughput for disks with higher load due to less available MEMS buffer space (Figure 6.7(b)) as compared to the integrated configuration which allows for all available MEMS buffer space to be shared. At the same time, the integrated MEMS buffer consolidates all the available MEMS buffer bandwidth for proportionate sharing and thus improves the overall throughput characteristics of the MEMS buffer bank. More available MEMS buffer bandwidth in turn reduces DRAM buffering cost.

In the integrated configuration, since all the drives may access any portion of the buffer/cache, contention for the resources makes the solution more complex than that for the partitioned configuration. To resolve space and bandwidth contention, new data placement and IO scheduling strategies are required. Since these resources are now part of a common pool, they must be dynamically allocated based on the load on the individual disk drives. The resource allocation strategy must accommodate both overload and under-load at the individual drives. At the same time, it is also important that bandwidth and space load on the individual devices in the MEMS device pool is balanced to minimize the DRAM requirement and improve the overall throughput of the storage system.

We now outline the resource allocation strategy for both space and bandwidth on the MEMS devices. To absorb overload at an individual drive, the resource allocator does not differentiate between disks and disk IOs. Its view of the RAID is equivalent to a single device from where all IOs are executed. Whenever a new stream is serviced on a disk drive, the MEMS space and bandwidth resources required for the stream is allocated based on the load on the individual MEMS devices.

One observation to make here is that the space and bandwidth requirements for a given stream are proportional to each other[1]. However, the bandwidth resource is not easily time-shared and a significant, non-uniform switching overhead exists when serving more than one stream. As a result, a balanced space-load does not imply a balanced bandwidth-load and vice-versa.

Resource allocation and Load balancing

A purely space-based or purely bandwidth-based resource allocation and load-balancing is unlikely to yield good overall performance. A bandwidth-loaded device may be the least space-constrained device. Similarly a space-constrained device, may possess adequate spare bandwidth. Our resource allocation strategy combines space-based and bandwidth-based allocation into one.

The time-cycle utilization of a MEMS device gives a good approximation of how resource constrained the device is on both counts. A space-constrained device utilizes more of the time-cycle for data transfers, whereas a bandwidth-constrained device utilizes more of the time-cycle for both switching as well as data transfers. The MEMS devices in the pool are sorted according to their time-cycle utilizations. When a new stream must be served, the MEMS device with the least utilized time-cycle is picked. If the buffer space available on the device is insufficient for the new stream, the next device in the sorted list is chosen. This procedure is followed till a device satisfying both space and bandwidth requirements is found.

Figure 6.8 shows the IO operations scheduled on the MEMS devices for the duration of a MEMS IO cycle. The number and nature of streams served (N) is the same as that in the partitioned case (shown in Figure 6.6). In the integrated case, all the available MEMS devices are utilized regardless of the load distribution on the disk drives. A simple round-robin strategy is used to

---

[1]Here, we assume that all disk drives operate the same IO cycle duration. In reality, however, this observation holds even under more relaxed assumption. Exploring the validity extent of this observation must be left to a separate study.
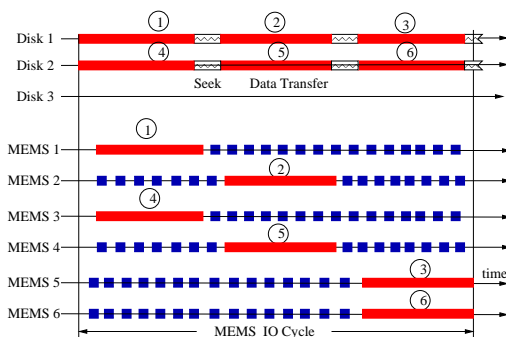
Figure 6.8: Integrated Scheduling.

balance the IOs on the MEMS buffer. Also notice that the MEMS IO cycle duration is considerably shortened. Since the DRAM buffering requirement is directly proportional to the length of the MEMS IO cycle, the integrated MEMS buffer/cache configuration minimizes the DRAM buffering cost for a given IO workload.

Although the above strategy may work for a simply load-imbalanced system, bit-rate heterogeneity adds additional complexity to the problem. When the load and bit-rates are both imbalanced, it is necessary to balance both the number of streams buffered as well as the bandwidth requirement of each MEMS device. One possible heuristic to this end is to assign each stream from a disk to a MEMS device in the bank in a round-robin fashion. This would distribute the number of streams equally over all the MEMS devices. If the variance in bit-rate of streams from a single disk is not significant, this strategy would work well. Another heuristic first tries to balance the bandwidth requirement of the MEMS devices. In the case that the bandwidth requirements are balanced, the heuristic assigns any new stream to the MEMS device which serves the minimum number of streams.

In the next section, we evaluate the performance of the integrated MEMS buffer using combination of the above approaches for a restricted domain of

load-imbalance and bit-rate heterogeneity.

6.3 Experimental Evaluation

This section presents an experimental evaluation of a streaming multimedia system equipped with MEMS storage. The evaluation presented in this section is based on the analytical model introduced in [63].

Parameter	FutureDisk	G3 MEMS	DRAM
RPM	$20,000$	–	–
Max. bandwidth [MB/s]	300	320	$10,000$
Average seek [ms]	2.8	–	–
Full stroke seek [ms]	7.0	0.45	–
X settle time [ms]	–	0.14	–
Capacity per device [GB]	1,000	10	5
Cost/GB [\$]	0.2	1	20
Cost/device [\$]	100–300	10	50–200

Table 6.1: Performance characteristics of storage devices in the year 2007.

To model the performance of the MEMS device, we closely followed one such model proposed by researchers at Carnegie Mellon University. The work in [71] describes their model comprehensively. For our experiments, we use the "3rd Generation" (G3) MEMS device model proposed in [71]. We obtained predictions for disk-drive and DRAM performance by using projections on current-day devices produced by Maxtor [50] and Rambus [59], respectively. These are summarized in Table 6.1.

In our experiments the average bit-rate of streams, $\bar{B}$, was varied within the range of 10KB/s to 10MB/s. Since the maximum bandwidth of the FutureDisk, R_{disk}, is 300MB/s, it can support tens of high-definition streams at a few megabytes per second each, more than a hundred compressed MPEG2 (DVD quality) streams at 1MB/s, or a thousand DivX (MPEG4) streams at

100KB/s, or even tens of thousands of MP3 audio at a bit-rate of 10KB/s. To minimize the mis-prediction of seek-access characteristics for the MEMS device, we assume that MEMS accesses, $\bar{L}_{mems}$, always experience the maximum device latency. We use scheduler-determined latency values, $\bar{L}_{disk}$, for disk accesses. The disk IO scheduler uses elevator scheduling to optimize for disk utilization.

