The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Simple Continuous Media Storage Server on Real-Time Mach

Hiroshi Tezuka and Tatsuo Nakajima
Japan Advanced Institute of
Science and Technology

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Simple Continuous Media Storage Server on Real-Time Mach

Hiroshi Tezuka[†] Tatsuo Nakajima
Japan Advanced Institute of Science and Technology
{tezuka,tatsuo}@jaist.ac.jp
http://mmmc.jaist.ac.jp:8000/

Abstract

This paper presents the design and implementation of a simple continuous media storage server: CRAS on Real-Time Mach. CRAS is a specially optimized storage system for retrieving multiple continuous media streams such as audio and video from a disk at constant rates for small scale distributed multimedia systems.

Many previous continuous media storage servers have focussed on high throughput for supporting as many video sessions as possible. However, these servers are too big and complicated for playback applications that retrieve continuous media data from the local disks of personal computers. Also, there are many continuous media systems requiring small continuous media storage servers that can be shared by a small number of applications. To reduce hardware costs, the servers should run on less powerful computers. This means that the previous big and complicated servers are not appropriate for such small scale environments.

We show that our simple continuous media server for small scale systems can guarantee the retrieval of continuous media data at a constant rate, and provide high throughput even though it is compact and simple.

1 Introduction

The increasing performance of microprocessors allows us to handle continuous media such as digital audio and video on personal workstations[11]. A continuous media storage system is one of the most important components for distributed continuous media systems. Since audio and video are timing-dependent continuous media, such storage systems should be able to ensure that multiple continuous media streams can be retrieved from disks at constant rates.

Several research groups have been building continuous media storage servers for video servers which can retrieve many streams concurrently[1, 5, 12, 13, 19]. The most important goal of these systems is to retrieve as many streams as possible. In addition, these systems focus on the special organization of data blocks on disks to reduce disk seeks , rotational latency, and the overhead of disk head scheduling. Disk striping is also used to increase the throughput of the disk. Since the storage systems require these mechanisms, the servers have included large complicated implementation of directory management, block caching, meta data management, and block allocation.

On the other hand, there are many personal and interactive continuous media applications that require access to local disks or a small shared storage server. For example, a typical application for future workstations is a travel coordinator for two people in different places. They use a map in a shared window on their respective systems, and check interesting sightseeing points in a video database while using a conferencing tool to talk to each other. This application should retrieve video clips at constant rates from the database stored on disks in the user's machines, and these retrievals must avoid interference from concurrent activities contending for the same disk.

Many commercial systems such as QuickTime[4] and Video for Windows[18], used by many commercial continuous media applications, can perform a constant rate stream retrieval easily because only one application at a time is supported, so there is no contending disk activity. However, future continuous media applications will outgrow such single task systems.

Small scale distributed continuous media applications also require simple continuous media storage servers. These servers should be compact and simple to minimize the hardware costs of the computers on which the servers run. Thus, traditional complicated storage servers that require a large amount of memory and powerful CPU may not be suitable for small scale

[†]He now belongs to Real World Computing Partnership Tsukuba Mitsui Building 16F, 1-6-1 Takezono, Tsukuba-shi, Ibaraki, 305, JAPAN.

applications.

Continuous Media Player[14], which runs on Unix, supports the retrieval of multiple movie files from disks. It provides a storage system that is designed for simultaneous accesses by many continuous media applications. The system can be integrated with existing Unix applications easily because it is executed as an Unix application, but it does not guarantee continuous media stream retrieval at constant rates. This means that the applications cannot play back high quality continuous media in a timely fashion.

In this paper, we describe the design and implementation of a simple continuous media storage system, CRAS, on Real-Time Mach[6, 7, 17] which has been developed by CMU, Keio University and JAIST. CRAS is more suitable for a wider variety of applications than traditional continuous media storage systems due to its use of a microkernel-based operating system. CRAS solves the problems described in the previous paragraphs by employing a real-time microkernel.

The following design decisions were made to keep CRAS small and compact:

- CRAS adopts the same disk layout policy as the Unix file system, Thus, both file systems access the same files, and functionality that does not require real-time constraints such as system administration is processed by the Unix file system.

- CRAS provides a single function, a constant rate retrieval for playback. This makes the size of CRAS compact. Also, it is easy to add a storage server that provides another function when necessary since the server can run on the Real-Time Mach microkernel as a user-level server.

- CRAS uses the real-time capabilities provided by Real-Time Mach to ensure constant rate retrievals. CRAS consists of several threads scheduled by the microkernel. This simplifies the structure of CRAS.

Real-Time Mach is currently integrated with the Lites server[3] which provides FreeBSD, NetBSD and Linux binary compatibility. Thus, CRAS on Real-Time Mach adds multimedia functionality to a traditional Unix environment.

In addition, We compare the throughput and delay jitter of CRAS with those of the Unix file system's using a movie player application, and we show that CRAS can not only meet the timing constraints of continuous media, but also provide high throughput even though it is compact and simple.

The remainder of this paper is structured as follows. Section 2 presents the design and implementation of our continuous media storage server, CRAS on Real-Time Mach. Section 3 evaluates the performance of CRAS and section 4 summarizes the paper.

2 CRAS: A Constant Rate Access Server

The current version of CRAS is implemented as a server on Real-Time Mach. The following design goals were considered in the design of CRAS:

1. It should be compact and simple, and it should not require changing the Unix file system format.

2. It should retrieve multiple continuous media streams at constant rates.

3. It should provide a data transmission mechanism between CRAS and applications that support dynamic QOS (Quality Of Service) control.

4. It should provide a high-level application interface for continuous media applications.

5. It should be usable by various types of applications[1] beyond those that continuous media storage systems can support.

