REAL-TIME DISK STORAGE AND RETRIEVAL OF DIGITAL AUDIO/VIDEO DATA

David P. Anderson
Yoshitomo Osawa[†]
Ramesh Govindan

CS Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720

August 8, 1991

ABSTRACT

The Continuous Media File System, CMFS, supports real-time storage and retrieval of continuous media data (digital audio and video) on disk. CMFS clients read or write files in "sessions", each with a guaranteed minimum data rate. Several sessions can exist concurrently, sharing a single disk drive. Clients can concurrently access non-real-time files on the same disk. CMFS addresses several interrelated design issues: 1) real-time semantics of sessions; 2) disk layout; 3) acceptance test for new sessions, and 4) disk scheduling policy. We use simulation to compare different design choices and to estimate the performance of CMFS under various load conditions and hardware assumptions.

[†] On leave as a Visiting Industrial Fellow from MO Business Development Division, Storage Systems Group, Sony Corporation, 2255 Okata, Atsugi, Kanagawa 243 Japan.

This work was supported by NSF Infrastructure Grant CDA-8722788, NSF PYI grant CCR-86-57529, the California MICRO program, and Sun Microsystems.

1. INTRODUCTION

Current magnetic, optical, and magneto-optical disks can provide sustained data rates of 5 to 10 million bits per second (Mbps) or more. This is enough for the storage and retrieval of many forms of digital audio and video (*continuous media*, or CM) data. For example, 44.1 KHz 16-bit stereo audio has a data rate of about 1.4 Mbps, and compressed video has data rates ranging from 64 Kbps up to several Mbps. However, because current general-purpose file systems cannot guarantee data rates, their utility for CM is limited.

We have developed a *Continuous Media File System* (CMFS) that allows clients to read and write files in "sessions", each with a guaranteed minimum data rate. Multiple sessions, perhaps with different data rates, can exist concurrently. CMFS can handle non-real-time traffic concurrently with these real-time sessions.

Our prototype version of CMFS runs as a user-level process on the SunOS 4.1 version of UNIX. The CMFS server accesses a SCSI disk via the UNIX raw disk interface and communicates with clients via TCP connections. Designed as a research testbed, the prototype lacks features such as dynamic file size, symbolic naming, and real-time writing. On a workstation with digital audio I/O hardware and local disk, a more complete version of CMFS might serve as a UNIX-type file system that also can store and playback phone messages or voice mail. Running on a network data server with many disks, CMFS might support a remotely-accessible database of video programs or CD-format music (in effect, providing a shared VCR and CD player).

To provide data rate guarantees, CMFS addresses the following interrelated issues:

- **Real-time semantics:** The CMFS client interface, described in Section 2, has flexible but well-defined real-time semantics.
- **Disk layout:** Section 3 gives the CMFS assumptions about disk layout.
- **Acceptance test:** Sections 4 describes how CMFS determines if a new session can be accommodated.
- **Disk head scheduling:** Several alternative policies for ordering disk read and write operations are discussed in Section 5.

Section 6 presents simulation-based analyses of various policy choices, and Section 7 discusses related work.

2. CLIENT INTERFACE

When a CMFS file is created it is designated as either *real-time* or *non-real-time*. Clients access real-time files in *sessions*, initiated using

```
request_session(
    int direction,          /* READ or WRITE */
    FILE_ID name,
    int offset,
    FIFO* buffer,
    TIME max_workahead,
    int rate);
```

If `direction` is `READ`, `request_session()` requests a session in which the given file is read sequentially starting from the given offset (henceforth assumed to be zero). The `max_workahead` and `rate` parameters are explained below. If the session cannot be accepted, an error code is returned. Otherwise, a session is established. Data is transferred to the client via a FIFO. Once a session has been accepted, CMFS will write bytes sequentially into the FIFO, and the client will remove these bytes sequentially. We do not specify exactly how this FIFO is implemented, or how the client removes data from it. The client might be a user-level program that uses system calls to fetch data, or a kernel-level activity transmitting data on a network connection.

Similarly, if `direction` is `WRITE`, `request_session()` requests a session to write the given file. The client transfers data into the FIFO buffer, and CMFS moves data from the buffer to the disk.

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 space or if `max_rate` is too large.

Non-real-time operations (which may be performed on either type of file) are UNIX-like: open, seek, read, write and close. CMFS provides two service classes for non-real-time access, *interactive* and *background*; the class is specified on open. Interactive access is optimized for fast response, background for high throughput. There are no performance guarantees for non-real-time operations.

2.1. The Semantics of Real-Time Sessions

A session is a "contract" between CMFS and its client; each guarantees that its behavior will obey certain rules. Essentially, CMFS promises to stay well ahead of a "logical clock" (which stops running if it catches up to the client), and the client promises not to get ahead of CMFS. This approach allows CMFS to handle variable-rate files and other non-uniform access in a simple way; CMFS is guided by client behavior, and need not know about data timestamps or file internals.

We use the following notation to describe the semantics of read sessions (see Figure 1).

R:	the `rate` argument to `request_session()`.
Y:	the `max_workahead` argument to `request_session()` (must be > 0).
t_{start}:	the time when the call to `request_session()` returns.
$P(t)$:	the index of the next byte to be put into the FIFO by CMFS at time t.
$G(t)$:	the index of the next byte to be removed from the FIFO by the client at time t.
$C(t)$:	the value of the logical clock at time t.
B:	the size of the FIFO buffer, in bytes.

The logical clock $C(t)$ is defined for $t \geq t_{start}$ as follows:

$$C(t_{start}) = 0 \tag{1}$$

$$\frac{dC(t)}{dt} = R \text{ if } C(t) < G(t)$$

$$\frac{dC(t)}{dt} = 0 \text{ if } C(t) = G(t)$$

In other words, the logical clock advances at rate R whenever it trails $G(t)$, and pauses

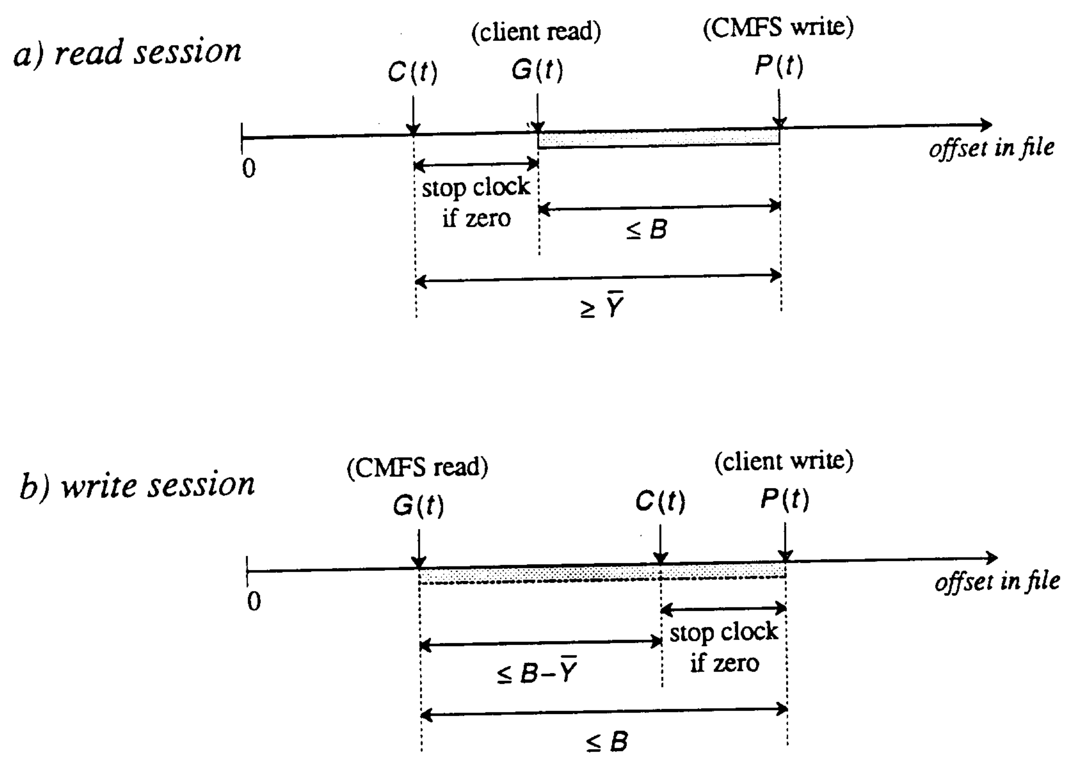

Figure 1: The semantics of read and write sessions are described in terms of a "put pointer" $P(t)$, a "get pointer" $G(t)$, and a logical clock $C(t)$. The shaded rectangles represent data in the FIFO.

otherwise.

For a read session, the following "Read Session Axioms" must hold for all $t \geq t_{start}$:

$$P(t) - G(t) \leq B \tag{2}$$

$$P(t) - C(t) \geq \bar{Y} \tag{3}$$

$$G(t) \leq P(t) \tag{4}$$

These conditions say that CMFS does not overflow the FIFO, CMFS allows the client to read ahead of the logical clock by up to $\bar{Y}$ bytes ($\bar{Y} > 0$), and the client does not read beyond the write point. CMFS therefore provides a guaranteed minimum data rate, but only as long as the client keeps up with its reading. There is no upper bound on the actual data rate; the client and CMFS can in principle work arbitrarily far ahead of the logical clock.

