The Continuous Media File System

David P. Anderson – University of California, Berkeley
Yoshitomo Osawa – Sony Corporation
Ramesh Govindan – University of California, Berkeley

ABSTRACT

Handling digital audio and video data ("continuous media") in a general-purpose file system can lead to performance problems. File systems typically optimize overall average performance, while many audio/video applications need guaranteed worst-case performance. These guarantees cannot be provided by fast hardware alone; we must also consider the interrelated software issues of file layout on disk, disk scheduling, buffer space management, and admission control. The Continuous Media File System (CMFS) is a prototype file system that addresses these issues.

Introduction

Support for digital audio/video (continuous media, or CM) has emerged as an important area in computer system design. CM capabilities will greatly expand the role of computer systems and will enrich the user interfaces of existing applications. Much effort is being directed toward "integrating" CM, that is, handling CM data in the same hardware and software framework as other data. This approach provides advantages in flexibility and generality; however, it introduces performance problems because CM traffic contends for hardware resources (CPU, disk, network) with other traffic.

Hardware speedup alone cannot solve these performance problems. It is necessary to schedule resources or limit workload (or both) in a way that reflects the performance requirements of CM traffic. For example, applications might vary data rates (e.g., reducing video resolution or frame rate) in response to changing load conditions. Alternatively, the system might allow applications to "reserve" resource capacity; resources must then be scheduled accordingly. The ability to reserve capacity is especially important for file systems, since in general the data rate of stored CM data is fixed.

CMFS (Continuous Media File System) is an experimental disk storage system for integrated CM. CMFS has the following properties:

- Clients of CMFS can reserve capacity in the form of *sessions*, each of which sequentially reads from or writes to a file with a guaranteed data rate.
- Multiple sessions, perhaps with different data rates, can exist concurrently, sharing a single disk drive.
- Non-real-time traffic is handled concurrently. Thus CMFS can be used as a general-purpose file system that handles CM data as well.

To provide these capabilities, CMFS addresses several interrelated design issues: disk layout, admission control (acceptance or rejection of new sessions) and disk scheduling. CMFS does not address high-level issues such as security, naming and indexing, or document structuring; these are left to higher levels [9, 10]. CMFS simply provides the ability to source or sink byte streams to/from storage at guaranteed rates. CMFS is influenced by several previous CM file systems [1, 5, 8, 11]. However, CMFS is more general than these systems: it supports read and write sessions, variable-rate files, and multiple sessions with different rates. It also supports non-real-time traffic more effectively.

The Client Interface to CMFS

What exactly is meant by "guaranteed data rate"? It is neither feasible nor desirable that CMFS should deliver data at a completely uniform rate, one byte every *X* seconds. Ideally, the semantics of a CMFS session should accommodate variable-rate files, work-ahead, and client pause/resume. We have developed a semantics that handles these cases in a simple and uniform way. Our semantics also provides a duality between reading and writing, thus simplifying CMFS.

To precisely describe the semantics of a CMFS session, we need a model for how CMFS interacts with its clients. Our model is as follows. Each session has a main-memory FIFO buffer for data transfer between CMFS and the client. For a read session, CMFS appends data to the FIFO and the client removes data (blocking when the FIFO is empty). For a write session, the client appends data (blocking when the FIFO is full) and CMFS removes it. Performance guarantees are defined entirely in terms of data insertion in, and removal from, the FIFO.

Further details are intentionally left unspecified, since the model can be realized in various ways. For example, a CM-capable file system may run in the OS kernel or in a protected user-mode virtual address space (see Figure 1). It may communicate with clients via traps (system calls) or via RPC; the

FIFO may be a memory-mapped stream [6] or a kernel data structure accessed by `read()` system calls.