In this section, we describe how CRAS has met these design goals.

2.1 Overview of CRAS

Figure 1 shows our typical playback application on Real-Time Mach. The application accesses CRAS for constant rate data retrieval; this is the only functionality supported by CRAS.

The implementation of 'Fast Forward', 'Fast Rewind', 'Step by Frame', and similar operations uses the Unix file system because these operations do not require critical real-time response. CRAS and the Unix file system can share files because they use the same disk layout. This approach makes CRAS simple and allows it to be highly optimized for a single function: constant rate data stream retrieval. CRAS does not support retrieving data at a rate faster than the rate at which astream was recorded since such a function is not necessary for most of applications.

One problem of the disk layout policy of the Unix file system is that the allocation of blocks in a single file may be random, making it difficult to provide a constant rate stream retrieval with high throughput. We minimized this problem by using the 'tunefs' command at file system creation time to indicate that

[1] In our research, we focus on building personal interactive continuous media applications.

blocks should be allocated as contiguously as possible [2].

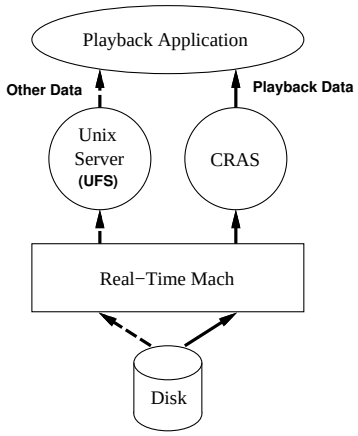

Figure 1: Architecture of a Playback Application on Real-Time Mach

A serious problem for continuous media servers is that they may trigger page faults that block threads in the servers, and interrupting a constant rate stream retrieval in progress. Our approach is to make the server small enough[3] so it can wire down all memory allocated for the server without severely impacting the amount of memory available to other applications. Also, the server is carefully designed to avoid accessing any non real-time OS servers during constant rate retrieval, so the priority inversion problem can be prevented from occurring[7].

Figure 2 shows an overview of how CRAS works. An application sends a request to CRAS to open a continuous media session, and retrieve media data from a file. CRAS performs an admission test to determine whether sufficient capacity is available to perform the requested retrieval. If the admission test fails, the application needs to wait for the completion of currently running retrievals. After the admission test passes, CRAS creates a shared memory between CRAS and the application for delivering media data from CRAS to the application(1).

CRAS schedules read requests of all admitted sessions periodically, and sends them to the kernel's device driver to start the disk operations(2). Then, the driver starts the operations(3). When completion interrupts are received by the device driver(4), CRAS is notified. Then, CRAS puts the media data retrieved from disk into the shared buffer between the applications and CRAS(5).

[2] Some recent Unix Fast File Systems adopt the same policy to improve sequential read throughput of files.

[3] CRAS consumes about (250KB + total buffer space) of physical memory.

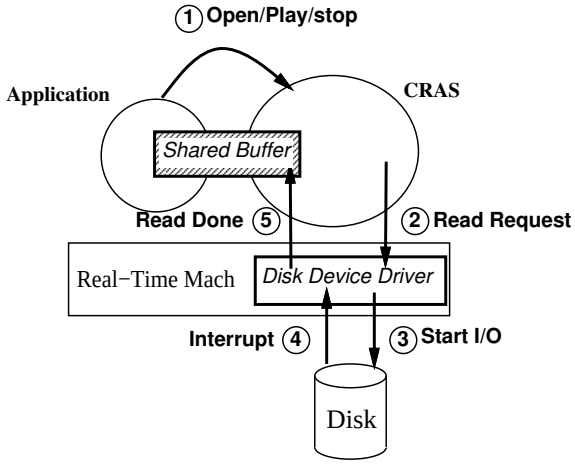

Figure 2: Interaction between an Application and CRAS

In Section 2.2, we describe a scheduling policy for guaranteeing constant rate retrievals. Section 2.3 presents an admission test used in CRAS. In Section 2.4, we describe the shared memory communication mechanism between CRAS and applications. Section 2.5 describes the application interface of CRAS. Finally, we describe how implementing CRAS on a microkernel enables it to be used in a variety of system configurations.

2.2 Retrieving Continuous Media at a Constant Rate

Continuous media data are usually accessed sequentially and it is easy to predict which data regions will be accessed in the future. In traditional storage systems, a read-ahead policy is used for improving sequential access performance, but it does not ensure that all data will arrive in memory by the time that it is needed. CRAS schedules pre-fetches of all data which are required in the near future and ensures that all required data will be fetched by the time it is needed.

CRAS periodically schedules pre-fetches of media data which are needed in the next period, and ensures that these data are fetched and placed in memory before the end of the current period. We call this period of CRAS its *interval time*. The interval time is determined by a tradeoff between the maximum number of streams supported by CRAS and the initial delay of the output streams. CRAS calculates the data sizes which are required in order to read media data within the interval time for the output streams. For example, CRAS retrieves all video frames that an application requires to play back within the interval time in a shared buffer, and the application fetches each video frame

from the shared buffer at the video stream's frame rate. CRAS optimizes throughput by reading continuous media data from disks up to 256K bytes at a time when the data is stored contiguously and by making all the read requests to disks in cylinder order to minimize the seek time. If the size of contiguous blocks is less than 256K bytes, CRAS reads the smaller blocks instead of reading a big block, decreasing the overall throughput.

Consider a video stream that was recorded at 30 fps. If an application wants to play back the video stream at 60 fps(Fast Forward), CRAS needs to retrieve all the video frames at twice the normal speed since CRAS cannot skip video frames during the retrieval. If the application will skip video frames during the retrieval and the retrieval does not require timeliness, the Unix file system should be used. As described in Section 2.4, an application can skip any video frames without disturbing the retrieval of CRAS although CRAS retrieves all video frames into the shared buffer.

