High Speed Networking and Multimedia Computing, IS&T/SPIE Symp. on Elec. Imaging Sci. & Tech., San Jose, CA, February 1994.

MPEG Video in Software: Representation, Transmission, and Playback

Lawrence A. Rowe, Ketan D. Patel, Brian C Smith, and Kim Liu
Computer Science Division - EECS
University of California
Berkeley, CA 94720
(rowe@cs.berkeley.edu)

Abstract

A software decoder for MPEG-1 video was integrated into a continuous media playback system that supports synchronized playing of audio and video data stored on a file server. The MPEG-1 video playback system supports forward and backward play at variable speeds and random positioning. Sending and receiving side heuristics are described that adapt to frame drops due to network load and the available decoding capacity of the client workstation.

A series of experiments show that the playback system adds a small overhead to the stand alone software decoder and that playback is smooth when all frames or very few frames can be decoded. Between these extremes, the system behaves reasonably but can still be improved.

1.0 Introduction

As processor speed increases, real-time software decoding of compressed video is possible. We developed a portable software MPEG-1 video decoder that can play small-sized videos (e.g., 160 x 120) in real-time and medium-sized videos within a factor of two of real-time on current workstations [1]. We also developed a system to deliver and play synchronized continuous media streams (e.g., audio, video, images, animation, etc.) on a network [2].Initially, this system supported 8kHz 8-bit audio and hardware-assisted motion JPEG compressed video streams. This paper describes the integration of a software decoded MPEG-1 video stream into this system.

The Continuous Media (CM) System provides full-function VCR commands, including forward and backward play at variable speeds and random positioning in the stream. It also resamples the audio stream so that sound is played at any speed. A best-effort model dynamically optimizes the playback frame rate within the constraints of the video file server and client machine CPU speeds and network bandwidth. The software MPEG-1 video decoding component of the CM System estimates resource needs, such as the CPU time required to decode a frame of video, and allocates available resources among various tasks, such as reading data from the network and decoding, dithering, and displaying frames.

The file representation of an MPEG-1 video stream is augmented with a frame index and a header that contains information needed by the CM System to play the stream (e.g., image size, frame rate, etc.). This representation is constructed by pre-parsing the video stream when it is loaded into a video file server.

This paper describes the MPEG-1 video file representation and the mechanisms required to support full-function playback. The remainder of the paper is organized as follows. Section 2 describes the software architecture of the system. Section 3 describes the file representation. Section 4 describes the mechanisms implemented for transmission and reception of video frames. Section 5 describes the adaptive rate control mechanisms. And, section 6 presents the results of experiments performed to evaluate the system.

2.0 CM System Architecture

The CM System is composed of programs built with a collection of libraries called the *CM Toolkit* (CMT). The *Continuous Media Player* (CMPlayer) is a video playback application implemented using CMT. Figure 1 shows the software organization of this application. The CMX process runs on the client workstation. It plays audio and video packets sent to it by the *CM Source* (CMS) process that runs on a video file server. CMX decodes the packets and schedules them for output to the audio device or X server. It communicates with the X server using shared memory. CMPlayer, CMX, and CMS processes are synchronized through a distributed, replicated object called a *Logical Time System* (LTS). The distributed objects are synchronized by synchronizing the system clocks on the processors using the Internet Network Time Protocol.

The CMPlayer establishes a connection to a local CMX process and instructs it to play video clips provided by a particular CMS. The local CMX connects to the appropriate CM, opens the clips, and creates an LTS to synchronize delivery of the clips. The LTS maps system time (i.e. the system clock) to logical playback time. Logical time is the current position within a

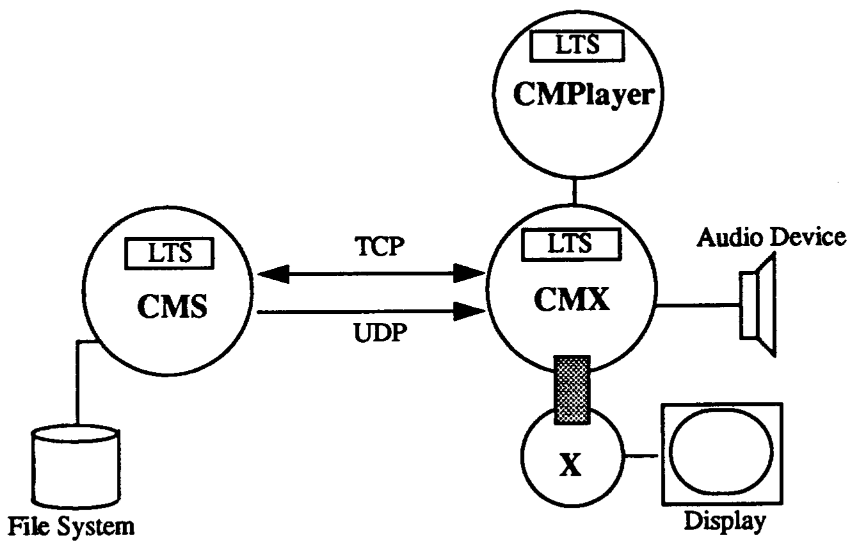

FIGURE 1. Software Organization of CMPlayer

stream. For example, if a 10 second video clip is being played, the beginning of the stream corresponds to logical time 0 and the end of the stream corresponds to logical time 10. The mapping is:

lts = *speed* * (SystemTime - *offset)*

For example, playing a video clip from the beginning at normal rate corresponds to setting *speed* to 1.0 and *offset* to 0.0. Random access to the 7th second of the video clip corresponds to setting *speed* to 0.0 and *offset* to 7.0. User requests to play, stop, or move to another stream position are implemented by setting the *speed* and *offset* slots in the LTS in the CMPlayer. These slot changes are implicitly distributed to the LTS objects in CMX and CMS which causes the requested action to happen.

