Optimizing Disk Performance for a Multimedia Storage Server*

Ian Alderman   Mamadou Diallo

Abstract

One potential bottleneck point in a multimedia server is the disk system. The capacity and sustained transfer rates for inexpensive disks have increased significantly in recent years, making it plausible to build a multimedia storage server from commodity hardware. However, currently available file systems were not designed for use with multimedia access patterns, which involve periodic transfer of large quantities of data with real time constraints. We compare the performance of several different methods of accessing the disk system on Solaris and FreeBSD. We conclude that the performance overhead of accessing data through existing file systems outweighs the convenience advantage, and that scheduling disk accesses to minimize seek times and reduce the variation in response time is important in providing peak performance.

1 Introduction

Networking technology promises to change the way we receive information. Although the new media has not yet replaced the old, applications such as interactive TV, on-demand course content, web based news supplemented with multimedia, and the rise of economically feasible music distribution systems such as those recently proposed by Napster and mp3.com, indicate that the change is coming.

The popular adage is that, "A picture is worth a thousand words." In digital storage, it's worse: a picture takes more than a thousand words worth of storage space, and continuous media, such as digital audio and video, takes up even more. For example, an eight minute audio track encoded using MPEG-1, Layer 3 at 192 Kbps takes 11.25 MB to store. A one hour MPEG-1 video (1.5 Mbps) takes 675 MB, And an MPEG-4 video of a 100 minute movie (4.0 Mbps) takes 3 GB to store. It is not feasible to store this quantity of data in memory for cost reasons: it must be stored on disk [6].

Multimedia file servers built from commodity hardware can store hundreds of gigabytes of data, and serve hundreds of customers simultaneously. The

*This research was performed as course work for CS-736: Advanced Operating Systems, taught by Remzi Arpaci-Dusseau, in Fall 2000, at the University of Wisconsin, Madison. Test hardware was provided by the SWORD project.

performance of these servers is not necessarily constrained by network capacity, since modern switches and network cards can handle gigabit speeds, and a single server could be attached to the network through several NICs.

Although disk capacity has increased significantly in recent years, disk performance has not increased as rapidly. In some prototype testing, we determined that a multimedia file server's disk interface can exhibit poor performance if designed incorrectly. We built a simple application that read data in the manner that a multimedia server would if each stream was served by a single process. We ran one instance of this simple application and measured the performance. Then we ran several instances concurrently and noticed that the overall throughput from the disk decreased when several streams were being accessed simultaneously.

Our goal in this paper is to determine the fastest interface to the disk system for a multimedia file server running on commodity hardware. We compare several different methods of I/O, including `read` and `pread`, `mmap`, and asynchronous reads through implementations of the POSIX.2 standard. We attempted to allow the system to optimize the concurrency and ordering of disk requests through system and user level threads, and compare the performance to single threaded versions. We compare the performance of FreeBSD and Solaris.

2 Motivation

Streaming media servers have different performance criteria from traditional file servers. Continuous media has, on one hand, a real time constraint: clients must receive data on a stream before playback, and the time at which a segment of a file must be received is determined by the time at which the client begins playback and the rate at which playback occurs. Although some jitter from the disk system and from the network is acceptable, to provide smooth playback, it must be kept to a minimum. On the other hand, there is not a performance reason why a segment should be received much before it is needed, in fact, timely delivery prevents exhausting client buffers. A key insight is that minimizing the tail of the response time distribution is more important for performance given the real time requirements of continuous media than minimizing the average response time [11].

Stated differently, blocks retrieved from a video file server are retrieved on a strict schedule, and missing deadlines in this schedule is acceptable only rarely.

Our goals in performing this research included:

- Quantifying the performance characteristics of various methods for accessing continuous media data stored on disk.
- Determining the maximum number of concurrent streams of data our test system could support given a certain playback rate, and the best method for retrieval.
- Determining why some retrieval methods performed better than others.

- Contributing to the development of a video file server for the SWORD project[1], by providing a tool for the analysis of various design decisions such as the amount of memory included and the number of disks attached, and by providing a basis for the implementation of the disk interface for this server.

3 Related and future work

Disk scheduling is a well understood area, and several groups have explicitly addressed the specific needs of multimedia storage systems [6, 10, 3, 5], as well as comparing various approaches to striping multimedia data across several disks for load balancing [7, 11, 9]. However, none of these papers explicitly measure disk performance, nor do they compare the performance of interfaces such as raw I/O and FFS.