Figure 3 shows the structure of CRAS. CRAS includes five threads: a request manager thread, a request scheduler thread, a deadline manager thread, an I/O done manager thread, and a signal handler thread. The request manager thread accepts open/close requests from applications and calculates the buffer size that must be read for each active stream within the interval time. The request scheduler thread sends read requests to the kernel for all active streams. At the first phase of each interval, the request scheduler thread gets all the data that has been retrieved in the previous interval from the I/O done queue. The request scheduler then puts this data and its associated timestamp in the shared memory buffer which is used to pass data to applications. This thread also requests all necessary disk reads for the next interval from the kernel. Now, the I/O done manager thread again accepts I/O done notifications from the kernel, and puts the media data into the I/O done queue. The deadline manager thread manages the deadline of the request scheduler thread and executes the recovery action from a missed deadline. Currently, CRAS notifies a warning message when a deadline is messed.

We changed two parts of Real-Time Mach to support CRAS. The first modification was to the queue in the disk driver. We divided the queue into two queues, one for normal activities, and another for real-time activities. CRAS uses the real-time queue and the Unix file system uses the non real-time queue. If there are any requests in the real-time queue, the requests are processed before the request in the non real-time queue. Each queue is sorted by using the traditional C-SCAN algorithm[4] to minimize total seek time. The second modification we made is add a new interface for reading larger data blocks from the kernel efficiently. Existing primitives for reading data from devices allocate buffers in kernel, making it difficult to manage buffers in CRAS explicitly. Thus, the new interface enables us to specify buffers that are managed in CRAS explicitly.

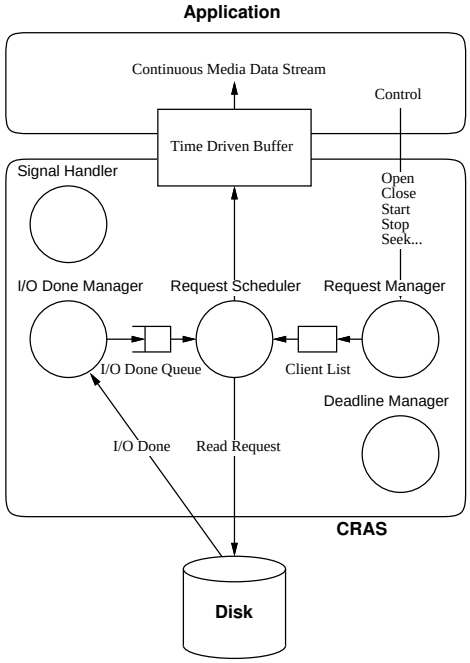

Figure 3: Structure of CRAS

2.3 Admission Test

When playing back continuous media, a host must not be interrupted by a resource shortage. Admission control is necessary to decide whether the real time requirements of a new continuous media stream can be satisfied before playback starts.

The admission test adopted in CRAS is based on estimating the time of data transfer. First, an application sends the data rate of a continuous media stream to CRAS when opening the file. Next, CRAS calculates the time for retrieving data [formula (1)] and the total size of buffer memory [formula (2)] that are required to handle the new stream based on the parameters of the disk (see Table 1) and the data rate of the new stream[5]. The data retrieval time is calculated as sum of two parts, data transfer time and overhead time. Overhead is sum of the time used by

[4] The disk arm moves unidirectionally across the disk surface toward the inner track. When there are no more requests for service ahead of the arm it jumps back to service the request nearest the outer track and proceeds inward again[2].

[5] Appendix B describes the derivation of these formulas.

other disk access activities, head seek time, rotational delay and command overhead. Appendix C describes total overhead time, O_{total}, in more detail.

T	Interval time of CRAS
B_{total}	Total buffer memory size
D	Disk data transfer rate
O_{total}	Total access overhead
C_{total}	Total size of chunks
R_{total}	Total data rate

Table 1: Parameters for Admission Test

$$T \geq \frac{O_{total} \times D + C_{total}}{D - R_{total}} \quad (1)$$

$$B_{total} \geq 2 \times (T \times R_{total} + C_{total}) \quad (2)$$

If the time for retrieving data is less than the interval time and the total size of buffer memory (including the buffer for the new stream) is less than the total memory size allocated for CRAS, a new stream is opened successfully and CRAS allocates buffers and starts to pre-fetch the stream. Otherwise, the admission test fails and an error indicating a resource shortage is returned to the application.

2.4 Time-Driven Shared Memory Buffer

There is a problem when using traditional FIFO buffers for communicating between client applications and the continuous media server. Since CRAS delivers data to buffers at a constant rate, when applications cannot fetch data from the buffers at the same rate, the buffers may overflow. For this situation, FIFO buffers have the undesireable logical property of discarding incoming new data before obsolete old data in the buffers.

Our system uses a logical clock per stream to control retrieval; this clock is distinct from the system clock. The speed of a stream determines the rate of advance of the associated logical clock. At the time when a stream is opened, the logical clock is set to zero, and its rate of advance is set to the original recording data rate of the stream. CRAS schedules pre-fetches according to the logical rate, and clients access media data using a logical clock.

The shared memory between CRAS and applications described in the previous section is called a *time-driven shared memory buffer*. This buffer enables clients to access media data without explicit communication with CRAS. When a new stream is opened, a new shared memory area is set up for the stream.