In a RAID system supplied with a MEMS buffer bank, the MEMS buffer can be managed in either of the two configurations: *partitioned* or *integrated*. While the partitioned scheme offers the advantages of simplicity and potentially lesser IO bus traffic, it has performance drawbacks in certain settings. We alluded to these settings earlier in Section 6.2. Additional parameters in a streaming RAID system may lend a partitioned MEMS buffer to be cost and performance bottleneck and even make it unsuitable for a region of the parameter space. The additional parameters in a streaming RAID system are:

- *Load imbalance (i)*

- *Bit-rate heterogeneity (h)*

Although these definitions for these parameters (presented in Section 6.2) are restrictive, we use these parameter definitions to conduct an initial evaluation of the partitioned and integrated configurations. Varying other parameters apart from the above two can create different workloads. However, we feel that these two parameters are sufficient to conduct an initial evaluation.

We now conduct an evaluation of the effect of these parameters on each configuration, namely the *partitioned* and *integrated* MEMS-buffer. For both parameters, we determine the MEMS buffer-space and the DRAM buffer required to support a given IO load.

6.3.1   Effect of load imbalance

We first explore the effect of an imbalanced load on the individual RAID disks on each configuration. In the following experiment, we assume that all the streams serviced by the system have the same bit-rate. The load distribution on the disk drives in the RAID is set to follow an exponential distribution.

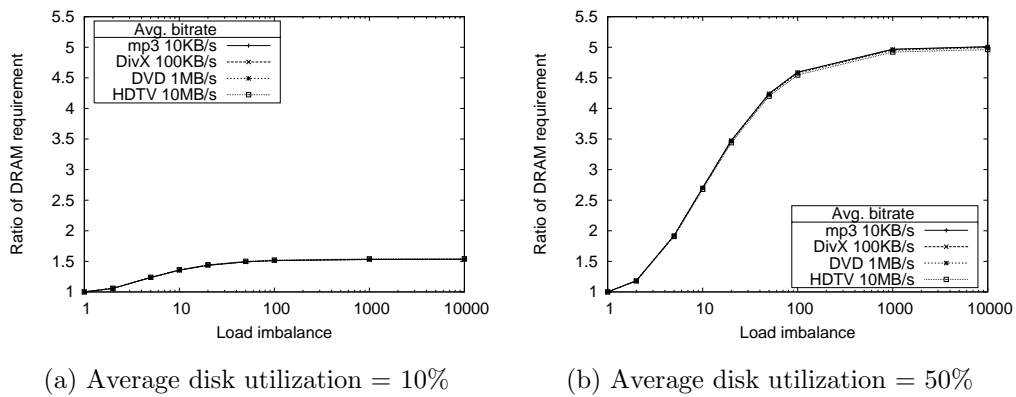(a) Average disk utilization = 10%   (b) Average disk utilization = 50%

Figure 6.9: Improvement in DRAM requirement of integrated over partitioned for different average disk utilization values.

Figure 6.9 depicts the improvement in DRAM requirement of the integrated configuration over the partitioned configuration. This improvement is the ratio of the DRAM requirement of the partitioned to the integrated configuration. When the disks are running at a low average utilization (Figure 6.9(a)), most of the disk as well as the MEMS devices are operating in the region below the knee of the disk utilization curve (depicted in Figure 6.7) for the partitioned configuration. Hence the average IO size, and consequently the DRAM requirement, is small. The integrated configuration, however, still offers potential improvement by reducing the DRAM requirement even in such a case. This is simply because the uneven disk load is evenly distributed over the MEMS bank and no single MEMS device is overloaded. When the disks are made to operate at a higher average utilization (Figure 6.9(b)), a significant number of the disks and MEMS

devices are now made to operate in the region above in the knee in the disk utilization curve. By distributing the load evenly over all the MEMS devices, the integrated configuration offers significant savings in DRAM requirement of as much as 5X.

We also note that the reduction in DRAM requirement does not vary significantly for different bit-rates. In fact, this improvement depends only on the product of the bit-rate and the number of streams served by the system, which is a constant for a given disk utilization. The slight difference we notice is due to the change in disk access overhead depending on the actual number of streams served.

Finally, we notice that the improvement levels off at higher values of load-imbalance. This quirky behavior is because of the way load-imbalance metric is defined. At higher values, although the load-imbalance value changes drastically, the actual load does not change significantly and therefore the DRAM requirements for either configuration do not change significantly.

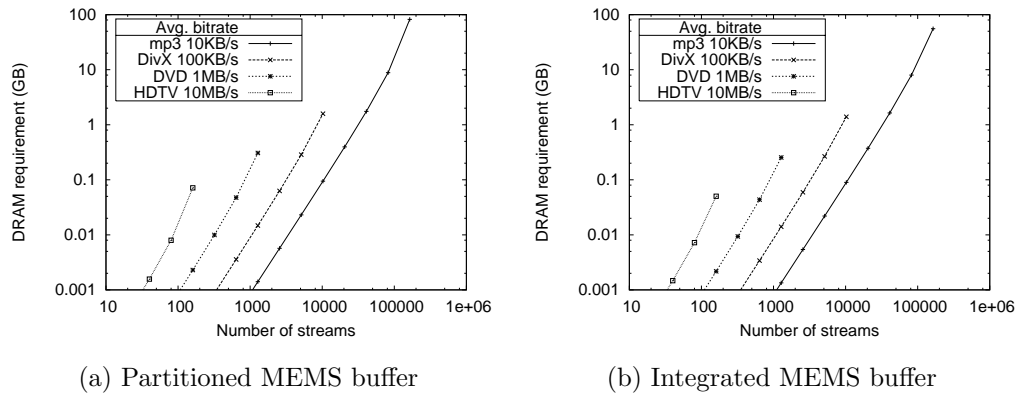(a) Partitioned MEMS buffer   (b) Integrated MEMS buffer

Figure 6.10: DRAM requirement for load imbalance $(i) = 2$.

