Experiences with Evaluating System QoS and Channel Performance on Media-On-Demand Systems [*]

Wonjun Lee
Dept. of Computer Sci. & Eng.
Korea University
Seoul, Republic of Korea

Jaideep Srivastava
Dept. of Computer Sci. & Eng.
University of Minnesota
Minneapolis, MN 55455 USA

Abstract

This paper presents the design and implementation of a continuous media file system, which has been implemented in the context of a distributed multimedia application development environment that has been prototyped. To make a performance analysis of file systems and distributed object services for continuous media provisioning, we validate the performance analysis of file system on media-on-demand (MOD) systems against that of a conventional file system through an experimental evaluation. **Keywords:** Multimedia, Quality of Service, Media-On-Demand System, Continuous Media, File System.

1 Introduction

While the mechanisms provided by the Unix File System (UFS) have been sufficient for most applications, there are important classes of applications where it is not been so. An increasingly important class of applications where UFS is not suitable are those requiring storage and retrieval of *continuous media (CM)*, i.e. audio, video, animation, etc. [1]. Following are some unmet file system needs of continuous media: *Placement and Storage Structures*. The key to designing high performance file systems to support CM requires that we use information about (i) the inherent temporal nature of continuous media, and (ii) QoS oriented nature of media access, to develop new solutions to these problems.

The initial work in the area, e.g. [2], focused on the principles behind continuous media storage and retrieval, and led to the development of the *incremental retrieval model*. A number of investigations, e.g. [3, 4, 5, 6], have developed techniques for continuous media access, and have carried out their evaluation, usually by means of simulation. Similarly, work in the area of file placement includes [7], where different approaches to storing continuous media data have been proposed. In some cases simulation-based evaluation has been performed. While some good incremental retrieval models exist, and a number of techniques for CM storage and access have been developed, there is a serious dearth of actual file system implementations which are suited for a broad range of multimedia applications. A number of efforts have focused on developing I/O storage and management systems for CM to fulfill the needs of specific applications, e.g. [8]. Conversely, given the fact that under low resource utilization whether the underlying system is cognizant of an application's real-time needs or not does not matter, there have been efforts to use high-performance general purpose file systems to manage CM data, as evidenced by the Tiger-Shark file system [9] which has shown good performance in some multimedia applications. In our opinion this approach suffers from two problems. First, it is not scalable since keeping the load low means building the sys-

[*] This work was supported by U.S. Army Research Lab (ARL) number DA/DAKF11-98-9-0359 to the University of Minnesota.

tem with very high capacity. Second, such a system will have a seriously unacceptable price-performance ratio.

Overall, we believe that a continuous media file system *must* use information about the inherent nature of continuous media in all its functions to ensure both high performance and a good price-performance ratio. Hence, it must be built on the principles of incremental retrieval,which is emerging as the model to capture the inherent nature of continuous media. To our knowledge, the file systems that fit this criteria are [10], where some principles of the incremental retrieval model have been used, especially in the design of the admission control strategy. However, one issue that is still lacking from any study we have seen is the evaluation of any file system from the viewpoint of QoS metrics. After all, a continuous media application's performance (and possibly correctness) needs are expressed in terms of its QoS specification, and hence the underlying system's performance must be measured in terms of how successful it is in attaining the specified QoS. In the multimedia networking community this is today the accepted way of evaluating communication protocols for continuous media. We believe that the file system community must do the same.

We have compared the experimental behavior of the Unix file system, as instantiated in the Solaris 2.7 operating system, with that of the Presto file system. The experimental data used conformed to that specified for motion JPEG (MJPEG) video.

The structure of this paper is as follows: in section 2 we describe the design considerations and architecture of the Presto file system. Section 3 provides details of PFS implementation. The first part of section 4 describes our experimental comparison of PFS and UFS for basic file operations, while the second part does the same for continuous media access. In section 5 we conclude the paper.

2 Design Considerations

To handle continuous media efficiently, we need to store and retrieve large amounts of multimedia information with continuous playback, providing user-specified QoS. The CM file system should provide larger data unit abstractions, such as *video frame* and *audio sample group*, unlike conventional file systems, e.g. UFS, which provide only a byte-oriented abstraction. A *unit* is a user-defined logical chunk of data, e.g. a frame for video data and a sequence of audio samples for audio data. Henceforth, the abstraction can support CM application's retrieval and storage needs via units in a CM stream. In addition, it should allow an application to randomly seek to a unit within a stream.

Another design objective for the CM file system is to provide efficient I/O access to the disk. This objective is crucial for CM retrieval and storage where real-time continuous delivery is required and where high volume I/O bandwidth is required. To do so, new storage and access strategy should provide timely delivery to applications via optimizing accesses, buffer management, and interface with the I/O scheduler.