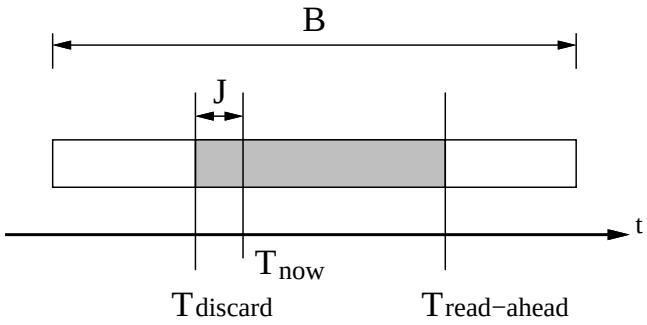

Figure 4: Time-Driven Shared Memory Buffer

CRAS puts the media data with its timestamp in the buffer. CRAS removes the media data automatically when the timestamp becomes greater than the logical clock's current time. Thus, the buffer always has enough space for storing media data retrieved from disks.

The merit of this approach is that it allows a client to change the rate at which it processes a media stream without changing the rate of retrieval in CRAS or stopping the current stream. The scheme is especially useful for building an application supporting dynamic QOS control[8]. The mechanism is difficult to implement using a traditional FIFO buffer, because new data is dropped when the buffer is full. *Time-driven shared memory buffer* solves the problem by discarding obsolete data from the buffer automatically based on timestamps of the data.

Figure 4 shows the structure of the time-driven shared memory buffer. In this figure, B stands for the total size of the buffers, and J is a short time which allows small jitters to occur. $T_{discard}$ defines the time when all data whose timestamp is less than this time to be discarded. T_{now} is the current time, and $T_{read-ahead}$ is the time when the next pre-fetches of media data will start. Here, $T_{discard}$ is defined as $T_{now} - J$. Clients read the data at the location pointed to by T_{now}. Time-driven shared memory buffer is attractive when two periodic threads are executing at different rates. Our scheme solves the problem by decoupling each period of a thread from other threads processing the same stream by using a time-driven shared memory buffer. The solution provides a better support for dynamic QOS control.

Consider how the time-driven shared memory buffer works when the rate of an application is different from the rate of a media stream. Let us assume a video stream with 30 fps, and the interval of CRAS is 1 second. CRAS retrieves 30 video frames for each 1 sec interval from a disk, and puts them into the shared

crs_open	Open a new continuous media stream
crs_close	Close a continuous media stream
crs_start	Start the logical clock of a continuous media stream
crs_stop	Stop the logical clock of a continuous media stream
crs_seek	Set the logical clock to the specified value
crs_get	Get the address of data chunk in the time-driven shared memory buffer specified by logical time

Table 2: Application interface of CRAS

buffer. Now, an application wants to play back the video stream at 10 fps. The application fetches video frames whose timestamps are equal to the logical clock that is updated every 100 ms. This means that the application only uses one of every three frames from the shared buffer. Then, the video frames are automatically removed from the buffer when their timestamps become greater than the logical clock. The scheme does not require a complex feedback mechanism when the rate of an application and a media stream are different.

2.5 Application Interface

The application interface of CRAS is different from conventional storage systems due to CRAS's timing-dependency. To ensure a constant rate stream retrieval, CRAS pre-fetches data blocks without explicit requests from applications. This requires a high-level application interface that is suitable for building continuous media applications, rather than a traditional file system interface.

CRAS does not provide an explicit read request for applications. In a traditional file system interface, pre-fetching cannot be controlled from users. However, pre-fetching is the most important mechanism for maintaining constant rate stream retrieval. Hence, the interface should support primitives to initiate pre-fetching to start a stream, and to stop pre-fetching in order to to suspend the stream. A traditional interface does not offer such primitives so it is not suitable for continuous media storage systems.

The application interface of CRAS is shown in Table 2. In contrast to a conventional interface that retrieves data by specifying a byte offset in a file, the CRAS interface retrieves a chunk of data specified by a logical clock value. Since the interface is based on the media data's logical clock, an application program can fetch the necessary data blocks according to its actual requirements.

In Table 2, *crs_open*, *crs_close*, *crs_start*, *crs_stop* and *crs_seek* are requests to CRAS. Unlike these, *crs_get* does not communicate with CRAS, because an application can get the data from its time-driven shared memory buffer. *Crs_open* is used to open a new continuous media stream, and *crs_close* closes the stream. *Crs_start* is used to start pre-fetching, and continuous media data is put at a constant rate into a time-driven shared memory buffer. *Crs_stop* stops the pre-fetching.

When an application opens a new continuous media stream by using *crs_open*, the application sends information about the timestamp, duration and size of each chunk to CRAS. The information is used for scheduling pre-fetches of media data and discarding obsolete media data in CRAS. Usually, this timing information is stored in a control file separate from the continuous media data file or is calculated by the client application. The timestamp of each block, which is used for reading data from applications and for discarding obsolete data, is calculated from the sum of the durations of all previous media blocks.

2.6 Customization of CRAS

Since it is difficult to implement a specialized storage systems for each new continuous media application, we have designed CRAS to be used by various types of applications ranging from a large-scale video-on-demand system to a small scale independent server accessed by several personal applications on Unix. User-level implementation of a continuous media storage system allows us to customize the system easily, and allows the system to execute multiple CRAS's simultaneously.

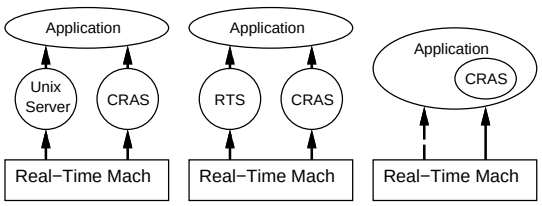

Figure 5: Customization of CRAS

Figure 5 shows several configuration of CRAS. The left configuration is the typical configuration where

a Unix Server being used. The middle configuration shows CRAS being used with RTS, a small operating system server specialized for embedded systems on Real-Time Mach[6]. The right configuration is more aggressive, because CRAS is linked with an application. This configuration is especially useful for supporting continuous media in embedded systems.