The CMS and CMX processes exchange control information through a virtual circuit connection (TCP). Continuous media data is transmitted from CMS to CMX through a datagram connection (UDP). CMX requests retransmissions if a packet is lost, and it deals with data that arrives out of order since UDP provides no guarantees of delivery.

For each media type, a set of procedures within CMS sends the appropriate data at the correct time. This set of procedures is called the *MediaSource*. Data is sent in response to an LTS value change. Each data frame is marked with a logical start and end time which defines a period during which the frame is valid for display. A set of procedures in CMX, called the *MediaDest*, receives, decodes and displays the data. CMX is responsible for displaying the data at the correct time.

Both CMS and CMX can make independent decisions to skip or drop frames to maintain synchronization with logical time. In addition, CMX sends rate control information to CMS to vary the rate at which frames are sent if it notices that the network is congested.

CM data is stored on disk in a clipfile. A clipfile is composed of a header with type-specific information (e.g., width and height in the case of a video clip or samples/second in the case of an audio clip), the actual data, and a set of tables describing frame specific information such as offset into the file. The exact format of a clipfile is dependent on the media type.

The CMPlayer uses scripts to synchronize playback of video and audio. Each script contains a list of video and audio clips to play. Each clip list specifies the hostname of the file server containing the clips. For each clip, a range within the clipfile is specified. For example, Figure 2 shows a script that plays frames 100 through 800 in clipfile *A* followed by frames 300 to 1000 in clipfile *B*, and finally frames 1000 to 1500 in clipfile *A*. The script also specifies that all of audio clipfile *C* is to be played.

```
#%!CM-Script-1.0
... header information (e.g., video size, frame length, etc.)
set videoClipList {
        host1.berkeley.edu
                {/data/video/A 100 800}
                {/data/video/B 300 1000}
                {/data/video/A 1000 1500}
}
set audioClipList {
        host2.berkeley.edu
                {/data/audio/C 0 end}
}
```

FIGURE 2. Example Script

Notice that in the example script, two different parts of video clipfile *A* are played at different times and that the hosts for the video and audio clip lists can be different. We have used the system to play videos over local and wide area networks (e.g., ethernet, FDDI, and Internet), and we have played video stored at a remote file server (UCSD) and synchronized it with audio on a local file server (UCB).

3.0 MPEG-1 Video Clipfile Format

Figure 3 shows the structure of an MPEG-1 video clipfile. The fixed size header at the beginning of the clipfile contains the following information:

MPEG-1 clipfile magic number
image width and height
frames per second (typically 30 fps)
total number of frames in the file
offset to the end of the MPEG data

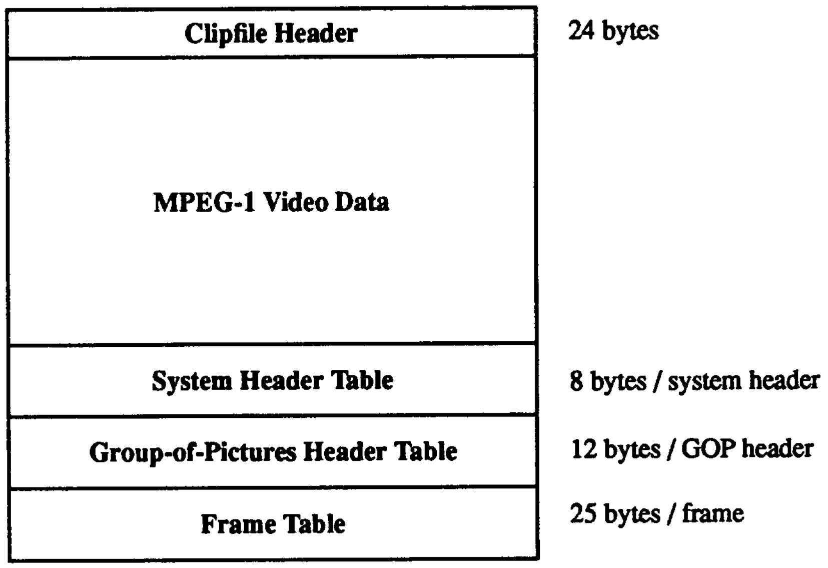

FIGURE 3. MPEG-1 Video Clipfile Format

The magic number specifies the file type. The width and height are gotten from the MPEG video stream for historical reasons. Currently this information is not used by either CMS or CMX. The offset to the end of the MPEG data is used by CMS to quickly find and read the tables of information found at the end of the file.

The MPEG-1 video data is a standard MPEG-1 video stream. The clipfile representation acts as a wrapper around the original MPEG-1 video stream. We chose this organization so the original stream representation could be easily extracted from an MPEG-1 video clipfile.