We hope to expand the research summarized in this paper to test new techniques to improve the performance of our storage server prototype. One obvious addition would be to include several disks in the benchmarks to determine the implementation complexity and assess the potential for contention on the system and SCSI busses.

In the database community, it is recognized that managing disks directly allows the application developer to perform optimizations explicitly, and prevents the operating system from attempting unsuitable optimizations [8]. We have quantified these differences specifically for the multimedia storage server application, which has quite different performance goals from databases, such as real-time requirements.

4 Methodology

Our test hardware consisted of a standard PC with two 500MHz Pentium II's, 512 MB of memory, and four Seagate ST39183W SCSI-2 8638MB drives attached to two Adaptec 2940 Ultra SCSI adapters. Solaris and FreeBSD were installed on the first two drives, and tests were performed on one of the others.

For the tests which involved the file system, we first created a new file system with 8 KB blocks, the maximum allowed under Solaris. We then populated the file system completely with 100 MB files, written sequentially 1MB per write. We thought that this might lead to misleading results if the files were accessed in the same order they were written, so we wrote them in a randomly permuted order, but one after the other. No other file system aging was done. We believe

[1] The SWORD project is a research effort at the University of Wisconsin-Madison the goals of which include "Scalable Wide-area On-demand Reliable Data Delivery." Research results to date include bandwidth conserving stream merging techniques [2] and caching strategies [1]. Analytic and simulation results are confirmed through the development of a prototype, which, in its current implementation, does not store video files, but rather performs stream merging and caching. Future plans include the possible development of a storage server which would provide a fast interface to a number of attached disks.

that this will produce somewhat optimistic results from the file system tests: that is, a real system may be somewhat more fragmented and provide somewhat poorer performance than that which we describe here.

For each test, we took as input a number of streams, a block size for each read, and a bit rate. All tests were written in C, and designed to compile on both Solaris and FreeBSD. Seven binaries were produced, each corresponding to a different I/O method. The specifics of each are described in greater detail below.

Each program first calculated the amount of time allotted for each *round*, where a round is defined to be the period of time allowed for retrieval of the given block size for the streams. For each round, two metrics were captured: whether or not the round was completed by its (strict) deadline, and how long the round took since it began. As described above, a strict deadline is defined by the playback schedule of the client, and consists of a point in time determined by the start of playback and the playback rate. Each test consisted of 100 rounds.

In addition to checking whether each round was completed by its strict deadline, we also captured the absolute amount of time each round took to complete and kept some statistical information on these times.

Between tests, we flushed the file system buffer cache by unmounting and remounting the file system.

5 I/O Methods

In this section, we discuss the implementation details of some of the I/O methods we employed. There are several different parameters we could vary, described in the subsections below.

- Operating system: we tested on Solaris and FreeBSD.
- I/O system call: `read` (`pread`), asynchronous I/O (`aio`), and `mmap`.
- Block size: we tested 1024 KB blocks and 512 KB blocks.
- Thread model: we tested a single thread of execution, system scheduled threads, and user library threads (pthreads). System scheduled threads are not available on FreeBSD.
- File system: through the file system, and through the associated character device.

However, we did not test all combinations for various reasons. For example, `mmap` is not available on character devices. We only tested multiple threads with `read`, since `aio` provided poorer performance.

5.1 Operating System

Our code provided cross-platform support between Solaris 8 / Intel and FreeBSD 4.2 (a custom kernel was built to include AIO support) through the use of `#define` macros; in general the interfaces were similar. We attempted to include Linux in our tests, but the lack of support for files larger than 2GB (required for accessing the raw I/O device) prevented us from doing this easily. We were able to configure the system to dual boot these two systems, so the hardware we tested was exactly the same.

The intent of performing the tests on multiple operating systems was to determine whether there were implementation specifics and differences in what options were configurable that would affect the results of the tests.

5.2 File system vs. Character Device

The difference between file system access and character device access was one of the most significant differences we tested and the differences provide our key result. Notably, file system performance differed significantly between the two implementations, with Solaris providing much poorer performance.

The differences between accessing files through the file system and through the character device (which appears as a single 8636 MB file to the operating systems) required a different binary for each test, but for tests that were performed both through the file system and the character device, the only differences between the two versions are in the code that opens and reads the data.