3 Evaluation and Application Experience

This section describes the evaluation of CRAS, and our experiences with using CRAS in a QuickTime player.

3.1 Evaluation of CRAS

The results in this section show the advantages of CRAS over the Unix file system and the importance of the real-time scheduling techniques in CRAS. We evaluated the basic performance of CRAS on a Gateway2000 P5-100 with a 100MHz Intel Pentium processor, 32MB of memory, 2GB of SCSI disk(Seagate ST32550N) whose data transfer bandwidth is about 6.5M bytes per second(Table 4 in Appendix A shows the actual disk parameters of the disk.), and a 10 Mbps Ethernet interface. We used a timer board which contains an AM9513 counter/timer module for measurements with accuracy to the nearest 1 micro second[6].

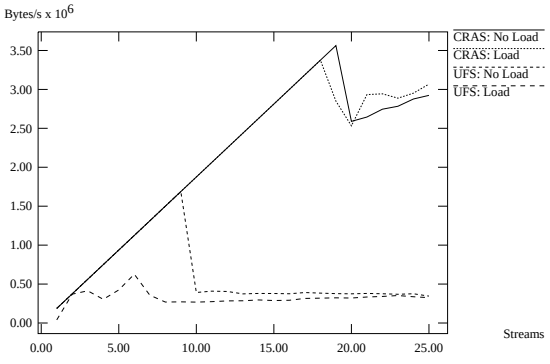

Figure 6: CRAS vs. UFS Throughput

Figure 6 shows the results of the first benchmark, demonstrating the advantages of CRAS over the Unix file system. In this benchmark, the data rate of each stream is 1.5Mbps[7], and either CRAS or Unix file system is used to read several streams simultaneously. The benchmark is evaluated both with no disk I/O activity and with other disk I/O activity[8]. The number of streams is varied from 1 to 25, and the actual throughput that CRAS and the Unix file system can achieve is measured in each case. In this benchmark, the interval time of CRAS was 0.5 second, and the initial delay time of each stream was 1 second. The results show that CRAS can support more streams, and provide a higher throughput than the Unix file system can achieve, because the Unix file system may cause priority inversions.

The result also shows that CRAS can achieve 55% of the disk's maximum transfer rate, and that background file access activities do not affect the throughput of CRAS. If a longer initial delay is allowed, CRAS can support more streams or higher data rates. For example, with 3 seconds initial delay, it can support more than 25 MPEG1 streams whose total throughput is 4.6MB/s(70% of disk bandwidth).

On the other hand, the Unix file system provides up to nine streams without other disk I/O traffic. Moreover, it cannot support even one stream when other disk I/O traffic is present.

Figure 7 is the result of the second benchmark, showing another advantage of CRAS over the Unix file system. In this benchmark, an application retrieves a video stream from each of the file systems, and we measured each frame's delay (the difference between current time and logical time) while running other activities that access the same disk. The result shows that the Unix file system causes larger delay jitters of video frames than CRAS even when both file systems achieve the same throughput.

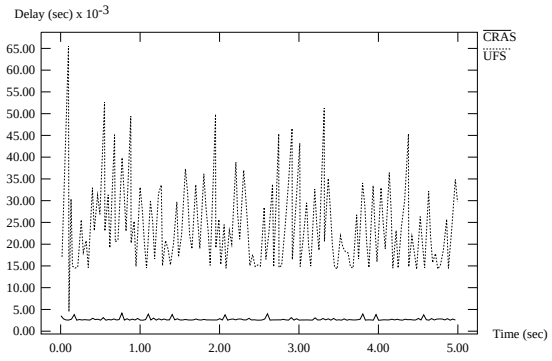

Figure 7: CRAS vs. UFS Delay

Figure 8 and 9 show the accuracy of our admission test. In the benchmarks, we employ two data

[6] Real-Time Mach supports high resolution timers using this timer board.

[7] This rate corresponds to a MPEG1 data stream.

[8] We executed two 'cat' programs which read movie files with the benchmark program. The priority of the benchmark program is higher than the priorities of 'cat' programs.

rates 1.5Mbps and 6Mbps[9], and we vary the number of streams from 1 to 20 for 1.5Mbps streams and from 1 to 5 for 6Mbps streams. Then, we measured the average and maximum ratio of the actual disk I/O time to the calculated I/O time, where 100% means that the CRAS's estimation of disk I/O time is perfect, and a lower ratio means that the estimation is more pessimistic. Further, we ran some disk I/O intensive applications together with CRAS. In Figure 8 and 9, "no_load" indicates that only CRAS was running, and "load" indicates that CRAS and these applications were simultaneously running.

The results show that the estimations of the admission test are very pessimistic when there are few streams, especially when the data rates of streams are low. This is because that the admission tests uses worst case seek time and rotational delay time to estimate total disk access time. If there are few streams or the data rates of streams are low, the time which takes to transfer data is short and overhead time dominates the calculated cost. On the other hand, the cases of 6Mbps streams with background activities yield an accuracy to about 70%. These results show that our admission control is over estimated when the data rate of each stream is low or small number of sessions are opened. If the data rate of a session is increased, the data transfer time will dominate the more pessimistic overhead estimates lessening the pessimism of our admission test.