We describe the semantics of a write session using the same notation as above. In this case, $P(t)$ is the index of the next byte to be inserted in the FIFO by the client, and $G(t)$ is the index of the next byte to be removed by CMFS. The logical clock $C(t)$ is defined by

$$C(t_{start}) = 0 \tag{5}$$

$$\frac{dC(t)}{dt} = R \text{ if } C(t) < P(t)$$

$$\frac{dC(t)}{dt} = 0 \text{ if } C(t) = P(t)$$

The following "Write Session Axioms" must hold for all $t \geq t_{start}$:

$$P(t) - G(t) \leq B \tag{6}$$

$$C(t) - G(t) \leq B - \bar{Y} \tag{7}$$

$$G(t) \leq P(t) \tag{8}$$

These conditions say that the client does not overflow the FIFO, that CMFS removes data from the FIFO fast enough so that the client can always write ahead of the logical clock by at least $\bar{Y}$, and that CMFS reads only valid data.

2.2. The Symmetry of Reading and Writing

In describing session acceptance and scheduling algorithms, the redundancy of treating read and write sessions separately can be avoided by observing the following symmetry between reading and writing. Suppose a write session S_W has fixed parameters t_{start}, B, $\bar{Y}$, and R and time-varying parameters $C_W(t)$, $P_W(t)$, $G_W(t)$. Consider an (imaginary) read session S_R having the same t_{start}, B, $\bar{Y}$ and R parameters, and for which

$$G_R(t) = P_W(t) \tag{9}$$

$$P_R(t) = G_W(t) + B \tag{10}$$

(see Figure 2). Each disk block written by CMFS in S_W advances P_W, and thus corresponds to a disk block read by CMFS in S_R.

Claim 1. *Suppose S_W is a write session, and S_R is defined as above. Then $C_R(t) = C_W(t)$ for all $t \geq t_{start}$, and S_R satisfies the Read Session Axioms (Eqs. 2, 3, and 4) if and only if S_W satisfies the Write Session Axioms (Eqs. 6, 7, and 8).*

Proof. $C_W(t) = C_R(t)$ is clear from substituting $G(t)$ for $P(t)$ in Eq. 5. The logical clock advances in S_R exactly when it advances in S_W, and at the same rate; therefore it always has the same value in both sessions. The equivalence of the Read and Write axioms then follows from substituting Eqs. 9 and 10 in Eqs. 2 and 3. □

Hence, 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.

2.3. Using the CMFS Interface

In applications such as playback of 8-bit digital audio, I/O begins on session acceptance and proceeds at a uniform rate. In other cases the restriction to a constant I/O rate is not desirable: 1) data may be grouped into large chunks (*e.g.*, video frames) that "occur" at a single moment; 2) data may have long-term rate variation (*e.g.*, because of variable-rate compression); 3) clients may delay initial I/O to synchronize multiple sessions; 4) clients may pause and resume I/O during sessions; 5) clients may "work

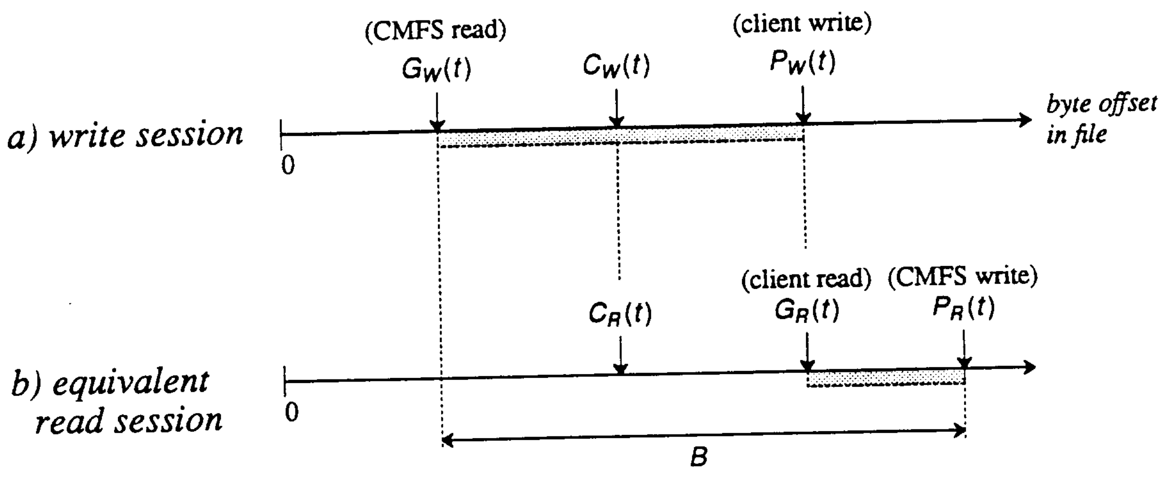

Figure 2: By interchanging empty/full and read/write, a write session (a) is transformed into a read session (b) that is equivalent with respect to scheduling.

ahead", filling up intermediate buffers to improve overall performance. The CMFS interface can accommodate all these requirements. To show this, we start by defining a general notion of temporal data.

Definition. A *bounded-rate file* F has parameters R and E, and the ith byte of F has a timestamp $T(i) \geq 0$ such that $T(i) \leq T(j)$ for $i < j$, and

$$i - j \leq R(T(i) - T(j)) + E \qquad (11)$$

for all $i > j$ (see Figure 3). In other words, the amount of data in a time interval of length T is at most $RT + E$. (R and E can be thought of as "maximum rate" and "block size" respectively.) The timestamps may be explicit (embedded in the data) or implicit.

Definition. Suppose a client reads a bounded-rate file F. We say that the file is *read in real time starting at t_0 with buffer size N* if 1) byte i is read (*i.e.*, removed from the FIFO) before $t_0 + T(i)$ and 2) at any time t, no more than N bytes i such that $T(i) > t - t_0$ have been read.

Intuitively, this means that the client reads the file fast enough to get the data on time, but slow enough so that only N bytes of buffer space are needed.

The CMFS interface allows a client to read a bounded-rate file in real-time using limited buffer space:

Claim 2. *Suppose a client creates a session reading a bounded-rate file F with parameters R and E, and the session begins (i.e., the* `request_session()` *call returns) at time 0. Then it is possible for the client to read F in real time starting at E/R with buffer size E.*

Proof. We will show the existence of a client behavior that is consistent with the Read Session Axioms and that satisfies the above definition of reading a bounded-rate file in real time. Let the client behave as follows. At time i/R (for each $i \geq 0$) the client removes the next byte n from the FIFO if $T(n) < (i+1)/R$. It then waits until time $(i+1)/R$.

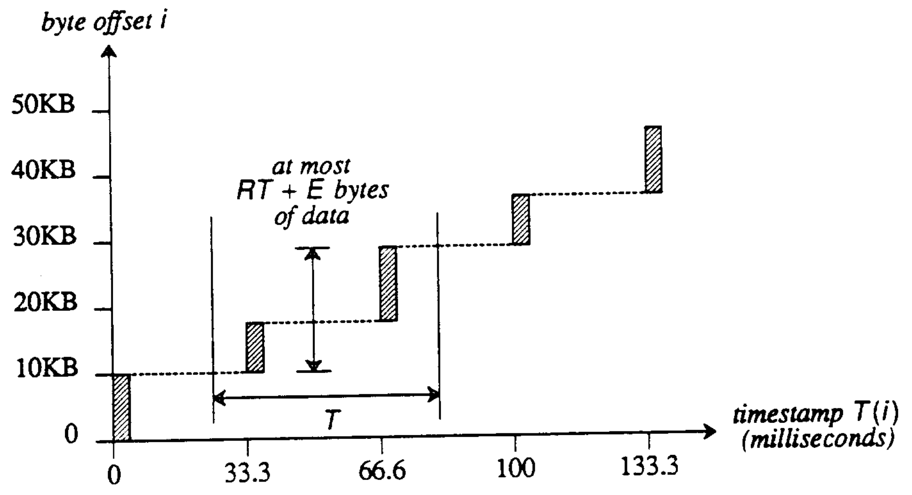

Figure 3: A *bounded-rate file* contains timestamped data. This example represents a file of 30 frames/second video data with a varying number of bytes per frame; each frame is a vertical segment of the stairstep. The file has parameters R and E; the number of bytes with timestamps in an interval of length T cannot exceed $TR + E$, as shown.

Hence in each "time slot" of length $1/R$ the client either reads a byte or skips to the next time slot. The clock $C(t)$ advances during a read slot and pauses during a skip slot. The client "works ahead" by at most one byte, so (since the workahead parameter Y of the session is at least one) the Read Session Axioms are obeyed.

To verify the claim, we must first show that each byte n is read no later than $T(n) + E/R$. Suppose otherwise. Then some byte n is read at time j/R with

$$j/R > T(n) + E/R \tag{12}$$

If no skips occurred in slots $0 \cdots j$ then let $i = 0$; otherwise let i be such that $i-1$ is the last skip slot before j. Let m be the index of the byte read in slot i. Then

$$T(m) > i/R \tag{13}$$

since otherwise m would have been read in slot $i-1$. Now combining Eqs. 12 and 13 we have

$$T(n) - T(m) < (j - i - E)/R \tag{14}$$

Since the client reads in every slot from i to j we have $j - i = n - m$. Therefore

$$n - m > (T(n) - T(m))R + E \tag{15}$$

which contradicts the assumption that F is a bounded-rate file.