Reading files larger than 2GB (including raw I/O access to partitions larger than 2GB) requires a modification to the original FFS on which the file systems for Solaris and FreeBSD are based; Solaris provides two interfaces to files, one of which uses 32 bit offsets and another which uses 64 bit offsets; we used the regular (32 bit) interface for the file system tests and the 64 bit interface for the raw access. FreeBSD uses large offsets by default.

To make sure that we weren't skewing our performance results, we ran a simple test using the raw I/O method and determined that reads at the beginning of the disk (6.9 seconds per 100 MB) are quite a bit faster than reads at the end of the disk (10.9 seconds per 100MB). In order to make sure that our tests took this variation into account, when a raw test was performed that used some number of streams less than 86, we spread the initial offsets of the streams out across the entire disk.

The following table illustrates the file system implementation performance differences between Solaris and FreeBSD. The data presented is the number of seconds to retrieve 100 MB, averaged across the entire disk.

	Solaris	FreeBSD
RAW I/O	7.8	7.8
File system	13.9	7.9

In Solaris, sequential raw reads of 100 MB across the entire disk averaged 7.8 seconds, while sequential reads of 100 MB files filling the file system averaged 13.9 seconds. In FreeBSD, sequential raw reads of 100 MB averaged 7.8 seconds, while sequential reads averaged 7.9 seconds. This poor file system performance for sequential reads on Solaris affected all of the results obtained through the file system.

5.3 Block Size

For the majority of our tests, we chose a block size of 1 megabyte, chosen based on results from [7]. Özden et. al. suggest that efficiency can be improved by allowing the system to perform sequential reads which are as large as possible and by minimizing the number of seeks that are required.

Since we get nearly 60 concurrent streams maximum performance from a two year old disk in our performance tests, and if we imagine that a production multimedia file server will have approximately 10 disks attached, we think that block sizes larger than 1 MB will result in a prohibitively expensive memory requirement. If we assume 10 disks and 60 streams per disk, the system will transfer 600 MB each round. While a round is being transfered from disk, the previous round also needs to stay in memory, so that it can be transfered over the network, so this 10 disk system will require 1.2 GB of memory. Larger blocks would increase the memory requirement, correspondingly increasing cost of the file server significantly.

Tests were performed using smaller block sizes, and the results confirmed the expected results that fewer streams were supported.

5.4 Blocking Reads

Our implementation of synchronous reads used `read` when reading through the file system, and `pread` when reading the character device. This was certainly the simplest implementation, and surprisingly provided the best performance, achieving near optimal throughput, despite allowing no concurrency in I/O requests.

5.5 Asynchronous reads

We expected that asynchronous reads would provide near peak performance through the file system due to our hope that the kernel and disk system would schedule reads in the most optimal order. We were disappointed because (as far as we've been able to determine) the system does not transfer each 1 MB block in an atomic operation, preventing the disk's firmware optimizations from being effective for large data transfers. From a resource management perspective, this is the correct approach, but for our application, it provides poorer performance.

The code that performed asynchronous reads initiates all reads at once (using `aioread` in Solaris, and the corresponding `aio_read` in FreeBSD), and then loops, waiting for each one to return (`aiowait`, `aio_waitcomplete`).

Asynchronous reads are included by default in Solaris, but require a kernel recompilation in FreeBSD. They are not available in Linux.

To the contrast of synchronous I/O mentioned above, once processes have requested data from the system and they can not be serviced immediately, they do not have to block until their request returns. Instead, they move on to doing something else and are notified of the completion of their request. One observation that can be immediately inferred from asynchronous I/O is that it requires a lot more overhead than its traditional synchronous counterpart. The system is now forced to keep track of more process state for the processes until their requests have been fulfilled. As a result, asynchronous I/O can turn out to be worse than expected (as will be shown shortly); for instance, as the number of requesting processes grows, performance degradation increases, leading to poor performance.

We believe that asynchronous I/O can provide performance improvements in some applications, with different performance requirements than ours. We refer the reader to [4], where it has been shown that in some benchmarks AIO outperforms synchronous I/O by as much as 50%.

In Solaris, a system thread is generated for each request. Once all the processes have been lined up for service, the threads are multiplexed around the required resource. As already mentioned, AIO boosts performance by allowing the calling process to perform other critical tasks until it can no longer proceed without accessing the results of the outstanding AIO operations. There are situations when the user may want to check the status of the outstanding AIO operations regularly. However, we want to minimize excess checking for performance reasons. Using polling to check the status also hurts performance. A better choice is to use asynchronous notification for the completion of AIO operations via perhaps a signal handler.