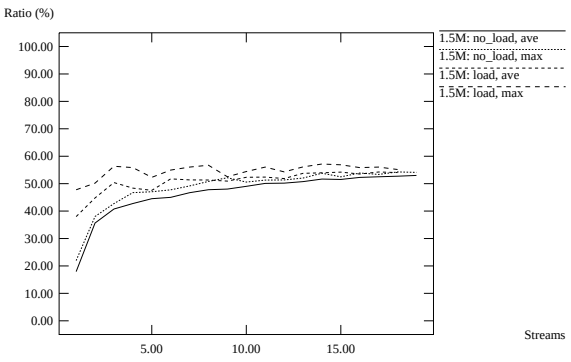

Figure 8: Accuracy of Admission Test (1): 1.5Mbps

Figure 10 shows the importance of real-time scheduling techniques to guarantee constant rate stream retrieval. In this benchmark, one stream of 1.5Mbps was retrieved from CRAS while other tasks that consume CPU time were also running. We measured each frame's delay under both fixed priority scheduling and round-robin scheduling. Under round-robin scheduling, delay jitters of retrieved data are much larger than

[9] The data rate corresponds to MPEG2 data stream.

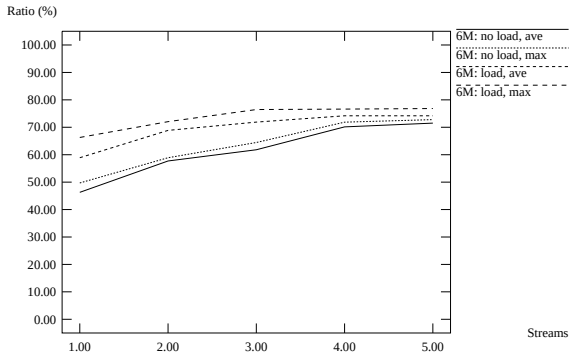

Figure 9: Accuracy of Admission Test (2): 6Mbps

under fixed priority scheduling. This result shows that real-time scheduling is very important to retrieve continuous media data at a constant rate.

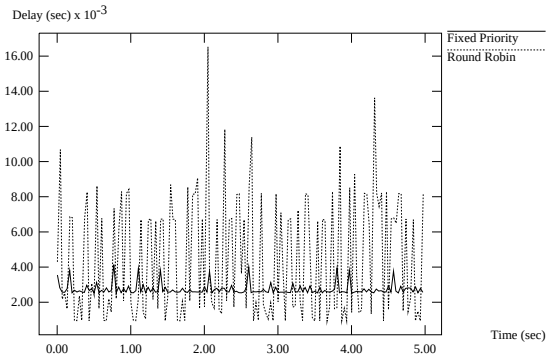

Figure 10: Effect of Real-Time scheduling

3.2 Experience and Discussion

We implemented a distributed QuickTime movie player, QtPlay[8, 16] which retrieves movie data from disks using CRAS and transmits it over the network using NPS[9]. QtPlay sends video streams to the X11 server and audio streams to an audio server for playback, and it can play multiple movies simultaneously. Our experience with implementing QtPlay using CRAS shows that CRAS's high level application interface is suitable for handling continuous media, and makes it easy to construct continuous media applications. This application demonstrates that CRAS can support constant rate stream retrievals even though it is very compact and simple, and that it is particularly suitable for playback applications.

Our QuickTime player can change the frame rate of

a movie at any time without notifying CRAS because CRAS's time-driven shared buffer enables applications to support this flexibility. This ability is very attractive for personal or small scale distributed environments because applications cannot be executed at a full quality due to the limited hardware capabilities of the machines used in these environments.

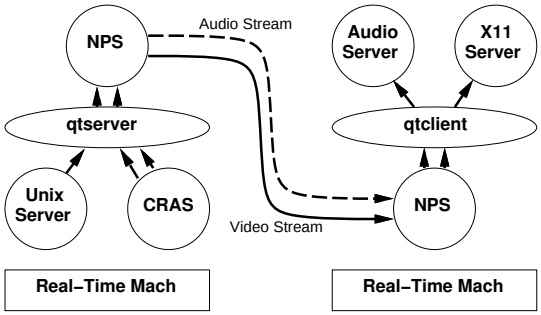

Figure 11: QuickTime Player on Real-Time Mach

We found three problems with CRAS in the personal environment. First, the sizes of video data compressed by JPEG or MPEG varies significantly. In this case, the rate of a stream is not constant. CRAS allocates buffers for retrieving within each interval time based on worst case bandwidth. If the average bandwidth is much less than the worst case bandwidth, much of the buffer space may not be used.

Our admission test algorithm causes the second problem. The result of the previous section shows that our admission test is more pessimistic when it is used by one or two sessions with low data rates. However, the rest of the throughput may be used by non real-time disk accesses.

Lastly, we adopt the Unix file system's disk layout policy with the changed parameter for allocating blocks contiguously as much as possible by 'tunefs' command. However, editing a continuous media file may make the layout of blocks random. Non-continuous data makes the seek time long, and the throughput of the disk is decreased. For example, adding a block in the middle of two contiguous blocks prevents CRAS from retrieving media data at a constant rate. Our approach needs to rearrange media files whose data blocks are allocated randomly.

4 Conclusions

In this paper, we presented the design and implementation of a continuous media storage server, CRAS, and showed that CRAS can retrieve media data without violating the timing constraints even though it is compact and simple.

CRAS provides mechanisms to support continuous media including a constant rate stream retrieval, an admission control mechanism, a high-level application interface, and a stream transmission mechanism for applications supporting dynamic QOS control. Also, CRAS demonstrates that a microkernel-based operating system is suitable for meeting the requirements of various types of applications in continuous media storage systems.