Figures 6.10 and 6.11 show the actual DRAM requirement for the partitioned and integrated configurations under different load conditions and for different
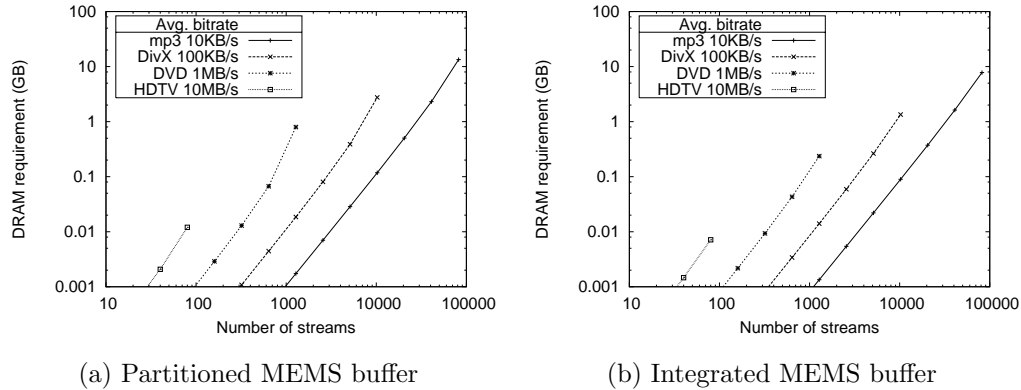
(a) Partitioned MEMS buffer   (b) Integrated MEMS buffer

Figure 6.11: DRAM requirement for load imbalance $(i) = 10$.

average bit-rates. For each curve, we also assume that all streams share the same bit-rate. We vary the total number of streams serviced by the system and load the system to the point when the disks simply do not possess the raw throughput required to support all the streams. As observed for the single-disk case, streaming 10MBps HDTV streams requires far lesser DRAM than 10KBps mp3 streams. Due to a balanced MEMS buffer load in the integrated configuration, it can support more number of streams than the partitioned case, given a fixed amount of DRAM.

Figure 6.12 shows the ratio of the DRAM requirements for the partitioned and integrated configurations for the same experiment. The trend shows an increase as the number of streams are increased for a given average bit-rate. This is expected because the load on the disks and MEMS devices increases thereby also increasing the actual difference in load. The maximum improvement is different for different bit-rates because at these points the total streamed bandwidths are also different. Savings in DRAM are as much as 1.47X for a load-imbalance of 2 and as much as 3.3X for a load-imbalance of 10. For a greater load-imbalance, these values will be higher.
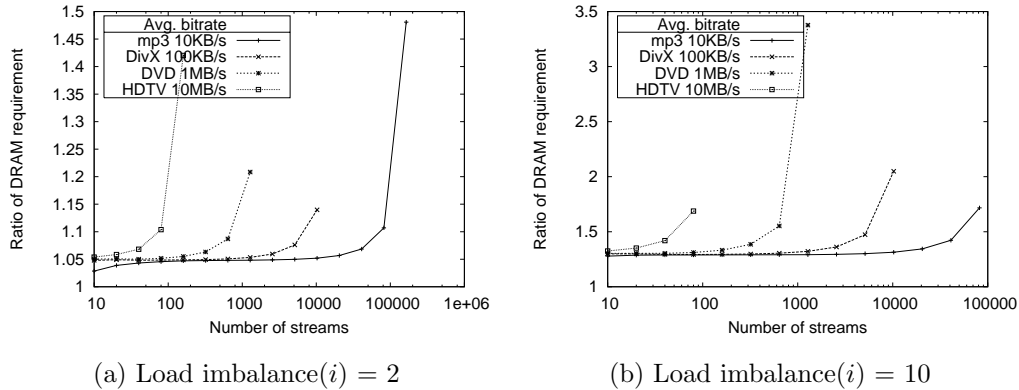
(a) Load imbalance$(i) = 2$ (b) Load imbalance$(i) = 10$

Figure 6.12: Improvement in DRAM requirement of integrated over partitioned for different load imbalance values.

The IO Bus Bottleneck: Although it seems as if the integrated configuration will always perform better than the partitioned, it may not be the case. If the storage hardware is set up such that the MEMS buffer is an independently accessed device on the IO bus, the IO bus bandwidth requirement is twice the total streamed bandwidth as mentioned earlier in Section 6.1. In Figure 6.13 we limit the IO bus bandwidth to the sum of the maximum throughputs of the eight RAID drives. The improvement of the integrated over the partitioned increases rapidly at lower disk utilizations. However, the integrated configuration performs worse than the partitioned over total system utilizations of 50% because of insufficient IO bus bandwidth.

6.3.2 Effect of bit-rate heterogeneity

Apart from load-imbalance, bit-rate heterogeneity is another factor for an uneven space load on the MEMS buffer. In the case of bit-rate heterogeneity the MEMS buffering requirement for each disk is different. The integrated configuration can handle bit-rate heterogeneity by distributing the MEMS buffering requirement for each disk over all the MEMS devices in the MEMS bank.
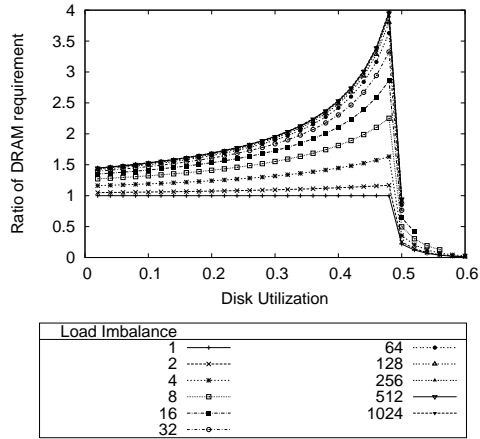
123

Figure 6.13: Improvement in DRAM requirement of integrated over partitioned in the presence of limited IO bus bandwidth.

It is important to point out here that bit-rate heterogeneity does not impact the total DRAM requirement. If all the disk throughputs are the same, the MEMS device throughputs are also the same. Since the total number of streams serviced by the entire system is also a constant, the product of the average IO size for the entire MEMS bank and the total number of streams gives us the DRAM requirement, a constant value. We now explore the effect of bit-rate heterogeneity on the MEMS buffering requirement for both configurations.

To obtain Figure 6.14, we conduct the following experiment. Keeping the total bandwidth requirement of each disk constant, we vary the average bit-rate streamed by the disks by using employing bit-rate heterogeneity (h). Disks with lower indices stream a lower average bit-rate than those with higher indices.