Finally we must show that at any time t, no more than E bytes i such that $T(i) > t - E/R$ have been read. Let i be such that $T(i) > t - E/R$. Then, given our specification of client behavior, byte i must have been read at time $t - E/R$ or later. The client reads at most E bytes in time E/R, so the claim holds. □

Claim 2 and its proof make the unrealistic assumption that client CPU processing is instantaneous and can be scheduled at precise instants. To guarantee real-time performance in practice, the client must: 1) compute bounds on the client CPU processing time; 2) make a "reservation" with the CPU scheduler for the needed CPU time; 3) based on the processing time bounds and the scheduling delay bounds provided by the CPU scheduler, compute an appropriate value for $\bar{Y}$ and use this in the `request_session()` call. The details depend on the form of the CPU scheduler's guarantees. Similar reservations must be made for other resources (*e.g.*, network bandwidth) used to handle the CM data stream.

The CMFS interface provides implicit flow control between client and CMFS. This flow control is well-suited to bounded-rate files, and can accomplish other goals as well:

- If a client wants to pause and resume a session (*e.g.*, because of user interaction), it can do so simply by stopping the removal of data from the FIFO; the logical clock will stop soon thereafter. Data rate and buffer-requirement guarantees will remain valid after the client resumes reading. (The pause/resume is equivalent to shifting the timestamps of the remaining data.)
- Suppose a client does synchronized playback or recording to/from multiple files, perhaps on different CMFS servers. The various `request_session()` calls may return at different times. To limit buffer space usage, the client should postpone doing I/O on any of the sessions until all of the `request_session()` calls have returned. The clocks of all sessions remain at zero during this period. When all the calls have returned, the client begins reading from the sessions on each server; the buffer space needed for synchronization then depends only on the skew of the start messages.
- The client can, if the hardware is fast enough, read arbitrarily far ahead of the logical clock. This "workahead" data can then be buffered (in distributed applications, the buffers may be spread across many nodes), protecting against playback glitches and allowing improved system response to transient workload.

3. DISK LAYOUT ASSUMPTIONS

We assume that the CMFS uses a single-spindle disk drive, so that all operations are done sequentially, and that the disk is read and written in fixed-sized *blocks*. The CMFS reservation and scheduling algorithms do not mandate a particular disk layout. Instead, we assume that the layout allows the following "bounding functions" U_F and V_F to be obtained:

(1) For a given file F and n, $U_F(n)$ is an upper bound on the time to read n logically contiguous blocks of F (including all seek and rotation time), independent of the initial position of the disk head and the starting block number to be read.

(2) $V_F(i, n)$ is an upper bound on the time needed to read the n blocks of file F starting at block i.

The bounds need not be tight; slackness in the bounds may, however, cause sessions to be rejected unnecessarily.

The functions U and V should take into account interrupt-handling latency, CPU overhead, and features (such as track buffering) of the disk controller. They can also reflect bad blocks detected when the disk is initialized. For example, they can be based on the assumption that each track has at most one bad block; tracks with more than one can then be left unused or devoted to non-real-time files.

On disk operations that fail (*e.g.*, because of checksum errors) most controllers notify the CPU of the failure, allowing the CPU to retry the operation. In general, of course, unbounded retries can interfere with performance guarantees. CMFS attempts to maintain "slack time" (Section 5.1), allowing it to retry failed operations without violating performance guarantees. If it runs out of slack time, it skips the operation, resulting in a block of undefined data in the file (write) or output stream (read). Such errors may be tolerable if their only effect is a "glitch" in the media stream, as is the case for many forms of CM data.

3.1. Examples of Disk Layouts

The CMFS prototype supports only fixed-size files, and file allocation is physically contiguous. Each file begins at some point within a cylinder, filling the remainder of that cylinder, zero or more adjacent cylinders, and part of a final cylinder.

To demonstrate possible bounds functions for this layout, we assume L_{seek_min} and L_{seek_max} are bounds on the 1-track seek time and the worst-case seek time respectively, L_{block} is the time to read one block, $L_{rotation}$ is the rotation time, and N is the number of blocks per cylinder. Furthermore, we assume that the controller does track-buffering; it reads a track into a local buffer immediately after seeking to it. Hence, if an entire track is read, rotational latency is negligible regardless of the order in which the sectors are read. Possible bounds functions are

$$U_F(n) = L_{seek\_max} + nL_{block} + \left\lceil \frac{n}{N} \right\rceil L_{seek\_min} + 2L_{rotation} \tag{16}$$

and

$$V_F(i, n) = L_{seek\_max} + nL_{block} + (k-1)L_{seek\_min} + \frac{j}{N}L_{rotation}$$

where k is the number of cylinders storing the n blocks of F starting at offset i, and j is the number of blocks not in F in the first and last of these cylinders.

This contiguous layout policy is feasible for read-only file systems or if disk space is abundant. For more flexibility, a variant of the of the 4.2BSD UNIX file system layout [5] 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 determine the `max_rate` parameter of the file. Bounds functions U and V can be computed from n, k, the size of a cylinder group, and the disk parameters. Allocation and compaction strategies would pose a complex set of issues; we do not discuss them here.

4. ACCEPTANCE TEST

CMFS can accept a new session S only if its data rate requirements, together with those of existing sessions, can be guaranteed. We now describe a procedure for making this decision.

We begin by introducing terminology. Suppose that sessions $S_1 \cdots S_n$ read files $F_1 \cdots F_n$ at rates $R_1 \cdots R_n$. An *operation set* ϕ assigns to each S_i a positive integer M_i. CMFS *performs* an operation set 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 (the order of operations is not specified).

Let $L(\phi)$ denote the worst-case elapsed time needed to perform ϕ based on the layout (Section 3):

$$L(\phi) = \sum_{i=1}^{n} U_{F_i}(M_i) \tag{17}$$

Let $D_i(\phi)$ denote $\frac{M_i A}{R_i}$, where A is the block size in bytes. Let $D(\phi)$ denote $\min_{1 \le i \le n} D_i(\phi)$. Intuitively, $D(\phi)$ is the period for which the data read in ϕ "sustains" the sessions.

A *workahead-augmenting set* (WAS) is an operation set ϕ such that $L(\phi) < D(\phi)$, or equivalently

$$M_i A > R_i L(\phi) \tag{18}$$

for all i. In other words, the data read in a WAS "lasts longer" than the worst-case time it takes to perform the operations.

An operation set ϕ is called *feasible* if, for all i,

$$M_i A + \bar{Y}_i \le B_i \tag{19}$$

where B_i is the size of the FIFO buffer used by S_i and $\bar{Y}_i$ is the workahead limit of S_i (Section 2.1). Intuitively, this means that the amount of data read in ϕ for each session fits in the corresponding FIFO, even in the extreme case where the client has used none of its "workahead allowance" $\bar{Y}_i$.

An *operation sequence* Φ is a pair (π, ϕ), where π is a permutation of $1 \cdots n$ and ϕ is an operation set. CMFS *performs* an operation sequence by doing the operations in ϕ in the order $\pi(1) \cdots \pi(n)$. Φ is called *workahead-augmenting* if ϕ is workahead-augmenting; likewise Φ is *feasible* if ϕ is feasible.

Let $W_i(t)$ denote $(P(t) - C(t) - \bar{Y})/R$, where P, C, $\bar{Y}$ and R are the parameters for S_i. W_i is the "workahead" for session i, *i.e.*, the duration of data currently buffered for the session, above and beyond the client's workahead allowance $\bar{Y}$. We say that the session *starves* if W_i becomes negative.

The *file system state* $W(t)$ at time t is the vector $<W_1(t) \cdots W_n(t)>$. Let Φ be an operation sequence. A state W is *safe relative to* Φ if

$$W_{\pi(j)}(t) \ge \sum_{i=1}^{j} L(\pi(i))$$

for all j (recall that $L(j)$ is the worst-case time needed for S_j's operation). Intuitively, this means that enough data is currently buffered so that, if Φ is performed immediately, no session starves before its operation is completed.

Claim 3. *If Φ is workahead-augmenting and feasible, then there is a system state that is safe relative to Φ.*

Proof. Consider the state in which all buffers are full. Then for session S_j we have

$$\begin{aligned} W_j(t) &= (P(t) - C(t) - \bar{Y})/R \\ &= (B_j - \bar{Y})/R \\ &\ge (M_j A)/R \\ &> L(\Phi) \\ &\ge \sum_{i=1}^{j} L(\pi(i)) \end{aligned}$$

The final three steps use Eqs. 19, 18, and 17 respectively. □

Claim 4. *Suppose that there is a feasible workahead-augmenting operation sequence Φ, and assume that at time t_0 the state W is safe relative to Φ. Then the CMFS can satisfy the Read Session Axioms (Eqs. 2 and 3) for all i and $t \geq t_0$.*

Proof. We prove this by defining disk scheduling policy, called the **Static** policy, that satisfies the axioms. The policy is as follows. Repeatedly apply the schedule given by Φ, with the following exception: if $P(t) + A - G(t) > B_i$ at the point of starting a block read for S_i, then immediately skip to the next session (since reading the block could cause a buffer overflow). It is clear that this policy preserves Eq. 2.