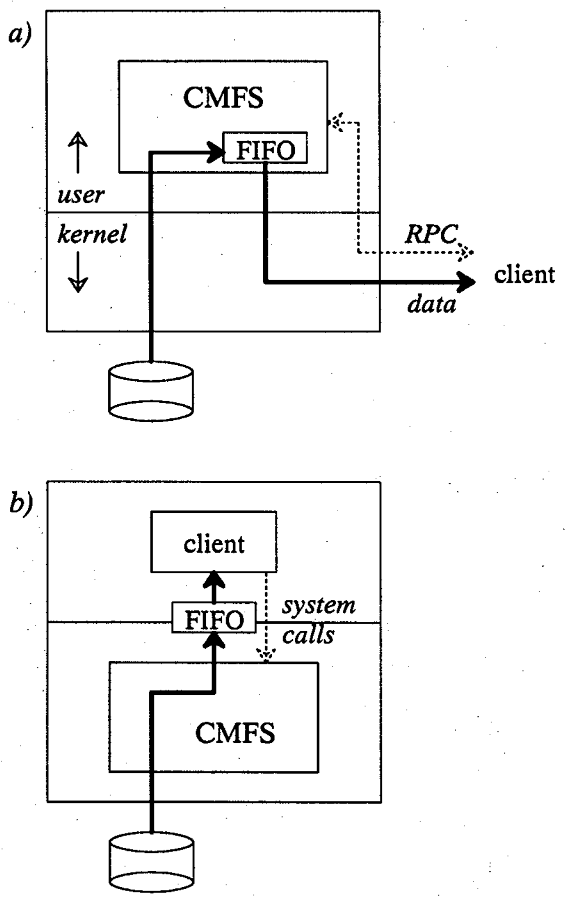

Figure 1: A CM-capable file system can run at the user level, communicating with its clients over network connections (a). Alternatively, it can be implemented in an OS kernel, with client data access by a shared-memory FIFO (b).

The CMFS prototype is implemented as a user-level UNIX process (Figure 1a), and clients communicate data via flow-controlled network connections. Data is removed from the FIFO of a read session whenever the corresponding network connection is ready to accept data.

The Semantics of a Session

The semantics of a CMFS session are defined in terms of a "logical clock" $C(t)$, the "get point" $G(t)$ (the start of the FIFO data) and the "put point" $P(t)$ (the end of the FIFO data). These are byte indices into the file; $C(t)$ is zero when the session starts ($t=0$). Each session has two parameters: R (its data rate) and $\bar{Y}$ (its "cushion"). Figure 2a depicts the semantics of a read session:

- The logical clock advances at rate R whenever it is less than the get point.
- The logical clock stops whenever it equals the get point.
- The put point is always at least $\bar{Y}$ ahead of the logical clock.

These rules imply that if the client removes one byte of data every $1/R$ seconds, it will never block; in other words, the client is guaranteed a data rate of R bytes/second. However, the flow of data need not be smooth or periodic. CMFS promises to stay ahead of the logical clock by a given positive amount (the cushion $\bar{Y}$), and the client's behavior determines how the clock advances. These semantics allow CMFS to handle variable-rate files and other non-uniform access in a simple way. CMFS is guided by client behavior; no explicit rate-control calls are needed, and CMFS need not know about file internals.

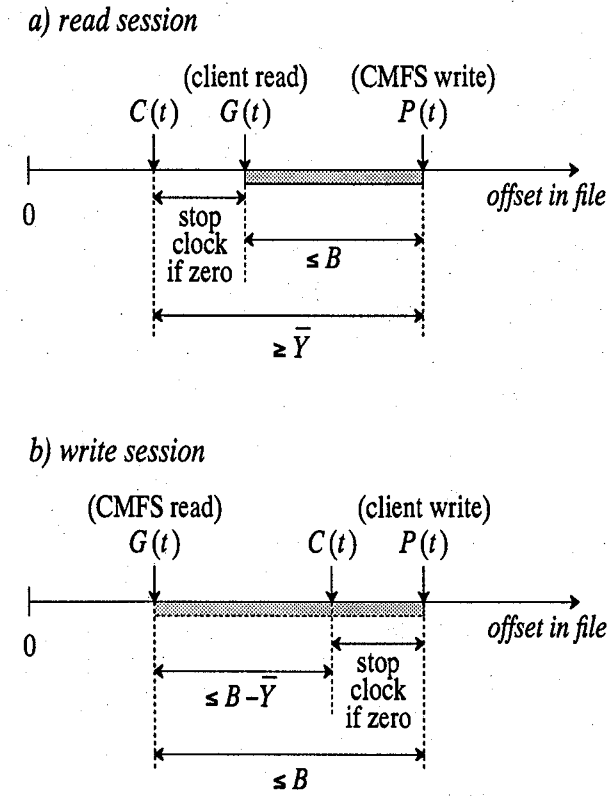

Figure 2: The semantics of read and write sessions are described in terms of a "put point" $P(t)$, a "get point" $G(t)$, and a logical clock $C(t)$. The shaded rectangles represent data in the FIFO.