Figure 6.14 presents the MEMS buffering requirement for each disk for $h = 2$. This represents a factor of two difference between the lowest average bit-rate to the highest average bit-rate. In a system that serves a heterogeneous mix of streaming media, this is reasonable to expect. For the integrated case, the MEMS buffering requirement is uniform across all the MEMS devices and does not depend on individual disks. It is therefore a horizontal line for each value
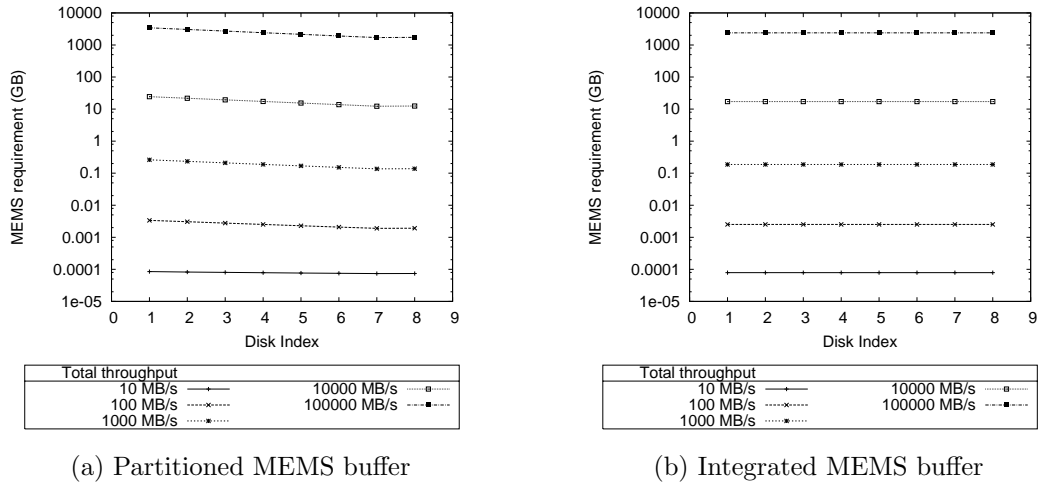
124

(a) Partitioned MEMS buffer   (b) Integrated MEMS buffer

Figure 6.14: MEMS buffer requirement for individual disks for bit-rate hetero-geneity $(h) = 2$.

of the total system throughput. For the partitioned case, since the number of streams serviced by a disk with a lower average bit-rate is greater, disks with lower average bit-rate require more MEMS buffer space than ones which serve high bit-rate streams. This variance in MEMS buffer space requirement, though small, can lead to inadequate MEMS buffer for certain disks and thereby reduce the overall throughput of the system.

Figure 6.15 presents the MEMS buffering requirement for each disk for $h = 1000$. If a system divides the streams over the disks based on their bit-rates, some disks may service streams with significantly lower bit-rate than others. There is a factor of 1000X in the bit-rates of mp3s and HDTV streams. That being the case, the variance in MEMS buffering requirement is a few orders of magnitude. With a 20GB partitioned MEMS buffer per disk, only system throughputs lesser than $10,000$MB/s can be supported. For the integrated case, this value is higher and can be estimated in the range of approximately $25,000$MB/s.

(a) Partitioned MEMS buffer   (b) Integrated MEMS buffer
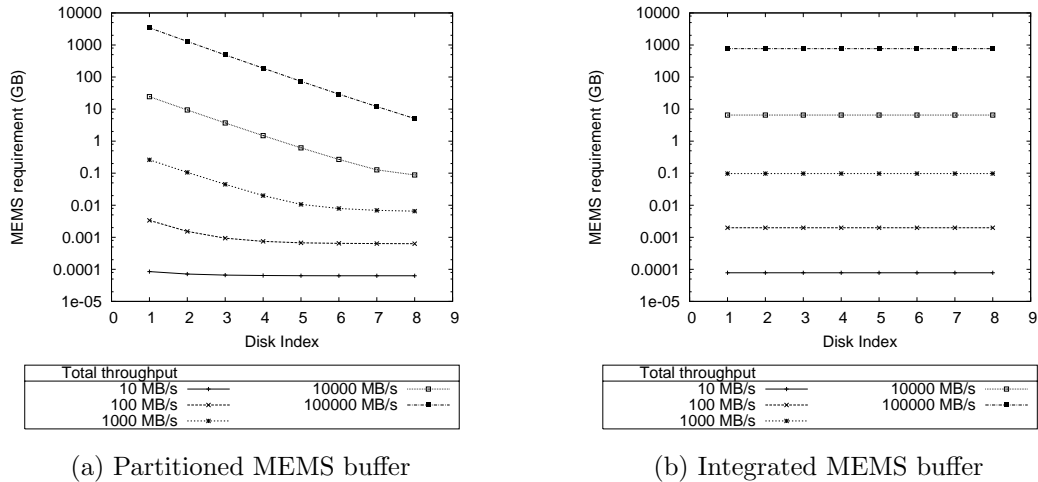
Figure 6.15: MEMS buffer requirement for individual disks for bit-rate heterogeneity $(h) = 1000$.

6.4 Summary

In this chapter, we proposed new architectures for storage systems using MEMS-based storage, an emerging technology, and evaluated its value for real-time multimedia applications. We explored the impact of integrating emerging storage technology into existing storage architectures on the class of streaming media applications. This approach was a specific-case solution for the more general problem of managing heterogeneous data and storage. We proposed rules-of-thumb for partitioning data types across storage devices. We presented hardware architectures and logical hardware abstractions as well as proposed and evaluate mechanisms for managing heterogeneous streaming storage systems that can also support real-time streaming data. We demonstrated that an integrated MEMS buffer can handle both load-imbalance and bit-rate heterogeneity in a RAID-based real-time storage system.

Chapter 7

Related Work

7.1 Quality of Service: Disk Drives

Video broadcasting and multicasting have been used in some cable-based or satellite-based movie-on-demand channels, in which requests for a movie arriving within a period of time are "grouped" (i.e., batched) together and served with a single stream [44, 42, 85]. Schemes have been proposed to support viewing interactivity in a broadcast (or multicast) environment [3, 15, 57]. However, these schemes often double the bandwidth requirement for clients or require a considerable amount of client buffering. In this paper, we propose a client/server dual architecture that manages media data for enabling interactivity between broadcasters/multicasters and clients.

Several schemes have been proposed for supporting fast-scans. These can be classified into three approaches:

1. *Increase playback rate.* This approach increases the playback frame rate for supporting a fast-scan operation. But, this method is impractical, since no TV can display more than 30 frames per second. Second, the IO bandwidth, memory use, and CPU overhead can be exceedingly high.