The system header table is composed of an entry for each system header in the MPEG data. Each entry contains a byte offset from the beginning of the clipfile and the length of the header. The group-of-pictures (GOP) header table is similar but also includes the index of the system header on which the GOP header depends.

The frame table is composed of an entry for each frame in the file. Each entry contains the index of the frame's system header, the index of the frame's GOP header, the byte offset to the frame from the beginning of the clipfile, the size of the frame, the frame type (either I, P, or B), and indices of the frame's past and future reference frames. For I and P frames that do not have a future reference frame, the index of the next I frame is stored instead. In the case of I frames which do not have a past reference, the index of the previous I frame is stored.

In total, the file representation adds 24 bytes for the clipfile header, 8 bytes for each system header, 12 bytes for each GOP header, and 25 bytes for each frame. The clipfile header, the system and GOP header tables, and the frame table are loaded into main memory by CMS when it opens the file. These data structures are used to quickly seek into the MPEG data for transmission of a particular system or GOP header or frame.

4.0 Transmitting and Receiving Frames

This section describes the heuristics for playing an MPEG-1 video stream on heterogeneous clients and networks with varying transmission bandwidth.

4.1 Transmitting frames

MPEGSource is the module within CMS responsible for sending MPEG-1 video data to CMX. It opens the clipfile, creates and allocates structures, and binds the distributed LTS object to the source.The LTS is distributed to the CMS when CMX requested it to open the clipfile. A timer event is created to notify *MPEGSource* when to send the next frame.

When *MPEGSource* is notified to send a frame by the timer event, the following steps are taken:

1. Decide which frame to send. The LTS maps the current system time plus a small prefetch delay (typically 0.5 seconds) to logical time. The frame number that should be displayed at that logical time is calculated.
2. Check reference frames. *MPEGSource* keeps a cache of the last two I or P frames sent to CMX. If the frame being sent depends on reference frames (i.e., a B or P frame), it checks to make sure the reference frames have been sent. If the reference frames are not cached, it sends the reference frame rather than the frame originally considered. Since frame dependencies have been pre-computed, these checks are very fast.
3. Send system and GOP headers. The last system and GOP header indices are also stored. If the last system and GOP headers sent are not the ones needed by the frame being sent, these headers are sent.
4. Calculate start and end times for display. The start display time is calculated from the frame number. The end display time calculation depends on the frame type. For B-frames, the end time is simply the start time plus 1/fps (i.e., the logical duration of one frame given the current frame rate). For I- and P-frames, the end time is simply the start time plus the time until the next I-frame. Since all subsequent P- and B-frames will depend on the I- or P-frame in question, CMX considers the frame valid until the next I-frame.

This algorithm prioritizes I-frames above P-frames and P-frames above B-frames. Since *MPEGSource* keeps track of the last two reference frames sent, it will not send a frame unless all information needed to decode it has been sent. As resources become constrained and frame rate is reduced, *MPEGSource* will drop B-frames first, then P-frames, and finally I-frames.

4.2 Receiving and decoding frames

MPEGDest is the module within CMX that receives, decodes, and displays frames sent to it by a CMS. *MPEGDest* uses an ordered dither to dither the 24-bit images into a 7-bit color space before it is displayed. The cost of dithering each frame is considered as part of the decoding process. Since decoding is CPU intensive, *MPEGDest* must decide whether a frame can be decoded and displayed before its logical end time. If not, it drops the frame.

To allow other parts of CMX to do work (e.g., decoding and displaying other media streams, reading data from the network, etc.), *MPEGDest* decodes a part of a frame at a time, and then puts itself at the end of the event queue. Each time *MPEGDest* is allowed to decode a frame, it times itself and updates a static variable with a weighted average of the number of macroblocks per second (mbs) it is able to decode. This value is used to decide how long it will take to decode a frame.

As frames are received by *MPEGDest*, they are put on a queue of decode requests. *MPEGDest* decides which frame to decode using the following algorithm:

1. If a frame depends on a frame that has not yet been received and decoded, it is not considered and the next frame in the queue is evaluated. This step allows frames to be received out of order.
2. If the current logical time is already past the end time of the frame, it is dropped.
3. If the current logical time plus the estimated time to decode the frame is less than the end time for the frame, it is selected for decoding.
4. If the current logical time plus the estimated time to decode the frame is not less than the end time for the frame and no other frames exist on the queue, it is selected for decoding.
5. If the current logical time plus the estimated time to decode the frame is not less than the end time for the frame and another frame exists on the queue, the next frame on the queue is evaluated. If MPEGDest estimates that the next frame will finish before its end time, the first frame is dropped and the next frame is selected for decoding.
6. If the current logical time plus the estimated time to decode the frame is not less than the end time for the frame, the frame is a B-frame, and the next frame on the queue is a P- or I-frame, the first frame is dropped and the next frame is selected for decoding.
7. Similarly, if the current frame does not have time to finish, the frame is a P-frame, and the next frame is an I-frame, the first frame is dropped and the next frame is selected for decoding.

Once frame decoding begins, it continues to completion, and the frame is displayed regardless of whether the end time has already passed. Although displaying the frame may be out of synch with logical time, this policy avoids wasting CPU resources decoding frames that will never be displayed. It also updates the display to be a frame closer to the correct frame than was previously displayed. The success of this policy depends on being able to estimate the time required to decode a frame before decoding begins.