Consider a particular session S_k during one "cycle" of Φ, starting at time $t = 0$ (see Figure 4). Let t_D denote the time at which S_k's operation ends. Then $t_D \leq \sum_{i=1}^{k} L(\pi(i))$. Eq. 3 holds during $[0, t_D]$ since $C(t)$ advances by at most R bytes/second, and $W_k(0) \geq t_D$. The operation for S_k either reads the full amount or is truncated; in either case $W_k(t_D) > L(\Phi)$, since Φ is workahead-augmenting. Let t_E denote the time at which the cycle ends. Then $t_E - t_D \leq \sum_{i=k+1}^{n} L_i$, and the clock $C(t)$ advances at a rate of at most R, so $W_k(t_E) \geq \sum_{i=1}^{k} L(\pi(i))$. Hence W_k remains positive during $[t_D, t_E]$.

Therefore no starvation occurs during the cycle, and at the end of one cycle the system state is again safe relative to Φ. Hence the Static scheduling policy will maintain the Read Session Axioms for all sessions. ☐

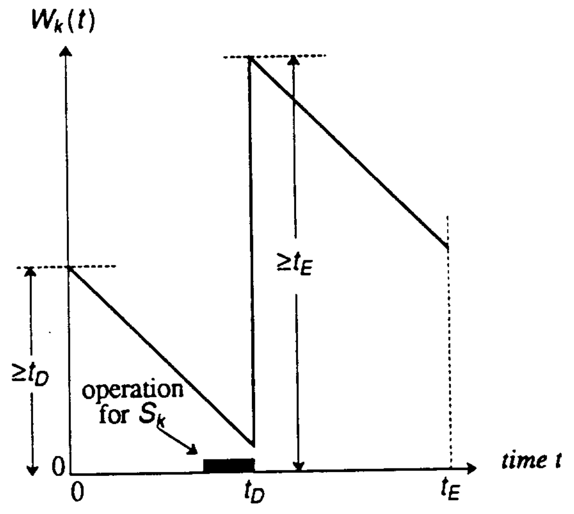

Figure 4: Diagram for the proof of Claim 4, showing the workahead W_k of a session S_k during one cycle of the Static scheduling policy. At the start of the cycle, S_k has enough workahead to last until time t_D, when its operation is finished. The amount of data read suffices for at least $L(\phi)$, which exceeds the length t_E of the cycle. Therefore W_k is always positive; *i.e.*, S_k never "starves".

4.1. The Minimal Feasible WAS

Claim 4 shows that CMFS can satisfy the data rates of a set of sessions if there is a feasible WAS. 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 $i \geq 0$ (see Figure 5). Let I_i denote the interval $(t_i, t_{i+1}]$. Let ϕ_i denote the operation set $<\left\lceil R_1 t_1 \right| \cdots \left\lceil R_n t_n \right|>$. Note that ϕ_{i+1} differs from ϕ_i by the addition of 1 block to all sessions whose data periods divide t_{i+1}; hence the sequence of ϕ_i is easy to compute. Note also that $L(\phi_i) < L(\phi_{i+1})$ for all i.

Claim 5. *If ϕ is an operation set such that $D(\phi) \in I_i$, then $L(\phi) \geq L(\phi_i)$.*

Proof. Any sequence ϕ for which $D(\phi) > t_i$ must read at least $\left\lceil \frac{t_i}{R_j} \right\rceil$ blocks for each session j, and hence $L(\phi) \geq L(\phi_i)$. □

Claim 6. *If there is a feasible WAS ϕ such that $D(\phi) \in I_i$, then ϕ_i is feasible.*

Proof. ϕ must read at least as many blocks for each session as does ϕ_i. Therefore, since ϕ is feasible, so is ϕ_i. □

Claim 7. *The following algorithm computes the minimal WAS:*

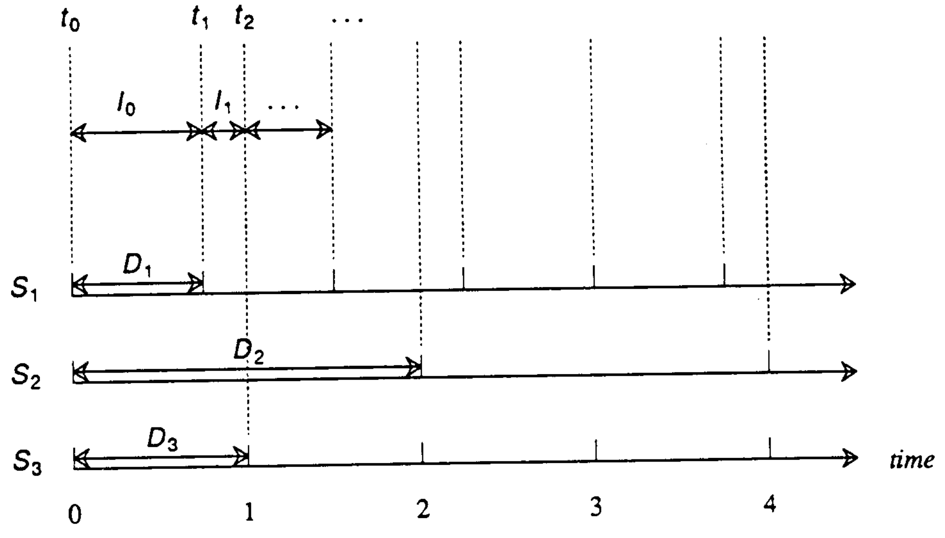

Figure 5: A block of data for session S_i has a "duration" D_i that depends on the data rate of S_i. The set of all multiples of these periods defines a set of intervals I_i. Within each interval there is a unique minimal-length operation set ϕ_i.

(1) Let $\phi_0 = <1, \cdots, 1>$ (this is the minimum length operation set 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 $M_i A + \bar{Y}_i \le B_i$ for all i) stop; there is no feasible WAS.

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

(4) Compute ϕ_{i+1} and go to (2).

Proof. Suppose the algorithm stops in step 3, returning a WAS in I_j. Let ϕ be the minimal-length WAS, and let i be such that $D(\phi) \in I_i$. It is not possible that $i<j$, since then ϕ_i is feasible (Claim 6) and workahead-augmenting, so the algorithm would have terminated at iteration i. It is also not possible that $i>j$, since then $L(\phi) \ge L(\phi_i) > L(\phi_j)$, contradicting the minimality of $L(\phi)$. Therefore $i=j$ and (from Claim 5) we must have $L(\phi_i) = L(\phi)$, so ϕ_i is the minimal WAS.

Finally, suppose that the algorithm terminates in step 2 for some i. Suppose that a feasible WAS ϕ exists, with $D(\phi) \in I_j$. By the above arguments $i \le j$. But then ϕ reads at least as many blocks for each session as does ϕ_i, so the buffer allocation feasible for ϕ is feasible for ϕ_i, which is a contradiction. □

4.2. Buffer Space Allotment

Suppose that a fixed amount B of buffer space is available for CMFS client FIFOs. How should this space be divided among the various clients? CMFS performs best when all sessions can "work ahead" by about the same time (see Section 6). In other words, the buffer space allocated to a session, beyond that needed for the client's workahead allowance $\bar{Y}$, should be roughly proportional to the data rate R.

CMFS therefore uses following policy. Let Y denote $\sum_{i=1}^{n} \bar{Y}_i$, and let R denote $\sum_{i=1}^{n} R_i$. Session S_j is allocated

$$\bar{Y}_i + \frac{(B-Y)R_j}{R} \tag{20}$$

bytes. This allocation may need adjustment in two cases. First, the above expression is real-valued; the actual allocation must be an integer, and possibly a multiple of some "allocation block" size. Second, each session must be given enough space to accommodate the minimal WAS (Section 4.1).

5. DISK SCHEDULING POLICY

On completion of each disk block I/O, CMFS decides which disk block to read or write next, and issues the appropriate command (seek, read, or write) to the disk controller. The algorithm for this decision constitutes a *disk scheduling policy*. Such a policy must

- satisfy the requirements of current sessions (see Section 2.1);
- delay the return of the `request_session()` call for a newly accepted session until it is safe to do so (we call this the *session startup* policy);
- efficiently service non-real-time workload.

Policies for real-time CPU scheduling, such as earliest-deadline-first [4], are not immediately relevant because of seeks. In this section we describe several possible disk scheduling policies. Some of these policies are defined in terms of *slack time*, which we will now define.

5.1. Slack Time

Suppose that the minimal WAS $\bar{\phi}$ takes worst-case time $L_1 + \cdots + L_n$, where $L_i = U_{F_i}(M_i)$ (*i.e.*, the time for an inter-file seek and the read of M_i blocks of file F_i). Let π be a permutation of $1 \cdots n$, and let Φ be the operation sequence $(\pi, \bar{\phi})$. For a given session j, let $H_j = W_j - \sum_{i=1}^{j} L_{\pi(i)}$, and let $H = \min_{i=1}^{n}(H_i)$.

The quantities H_j and H, which are time-varying since they depend on the workaheads W_i, have the following intuitive meaning. If CMFS performed Φ starting now, the workahead for S_i would not fall below H_i; in fact no sessions would starve if Φ were postponed for H seconds. In this sense, H is the "slack time" during which the CMFS is free to do non-real-time operations or workahead for real-time sessions.