2. *Use separate fast-scan streams [10, 76, 75, 82].* This approach cannot be

used in the broadcast (or multicast) scenario because a program is broadcast at a single rate. We can add a unicast channel to deliver fast-scan data, but this provision would require additional network bandwidth.

3. *Skip frames in the regular stream [24].* The frame-skipping approach, if not designed carefully, can cause low IO-resolution and consequently low system throughput. However, if we break a sequential IO into small IOs to read every k^{th} frame, we can end up with worse, rather than better, IO performance. We believe that skipping frames is nevertheless the right approach for supporting fast-scans under our dual client/server setting. In this study, we have proposed methods to improve IO resolution and to reduce IO overhead in order to implement this approach efficiently.

Data placement and IO scheduling for improving disk throughput have been studied extensively in the context of multimedia file systems [8, 55, 76, 75, 82, 98]. Most work has been done on read-only systems like video-on-demand, and have not explored simultaneous storage and retrieval of video streams. Even in read-only systems, low-level device optimizations have not been explored. Our work offers a low-level optimized solution to simultaneous storage and retrieval of continuous media. Low-level data placement and IO scheduling strategies can also improve the performance of read-only systems like multimedia data servers. We differentiate our approach from previous efforts in two respects:

1. *Fine-grained* device management: We collect detailed disk parameters directly from a disk to make more effective real-time device management decisions.

2. *Integrated* device management: We provide an integrated strategy, from disk feature extraction, data organization, and data placement, to IO scheduling. We show that trade-offs often exist between design parameters, and we propose methods to find optimal trade-offs under different workload scenarios.

Several previous studies have focused on the problem of disk feature extrac-

tion. Worthington, Ganger, et al. [68, 92] extract disk features in order to model disk drives accurately. Both studies rely on interrogative SCSI commands to extract the LBA-to-PBA mapping from the disk. Patterson et al. [81] present several methods for empirical feature extraction, including an approximate mapping extraction method. In [1], the authors propose methods for obtaining detailed temporal characteristics of disk drives, including methods for predicting rotational delay. Diskbench is open source and available for download [20]. At this time, we are not aware of any other disk feature extraction tool which is currently available for download as open source, and which runs on a widely available operating system without any kernel modifications.

Multimedia file-system efforts in recent years include [33, 35, 49, 52, 74, 80]. Of these, the industry initiatives usually do not disclose their implementations. To the best of our knowledge, unlike XTREAM, existing admission controllers for streaming multimedia systems do not use disk modeling based on low-level disk profiling. In our work, most of the storage management policies are based on accurate low-level profiling of disk parameters.

To decrease the response time of disk IO requests, we proposed *semi-preemptible* disk IO. Before the pioneering work of [14, 52], it was assumed that the nature of disk IOs was inherently non-preemptible. In [14], the authors proposed breaking up a large IO into multiple smaller chunks to reduce the data transfer component ($T_{transfer}$) of the *waiting time* ($T_{waiting}$) for higher-priority requests. A minimum chunk size of one track was proposed. In this paper, we improve upon the conceptual model of [14] 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.

7.2 Quality of Service: MEMS storage

Multimedia workloads fall into two main categories: (1) *best-effort* and (2) *real-time*. Text and image data require a *best-effort* service. In most cases, retrieval of these data require prompt service and must be starvation-free. Continuous data forms like audio and video fall into the *real-time* category, imposing real-time requirements on data retrieval. The *best-effort* class of multimedia data usually displays spatial and/or temporal locality attributes. Traditional caching policies [77] are well suited for such applications. The work of [71] shows that including a MEMS-based cache in the storage hierarchy can improve I/O response time by up to 3.5X for best-effort data.The focus of this work is to examine if such an addition can also improve the performance of the real-time class of multimedia data.

To service real-time disk IOs, two classes of scheduling algorithms have been proposed. *Quality Proportion Multi-Subscriber Servicing* (QPMS or *time-cycle* based scheduling) [60] and the *Earliest Deadline First (EDF)* [14] are representative scheduling algorithms. Several improvements to these algorithms have been since investigated [8, 84, 52, 73]. In this thesis, we built upon the time-cycle model [60] for scheduling continuous media on MEMS devices. [1]

Scheduling multimedia streams on a k-device MEMS bank is similar to scheduling streams on a disk array. However, using a MEMS bank as a disk buffer must consider the real-time requirements between the disk and MEMS storage as well as between MEMS storage and DRAM. On the other hand, disk load balancing policies [11, 66, 90, 91] are directly applicable to a MEMS bank when it is used as a cache. We investigated two representative policies from previous work on disk arrays for a MEMS bank.

---

[1]Of the two classes of continuous media schedulers, the *time-cycle* based scheduler has gained popularity due to its simplicity and inherent support for timing guarantees for individual streams. Apart from this, the time-cycle service model also simplifies admission control and the incorporation of starvation-free service for non-real-time data.

Research effort is also underway to explore other aspects and applications of MEMS-based storage. In [96], the authors exploit the two-dimensional nature of MEMS-storage access and develop a tabular data placement scheme for relational data. They show that using MEMS storage can improve IO utilization and cache performance for relational data. In a recent study, the authors have also demonstrated data placement techniques on the MEMS device for the general problem of declustering two-dimensional data [97]. In [43], the authors present power conservation strategies for MEMS-based storage devices. There has been a fair amount of interest in modeling and simulating MEMS-based storage [23, 29, 30]. So far in our work, we have used the CMU model [30] for the MEMS-based storage device.

Using assistive storage technologies can enhance the performance of disk-based storage systems significantly. MEMS-based storage devices in particular play a complementary role to that of disk drives and can aid in designing high-performance storage systems. This approach to improving storage system performance is gaining momentum. Schlossler et. al. [71] have proposed using MEMS-based storage devices as a disk cache and have reported a 3.5X improvement in IO response time for best-effort data. Uysal et. al. [87] report that replacing disk arrays entirely or partially with MEMS-based storage improves storage latency as well as throughput significantly for file-system and database traces over conventional arrays. Ying et. al. [95] also report about lower power consumption of MEMS-based storage devices compared to disk drives.

The emergence of MEMS-based storage devices may change existing storage architectures. One interesting aspect is the combination of disk drives with MEMS storage. There are many possible ways to combine disk drives with MEMS storage. In [87], the authors investigate few disk array architectures where the use of MEMS storage is most beneficial. They propose a number of novel architectures to examine the potential placements of MEMS-based storage in a disk array. They show that replacing disks with MEMS-based storage or

131

by using hybrid MEMS/disk arrays, can provide substantial improvements in performance and cost/performance over conventional arrays.