The semantics of a write session are as follows (see Figure 2b):

- The logical clock advances at rate R whenever it is less than the put point.
- The logical clock stops whenever it equals the put point.
- The get point trails the logical clock by at most $B - \bar{Y}$ bytes, where B is the size of the FIFO.

The Duality of Reading and Writing

In describing the algorithms used by a real-time file system, it is tedious to have to deal with the read and write cases separately; they are similar but they differ in some crucial respects. In CMFS this problem disappears because reading and writing are dual in the following sense. Given a write session, consider a read session for the same file; the FIFO of the read session is identical to the write session, but with "empty" and "full" interchanged (see Figure 3). As shown in [4], the dual read session obeys the rules for read sessions (described above) if and only if the original write session obeys its corresponding rules. (In fact, the semantics of sessions were designed to make this hold.)

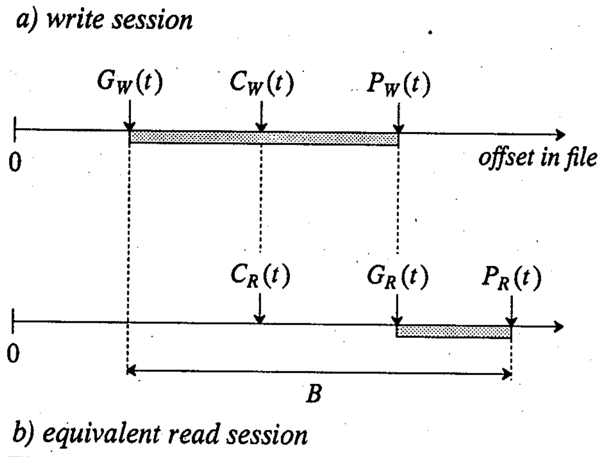

Figure 3: By interchanging empty/full and read/write, a write session (a) is transformed into a read session (b) that is equivalent with respect to scheduling.

Thus, from the point of view of scheduling in CMFS, reading and writing are essentially equivalent. The main difference is the initial condition: an empty buffer for a write session corresponds to a full buffer for a read session. In describing CMFS's algorithms for scheduling and session acceptance, we will refer only to read sessions.

Using CMFS Semantics

The CMFS interface provides an implicit rate control: the client can stop the logical clock by simply not removing data from the FIFO. This control mechanism can serve several purposes:

- CMFS adapts automatically to files that have variable data rate (e.g., variable-rate compressed video) or instantaneous "chunks" such as video frames (this is formalized in [4]).
- A client can pause a session by stopping the removal of data from the FIFO. Performance guarantees will remain valid after the client resumes reading.
- The initial pause that occurs when files from several CMFS servers are played synchronously at a single I/O server (such as ACME [3]) is handled automatically.
- If the hardware is fast enough the client can read arbitrarily far ahead of the logical clock. This "workahead" data can then be buffered (both within CMFS and at other points), protecting against playback glitches and improving the performance of other traffic (see [2]).

CMFS is intended to be part of a real-time distributed system in which each shared resource (CPU, disk, network) can be reserved in "sessions", each of which handles a data stream and has an upper bound on its delay. This "meta-scheduling" scheme is described elsewhere [2]; the connection with CMFS is that a session's "cushion" parameter Y should be at least as large as the delay bound of the resource (usually a CPU) handling the data after removing it from the FIFO. This ensures that the logical clock never stops accidentally due to delay in CPU processing.

The CMFS Control Interface

The operations (RPCs or system calls) to create and start a CMFS session have the following form:

```
ID request_session(
    int direction,
    FILE_ID name,
    int offset,
    FIFO* buffer,
    TIME cushion,
    int rate);
start_clock(ID id);
```

If `direction` is `READ`, `request_session()` requests a session in which the given file is read sequentially starting from the given offset. If the session cannot be accepted, an error code is returned. Otherwise, a session is established and its ID is returned. `Start_clock()` starts the session's logical clock. The client is notified (via an RPC or exception) when the end of the file has been reached. CMFS also provides a `seek()` operation that flushes data currently in the FIFO and repositions the read or write point.

A real-time file is created using

```
create_realtime_file(
    BOOLEAN expandable,
    int size,
    int max_rate);
```

`expandable` indicates whether the file can be dynamically expanded. If not, `size` gives its (fixed) size. `max_rate` is the maximum data rate (bytes per second) at which the file is to be read or written. CMFS rejects the creation request if it lacks disk space or if `max_rate` is too high.