3 Implementation Details

PFS is implemented on a raw disk partition of UNIX so as to bypass the UNIX block buffer cache and allow the imposition of customized access structures. A PFS partition is divided into one or more *extents*, and each extent consists of a number of *units*. A CM stream is stored in an extent. Information about each stream is stored in a structure referred to as the *inode*. A *super block* is used to maintain the formatting data and extent map of free extents. The overview of disk layout is shown in Figure 1.

The formatting information in the super block includes: (1) number of extents in partition, (2) size of each extent in blocks, (3) size of each block in physical sectors. The free extents in a partition are indexed by a bitmap. A first fit algorithm is used to allocate a free extent to a stream. A command called *Pformat* is used to format the raw partition and create the super block.

There is a unique inode assigned to each extent. Each inode is 128 bytes long, and includes the following fields: (1) name of a stream, (2) name of extent, (3) number of units in the stream, (4) stream type.

An *extent* consists of a number of units. The unit size is variable. Each extent is divided into two regions. The first region is the index region, storing the size and starting address of each unit. The other region is the data region storing a sequence of units, with each unit starting on a physical sector boundary.

4 MOD Server

We prototyped several versions of MOD server systems which support various network protocols, and integrated our continuous media file system with the MOD servers. Typical MOD server system includes a server and one or more clients, and may serve the clients concurrently. Figure 2 depicts the architecture of an MOD server which plays out multiple streams to the requesting clients across the network. The MOD server has four major components. The *Network Manager* responds to clients' connection requests. The *QoS Manager* is responsible for admission control and I/O scheduling. Each *Proxy Server* communicates with a client, receiving continuous media stream operation requests and sending the CM data by network.Each *I/O Manager* reads out CM data from disks for a proxy server. There are as many proxy servers and I/O managers as clients. Client has two main components, one *Client N/W Controller* and one *CM Player*. We have conducted performance analysis for our novel CM file systems in the MOD server system environments. Details about the results will be present in section 5.

5 Performance Analysis

5.1 On Standard File Operations

We demonstrate the advantages of PFS over UFS based on experiments that use standard file operations. The experiments were carried out on a Sun Ultra Sparcstation with a SCSI disk. The focus of our experiments was to verify that for standard file operations the performance of PFS is comparable to that of UFS.

The metrics we consider in this section are disk throughput, buffer requirement and service cycle length. Since we are interested in the performance of the disk system, it was important to ensure that the file reads were not serviced from the file cache. Hence, we used different files in each test, and chose to use each file only once to bypass the effects of system I/O buffering. To eliminate the effects of random fluctuations, we repeat each operation several times and use the average value or should it be minimum value.

In both PFS and UFS, the relationship between buffer size(b) and data access time(t) is:

$$t = (1/r_o)b + t_o \tag{1}$$

$$r = r_o/(1 + r_o t_o/b) \tag{2}$$

Here, t is the access time, r is data rate, buffer size(b) is the data size in a single read, r$_o$ is the possible maximum data rate, and t$_o$ is the constant overhead for a single read. In Unix, r$_o$ depends solely on the buffer size. In PFS, r$_o$ depends on both the unit size and the number of units in a single read.

Fig.3 shows the relationship between the data rate and the unit size. The graph shows that PFS has a higher maximum possible data rate. In PFS, data rate increases with unit size, and the more number of units per read, the higher the achievable data rate. In UFS, our experimental observations are that when the buffer size is smaller than 10KB, the data rate increases with the buffer size. When the buffer size is larger than 10KB, the data rate remains constant. When the unit size is small (below 8KB), UFS is faster than PFS. When the unit size is above 10KB, PFS is faster than UFS. In a CM Stream, we store a frame of CM data as a unit. UFS is optimal for small frames while PFS is optimal for large frames. For 640 by 480 JPEG frames, where the size of the compressed frame is about 18KB, PFS has a higher data rate than UFS. From Fig.3, we determine two ways to improve the throughout of PFS. One is to group a number of fames in to a unit. The other is to read out more units in a single read, called *group read*. The data rate increases with group size.

When there are requests for multiple streams from PFS, they compete for I/O bandwidth and other system resources, and therefore require larger buffers to handle the context switch overhead. Hence, to maintain a fixed data rate we need a longer service cycle. UFS can support up to 16 concurrent JPEG streams, each at the rate of 540KB/s. When the number of concurrent streams is more than 16, the required data rate is not satisfied. The buffer size of each stream remains the same as the frame size while the number of streams goes from 1 to 16. Under the same conditions, PFS can support up to 23 concurrent JPEG streams, while the buffer size of each stream monotonically increases with the number of streams. In PFS, when there are less than three concurrent streams, the buffer is the same as frame size. As more concurrent streams enter, we must increase the buffer size and do group read to meet the required consumption rate.