Claim 8. *Let $\bar{\pi}$ be the ordering of sessions by increasing value of W. Then, among all permutations π, $\bar{\pi}$ gives the maximal value of H.*

Proof. Consider a permutation π in which W is not increasing; in particular suppose $W(S_{\pi(k)}) > W(S_{\pi(k+1)})$. Let B and C denote the slack times of the two sessions. If we reverse the order of the two sessions in π, then for the new slack times D and E we have $C < D$ and $C < E$ (see Figure 6). The slack times of other sessions remain unchanged. Hence the minimum of the slack times is not decreased by reversing the order. □

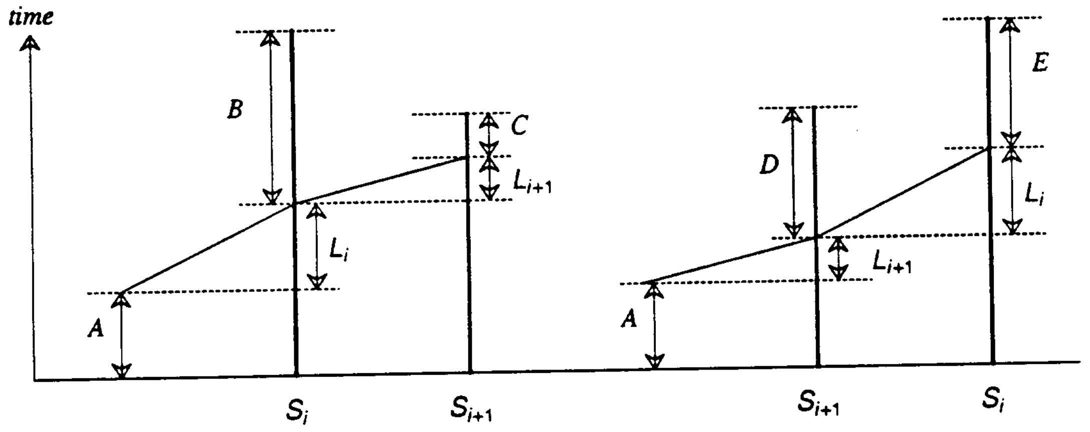

Figure 6: Global slack time is maximized by ordering sessions by increasing workahead. Suppose a sequence has two sessions S_i and S_{i+1} that are not in this order. The bold lines represent their workaheads W, and their slack times are B and C as shown (A denotes the maximum time needed for operations preceding i; L_i is the time bound for the operation of session S_i). By reversing the order of the two sessions, the slack times are D and E. Simple algebra shows that $C < D$ and $C < E$. Global slack time is the minimum of the session slack times, so the result follows.

We therefore consider only the increasing-workahead ordering $\bar{\pi}$ of sessions. Let $\bar{\Phi}$ denote $(\bar{\pi}, \bar{\phi})$, and let H_i and H denote the corresponding slack times at a particular moment.

It is important to note that $H < 0$ does not imply that starvation has occurred or will occur. H is based on the pessimistic assumption that an inter-file seek is needed prior to every operation in the WAS. For example, if $H < 0$ during of a multi-block file operation, CMFS is simply compelled to finish the current operation; starting a new WAS would incur an inter-file seek.

5.2. Real-Time Scheduling Policies

Having defined slack time, we can now describe several possible disk scheduling policies. These policies all avoid starvation; their relative performance is discussed in Section 6.

(1) **The Static/Minimal policy** (a special case of the Static policy described in Section 4) simply repeats the minimal WAS. This policy avoids starvation, as shown in the proof of Claim 4. However, its use of short operations causes high seek overhead, so the performance of non-real-time traffic suffers.

(2) The **Greedy policy** does the longest possible read for each session. At each iteration, it computes the slack time H, finds the session S with smallest workahead, and reads the greatest number n of blocks for S such that 1) $V_F(i, n) \leq H$ and 2) the blocks fit in currently available buffer space.

(3) The **Cyclical Plan policy** differs from Greedy in that, instead of devoting all the slack time to readahead for one session, it tries to distribute it among the sessions in a way that maximizes slack time. It computes H and augments the minimal WAS $\bar{\Phi}$ with H seconds of additional reads (these reads are done, for each session S_i, immediately after the read for S_i in $\bar{\Phi}$). The policy distributes workahead by identifying the "bottleneck session" (that for which H_i is smallest) and schedules an extra block for it, updating H_i and H; this is repeated until H is exhausted. The resulting schedule determines the number for blocks read for the least-workahead session; when this read completes, the procedure is repeated.

In both the Greedy and Cyclical Plan policies, the least-workahead session 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* the least-workahead session, yielding what we call **Aggressive** versions of each policy.

All policies skip to the next session when a buffer size limit is reached. If at some point all buffers are full, then no operation is started. When a client subsequently removes sufficient data from a FIFO, the policy is restarted.

5.3. Non-Real-Time Operations

A non-real-time operation N can arrive at any time. N is queued if other non-real-time operations are queued or in progress. Otherwise, CMFS must decide whether to start N immediately ("preempting" the current real-time cycle) or defer it. To decide this, CMFS computes the current value of H. If N has worst-case latency L (including seek time) then it can safely be started if $L \leq H$. While N is being handled, H decreases at rate one. Further non-real-time operations, arriving during N, may be handled immediately if H will remain nonnegative. When non-real-time operations are suspended (because they are finished or H is too low) the real-time scheduling policy is resumed.

The policy of servicing a non-real-time operation whenever it is safe to do so may tend to keep H low. This forces the scheduler to do short real-time operations (close to the minimal WAS), causing the system to run inefficiently. To avoid this, it may be preferable to do non-real-time operations only when slack time exceeds some nonzero threshold.

To avoid the seek overhead of rapidly alternating between real-time and non-real-time operations, CMFS uses the following *slack time hysteresis* policy for non-real-time workload. An interactive operation is started only if $H \in [H_{I1}, H_{I2}]$. Once H falls below H_{I1}, no further interactive operations are started until H exceeds H_{I2}.

Similarly, background operations are done within a hysteresis interval $[H_{B1}, H_{B2}]$. No background operation is started if an interactive operation is eligible to start. In Section 6 we examine the effects of hysteresis, and of the hysteresis parameters, on system performance.

5.4. Session Startup

A newly-accepted session is said to *start* when its `request_session()` call returns. This must occur only when the system state is safe with respect to the new WAS. A special mechanism is needed for handling 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 feasible WASs for the sets $S_1 \cdots S_n$ and $S_1 \cdots S_{n+1}$ respectively. S_{n+1} is started as follows.

CMFS adjusts FIFO buffer sizes according to the procedure described in Section 4.2. It can shrink a buffer by discarding data from the end of the FIFO if needed (it must later reread the data from disk). The scheduler then goes into "startup mode" during which its policies are changed as follows:

(1) Non-real-time operations are queued for later execution.

(2) For scheduling purposes, slack time H is computed relative to ϕ_n. However, in the Cyclical Plan policy the allocation of slack time for workahead is done relative to ϕ_{n+1}, using a session ordering in which the new session appears first (however, no I/O for S_{n+1} is done during this phase).

(3) When the system state is safe with respect to ϕ_{n+1}, then a read of $\phi_{n+1}(n+1)$ blocks for S_{n+1} is started. When this read is completed, the system 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.

The above policy is sequential: if several sessions are accepted at about the same time, they are started in sequence. It would also be possible to parallelize startup of multiple sessions; we omit this for simplicity. Step (3) can be omitted for write sessions because the equivalent read session starts with a full buffer (Section 2.2).

6. PERFORMANCE

In this section we study the performance of CMFS as a function of disk scheduling policies and hardware parameters. We give performance measurements obtained by simulation, using the source code of the CMFS prototype with the disk I/O calls replaced by calls to an event-based simulator. All actions other than disk I/O are modeled as instantaneous; *i.e.*, we do not model the CPU time needed to copy data between buffers or to execute policy algorithms. We assume that an interrupt is generated for every block I/O completion (this is not realistic for some disk interfaces). 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.

6.1. Number of Concurrent Sessions

Figure 7 shows the maximum number of concurrent sessions accepted by CMFS as a function of system buffer size. This is shown for two different session data rates: 64 Kbps and 1.4 Mbps. In each graph, curves are given for three different disk types: 1) 39 ms maximum seek time and 11.8 Mbps transfer rate (CDC Wren V), 2) 35 ms maximum seek time and 8.4 Mbps transfer rate (CDC Wren III) and, 3) 180 ms maximum seek time and 5.6 Mbps transfer rate (Sony 5.25" optical disk).

The disk transfer rate imposes an upper bound on the number of concurrent sessions that can be accepted. Unbounded buffer space is needed as this limit is approached. To reach 90% of the limit with a type 1 disk requires 4 MB for 1.4 Mbps sessions and 85 MB for 64 Kbps sessions. The efficiency depends on the length of operations in the minimal WAS; 64 Kbps sessions require a proportionally longer WAS and therefore more buffer space. When the number of accepted sessions is fixed, a disk with higher seek time needs a longer minimal WAS and therefore more buffer space.