We considered implementation of the more traditional asynchronous read options in UNIX, `select` and `poll`, but found that they don't work on normal files: both always return true, indicating that a non-blocking read is possible, when in fact this is not the case with 1 MB reads.

5.6 Mmap

Through the file system, `mmap` provides the best performance on Solaris, despite our extremely simple implementation (we transformed each blocking `read` call into an `mmap` call followed by a checksum of each 512th bit to confirm that the data had in fact been copied into memory). It would be interesting to compare this technique with the asynchronous alternative, issuing one `mmap` call for each stream first and then attempting to do the reads, although our experience with asynchronous disk scheduling was disappointing. Single threaded `read` performed slightly better than `mmap` under FreeBSD. Neither Solaris nor FreeBSD allowed `mmap` to a character device.

5.7 Multithreading vs. Single Threading

Another approach that we took to allowing the system to optimize the order in which multiple asynchronous reads are performed was to spawn several threads, each of which waits on a signal to start reading at the beginning of each round.

This code was more complicated and took longer to implement and debug. We implemented threads using the `pthreads` interface, which is shared between Solaris and FreeBSD. Solaris allows the programmer to set the level at which threads are scheduled to be "system"; FreeBSD does not. On Solaris, both system level threads and user level threads performed the same, however. FreeBSD does not support system level threads, and we were unable to run some of the tests using the file system and threads to completion perhaps due to implementation errors, in our code or in the system libraries; the same tests completed on Solaris.

6 Results

Table 1 summarizes the results we obtained through the various I/O methods we implemented (using our strictest method for calculating the maximum number of streams and 1MB reads). This strict method allowed a maximum of one round to complete after its deadline as defined by the start time and the bit rate of the stream.

method	*RAW I/O*		*File System*	
	Solaris	FreeBSD	Solaris	FreeBSD
`read`	63	63	25	53
`aio`	33	33	23	39
threads (user)	29	49	18	N/A
`mmap`	N/A	N/A	31	51

Table 1: Comparison of maximum number of 1.5 Mbps streams for various I/O methods (1 MB block size).

The maximum number of streams (63) was obtained through the raw interface on both Solaris and FreeBSD. This number is quite close to the expected maximum derived in two different ways.

Özden et. al. suggest that the maximum number of streams q for a given disk must satisfy the following equation [7] (the parameters are from vendor supplied information for the disk we're using):

display rate	r_{disp}	1.5 Mbps
inner track disk transfer rate	r_{disk}	120 Mbps
settle time	t_{settle}	.6 ms
seek time	t_{seek}	17 ms
rotational latency	t_{rot}	8.33 ms
block size	d	1 MB

$$q \cdot \left(\frac{d}{r_{\text{disk}}} + t_{\text{rot}} + t_{\text{settle}} \right) + 2 \cdot t_{\text{seek}} \leq \frac{d}{r_{\text{disp}}}$$

This calculation results in a maximum of 70 streams.

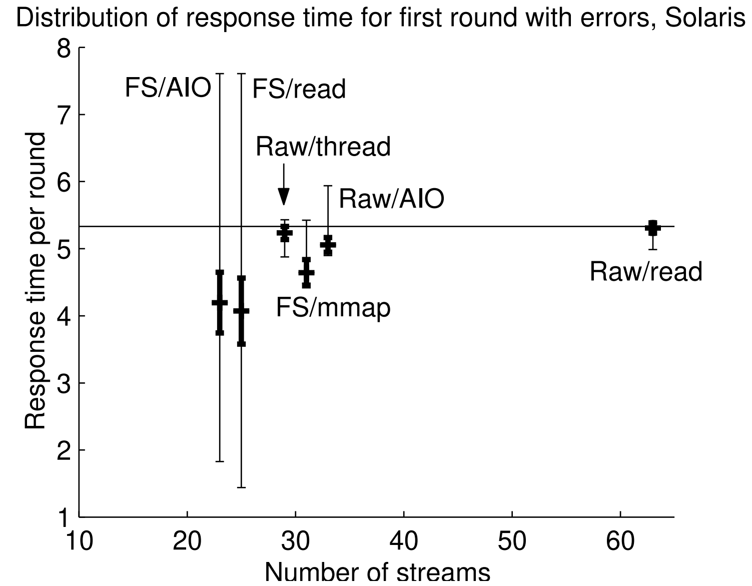
The other method for calculating the maximum number of streams involves the observed average sequential transfer rate (100 MB / 7.8 s = 102.6 Mbps):