Non-Real-Time Access

CMFS also supports non-real-time file access. There are two service classes: *interactive* and *background.* Interactive access is optimized for fast response, background for high throughput. The interface for non-real-time access is like that of a UNIX file system, except that the `open()` call specifies the service class. There are no performance guarantees for non-real-time operations.

Disk Layout and Performance Assumptions

To make performance guarantees, we need information about the speed of disk operations, which is determined largely by the disk layout. Many layout policies are possible, and there is no single best policy. For example, contiguous layout minimizes seek time within files, but it may cause external fragmentation, and it is difficult to extend existing files. So instead of adopting a particular policy, we just assume that the disk is read and written in *blocks* of fixed size (a multiple of the hardware sector size), and that the layout has "bounding functions" U_F and V_F:

- For a given file F, $U_F(n)$ is an upper bound on the time to read n logically contiguous blocks of F, independent of the position of the disk head and the starting block number to be read.
- $V_F(i, n)$ is an upper bound on the time needed to read the n blocks of file F starting at block i.

The functions should take into account sector interleaving, interrupt-handling latency, the CPU time used by CMFS itself, and features (such as track buffering) of the disk controller.

Our CMFS prototype uses contiguous allocation. The number of sectors per block is a fixed parameter. Ignoring CPU overhead and other factors, bounds functions for this policy are easy to derive (see [4]). Contiguous layout, however, is only feasible for read-only file systems or if disk space is abundant. For more flexibility, a variant of the 4.2BSD UNIX file system layout [7] could be used. A real-time file might consist of clusters of n contiguous blocks, with every sequence of k clusters constrained to a single cylinder group. n and k are per-file parameters; they are related to the file's `max_rate` parameter.

Admission Control

CMFS accepts a new session only if its data rate, together with the rates of existing sessions, can be guaranteed. One way to decide this is to see whether any static schedule (that cyclically reads fixed numbers of blocks of each session) satisfies the rate requirements of all session and fits in the available buffer space.

To be more precise, suppose that sessions $S_1 \cdots S_n$ read files $F_1 \cdots F_n$ at rates $R_1 \cdots R_n$. A *schedule* ϕ assigns to each S_i a positive integer M_i. CMFS *performs* a schedule by seeking to the next block of file F_i, reading M_i blocks of the file, and doing this for every session S_i. From the functions U and V we can get an upper bound $L(\phi)$ on the elapsed time of performing ϕ.

The data read in ϕ "sustains" S_i for $\frac{M_i A}{R_i}$ seconds, where A is the block size in bytes. $D(\phi)$, the period for which the data read in ϕ sustains all the sessions, is the minimum of these periods. If the data read in ϕ "lasts longer" than the worst-case time it takes to perform ϕ (i.e., if $L(\phi) < D(\phi)$), we call it a *workahead-augmenting schedule* (WAS).

If the amount of data read for each session in a schedule ϕ fits in the corresponding FIFO, we say that ϕ is *feasible*. It is easy to show the following (see [4]): **CMFS can safely accept a set of sessions if there is a feasible workahead-augmenting schedule.**

We now describe an algorithm to compute the *minimal feasible WAS* ϕ (the feasible WAS for which $L(\phi)$ is least). Clearly, a minimal feasible WAS exists if and only if a feasible WAS exists.

Suppose that sessions $S_1 \cdots S_n$ are given. Let D_i be the "duration" of one block of data for S_i, given by A/R_i. Let $\{t_0 < t_1 < \cdots\}$ be the set of numbers of the form kD_i for $k \geq 0$ and $i \geq 0$. Let I_i denote the interval $(t_i, t_{i+1}]$. Let ϕ_i denote the schedule $< \lceil t_i/D_1 \rceil \ \cdots \ \lceil t_i/D_n \rceil >$. Note that ϕ_{i+1} differs from ϕ_i by the addition of 1 block to all sessions whose block durations divide t_{i+1}; hence the sequence of ϕ_i is easy to compute.

The following algorithm computes the minimal WAS (the proof is in [4]):

(1) Let $\phi_0 = <1, \cdots, 1>$ (this is the minimal schedule for which $D(\phi) \in I_0$).

(2) If ϕ_i is infeasible (i.e., there is no allocation $< B_1 \cdots B_n>$ of buffer space to client FIFOs such that $M_i A + Y_i \leq B_i$ for all i) stop; there is no feasible WAS.

(3) If $L(\phi_i) \leq D(\phi_i)$ stop; ϕ_i is the minimal feasible WAS.