4.3 Playing Backwards

Playing backwards is difficult, because an arbitrary number of frames may have to be decoded before the desired frame can be decoded. For example, given the following frame sequence and logical time at frame 10:

	I	B	B	P	B	B	P	B	B	I
frame #:	1	2	3	4	5	6	7	8	9	10

Frames 1, 4, and 7 must be decoded before frame 9 can be decoded.

CMS and CMX see no logical distinction between playing forwards and backwards. CMS receives an LTS event to send a frame. It knows the frame to be sent, its type, and the reference frames cached in CMX. It can calculate what frame should be sent and sends it. Similarly, CMX has a queue of frames to decode and display. The only difference is that the clock runs backwards (i.e., *speed* < 0). Consequently, the mechanisms described above also work for backwards play.

In deciding which frame to send, *MPEGSource* will recursively send reference frames on which other frames depend until it finds an I-frame. The next time it attempts to send a frame, a similar recursion will occur until the P-frame that depends on the I-frame is reached. This selection process continues until it reaches a point in the stream where it has already sent the requisite reference frames. Unfortunately, this creates a bursty play rate since it must send all the I- and P-frames at the beginning of a sequence before sending frames that depend on them. When playing backwards through such a sequence, the frames that actually get sent are not scheduled for display until some time into the future, creating a latency period. As we progress backwards toward the beginning of the sequence, we catch up to the logical display times of the frames already sent which results in a flurry of already decoded frames being displayed.

We could further optimize backwards play by keeping more state about reference frames already sent. Currently, information about only the last two reference frames sent is kept. When playing backwards, however, frames often depend recursively on many different reference frames. Increasing the amount of state information kept would avoid retransmitting and redecoding reference frames.

4.4 Implementing Random Access

Random access implies seeking to an arbitrary logical time with *speed* equal to 0. Random access is implemented with the same mechanisms described above with only a slight difference. Because logical time is not advancing, the end display time is infinite so all frames sent will be decoded. *MPEGSource* sends the required reference frames and the frame corresponding to the logical time. *MPEGDest* decodes the frames in proper decode order ending with the frame corresponding to the logical time originally requested.

4.5 Rate control mechanisms

Because MPEG software decoding is CPU intensive, the bottleneck for playback is often the rate at which CMX is able to decode a frame. Although CMX receiving mechanisms allow it to drop frames that will never be decoded in time, a preferred solution would decrease the frames per second being sent in the first place [2]. The heuristic used to control frame transmission periodically checks the difference between frames decoded and frames received. Currently, this check occurs approximately every 0.25 seconds. If the number of frames received is greater than the number of frames decoded, CMX instructs CMS that the current frame rate is too high. If the number of frames received is less than or equal to the number of frames decoded and the current frame rate is less than the ideal frame rate, CMX instructs CMS that the frame rate can be increased. In this way, CMX dynamically controls the rate at which frames are being sent. Ideally, this heuristic will prevent CMS from sending data that will never be used.

5.0 Experimental Results

We ran a series of experiments to investigate how well these heuristics work. The system was tested with CMS on a SPARC 1+ with a local SCSI disk and CMX on a SPARC 10. Logging procedures recorded control, transmission, and decoding events. The system was configured to play an MPEG stream at increasing frame rates to test the frame dropping and rate control heuristics. Specifically, we were interested in answering three questions:

1. How much overhead is incurred by the CM System?
2. How well do the rate control heuristics estimate the optimal playback frame rate?
3. How uniform are frame drops among frames of equal value?

5.1 Overhead of the system

The overhead of the CM System was measured by comparing playback using the CM System with playback using the Berkeley MPEG-1 video viewer that reads data from a local file, decodes each frame, and displays it as fast as possible.

Two different MPEG streams were used for testing. The characteristics of the two streams are given in the following table.

TABLE 1. Characteristics of test streams

Name	Size	# Frames	File Size	Compression
canyon.mpg	144 x 112	1758	1744060	49:1
flowg.mpg	320 x 240	148	690185	49:1

Figures 4 and 5 show graphs of the frames per second (fps) played by the CM System at different target frame rates and the performance of the simple MPEG decoder. The CM system overhead is the difference between the two lines in the graphs. As target frame rate increases, overhead increases as the frame rate control and frame dropping heuristics require more CPU resources. For both streams, playback rates flatten as the target rate increases which suggests that overhead in the system reaches an upper bound.

The difference between the CM System and the MPEG viewer can be expressed as a percentage of the playback rate of the MPEG viewer. For the canyon stream, overhead ranges from 7% at 24 fps to 28% at 60 fps. The larger flower garden stream overhead ranges from 20% at 5 fps to 45% at 30 fps. Although 45% is a very high overhead, it is high because the Sparc 10 is relatively slow as a decoder. When this example is run on an HP 750, the percentage overhead is below 20%. Hence, the CM system overhead is about 20%. This percentage can be improved by more tuning of the code that implements the heuristics.