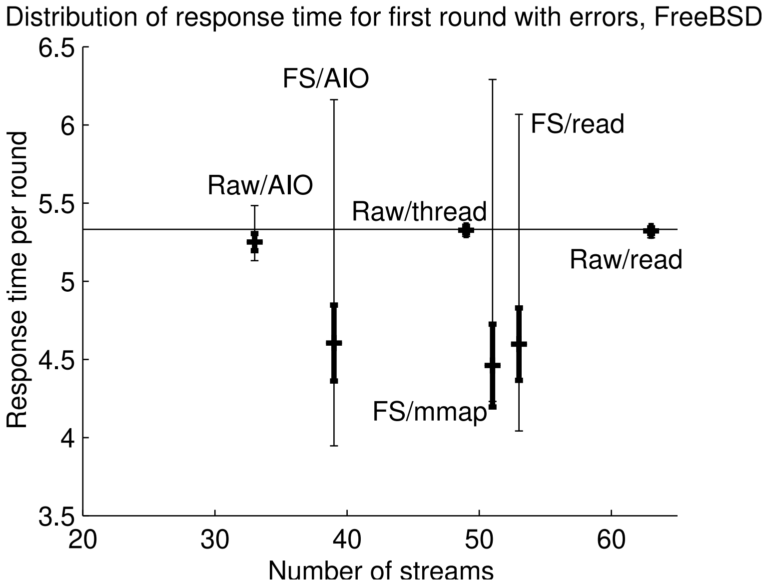
$$n_{\text{streams}} = \frac{r_{\text{xfer}}}{r_{\text{disp}}} = \frac{102.6 \text{ Mbps}}{1.5 \text{ Mbps / stream}} = 68 \text{ streams}$$

Given these calculations, our target of nearing the maximum throughput of the disk is achieved.

The following charts draw out some of the reasons why various techniques fail to perform optimally. Our tests measured the maximum number of streams attainable by incrementally increasing the number of streams and reading 100 MB for each stream. In each round per test, the amount of time required to complete the round was captured. The maximum number of streams without errors was calculated by subtracting one from the number streams in the first test that contained errors. The distribution of the response times in the first test that contained errors is depicted here; complete plots of the response times for several of the methods are provided later in the paper.

In these plots, the X axis indicates the number of streams for which the response data is collected; for each method the number of streams selected is the least with any errors at all. In the Y axis, he darker line indicates the mean (the horizontal line in the middle) and standard deviation (extending above and below the mean) of the response times for that method at that number of streams, while the thinner line indicates the maximum and minimum response time.





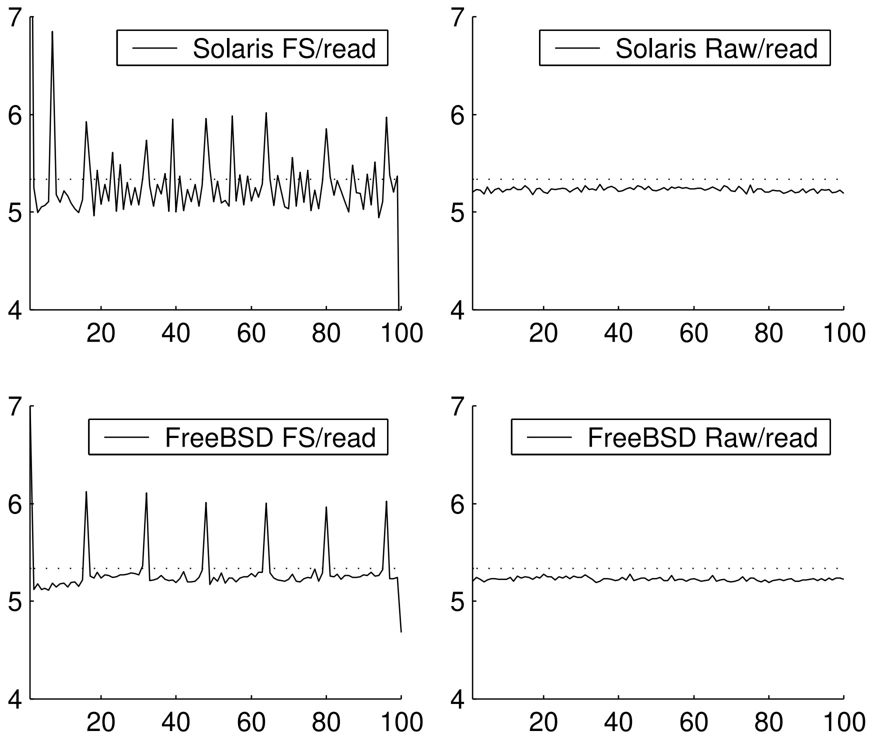
For example, in Solaris, the number of streams in which an error was first observed for the FS/`read` method was 23, the mean was 4.07, the standard deviation was 0.49, the maximum was 7.6, and the minimum was 1.44.