(4) Compute ϕ_{i+1} and go to (2).

Disk Scheduling

When a disk block I/O completes, CMFS decides which disk block to read or write next, and it issues the appropriate command (seek, read, or write) to the disk device driver. The algorithm for this decision (the *disk scheduling policy*) must prevent starvation of sessions, and it should handle non-real-time workload efficiently.

As with any disk-based file system, seek overhead is a dominant concern in CMFS scheduling. It is desirable to perform long (multi-block) operations. However, it is only possible to perform long operations if the system is far enough "ahead of schedule" so that no sessions will starve before the operation is complete. We use the term *slack time* to mean the maximum time that CMFS can defer doing any reads for sessions.

The slack time, denoted H, is computed as follows. Suppose that the minimal WAS ϕ takes worst-case time $L_1 + \cdots + L_n$, where $L_i = U_{F_i}(M_i)$. Let W_i denote the "workahead" for session i; that is, the temporal value of data in the FIFO minus the session's cushion Y. Assume sessions are numbered so that $W_1 \le W_2 \le \cdots W_n$ (this ordering maximizes slack time). If the operations in ϕ is performed immediately in this order, the workahead of session j will not fall below $H_j = W_j - \sum_{i=1}^{j} L(i)$. CMFS can therefore safely defer starting ϕ for a period of $H = \min_{i=1}^{n}(H_i)$.

CMFS factors the disk scheduling policy into three parts:

- A *non-real-time policy* decides whether a non-real-time operation can be initiated.
- If a non-real-time operation is not initiated, a *real-time policy* decides which session to work on.
- A *startup policy* is in effect when sessions have been accepted but not yet started.

We describe these sub-policies separately.

Real-Time Policies

We have implemented and studied several real-time policies (the choice of policy is a CMFS option):

- **The Static/Minimal policy** simply repeats the minimal WAS.
- The **Greedy policy** does the longest possible operation for S_1 (the session with smallest workahead). It reads blocks for S_1 for a period of $H + L_1$; i.e., it uses the entire slack time for workahead on S_1.
- The **Cyclical Plan policy** tries to divide slack time workahead among the sessions in a way that maximizes future slack time. The policy distributes workahead by identifying the "bottleneck session" (that for which H_i is smallest) and schedules an extra block for it. This is repeated until H is exhausted. The resulting schedule determines the number for blocks read for S_1; when this read completes, the procedure is repeated.

All policies skip to the next session when a buffer size limit is reached. If at some point all buffers are full, no operation is done; when a client subsequently removes sufficient data from a FIFO, the policy is restarted. In both the Greedy and Cyclical Plan policies, the least-workahead session S_1 is serviced immediately. Therefore the value of H used by these policies can be computed as the minimum of the slack times of all sessions except S_1, yielding **Aggressive** versions of the policies.

Non-Real-Time Policy

Recall that CMFS has two classes of non-real-time traffic: interactive and background. The goal of the non-real-time policy is to provide fast response for interactive traffic and high throughput for background traffic.

A non-real-time operation with worst-case latency L can safely be started if $L \le H$. However, doing so may make H close to zero. This will subsequently force CMFS to do short real-time operations (close to the minimal WAS), causing high seek overhead.

Instead, CMFS uses a *slack time hysteresis* policy for non-real-time workload. An interactive operation is started only if $H \ge H_{I1}$. Once H falls below H_{I1}, no further interactive operations are started until H exceeds H_{I2}. Similarly, background operations are done with hysteresis limits $[H_{B1}, H_{B2}]$. No background operation is started if an interactive operation is eligible to start.

If the hysteresis limits are set appropriately, this policy has two benefits: 1) it keeps slack time from becoming close to zero, and 2) it avoids the seek overhead of rapidly alternating between real-time and non-real-time operations.

Startup Policy

When CMFS accepts a session request, it must delay returning from the `request_session()` call until the system state is "safe" (i.e., slack is positive) with respect to the new WAS. CMFS uses the following policy during this startup phase.