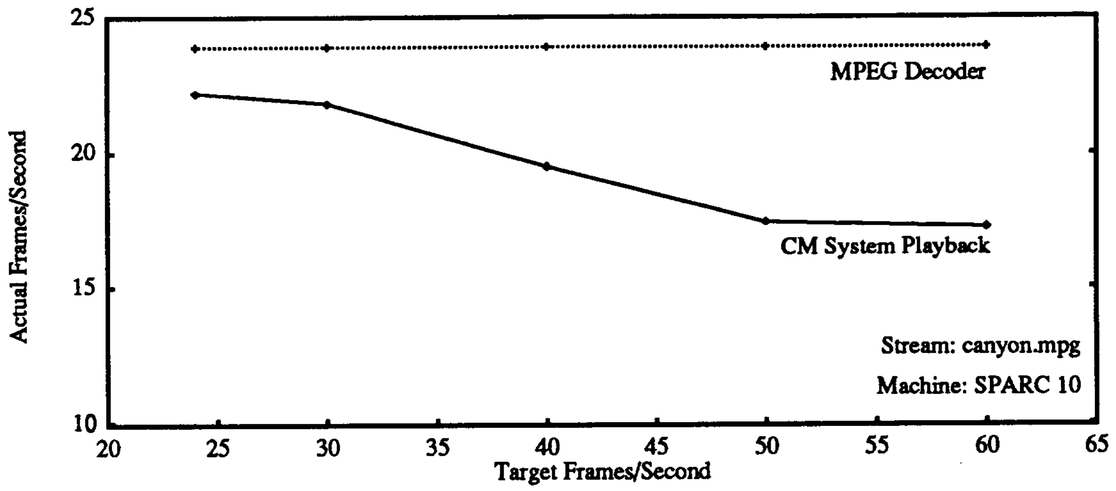

FIGURE 4. Overhead in CM System: canyon.mpg

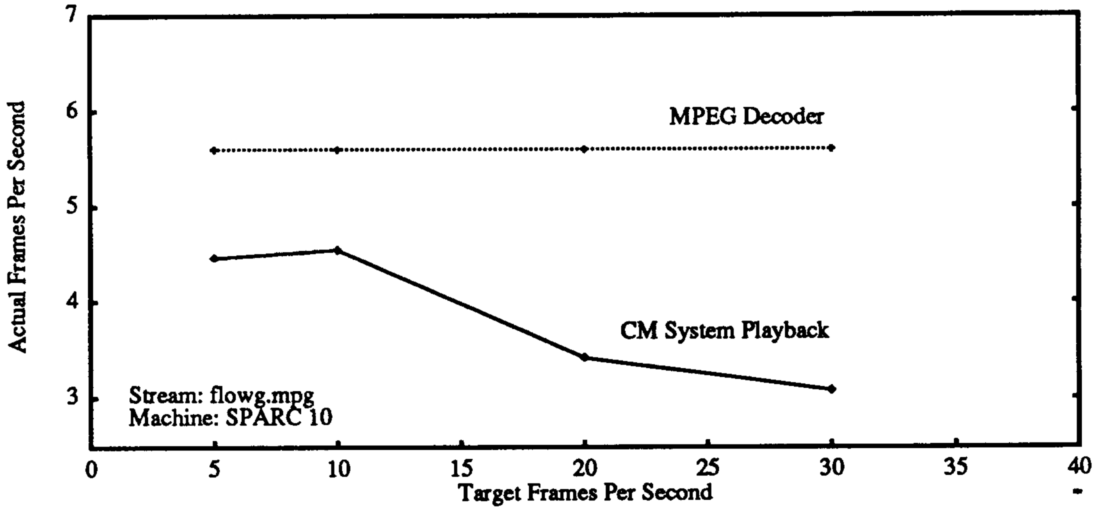

FIGURE 5. Overhead in CM System: flowg.mpg

5.2 Heuristic Performance

Two sets of heuristics control playback frame rate. One set controls frame sending within CMS and the other controls frame decoding within CMX. Our second experiment attempts to assess the impact of these heuristics. Exploring the mechanism on the send side first, Figures 6 and 7 show graphs plotting frames sent and frames actually played versus increasing target frame rates for the two different streams. Both graphs show that some frames sent are not displayed. As target frame rates exceed decoding capacity, the source side continues to send frames that can not be decoded. At some threshold, however, the number of frames sent is reduced to a more reasonable level.

The difference between frames sent and frames played is caused by several factors. First, the rate control does not calculate an optimal frame rate. Instead, adaptive frame rate control is done incrementally by slowing down the frame rate if more frames are being received than played and speeding up the frame rate if more frames are being played than being received. These mechanisms eventually bounce between slightly too fast where some frames are dropped and slightly too slow where no