Although the current version of CRAS has no capability for writing continuous media files at constant rates, it is easy to add it. To limit the size of these modifications, the Unix file system must be modified to allocate data blocks in advance when a file is created or expanded. CRAS can then write continuous media data at constant rates to the allocated blocks via the same algorithm used for retrieving continuous media data.

Acknowledgements

We would like to thank the members of the MMMC Project in JAIST for their valuable comments and inputs to the development of Real-Time Mach and also for integrating Lites with Real-Time Mach. We are also grateful to David Black, the Usenix shepherd for our paper, who has provided many helpful comments.

References

[1] D.A.Anderson, et. al. "A File System for Continuous Media", ACM Transaction on Computer Systems, Vol.10, No.4, 1992.

[2] H.M.Deitel, "An Introduction to Operating Systems", Addison-Wesley, 1984.

[3] J.Helander, "Unix under Mach: The Lites Server", Helsinki University of Technology, Master's Thesis, 1994.

[4] E.M.Hoffer, et. al., "QuickTime: An Extensible Standard for Digital Multimedia", *In Proceedings of IEEE COMPCON, 92*, 1992.

[5] P.Lougher, D.Shepherd, "The Design of a Storage Server for Continuous Media", The Computer Journal, Vol.36, No.1, 1992, pp.32-42.

[6] T.Nakajima, T.Kitayama and H.Tokuda, "Experiments with Real-Time Servers in Real-Time Mach", *In Proceedings of the USENIX 3rd Mach Symposium*, 1993.

[7] T.Nakajima, T.Kitayama, H.Arakawa, and H.Tokuda, "Integrated Management of Priority Inversion in Real-Time Mach", *In Proceedings of the Real-Time System Symposium*, 1993.

[8] T.Nakajima and H.Tezuka, "A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach", *In Proceedings of the ACM Multimedia '94*, 1994.

[9] T.Nakajima, "NPS:User-Level Real-Time Network Engine on Real-Time Mach", In Proceedings of First International Workshop on Real-Time Computing System and Applications, 1994.

[10] T.Nakajima and H.Tokuda, "Design and Implementation of a User-Level Real-Time Network Engine", IS-RR-94-14S, Research Report, Japan Advanced Institute of Science and Technology, 1994.

[11] S.Ramanathan, H.M. Vin, and P.V.Rangan, "Towards Personalized Multimedia Dial-Up Services", Computer Networks and ISDN Systems, 1994.

[12] K.K.Ramakrishnan, L.Vaitzblit, C.Gray, U.Vahalia, D.Ting, P.Tzelnic, S.Glaser, W.Duso, "Operating System Support for a Video-on-Demand File Service", Multimedia Systems, Vol.3, No.2, Springer-Verlag, 1995.

[13] P.V. Rangan, Harrick M.Vin, "Efficient Storage Techniques for Digital Continuous Multimedia", IEEE Transactions on Knowledge and Data Engineering, August, 1993.

[14] L.A.Rowe and B.C.Smith, "A Continuous Media Player", *In Proceedings of Third International Workshop on Network and Operating System Support for Digital Audio and Video*, November, 1992.

[15] C.Ruemmler and J.Wilkes, "An Introduction to Disk Drive Modelling", IEEE Computer vol.27, num.3, pp.17-28, Mach 1994.

[16] H.Tezuka and T.Nakajima, "Experiences with building a Continuous Media Application on Real-Time Mach", In Proceedings of Second International Workshop on Real-Time Computing Systems and Applications, 1995.

[17] H.Tokuda, T.Nakajima, and P.Rao, "Real-Time Mach: Towards a Predictable Real-Time System", *In Proceeding of the USENIX 1st Mach Symposium*, October, 1990.

[18] Microsoft Windows, "Multimedia: Programmer's Workbook", Microsoft Press, 1991.

[19] H.M.Vin, P.V.Rangan, "Designing a Multi-User HDTV Storage Server", IEEE Journal on Selected Areas in Communication, Vol.11, No.1, January, 1993.

A Disk Parameters

Parameters used in the admission test of CRAS are summarized in Table 3. In the table, T_{rot} is calculated by a disk's spindle rotation speed, and D, T_{seek_max}, T_{seek_min} and T_{cmd} are measured using small benchmark programs.

N	Number of continuous media stream
T	Interval time of CRAS
B_{total}	Total buffer memory size
D	Disk data transfer rate
T_{seek_max}	Maximum head seek time
T_{seek_min}	Minimum head seek time
T_{rot}	Disk rotational latency
T_{cmd}	Disk command overhead
B_{other}	Maximum block size of other disk traffic
O_{total}	Total access overhead
C_{total}	Total size of chunks
R_{total}	Total data rate
O_{other}	Delay time by other disk traffics
O_{seek}	Total head seek time
O_{rot}	Total disk rotational delay
O_{cmd}	Total disk command overhead

Table 3: Parameters for Admission Test

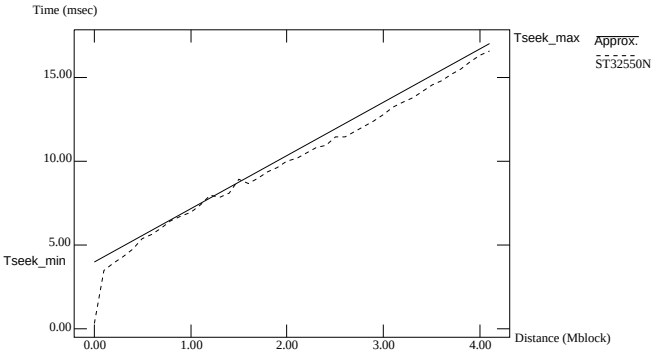

Figure 12: Disk Seek Time

Figure 12 shows the measured seek time of the disk that we used. T_{seek_max} and T_{seek_min} are obtained by approximating the measured results linearly. Table 4 shows the measured parameters of the disk.