Suppose sessions $S_1 \cdots S_n$ are currently active, and session S_{n+1} has been accepted but not yet started. Let ϕ_n and ϕ_{n+1} denote the minimal WASs for the sets $S_1 \cdots S_n$ and $S_1 \cdots S_{n+1}$ respectively. The scheduler enters "startup mode" during which its policies are changed as follows:

- Non-real-time operations are postponed.
- For scheduling purposes, slack time H is computed relative to ϕ_n.
- When $S_1 \cdots S_n$ have positive slack with respect to ϕ_{n+1}, a read for S_{n+1} (of the number of blocks given by ϕ_{n+1}) is started. When this read is completed, the state is safe for all $n+1$ sessions. The `request_session()` call for S_{n+1} is allowed to return, ϕ_{n+1} becomes the system's WAS, and the system leaves startup mode. This step is omitted for write sessions because the FIFO of the dual read session (as described earlier) is initially full.

CMFS Performance

The CMFS prototype is written in C++ and runs as a user-level process on SunOS 4.1. Depending on a compile-time flag, disk I/O is either simulated or real. The simulator keeps track of the disk head radial and rotational position, and models disk latencies realistically. Real I/O is done to a SCSI disk via the UNIX raw-disk interface.

Performance on UNIX

We tested a "real I/O" version of CMFS on a Sun 4/110 connected to a CDC Wren III via a SCSI interface. To obtain functions *U* and *V*, we measured the time of I/O operations. We found that the time to read a (512 byte) sector on the same track as the previously-read sector, and at a rotational distance of *n* sectors, varied from 6 msecs for $n=11$ to 21 msecs for $n=10$ (i.e., the disk spins 10 sectors before another read command is handled). The average time needed to read a sector at the same rotational position on an adjacent cylinder was 24 msecs, and on an extreme cylinder 54 msecs.

We then measured the CPU time used by CMFS. In a typical situation (1 MB buffer, two 1.4Mbps sessions, background traffic, Greedy policy) the average time per scheduling decision was 250 usecs. Cyclical Plan policy was slightly slower: 303 usecs average. The maximum times (impossible to measure precisely on UNIX) were in the range of 2 to 4 msecs.

Because of the high per-read overhead of UNIX, we used a block size of 35 sectors (1 track). We then defined functions *U* and *V* based on the measured times for UNIX I/O and CMFS execution. With 2 MB of buffer space, this version of CMFS accepts 39 64Kbps (telephone-rate) or 2 1.4Mbps (CD-rate) sessions. Starvation occurs about once per minute, perhaps due to UNIX CPU scheduling. With a workload consisting of 2 CD-rate sessions and random interactive arrivals at a rate of 5/sec., the mean response time is about 80 ms. With 2 CD-rate sessions, the throughput of background traffic is 1.424 Mbps. These results are in rough agreement with the simulator using the same system parameters.

Simulation Studies

Because of the scheduling vagaries of UNIX and the restriction to our available disk drives, the "real I/O" version of CMFS is not well-suited to performance studies. Instead, we did simulation-base studies using the CMFS simulation mode. Unless otherwise stated, the simulations use the Cyclical Plan policy and assume a disk with 11.8 Mbps transfer rate and 39 ms worst case seek time. Block size is 512 bytes.

Figure 4 shows the maximum number of concurrent sessions accepted by CMFS as a function of total buffer space, for data rates of 64 Kbps and 1.4 Mbps. Curves are given for three different disk types: 39 ms maximum seek time and 11.8 Mbps transfer rate (CDC Wren V), 35 ms maximum seek time and 8.6 Mbps transfer rate (CDC Wren III) and, 180 ms maximum seek time and 5.6 Mbps transfer rate (Sony 5.25" optical disk).