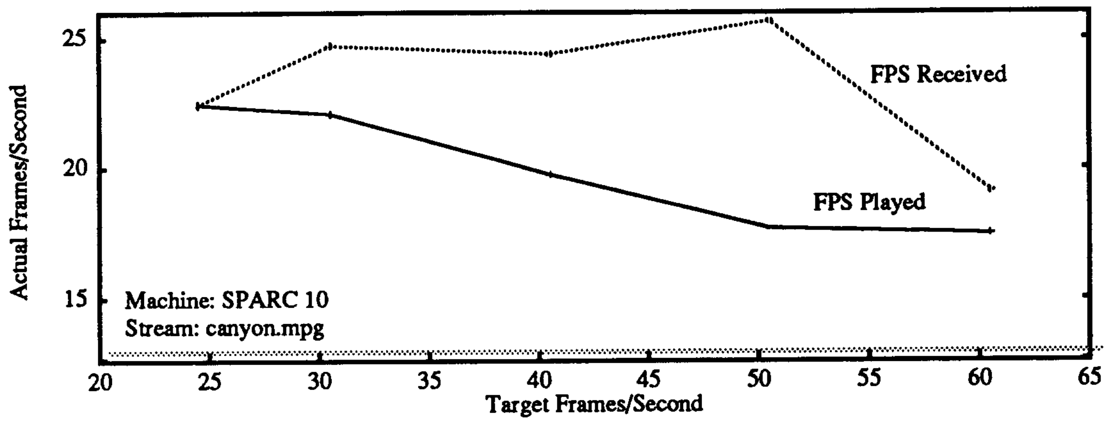

FIGURE 6. Frames Received and Played: canyon.mpg

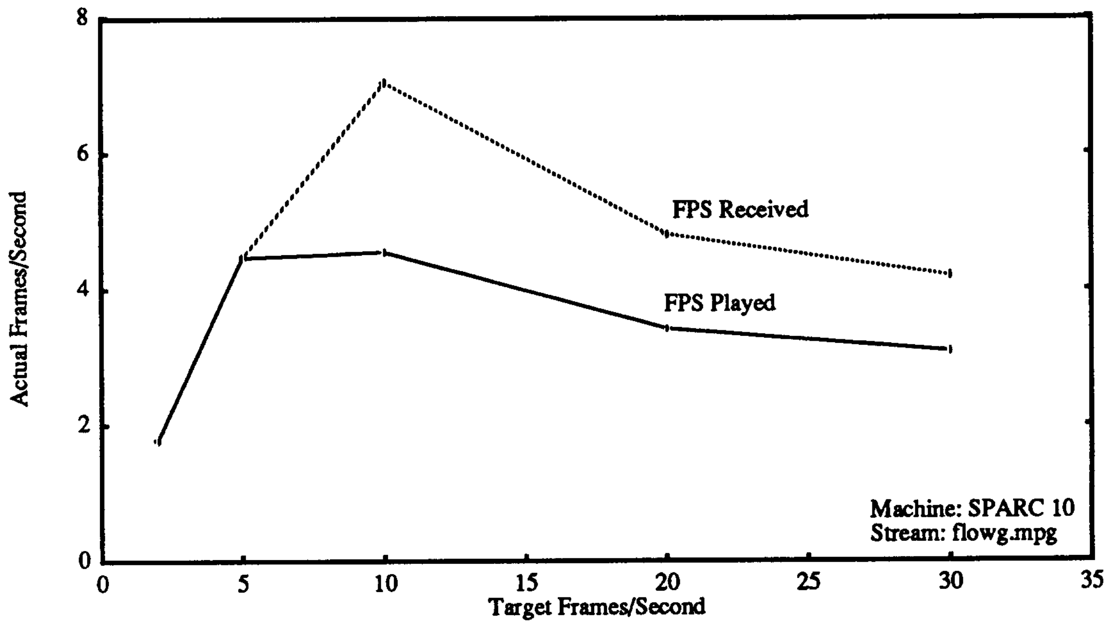

FIGURE 7. Frames Received and Played: flowg.mpg

frames are dropped. The resulting number of frames sent will be a little higher than the number of frames decoded and played. One way to improve the system is to look ahead in the stream and drop frames to allow time to decode large frames in the future (i.e., frames that will take a long time to decode) earlier in time so they are less likely to be late.

Second, as the target frame rate is increased, the sending frame rate negotiated between CMS and CMX which is limited by decoding ability becomes a smaller percentage of the target frame rate. Eventually, depending on the encoding pattern of the MPEG stream, CMS will begin to drop frames because the requisite reference frames have not been sent. This effect can be seen as the drop in frames sent between 10 fps and 20 fps for the flower garden stream and 50 fps and 60 fps for the canyon stream.

A more accurate measure of the effectiveness of the frame rate controls can be shown by examining the percentage of bytes sent that are actually used. Since I- and P-frames are much larger than B-frames, sending B-frames that are dropped is less consequential than sending I- or P-frames that are dropped. Figure 8 shows a graph plotting the percentage of bytes sent that were actually decoded and played versus increasing target frame rates for both MPEG stream. This graph reveals that byte utilization remains very high and relatively flat even as the target frame rate becomes exceedingly large relative to decoding ability. Consequently, we believe the CMS rate control mechanisms work well.