6.2. Performance of Disk Scheduling Policies

Since non-real-time operations can be done only if there is enough slack time, the main criterion for disk scheduling policies is how quickly they increase slack time. To study this, we simulated CMFS with three concurrent 1.4 Mbps sessions, no non-real-time traffic, and 4 MB system buffer size. At time zero, system slack is zero and all

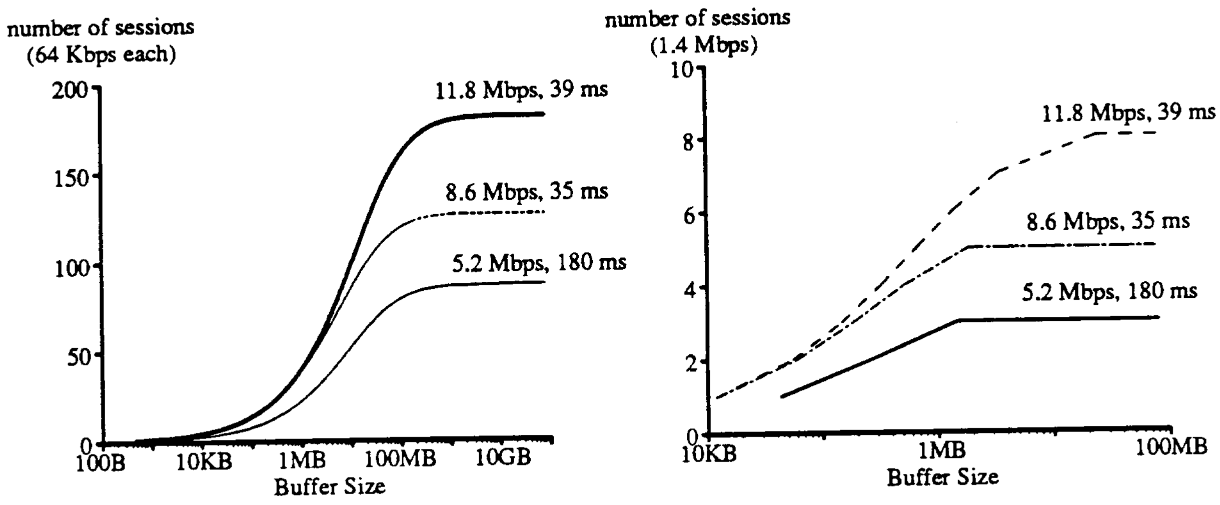

Figure 7: The number of sessions that can be accepted by CMFS depends on the available buffer space. The disk transfer rate imposes an upper limit on the number of sessions; to reach 90% of the limit with a type 1 disk requires 4 MB for 1.4 Mbps sessions and 85 MB for 64 Kbps sessions.

sessions have equal workahead. The results are shown in Figure 8.

At low values of system slack (up to 3 seconds) Cyclical Plan increases system slack somewhat faster than Greedy because it allocates workahead more evenly between sessions. CMFS is likely to be in this state during session startup. During steady-state operation, system slack will usually be higher. In this range, all the policies perform about the same, except Static/Minimal, which performs worse.

6.3. Response Time of Interactive Traffic

To study the effect of real-time traffic on interactive traffic, we simulated a fixed number of real-time sessions together with interactive requests that read randomly positioned blocks from disk. The interactive request arrival is Poisson with mean arrival rate λ. We define the *response time* of an interactive request as the time from its arrival to the start of the disk operation; the delay of the operation itself, including the seek, is not included.

The effect of hysteresis parameters is most noticeable under heavy load (otherwise slack remains high and hysteresis is not exercised). Figure 9 plots mean interactive response time against H_{i2}, for different values of H_{i1}, under a heavy load. We observe that:

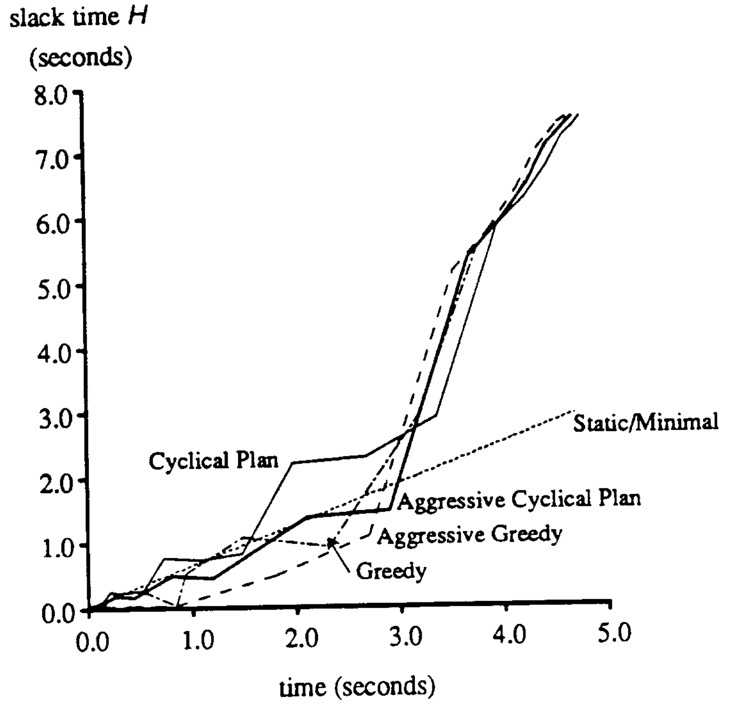

Figure 8: Disk scheduling policies build up slack at different rates. Above a certain point, all the policies except Static/Minimal perform about the same.

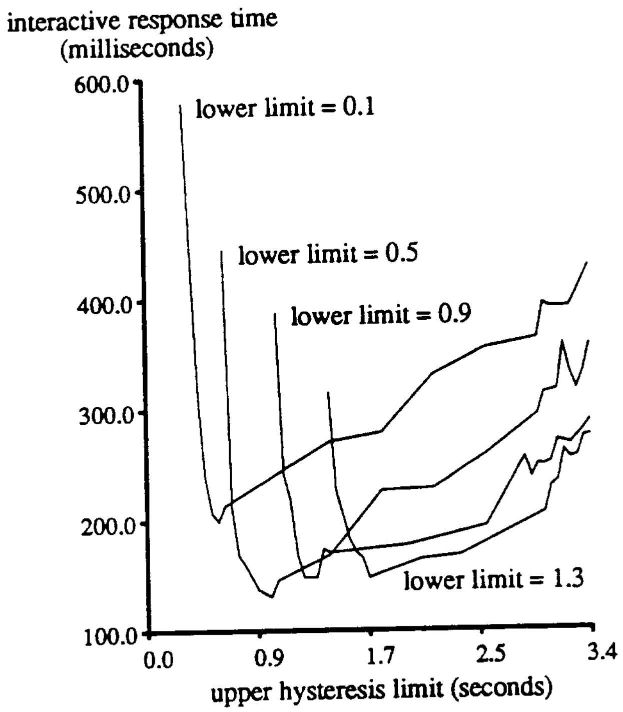

Figure 9: Interactive response times fall steeply and then rise gradually with increasing H_{l2}. Also, interactive response is poor for very small values of H_{l1}. This experiment was conducted with 2 MB total buffer (H_{buffer} is 3.4 seconds), 20 interactive arrivals per second and three 1.4 Mbps sessions.

- For fixed H_{l1} and increasing H_{l2}, interactive response time drops steeply and then rises gradually. When H_{l1} nearly equals H_{l2}, interactive response is poor because the system enters a mode in which it switches rapidly between real-time and non-real-time operations, causing high seek overhead. As H_{l2} increases, this oscillation becomes less frequent and response improves. Since interactive requests are queued while slack builds up from H_{l1} to H_{l2}, interactive response degrades if H_{l2} is increased past a certain point.
- When H_{l1} is very small (*e.g.*, 0.1 sec in Figure 9), system slack builds up slowly from H_{l1} to H_{l2} so interactive response is poor. A similar effect is observed if H_{l2} is close to the *buffer limit of slack*, H_{buffer} (the maximum slack that the system can accumulate for fixed buffer size).[1] For all H_{l1}, the interactive response is best when $H_{l2} - H_{l1}$ is about one second.

[1] A first approximation to H_{buffer} is $W - nL/C$, where W is the maximum workahead a session can accumulate, n is the number of sessions, L is the worst case seek/rotation time, and C is the fraction of disk bandwidth not used by real-time sessions.

For the disk parameters and session rates we have considered, a "rule of thumb" for hysteresis limits is as follows: set H_{l1} to $H_{buffer}/3$, and set $H_{l2} - H_{l1}$ to the smaller of $H_{buffer}/3$ and one second.

Figure 10 plots mean interactive response time as a function of arrival rate, for different values of system buffer size. If interactive arrival rate is low, system slack stays near H_{buffer} (Figure 11) and most interactive requests are service without waiting for real-time traffic. At higher arrival rates (in Figure 10, about 5 per second for the 500 KB case and 10 per second for 1 MB), interactive response degrades because system slack sometimes reaches the lower hysteresis limit H_{l1}, and interactive requests then are blocked until slack reaches H_{l2}.