The Raw/`read` method performs better than the others on both Solaris and FreeBSD. In addition, synchronous reads through the file system perform second best on FreeBSD, but only near the worst on Solaris; slower than any method on FreeBSD. In fact, all of the file system methods performed much worse on Solaris than on FreeBSD.

With the exception of `mmap` on Solaris, the variation in response time from the file system methods was significantly greater than it was in any of the raw methods. On both Solaris and FreeBSD, both Raw/AIO and Raw/threads suffered from a performance problem that is not related to a high variation in response time.

The following charts illustrate the distribution of the response times for four tests directly: the X axis here is the round number, and the Y axis is the number of seconds that round took to complete in this test. In each chart, the dotted line is the 5.3 second deadline for each round. The charts on the left show clearly how much more variable synchronous reads through the file system are than through the raw interface, and give some insight to what degrades performance for these methods.

The distributions selected were those with the mean service time at the maximum observed less than the deadline. Solaris was able to deliver just 36 streams through the file system under this condition, while FreeBSD delivered 61 streams. Both delivered 63 streams using the raw interface.



Note that the first round takes much longer than most of the other rounds, and that the last round takes much less. We surmise that this is because the first round is performing some prefetching and that the last round's data has been fetched already during the previous round. The pattern of access in FreeBSD is much more regular; regular spikes occur perhaps indicating that more data than normal is being retrieved in these rounds.

7 Conclusions

We tested the performance characteristics of various I/O techniques on Solaris and FreeBSD and their suitability for a multimedia storage server application. We found that in Solaris, file system performance for sequential reads is quite poor, and that on both systems, file system performance was more highly variable (and thus unsuitable for a multimedia storage server) than through the raw interface to the disk. We found that explicitly scheduling disk requests to minimize seek time resulted in better performance than allowing the system to schedule disk requests using asynchronous I/O or threads.

References

[1] J. M. Almeida, D. L. Eager, and M. K. Vernon. A Hybrid Caching Strategy for Streaming Media Files. In *Proc. Multimedia Computing and Networking 2001*, January 2001.

[2] D. Eager, M. Vernon, and J. Zahorjan. Minimizing Bandwidth Requirements for On-Demand Data Delivery. Technical Report 1418, Computer Science Department, University of Wisconsin, Madison, July 2000.

[3] R. Haskin and F. Schmuck. The Tiger Shark File System. In *Proceedings of 41st IEEE Int. Conf.: Technologies for the Information Superhighway*, pages 226–231, Februrary 1996.

[4] S. Leung. Programming Asynchronous I/O in Solaris 2. WWW, 1996. http://www.sunworld.com/sunworldonline/swol-03-1996/swol-03-aio-2.html.

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

[6] B. Özden, R. Rastogi, and A. Silberschatz. On the storage and retrieval of continuous media data. In *Conference on Information and Knowledge Management*, November 1994.

[7] B. Özden, R. Rastogi, and A. Silberschatz. Disk striping in video server environments. In *Proceedings IEEE International Conference on Multimedia Computing and Systems (ICMCS'96), Hiroshima, Japan*, June 1996.

[8] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, August 1999.

[9] J.R. Santos, R. Muntz, and B. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of ACM SIGMETRICS 2000, Santa Clara, CA*, pages 44–55, June 2000.

[10] P. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of ACM SIGMETRICS Conference, Madison, WI*, pages 44–55, June 1998.

[11] P J. Shenoy and H M. Vin. Efficient striping techniques for multimedia file servers. In *Proceedings of the Seventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97), St. Louis, MO*, pages 25–36, May 1997.