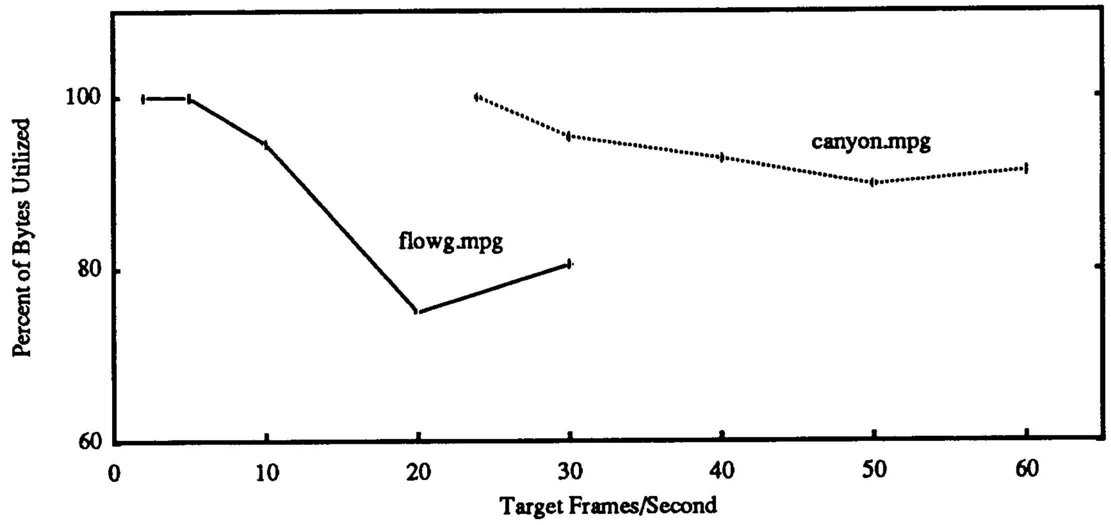

FIGURE 8. Byte Utilization: canyon.mpg and flowg.mpg

The destination side mechanisms are responsible for choosing which frames to decode and play. Figures 9 and 10 show graphs plotting frames received and frames played by type versus increasing target frame rate for each stream. B-frames are dropped exclusively until the target frame rate is twice the decoding ability. Even at this point, few P-frames are dropped while the number of B-frames dropped continues to increase until no B-frames are being played at all. After this point P-frame drops begin to increase. For all values of target frame rate, I-frames are not dropped at all. This behavior matches our design goal.