6.4. Throughput of Background Traffic

To estimate the effect of real-time traffic on background traffic throughput, we simulated three 1.4 Mbps sessions and a single background task that sequentially reads a long, contiguously-allocated file. We define the *background throughput fraction* T as the

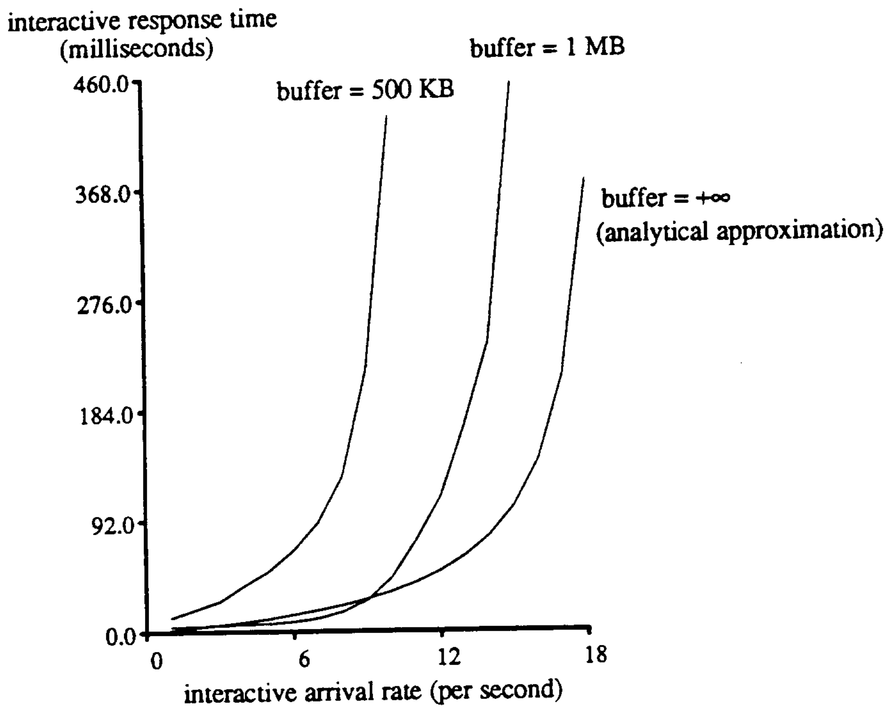

Figure 10: Interactive response time as a function of arrival rate, for different values of system buffer size. In this experiment there were four concurrent 1.4 Mbps sessions. The hysteresis limits (0.04, 0.08) and (0.35, 0.65) were used for the 500 KB and 1 MB cases respectively. The "infinite buffer" curve was obtained analytically, modeling the filesystem as an M/G/1 queue.

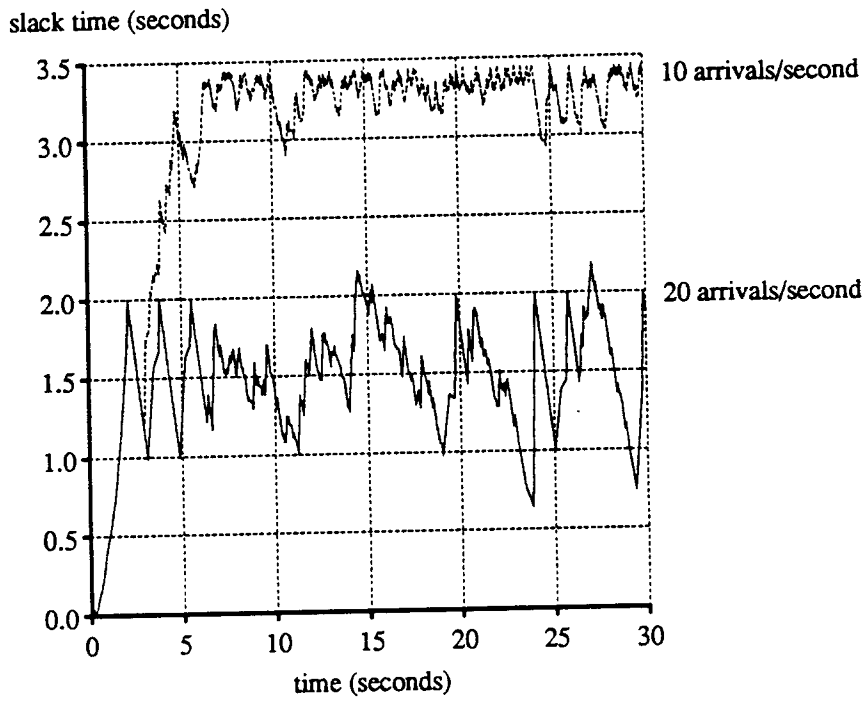

Figure 11: This graph shows how slack time varies in the presence of interactive non-real-time traffic. Three 1.4 Mbps sessions start at time zero, the system has 2MB of buffer space (H_{buffer} is 3.4 seconds), and hysteresis limits are (1 second, 2 second). If the interactive arrival rate is low (10 per second in this case), system slack stays near H_{buffer}. For high arrival rates (20 per second), slack oscillates between the hysteresis limits.

fraction of residual disk bandwidth (*i.e.*, disk bandwidth not taken up by real-time sessions) used by the background task. We describe below the effects of hysteresis and system buffer size on T.

Figure 12 plots T against lower and higher hysteresis limits. For the same reasons as discussed in Section 6.3, T is low if H_{B1} is very small, H_{B2} is close to H_{buffer}, or $H_{B2} - H_{B1}$ is small. In this case (since throughput, rather than response time, is the goal) there is no penalty if $H_{B2} - H_{B1}$ is large.

For the disk parameters and session rates we have considered, a "rule of thumb" for background hysteresis parameters is: set H_{B1} to $H_{buffer}/4$, and set H_{B2} to 95% of H_{buffer}.

Figure 13 plots optimal T (defined as the maximum in Figure 12) against buffer space. Throughput increases with system buffer space. With larger buffer space, there is less seek overhead in building system slack from H_{B1} to H_{B2} so more effective disk bandwidth is available.

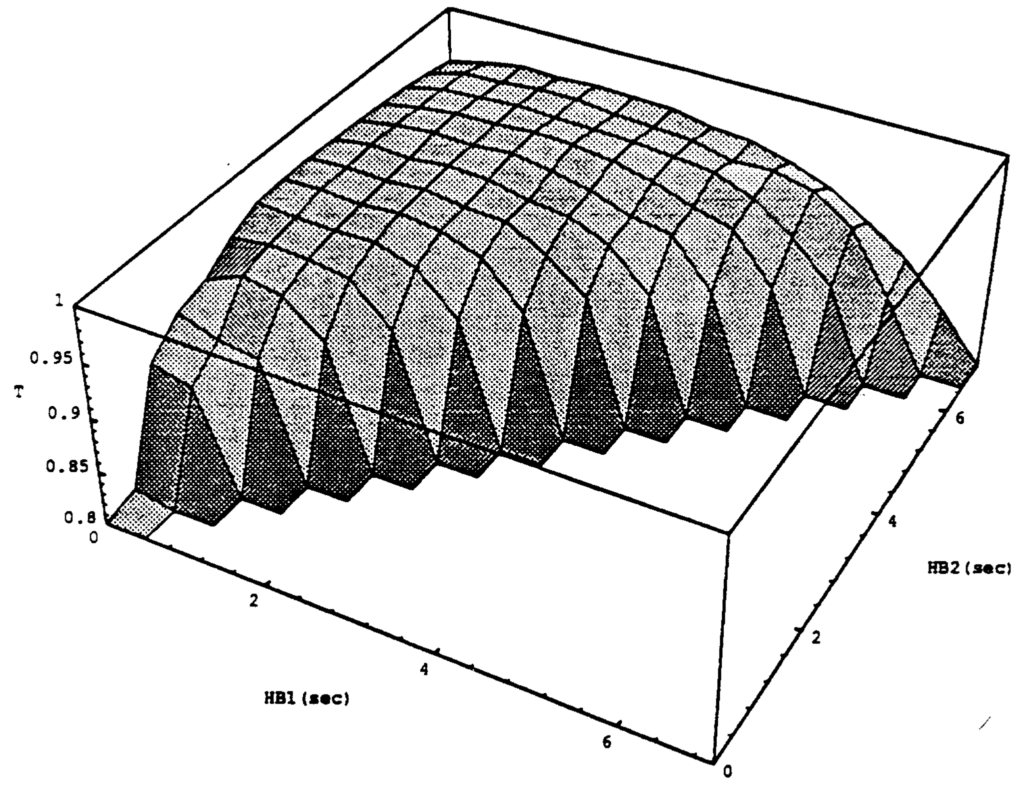

Figure 12: Background throughput fraction T as a function of the hysteresis limits H_{B1} and H_{B2}. H_{B1} is plotted on the X-axis and H_{B2} on the Y-axis. Buffer size is 4 MB, and the real-time workload is three 1.4 Mbps sessions.

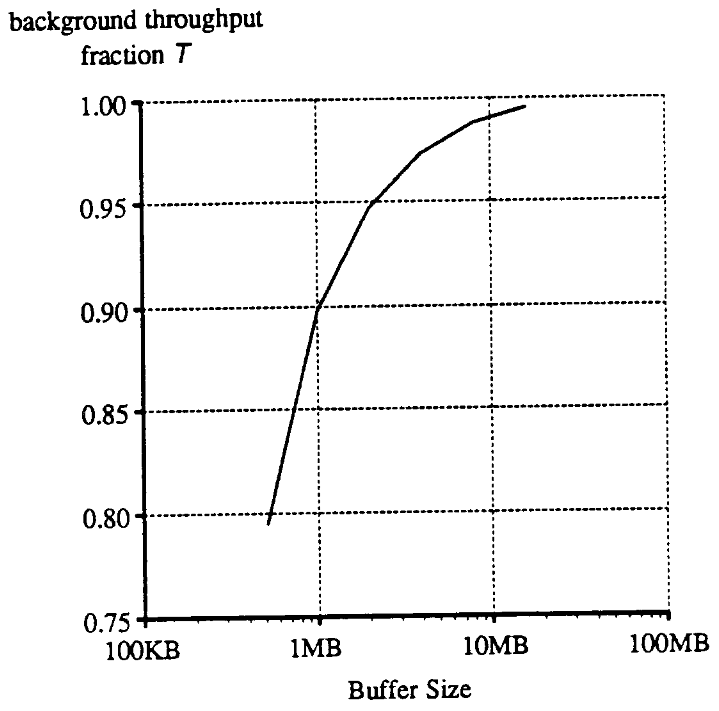

Figure 13: Throughput increases with increasing system buffer size. The graph plots the maximum background throughput at different buffer sizes. Real-time workload consists of three 1.4 Mbps sessions.