Following a similar direction, we extended our initial work on caching and buffering real-time data on MEMS-based storage [63] to address the problem of managing multiple disks and MEMS-based storage banks. We introduce the general problem of managing heterogeneous data and storage and propose rules-of-thumb for partitioning data types across different storage components. We propose high-level hardware architectures for a heterogeneous storage system that contains both disk drives and MEMS-based storage components. We then propose abstractions for these architectures, specifically the partitioned and integrated MEMS abstractions. Finally, we propose and evaluate data allocation and real-time IO scheduling strategies for a disk-MEMS heterogeneous storage system.

Chapter 8

Concluding Remarks

In this chapter, we present a brief summary of the dissertation highlighting its significant contributions. We then present possible directions in which the work presented here can be extended in the future.

8.1 Dissertation Summary

In this dissertation, we addressed the problem of real-time storage using two principal approaches. First, we fine-tuned existing disk-only storage systems by proposing, implementing, and evaluating new strategies for providing guaranteed-rate IO (using real-time IO scheduling and admission control), best-effort IO completion (by bandwidth reservation for non-real-time IOs), high-throughput (using data placement and IO scheduling), and short response time (using semi-preemptible IO). Unlike earlier approaches, these strategies are designed and implemented using accurate knowledge of low-level disk parameters provided by Diskbench, our disk profiling tool. These strategies lead to significant improvement in storage system performance over existing solutions and traditional file-systems.

Second, we proposed new architectures for storage systems using MEMS-

based storage, an emerging technology, and evaluated its value for real-time multimedia applications. We explored the impact of integrating emerging storage technology into existing storage architectures on the class of streaming media applications. This approach was a specific-case solution for the more general problem of managing heterogeneous data and storage. We proposed rules-of-thumb for partitioning data types across storage devices. We presented hardware architectures and logical hardware abstractions as well as proposed and evaluated mechanisms for managing heterogeneous streaming storage systems that can also support real-time streaming data. We demonstrated that an integrated MEMS buffer can handle both load-imbalance and bit-rate heterogeneity in a RAID-based real-time storage system.

8.2 Future Work

The research presented in this dissertation can be further explored and extended in at least two directions.

With respect to disk-only real-time storage solutions, we can improve upon the Diskbench-based approaches presented here. Semi-preemptible IO [18] is a powerful capability which has not been fully explored [19]. In a multi-programmed, multi-priority IO system, Semi-preemptible IO provides the additional capability of preempting long-running IO jobs in favor of retrieving more urgently required data. Although dominant in real-time CPU scheduling, preemptive scheduling algorithms is so far unexplored territory in the storage world for scheduling IO requests. Classical real-time CPU scheduling theory cannot be directly applied to storage devices, since they exhibit a significant, non-uniform switching overhead. Existing theories for real-time scheduling need to be modified, or new theories need to be developed, to account for this difference.

With respect to heterogeneous storage systems, we have proposed a possible

direction toward building real-time storage systems comprising of disk drives and MEMS-based storage and have demonstrated that they are superior in terms of both cost and performance metrics. We have however just scratched the surface of this emerging area and a lot more needs to be done to develop a generalized architecture and design for heterogeneous storage systems that comprise of multiple storage device types and that can simultaneously support a wide variety of data. Such systems bring up issues of architectural framework, space and bandwidth partitioning, IO bus contention, and support for real-time as well as non-real-time data within the same system. Heterogeneous storage systems are critical to support the next generation of applications and services in a cost-effective manner.

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] AKAMAI TECHNOLOGIES, *Content Delivery Services*, http://www.akamai.com/, (1998).

[3] K. C. ALMEROTH AND M. H. AMMAR, *The Use of Multicast Delivery to Provide a Scalable and Interactive Video-on-Demand Service*, IEEE Journal on Selected Areas in Communications, 14 (1996), pp. 1110–1122.

[4] T. E. ANDERSON, D. E. CULLER, AND D. A. PATTERSON, *A Case for NOW (Network of Workstations)*, IEEE Micro, (1995).

[5] L. R. CARLEY, G. R. GANGER, AND D. NAGLE, *MEMS-based Integrated-Circuit Mass-Storage Systems*, Communications of the ACM, 43 (2000), pp. 73–80.

[6] S.-H. G. CHAN AND F. TOBAGI, *Distributed Servers Architecture for Networked Video Services*, IEEE/ACM Transactions on Networking, (to appear).

[7] E. CHANG AND H. GARCIA-MOLINA, *BubbleUp - Low Latency Fast-Scan for Media Servers*, Proceedings of the 5th ACM Multimedia Conference, (1997), pp. 87–98.

[8] ——, *Effective Memory Use in a Media Server*, Proceedings of the 23rd VLDB Conference, (1997), pp. 496–505.

[9] K. CHANG, *A New System for Storing Data: Think Punch Cards, but Tiny*, The New York Times, (2002).

[10] M. S. CHEN AND D. D. KANDLUR, *Stream Conversion to Support Interactive Video Playout*, IEEE Multimedia, 3 (1996), pp. 51–58.

[11] A. L. CHERVENAK AND D. A. PATTERSON, *Choosing the Best Storage System for Video Service*, Proceedings of ACM Multimedia 95, (1995), pp. 109–118.

[12] T. CLARKE, *The TerraFly project*, Nature Electronic Magazine, (2001).

[13] 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).

[14] S. J. DAIGLE AND J. K. STROSNIDER, *Disk Scheduling for Multimedia Data Streams*, Proceedings of the IS&T/SPIE, (1994).

[15] A. DAN, P. SHAHABUDDIN, D. SITARAM, AND D. TOWSLEY, *Channel Allocation under Batching and VCR Control in Video-on-Demand Systems*, Journal of Parallel and Distributed Computing, 30 (1995), pp. 168–179.

[16] T. E. DENEHY, A. C. ARPACI-DUSSEAU, AND R. H. ARPACI-DUSSEAU, *Bridging the Information Gap in Storage Protocol Stacks*, Proceedings of the USENIX Annual Technical Conference, (2002), pp. 177–190.

[17] Z. Dimitrijevic, R. Rangaswami, and E. Chang, *The XTREAM Multimedia System*, IEEE Conference on Multimedia and Expo, (2002).