Fig.4 shows the relationship between the number of concurrent streams, and the length of the service cycle. In UFS, as the number of concurrent streams increases, the length of the service cycle increases linearly. In PFS, when there is only one stream, the length of the service cycle is very small. As the number of concurrent streams increases, the length of the service cycle increases faster than that in UFS. This shows that the overhead of resource contention of concurrent streams in PFS is more than that in UFS. This relationship between the number of concurrent streams and the length of service cycle is similar to the relationship between the number concurrent streams and the buffer size. We can deduce it from equation 1:

$$T > (sr_o t_o/(r_o - sR) \quad (3)$$

$$B > r_o R t_o/(r_o - sR) \quad (4)$$

$$S_{max} = r_o/R \quad (5)$$

Here, T is the length of the service cycle, B is the buffer size, S_{max} is the maximum number of concurrent streams, s is the number of concurrent streams, and R is the consumption rate. For a given number of streams, the length of the service cycle for PFS is longer than that for UFS, as shown in figure 4. This happens because PFS's approach to handling higher workload is to expand the length of the cycle, and thereby increase the efficiency of the I/O system by reducing context switch overhead. This is not without its drawback, since a larger buffer is required by PFS, compared to UFS, for a given workload. However, as we can observe, this strategy is better overall since it allows PFS to handle a larger range of workload, i.e. 23 streams, as compared to a maximum of 16 streams for UFS.

5.2 On QoS Metrics

This experiment is designed to measure the effect of UFS and PFS approaches to file management on drift profiles of streams, which specify the average and bursty deviation of schedules for frames from ideal expected points in time. We measured the difference between the ideal rendition time and the actual rendition time as a unit granule drift. The aggregate drift (ADF: Accumulated Drift Factor) is the sum of unit granule drifts over some interval, and the consecutive drift is the sum of consecutive non zero drifts.

In our experiments, UFS is turned out not to be well suited to support perceivable continuity of CM streams which are sensitive to drifts. Through this experiment, we can say that PFS preserves more faithfully the characteristics of real-time applications than UFS does (in Figure 5). When we increase the number of concurrent streams (1 to 20), the ADF values are increased. In measuring drift factors for single and/or multiple stream(s), we get better thoughput results (less loss) in PFS than in UFS. Comparing single stream to multiple streams, the former situation leads to lower loss than the latter for both PFS and UFS.

6 Conclusions

In the paper, we present results from the experimental evaluation of a continuous media file system, which has been implemented in the context of a distributed multimedia application development environment. Our on-going work is developing storage and access mechanisms that take advantage of QoS specificatinos to optmize system performance [11]. Putting efficient caching and buffering scheme into the current version of file system will be another future work.

7 Acknowledgments

The ideas implemented in PFS and performance studies have benefitted a lot by cooperation and discussion with a number of people. Specifically, we would like to thank Deepak Kencham of IBM Almaden and Difu Su of Yahoo USA, who mainly contributed on the initial version of this work.

References

[1] R. Steinmetz and K. Nahrstedt, eds., *Multimedia: Computing, Communications and Applications.* New Jersey: Prentice Hall, 1995.

[2] H. Vin and P. Rangan, "Efficient Storage Techniques for Digital Continuous Multimedia," *IEEE Trans. Knowledge and Data Engineering*, vol. 5, no. 4, pp. 564–573, 1993.

[3] H. V. P. Shenoy, "Cello: A Disk Scheduling Framework for Next-Generation Operating Systems," in *Proceedings of ACM SIGMETRICS*, June 1998.

[4] R. Tewari, H. Vin, A. Dan, and D. Sitaram, "Resource-based Caching for Web Servers," *ACM Multimedia Systems Journal*, 2001.

[5] A. Dan, D. Sitaram, and P. Shahabuddin, "Dynamic Batching Policies for an On-Demand Video Server," in *Proceedings ACM Multimedia 94*, pp. 15–24, ACM Press, October 1994.

[6] A. Reddy and J. Wyllie, "I/O Issues in a Multimedia System," *Computer*, vol. 27, no. 3, pp. 69–74, 1994.

[7] D. Kenchammana-hosekote and J. Srivastava, "Retrieval Techniques for Compressed Video Streams," in *Proceedings of Multimedia Computing and Networking SPIE IS&T*, January 1996.

[8] C. Federighi and L. Rowe, "The Design and Implementation of the UCB Distributed Video-On-Demand System," in *Proc. IS&T/SPIE 1994 Int'l Symp. electronic Images: Science and Technology* 1994.

[9] R. Haskin and F. Schmuck, "The Tiger Shark File System," in *COMPCON 96*, 1996.