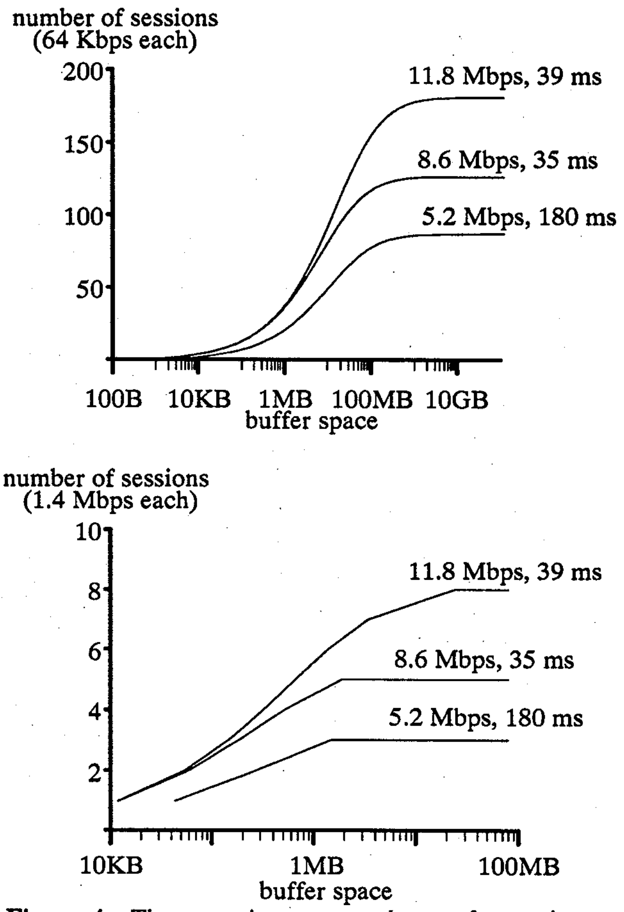

Figure 4: The maximum number of sessions depends on the available buffer space. The disk transfer rate imposes an upper limit on the number of sessions; to reach 90% of the limit with the 11.8 Mbps disk requires 4 MB of buffer space for 1.4 Mbps sessions and 85 MB for 64 Kbps sessions.

An important criterion for real-time scheduling policies is how quickly they increase slack. To study this, we simulated CMFS with three concurrent 1.4 Mbps sessions, no non-real-time traffic, and 8 MB buffer space. From the results (Figure 5) we see that Cyclical Plan performs slightly better than Greedy when slack is low, but that Greedy quickly catches up. Static/Minimal, because it cannot do long operations, performs much worse at higher slack levels. With appropriate hysteresis values CMFS maintains moderate to high slack levels during steady-state operation; thus the dynamic policies are preferable.

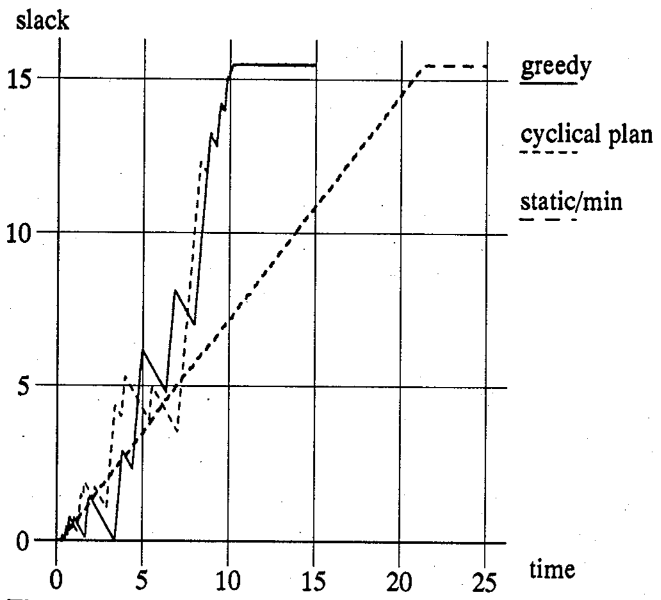

Figure 5: Disk scheduling policies build up slack at different rates. The Aggressive Cyclical Plan (solid line), Aggressive Greedy (dotted) and Static/Minimal (dashed) policies are shown here.

To study the performance of interactive non-real-time traffic, we ran simulations combining Poisson arrivals of interactive requests (each reading a random block) with several sessions. We then measured the average response time of the interactive requests. Details are given in [4]; the main results are:

- The effect of hysteresis parameters is greatest under heavy load; otherwise slack remains high and hysteresis is not exercised.
- Reasonable "rule of thumb" values for the hysteresis parameters are $H_{I1} = .3H_{max}$ and $H_{I2} - H_{I1} = \min(.3H_{max}, 0.5)$, where H_{max} is the upper limit on slack time imposed by buffer space.
- Response time decreases with increasing buffer space; in scenarios involving CD-rate sessions, the "knee" was around 1 MB.

To study the effect of real-time traffic on background traffic throughput, we simulated several sessions and a single background task that sequentially reads a long, contiguously-allocated file. We define the *background throughput fraction* T as the fraction of residual disk throughput (i.e., disk throughput not taken up by real-time sessions) used by the background task. We found that T was maximized for (roughly) $H_{B1} = .25H_{max}$ and $H_{B2} = .9H_{max}$. Again, increasing buffer space improves non-real-time performance; a buffer of at least 1 MB was needed to attain a value of T above 0.9.