[18] ——, *Design and Analysis of Semi-Preemptible IO*, Usenix File and Storage Technologies, (2003).

[19] ——, *Preemptive RAID Scheduling*, UCSB Technical Report, (2004).

[20] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya, *Diskbench URL*, http://www.cs.ucsb.edu/∼zoran/diskbench/, (2002).

[21] ——, *Diskbench: User-level Disk Feature Extraction Tool*, UCSB Technical Report, (2004).

[22] 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).

[23] I. Dramaliev and T. Madhyastha, *Optimizing Probe-based Storage*, Proceedings of USENIX File and Storage Technologies, (2003), pp. 103–114.

[24] W. Feng, F. Jahanian, and S. Sechrest, *Providing VCR Functionality in a Constant Quality Video-On-Demand Transportation Service*, ICMCS, (1996).

[25] T. Gandhi and M. Trivedi, *Motion Analysis of Omni-Directional Video Streams for a Mobile Sentry*, ACM International Workshop on Video Surveillance, (2003).

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google File System*, ACM SOSP, (2003).

[27] G. GIBSON ET. AL., *A Cost-Effective, High-Bandwidth Storage Architecture*, Proceedings of the 8th International Conference on Arhitectural Support for Programming Languages and Operating Systems(ASPLOS-VIII), (1998).

[28] G. GILDER, *Fiber Keeps Its Promise*, Forbes, (1997).

[29] J. L. GRIFFIN, J. SCHINDLER, AND S. SCHLOSSLER, *Timing-accurate Storage Emulation*, Proceedings of USENIX File and Storage Technologies, (2002).

[30] J. L. GRIFFIN, S. SCHLOSSER, G. GANGER, AND D. NAGLE, *Modeling and Performance of MEMS-based Storage Devices*, Proceedings of ACM SIGMETRICS, (2000).

[31] ——, *Operating Systems Management of MEMS-based Storage Devices*, Proceedings of the 4th Symposium on Operating Systems Design and Implementation, (2000).

[32] E. GROWCHOWSKI, *Emerging Trends in Data Storage on Magnetic Hard Disk Drives*, Datatech, ICG Publishing, (1988), pp. 11–16.

[33] R. HASKIN AND F. SCHMUCK, *The Tiger Shark Filesystem*, IEEE COMPCON, (1996).

[34] HEWLETT-PACKARD LABORATORIES, *Storage Systems Program*, http://www.hpl.hp.com/research/storage.html.

[35] M. HOLTON AND R. DAS, *XFS: A Next Generation Journalled 64-bit filesystem with Guaranteed Rate IO*, SGI Technical Report, (1996).

[36] K. HUA AND S. SHEU, *Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-on-demand Systems*, ACM Sigcomm, (1997), pp. 89–101.

[37] D. M. JACOBSON AND J. WILKES, *Disk Scheduling Algorithms based on Rotational Position*, HPL Technical Report, (1991).

[38] T. JOHNSON AND A. ZHANG, *A Framework for Supporting Quality-Based Presentation of Continuous Multimedia Streams*, Proceedings of the 4th IEEE Conference on Multimedia Computing and Systems, (1996), pp. 169–176.

[39] R. E. KALMAN, *A New Approach to Linear Filtering and Prediction Problems*, Transactions of the ASME–Journal of Basic Engineering, 82 (1960), pp. 35–45.

[40] KIONIX INC., http://www.kionix.com/, (2002).

[41] S.-H. LEE, K.-Y. WHANG, Y.-S. MOON, AND I.-Y. SONG, *Dynamic Buffer Allocation in Video-on-Demand Systems*, Proceedings of ACM SIGMOD International Conference on Management of Data, (2001), pp. 343–354.

[42] V. O. K. LI AND W. LIAO, *Distributed Multimedia Systems*, Proceedings of the IEEE, 85 (1997), pp. 1063–1108.

[43] Y. LIN, S. A. BRANDT, D. D. E. LONG, AND E. L. MILLER, *Power Conservation Strategies for MEMS-Based Storage Devices*, Proceedings of IEEE MASCOTS, (2002), pp. 53–63.

[44] T. LITTLE AND D. VENKATESH, *Prospects for Interactive Video-on-Demand*, IEEE Multimedia Magazine, (1994), pp. 14–24.

[45] C. LIU AND J. LAYLAND, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, ACM Journal, (1973).

[46] C. R. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle, *Towards Higher Disk Head Utilization: Extracting Free Bandwith From Busy Disk Drives*, Proceedings of the OSDI, (2000).

[47] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, *TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks*, Proceedings of the Symposium on Operating Systems Design and Implementation, (2002).

[48] D. Makaroff, G. Neufeld, and N. Hutchinson, *An Evaluation of VBR Disk Admission Algorithms for Continuous Media File*, Proceedings of the 5th ACM Multimedia Conference, (1997), pp. 143–154.

[49] C. Martin, P. Narayan, B. Ozden, and R. Rastogi, *The Fellini Multimedia Storage System*, Journal of Digital Libraries, (1997).

[50] Maxtor Corporation, *Atlas 10KIII-U320 Product Datasheet*, (2002).

[51] M. McKusick, W. Joy, S. Leffler, and R. Fabry, *A Fast File System for UNIX**, ACM Transactions on Computer Systems 2, 3 (1984), pp. 181–197.

[52] A. Molano, K. Juvva, and R. Rajkumar, *Guaranteeing Timing Constraints for Disk Accesses in RT-Mach*, Real Time Systems Symposium, (1997).

[53] G. E. Moore, *Cramming More Components Onto Integrated Circuits*, Electronics, 38 (1965).

[54] Nanochip Inc., http://www.nanochip.com/, (2002).

[55] R. Ng and J. Yang, *Maximizing Buffer and Disk Utilizations for News On-Demand*, Proceedings of the 20th VLDB Conference, (1994), pp. 451–462.

[56] NUA INTERNET SURVEYS, *Streaming Media Grows Popular in US*, http://www.nua.ie/surveys/, (2001).

[57] W.-F. POON AND K.-T. LO, *Design of Multicast Delivery for Providing VCR Functionality in Interactive Video-on-demand Systems*, IEEE Transactions on Broadcasting, 45 (1999), pp. 141–8.

[58] D. PSALTIS AND G. W. BURR, *Holographic Data Storage*, IEEE Computer, 32 (1998), pp. 52–60.

[59] RAMBUS INC., *RDRAM*, http://www.rambus.com/, (2002).