B Admission Test of CRAS

In this section, we describe how to derive formulas (1) and (2) presented in Section 2.3.

D	6.5MB/s
T_{seek_max}	17ms
T_{seek_min}	4ms
T_{rot}	8.33ms
T_{cmd}	2ms
B_{other}	64KB

Table 4: Actual Disk Parameters of our system

We indicate the parameters of each data stream with suffix i. A_i denotes the amount of data that is retrieved for each stream within an interval time T. A_i is derived as follows.

$$A_i \geq T \times R_i + C_i \tag{3}$$

Also, interval time T should satisfy the following formula where $O_i + A_i/D$ indicates time which requires to retrieve media data for session i within the interval.

$$T \geq \sum_{i=1}^{N} (O_i + \frac{A_i}{D}) \tag{4}$$

Substituting formula (3) into (4), the interval time T of CRAS must satisfy the next formula.

$$T \geq \frac{O_{total} \times D + C_{total}}{D - R_{total}} \tag{5}$$

$$O_{total} = \sum_{i=1}^{N} O_i, \quad C_{total} = \sum_{i=1}^{N} C_i, \quad R_{total} = \sum_{i=1}^{N} R_i$$

Then, the total amount of data A_{total} which must be retrieved within interval time T, buffer size of each stream B_i and total amount of buffer B_{total} are calculated in the following way.

$$\begin{aligned} A_{total} &= \sum_{i=1}^{N} A_i \\ &= T \times R_{total} + C_{total} \end{aligned} \tag{6}$$

$$\begin{aligned} B_i &= 2 \times A_i \\ &= 2 \times (T \times R_i + C_i) \end{aligned} \tag{7}$$

$$\begin{aligned} B_{total} &= \sum_{i=1}^{N} B_i \\ &= 2 \times (T \times R_{total} + C_{total}) \end{aligned} \tag{8}$$

C Calculating Disk Access Overhead

In this appendix, we describe the calculation of the disk access overheads that our admission test uses in details. The disk access overhead is calculated as the sum of an overhead of other disk access activities(C.1) a command overhead(C.2), a head seek time(C.3), and a disk rotational delay(C.4).

C.1 Overhead of Other Disk Access Activities: O_{other}

CRAS cannot access a disk when another activity is accessing the disk since its disk access operations are not interrupted. Thus, the time which is consumed by the activity is taken into account as an overhead that delays the disk accesses by CRAS. The worst case overhead is calculated as follows.

$$O_{other} = T_{cmd} + T_{seek_max} + T_{rot} + \frac{B_{other}}{D} \tag{9}$$

C.2 Command Overhead: O_{cmd}

Command overhead T_{cmd} is the time which is necessary to setup each disk access operation. When N reads are performed, total command overhead O_{cmd} is as follows.

$$O_{cmd} = N \times T_{cmd} \tag{10}$$

C.3 Head Seek Time: O_{seek}

Since the seek time is not proportional to cylinder distance[15], it is difficult to calculate the seek time from one cylinder to another cylinder. Thus, we use the linear approximation to estimate seek time T_{seek} in the following way.

$$\begin{aligned} T_{seek}(x) &= \alpha \times x + \beta \\ \alpha &= \frac{T_{seek_max} - T_{seek_min}}{N_{cyl}} \\ \beta &= T_{seek_min} \end{aligned}$$

In the above formula, x is a cylinder distance and N_{cyl} is the total number of cylinders of the disk.

Since CRAS sorts disk access requests in cylinder order, we obtain the following formula for estimating the total seek time to access N(≥ 2) streams, where $d_{i,j}$ indicates the cylinder distance of stream i and j.

$$O_{seek} = T_{seek_max} + \sum_{i=1}^{N-1} T_{seek}(d_{i,i+1})$$

$$= T_{seek_max} + \alpha \times d_{1,N} + (N-1) \times \beta$$

Assuming the worst case, $d_{1,N}$ is equal to N_{cyl}. We obtain O_{seek} as follows.

$$\begin{aligned} O_{seek}(1) &= T_{seek_max} && (11) \\ O_{seek}(N) &= T_{seek_max} + \alpha \times N_{cyl} + (N-1) \times \beta \\ &= 2 \times T_{seek_max} \\ &\quad + (N-2) \times T_{seek_min} && (12) \end{aligned}$$

In formula (12), T_{seek_max} indicates the worst case seek time for moving heads to the outer track. The seek is required to take into account non real-time disk accesses. $T_{seek_max} + (N-2) \times T_{seek_min}$ indicates the total seek time for processing all requests that are sorted in cylinder order within an interval.

C.4 Disk Rotational Delay: O_{rot}

A maximum rotational delay for accessing an arbitrary data block is equal to the disk rotation time T_{rot}. Thus, total disk rotational delay O_{rot} is calculated as follows.

$$O_{rot} = N \times T_{rot} \qquad (13)$$

C.5 Total Overhead: O_{total}

From formulas (9), (10), (11), (12) and (13), we can calculate the total overhead time O_{total} for accessing 1 stream in formula (14) and N ($\geq$2) streams in (15).

$$\begin{aligned} O_{total} &= O_{other} + O_{seek} + O_{rot} + O_{cmd} \\ O_{total}(1) &= \frac{B_{other}}{D} + 2 \times (T_{seek_max} + T_{rot} + T_{cmd}) && (14) \\ O_{total}(N) &= \frac{B_{other}}{D} + 3 \times T_{seek_max} + (N-2) \times T_{seek_min} \\ &\quad + (N+1) \times (T_{rot} + T_{cmd}) && (15) \end{aligned}$$