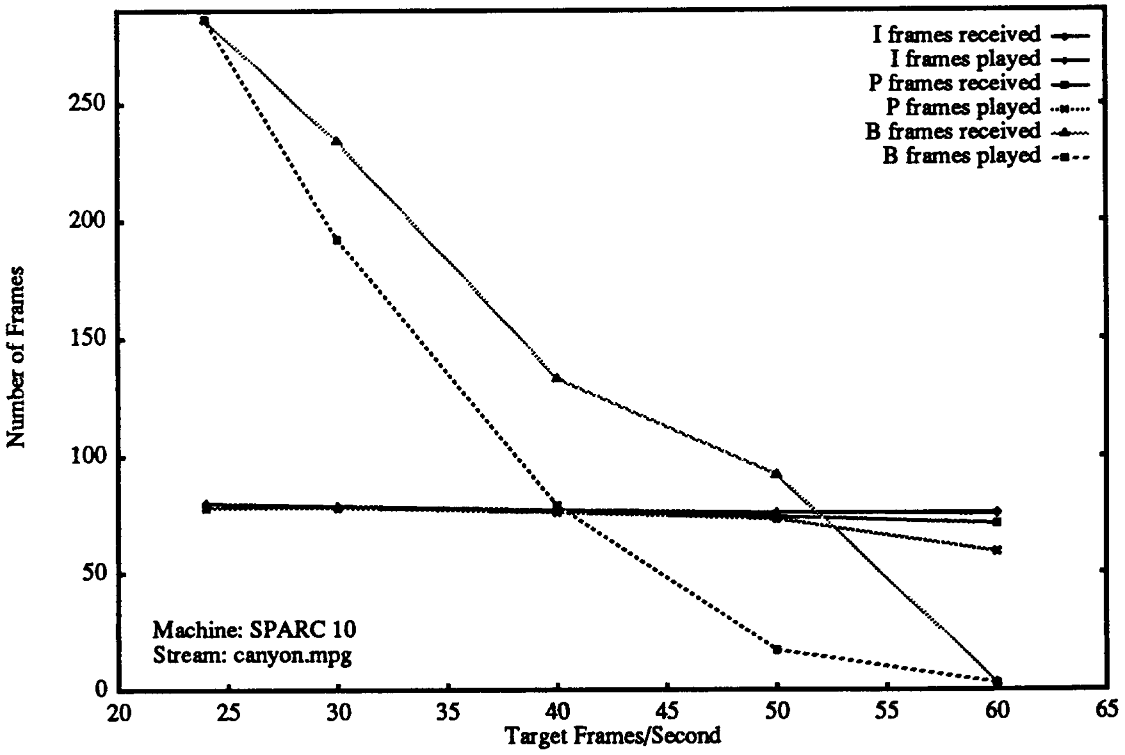

FIGURE 9. Frames received and played by type: canyon.mpg

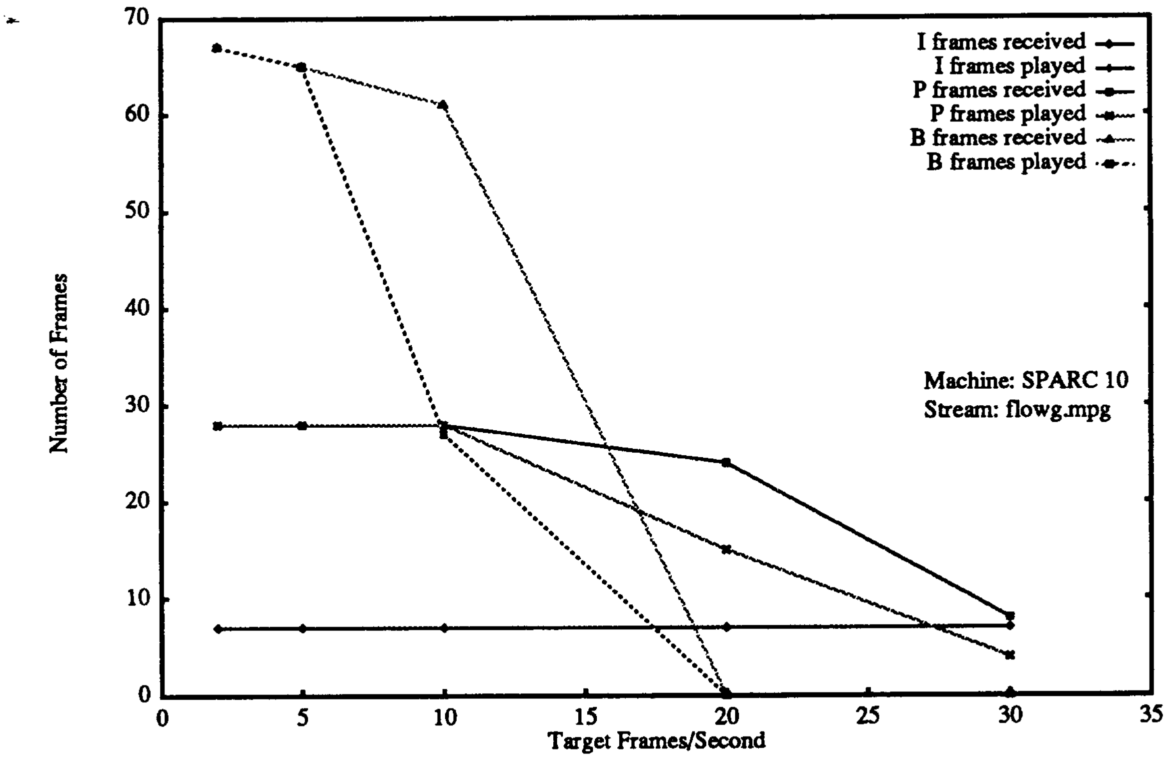

FIGURE 10. Frames received and played by type: flowg.mpg

5.3 Smoothness of Play

The third question we investigated deals with frame drops among frames of equal value. While it is important to play as many frames as possible, we believe smooth playback at a low frame rate is better than jerky playback at a high frame rate. In other words, a small standard deviation in jitter is better than a higher frame rate. To test the behavior of the heuristics in this case, we constructed an MPEG stream with a frame type pattern of an I-frame followed by 14 B-frames (i.e., no P-frames). Using this stream, we collected frame drop information as target frame rates exceeded decoding capacity. Figures 11 through 13 show patterns of frames received, decoded, and played by position within the recurring 15 frame pattern. Position 0 represents an I-frame and positions 1 through 14 represent B-frames.

Pattern:	I	B	B	B	B	B	B	B	B	B	B	B	B	B	B
% playback															
67%	I	B	B	B	B	B	B	B	B	B	B	B	B	B	B
9%	I		B	B	B	B	B	B	B	B	B	B	B	B	B
5%	I			B	B	B	B	B	B	B	B	B	B	B	B

FIGURE 11. Playback frame pattern: 10 frames/second

Pattern:	I	B	B	B	B	B	B	B	B	B	B	B	B	B	B
% playback															
10%	I								B						
8%	I							B							
7%	I										B				
6%	I														
5%	I						B								
5%	I									B					

Figure 12. Playback frame pattern: 15 frames/second

Pattern:	I	B	B	B	B	B	B	B	B	B	B	B	B	B	B
% playback															
23%	I														
15%	I													B	
13%	I														B
11%	I									B					
8%	I								B						
8%	I										B				

Figure 13. Playback frame pattern: 20 frames/second

The 10 fps case represents frame dropping patterns as target frame rates begin to exceed decoding ability. The first frames to be dropped are in positions 1 and 2. This pattern is a result of the high priority of the I-frame at position 0. At 15 fps, the target rate exceeds the capacity of the decoder. On average, only 2 frames per pattern are played. Optimally, the heuristics should choose frames 0 and either frame 8 or 9. This, however, only occurs 18% of the time. At 20 fps, the system drops all B-frames in the pattern over 20% of the time, yielding periodic playback of I-frames. When two frames in a pattern are played, however, B-frames late in the pattern are played, creating a bursty playback. So, the heuristics work reasonably well, but they can be improved if more global information about the pattern is used.

6.0 Summary

This paper described the design and implementation of a software decoder for MPEG-1 video that will play synchronized audio and video across a network with varying bandwidth and variable speed decoding hardware. We are continuing to improve the heuristics used for adaptive frame rate control and backwards play. Our future plans are to explore the heuristics to support multiresolution video streams.

7.0 Acknowledgments

This research was supported by the National Science Foundation (Grant MIP 90-14940), Fujitsu Network Transmissions Systems, Inc., Hewlett-Packard Company, and Hitachi America, Ltd.

8.0 References

1. K. Patel, B.C.Smith, and L.A. Rowe, "Performance of a Software MPEG Video Decoder", *Proceedings ACM Multimedia 93*, Anaheim, CA, August 1993, pp. 75-82.
2. L.A. Rowe and B.C. Smith, "A Continuous Media Player", *Proceedings 3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, San Diego, CA, November 1992.