[60] P. V. RANGAN, H. M. VIN, AND S. RAMANATHAN, *Designing and On-Demand Multimedia Service*, IEEE Communications Magazine, 30 (1992), pp. 56–65.

[61] R. RANGASWAMI, E. CHANG, AND Z. DIMITRIJEVIC, *Disk-aware Data Mangement for Interactive Media Services (Extended Version)*, http://www-db.stanford.edu/~echang/ics-extended.ps, Technical Report, (2001).

[62] R. RANGASWAMI, Z. DIMITRIJEVIC, E. CHANG, AND K. E. SCHAUSER, *MEMS-based Disk Buffer for Streaming Media Servers (extended version)*, http://www.cs.ucsb.edu/~raju/mems-x.pdf, (2002).

[63] ——, *MEMS-based Disk Buffer for Streaming Media Servers*, Proceedings of IEEE International Conference on Data Engineering, (2003), pp. 619–630.

[64] R. RANGASWAMI, Z. DIMITRIJEVIC, K. KAKLIGIAN, E. CHANG, AND Y.-F. WANG, *The SFinX Video Surveillance System*, IEEE Conference on Multimedia and Expo, (2004).

[65] P. V. Rangen, H. M. Vin, and S. Ramanathan, *Designing an On-Demand Multimedia Service*, IEEE Communications Magazine, (1992), pp. 56–64.

[66] 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), pp. 44–55.

[67] J. P. Scheible, *A Survey of Storage Options*, IEEE Computer, 35 (2002), pp. 42–46.

[68] J. Schindler and G. R. Ganger, *Automated Disk Drive Characterization*, CMU Technical Report CMU-CS-00-176, (1999).

[69] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger, *Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks*, . Proceedings of the USENIX Conference on File and Storage Technologies, (2004).

[70] S. W. Schlosser and G. R. Ganger, *MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole?*, Proceedings of the USENIX Conference on File and Storage Technologies, (2004).

[71] S. W. Schlosser, J. L. Griffin, D. Nagle, and G. R. Ganger, *Designing Computer Systems with MEMS-based Storage*, Architectural Support for Programming Languages and Operating Systems, (2000), pp. 1–12.

[72] Seagate, *SCSI Interface, Product Manual 2*, http://www.seagate.com/support/disc/manuals/scsi/38479j.pdf, (1999).

[73] C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri, *On Scheduling Atomic and Composite Multimedia Objects*, IEEE Transactions on Knowledge and Data Engineering, 14 (2002), pp. 447–455.

143

[74] 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).

[75] P. J. SHENOY AND H. M. VIN, *Efficient Support for Scan Operations in Video Servers*, Proceedings of the 3rd ACM International Conference on Multimedia, (1995), pp. 131–140.

[76] ——, *Efficient Support for Interactive Operations in Multi-resolution Video Servers*, ACM Multimedia Systems, 7 (1999).

[77] A. J. SMITH, *Cache Memories*, ACM Computing Surveys, (1982), pp. 473–530.

[78] C. STAUFFER AND E. GRIMSON, *Learning Patterns of Activity Using Real-Time Tracking*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 22 (2000).

[79] SUN MICROSYSTEMS, *The Intelligent Storage Network*, http://www.sun.com/storage/vision/, (1998).

[80] V. SUNDARAM, A. CHANDRA, P. GOYAL, P. SHENOY, J. SAHNI, AND H. VIN, *Application Performance in the QLinux Multimedia Operating System*, ACM Multimedia, (2000).

[81] N. TALAGALA, R. H. ARPACI-DUSSEAU, AND D. PATTERSON, *Microbenchmark-based Extraction of Local and Global Disk Characteristics*, UC Berkeley Technical Report, (1999).

[82] 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).

[83] D. A. THOMPSON AND J. S. BEST, *The Future of Magnetic Data Storage Technology*, IBM Journal of Research and Development, 44 (2000).

[84] T.-P. J. TO AND B. HAMIDZADEH, *Dynamic Real-time Scheduling Strategies for Interactive Continuous Media Servers*, ACM/Springer Multimedia Systems, 7 (1999), pp. 91–106.

[85] F. A. TOBAGI, *Distance learning with digital video*, IEEE Multimedia Magazine, (1995), pp. 90–94.

[86] J. W. TOIGO, *Avoiding a Data crunch*, Scientific American, 279 (2000), pp. 58–74.

[87] M. UYSAL, A. MERCHANT, AND G. A. ALVEREZ, *Using MEMS-based Storage in Disk Arrays*, Proceedings of Usenix File and Storage Technologies, (2003), pp. 89–101.

[88] 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), pp. 323–340.

[89] C. WANG, D. J. ECKLUND, E. F. ECKLUND, V. GOEBEL, AND T. PLAGEMANN, *Design and Implementation of a LoD System for Multimedia Supported Learning for Medical Students*, iWorld Conference on Educational Multimedia, Hypermedia & Telecommunications ED-MEDIA, (2001).

[90] J. WOLF, P. YU, AND H. SHACHNAI, *Disk Load Balancing for Video-on-Demand Systems*, ACM Multimedia Systems, (1997).

[91] J. L. Wolf, P. S. Yu, and H. Shachnai, *DASD Dancing: A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer Systems*, Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, (1995), pp. 157–166.

[92] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes, *Online Extraction of SCSI Disk Drive Parameters*, Proceedings of ACM Sigmetrics Conference, (1995), pp. 146–156.

[93] B. L. Worthington, G. R. Ganger, and Y. N. Patt, *Scheduling Algorithms for Modern Disk Drives*, Proceedings of the ACM Sigmetrics, (1994), pp. 241–251.

[94] 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 Annual ACM International Conference on Multimedia (ACMMM), (2003).

[95] L. Ying, S. Brandt, D. Long, and E. Miller, *Power Conservation Strategies for MEMS-based Storage Devices*, Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002), (2002).

[96] H. Yu, D. Agrawal, and A. E. Abbadi, *Tabular Placement of Relational Data on MEMS-based Storage Devices*, International Conference on Very Large Data Bases, (2003).

[97] ——, *Declustering two-dimensional Datasets over MEMS-based Storage*, International Conference on Extending DataBase Technology, (2004).

[98] P. Yu, M.-S. Chen, and D. Kandlur, *Grouped Sweeping Scheduling for DASD-based Multimedia Storage Management*, Multimedia Systems, 1 (1993), pp. 99–109.

146

[99] 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).