6.5. Session Startup Time

To study startup time for new sessions, we ran a simulation in which requests for six read sessions[2] arrive at time zero. Figure 14 shows the start times of the sessions. It can be seen that the difference between successive start times progressively increases; this is because the workaheads of all previously started sessions' have to be augmented to the new minimal WAS. When several requests arrive simultaneously, startup times are on the order of one second. This is similar to the startup time of a consumer VCR. However, it is too large for applications that require instantaneous response, such as interactive musical performance using sounds stored on disk (however, this problem can be solved by storing an initial segment of each sound file in memory).

7. RELATED WORK

Structural issues for multi-media files (sharing, parallel composition, annotations, *etc.*) have been addressed in the Xerox Etherphone system [11], the Sun Multimedia File System [10], and the Northwestern Network Sound System [9]. These projects do not concentrate on performance or scheduling issues, and the systems cannot make

[2] Write sessions can usually be started immediately; see Section 2.2.

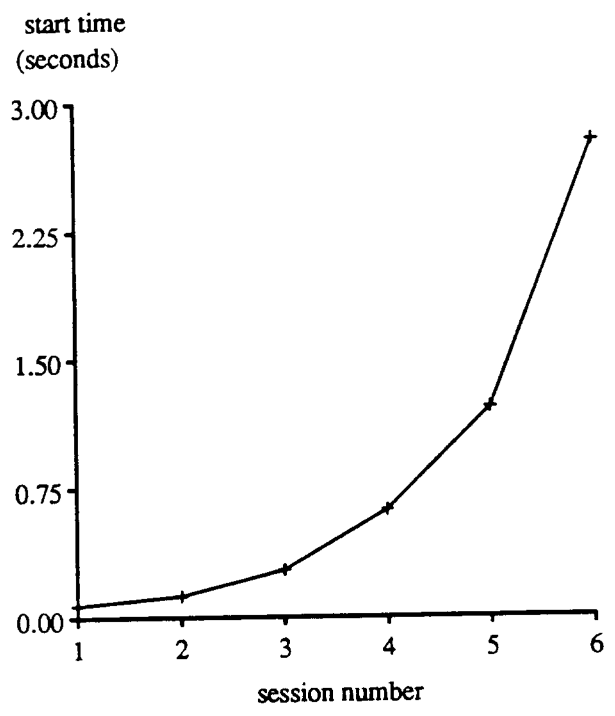

Figure 14: When six 1.4 Mpbs session requests arrive simultaneously at time zero, their actual start times are staggered as shown.

performance guarantees.

Other projects have addressed performance but without hard guarantees. Abbott gives a qualitative discussion of disk scheduling for playback of multiple audio tracks [1]. He compares a "balanced" policy in which read-ahead is divided among sessions, to a shortest-seek-first policy. His analysis does not, however, provide an acceptance test or performance guarantees. Rangan *et al.* [8] describe a prototype file system for storing and retrieving CM streams. Their file system supports the storage and retrieval of a single data stream only, concurrent non-real-time access to the disk is not permitted, and no performance guarantees are made.

Park and English [6] describe a system supporting single channel audio playback. Non-real-time traffic may concurrently access the disk, causing available disk bandwidth to change. As an alternative to disk bandwidth reservation for the audio channel, they propose changing the data rate of the channel dynamically, to accommodate non-real-time workload. The high data rate is chosen if the workahead on the stream is above a fixed threshold. This strategy does not guarantee a minimum data rate.

Yu *et al.* [12] discuss the layout of interleaved data streams with different data rates on a compact disk for guaranteed-performance playback. Their assumptions (single session, fixed rates, small buffers, no non-real-time traffic) are more restrictive than ours.

Gemmell and Christodoulakis [3] describe a file system supporting multiple audio channel playback. Non-real-time traffic may concurrently access the disk. Like CMFS, this work provides a basis for hard performance guarantees. However, it differs from CMFS in several respects. The channels must have the same (constant) data rate and must start at the same time. The scheduling policy is static: the system repeatedly applies a single feasible WAS for the audio channels, and reserves "free" time during each operation sequence to service non-real-time traffic. For non-real-time traffic, this static policy may perform worse than CMFS because: CMFS can "interrupt" a WAS, allowing non-real-time traffic to start immediately; CMFS can use accumulated system slack to handle long bursts of non-real-time traffic.

8. CONCLUSION

The Continuous Media File System (CMFS) provides guaranteed-performance read and write "sessions". Several such sessions can coexist with each other and with non-real-time traffic on the same disk. We have described the design of CMFS, and have discussed its performance. The central ideas of CMFS include the following:

- *Semantics:* The temporal semantics of a CMFS session are defined rigorously (Section 2.1), but they include a factor $\bar{Y}$ that allows "slack" in the client CPU scheduling. The semantics support a range of client requirements, including variable-rate data, starting and stopping, synchronization of multiple streams, and client workahead.
- *Layout:* CMFS does not mandate a particular disk layout, but simply requires that bounds functions U and V can be obtained (Section 3). Depending on the usage, a static contiguous policy or dynamic UNIX-type policy might be used.
- *Session acceptance:* The acceptance test checks for the existence of a *feasible WAS* (Section 4.1) to decide if a new session can be accepted based on layout parameters and available buffer space. It adjusts the allocation of buffer space among existing sessions to optimize system performance.
- *Disk scheduling:* Several disk scheduling policies for real-time traffic are possible. We examined the Greedy and Cyclical Plan policies and their Aggressive variants. These policies are all significantly better than the Static/Minimal policy, and there was little difference among them. The performance of all traffic types improves, up to a point, with increasing buffer space.
- *Concurrent non-real-time access:* CMFS handles non-real-time as well as real-time file access. Disk space can be dynamically used for various purposes; there is no need for separate disks or partitions for real-time files. CMFS uses the notion of *slack time* to decide when non-real-time traffic can be handled. The *slack time hysteresis* policy maintains high slack time and allows long non-real-time operations to complete without interruption.

8.1. Refinements and Future Work

The following observations suggest possible improvements to CMFS. First, for a session S_i, the graph of workahead W_i as a function of time is roughly a "sawtooth" function. If we consider two sessions S_1 and S_2 that have opposite phases in the scheduling cycle, then $\max(W_1 + W_2)$ is generally less than $\max(W_1) + \max(W_2)$. S_1 and S_2 can therefore share buffer space, possibly improving non-real-time performance or increasing the number of sessions that can be accepted. Second, the scheduling policy could take disk head position into account in various ways. For example, it could yield a session ordering that is more efficient than smallest-workahead-first (Section 5.1).

Similarly, the use of a policy such as SCAN [2] for ordering non-real-time operations could improve their performance.

File layout policies that allow variable-size (extendable) real-time files should be investigated. The goal of such a policy is to provide bounds functions *U* and *V* that are close to those of the contiguous policy, while allowing high utilization of disk space. The overhead of allocation map and index block I/O must be considered. Dynamic compaction of the disk, to make contiguous space available, might be desirable for such a policy.

Although we have presented the CMFS algorithms in the context of a single-spindle disk drive, they are equally applicable to a disk array in which files are "striped" across multiple disks [7]. A client-level session could be composed of sessions on multiple disks, with each disk reserved and scheduled as described here. This could be used to provide sessions with data rates higher than those of the underlying disk drives. It could have benefits such as load-balancing and increased availability even for sessions with data rates lower than individual disks.

ACKNOWLEDGEMENTS

Discussions with Vassilios Polimenis, George Homsy, and Mark Moran contributed to the CMFS design.

REFERENCES

1. C. Abbott, "Efficient Editing of Digital Sound on Disk", *J. Audio Eng. Soc. 32*, 6 (June 1984), 394.
2. P. J. Denning, "Effects of Scheduling on File Memory Operations", *Proceedings of the AFIPS National Computer Conf. Proc. Spring Joint Computer Conference*, 1967, 9-21.
3. J. Gemmell and S. Christodoulakis, "Principles of Delay Sensitive Multi-media Data Storage and Retrieval", *ACM TOIS (to appear)*, 1991.
4. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *J. ACM 20*, 1 (1973), 47-61.
5. 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.
6. A. Park and P. English, "A Variable Rate Strategy for Retrieving Audio Data From Secondary Storage", *Proceedings of the International Conference on Multimedia Information Systems*, Singapore, Jan. 1991, 135-146.
7. D. Patterson, G. Gibson and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *ACM SIGMOD 88*, Chicago, June 1988, 109-116.
8. P. V. Rangan, W. A. Burkhard, R. W. Bowdidge, H. M. Vin, J. W. Lindwall, K. Chan, I. A. Aaberg, L. M. Yamamoto and I. G. Harris, "A Testbed For Managing Digital Video and Audio Storage", *Proceedings of the 1991 Summer USENIX Conference*, Nashville, TN, June 10-14, 1991, 199-208.

9. J. M. Roth, G. S. Kendall and S. L. Decker, "A Network Sound System for UNIX", *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, Aug. 19-22, 1985, 61-67.

10. D. Steinberg and T. Learmont, "The Multimedia File System", *Proc. 1989 International Computer Music Conference*, Columbus, Ohio, Nov. 2-3, 1989, 307-311.

11. D. B. Terry and D. C. Swinehart, "Managing Stored Voice in the Etherphone System", *Trans. Computer Systems 6*, 1 (Feb. 1988), 3-27.

12. 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.