Finally, we studied startup time for read sessions. We ran a simulation in which requests for six sessions 1.4 Mbps arrive at time zero. The first session starts in about 0.1 seconds. The gap between the start times of successive session then increases rapidly; the sixth session takes about 2.0 seconds to start. This is because the workaheads of existing sessions have to be increased to accommodate the new minimal WAS, which becomes longer as more sessions are added.

Conclusion

CMFS shows that it is possible for a file system to simultaneously handle multiple sessions with different data rate guarantees, together with non-real-time workload. This is an important step in the integration of audio/video in general-purpose computer systems. CMFS contributes new ideas in its acceptance test and scheduling policies, and also in the flexible but rigorous semantics of sessions.

Our performance experiments show that 1) for real-time traffic, dynamic scheduling policies (such as Greedy and Cyclical Plan) perform best; 2) significant buffer space is needed for the system to perform near the limits of the disk drive; 3) slack-time hysteresis limits can have a large effect on non-real-time performance.

There are many directions for further work in CM file systems: integration of low-level servers like CMFS with higher levels, extension to parallel I/O architectures, dynamic layout and compaction schemes, and improved scheduling policies for non-real-time workload, to name a few.

Acknowledgements

This research was supported by NSF, Sun Microsystems, Sony Corp., and ICSI. Thanks to George Homsy, Vassilios Polimenis and Mark Moran for helpful discussions about CMFS, and to Eric Allman for encouragement and advice.

References

[1] C. Abbott, "Efficient Editing of Digital Sound on Disk", J. Audio Eng. Soc. 32, 6 (June 1984), 394.

[2] D. P. Anderson, "Meta-Scheduling for Distributed Continuous Media", UC Berkeley, EECS Dept., Technical Report No. UCB/CSD 90/599, Oct. 1990.

[3] D. P. Anderson and G. Homsy, "A Continuous Media I/O Server and its Synchronization Mechanism", IEEE Computer, Oct. 1991, 51-57.

[4] D. P. Anderson, Y. Osawa and R. Govindan, "Real-Time Disk Storage and Retrieval of Digital Audio and Video", ACM Trans. Computer Systems, to appear. Also UC Berkeley

EECS Dept. Technical Report No. UCB/CSD 91/646.

[5] J. Gemmell and S. Christodoulakis, ''Principles of Delay Sensitive Multi-media Data Storage and Retrieval'', ACM TOIS, to appear.

[6] R. Govindan and D. P. Anderson, ''Scheduling and IPC Mechanisms for Continuous Media'', Proc. of the 13th ACM Symp. on Operating System Prin., Pacific Grove, California, Oct. 14-16, 1991, 68-80.

[7] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry, ''A Fast File System for UNIX", ACM Transactions on Computer Systems 2, 3 (Aug. 1984), 181-197.

[8] P. V. Rangan and H. M. Vin, ''Designing File Systems For Digital Audio and Video'', Proc. of the 13th ACM Symp. on Operating System Prin., Pacific Grove, California, Oct. 1991, 81-94.

[9] D. Steinberg and T. Learmont, ''The Multimedia File System'', Proc. 1989 International Computer Music Conference, Columbus, Ohio, Nov. 2-3, 1989, 307-311.

[10] D. B. Terry and D. C. Swinehart, ''Managing Stored Voice in the Etherphone System'', ACM Trans. Computer Systems 6, 1 (Feb. 1988), 3-27.

[11] C. Yu, W. Sun, D. Bitton, R. Bruno and J. Tullis, ''Efficient Placement of Audio Data on Optical Disks for Real-Time Applications'', Comm. of the ACM 32, 7 (1989), 862-871.

Author Information

David P. Anderson has been an Assistant Professor at UC Berkeley since 1985. He is currently studying Macintosh programming and jazz piano. Contact him at anderson@icsi.berkeley.edu or 1627 Blake St., Berkeley CA 94703.

Yoshi Osawa recently spent a year as Visiting Industrial Fellow at UC Berkeley. After graduating from the University of Tokyo in 1985 he has worked for Sony, where his address is Sony Corporation, Electric Devices Group, Storage Systems Business Unit, Magneto-Optical Disk Drive Division, 2255 Okata, Atsugi, Kanagawa 243, Japan. His email address is osawa@strg.sony.co.jp.

Ramesh Govindan received his undergraduate degree from IIT-Madras and is completing his Ph.D. degree in CS at UC Berkeley. He can be reached at Evans Hall, UCB, Berkeley CA 94720 or ramesh@icsi.berkeley.edu.