[10] C. Martin, P. S. Narayanan, B. Ozden, R. Rastogi, and A. Silberschatz, "The Fellini Multimedia Storage Server," in *Multimedia Information Sotrage and Management* (S. M. Chung, ed.), Kluwer Academic Publishers, 1996.

[11] W. Lee and J. Srivastava, "An Algebraic QoS-based Resource Management Model for Competitive Multimedia Applications," *Multimedia Tools and Applications, Kluwer Academic Publishers*, vol. 13, no. 2, 2001.

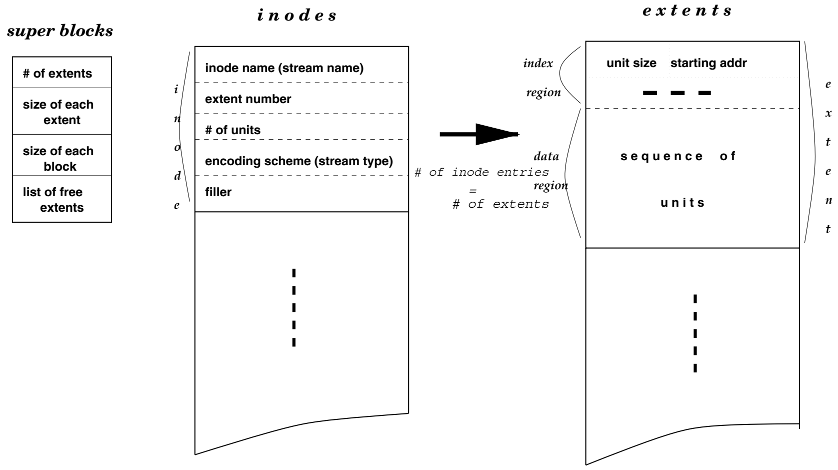

Figure 1: Disk Layout in PFS

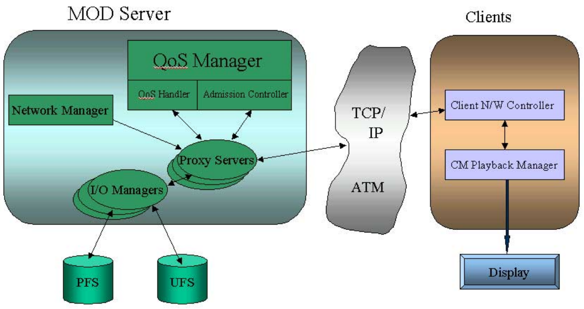

Figure 2: MOD Server Architecture

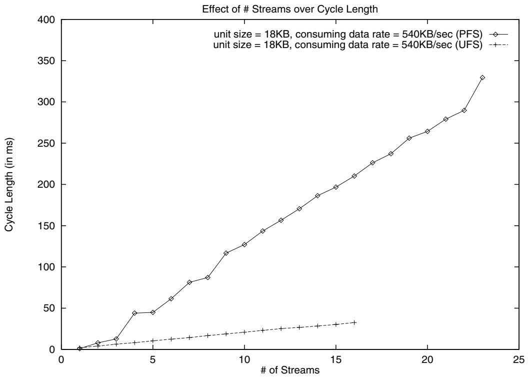

Figure 4: Effect of Number of Concurrent Streams on Cycle Length

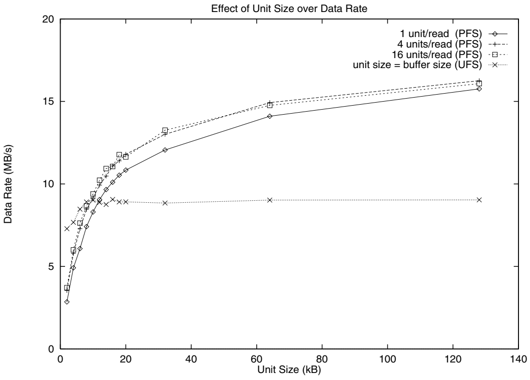

Figure 3: Effect of Unit Size on Data Rate

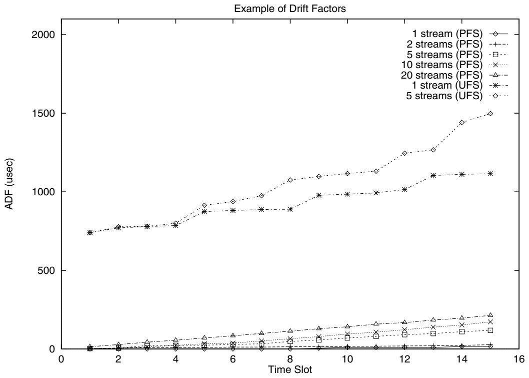

Figure 5: Unit Sequencing Drift Factors