A Server-centric Streaming Model[1]

Jin Hwan Jeong, Chuck Yoo
Department of Computer Science and Engineering, Korea University
{jhjeong, hxy}@os.korea.ac.kr

Abstract

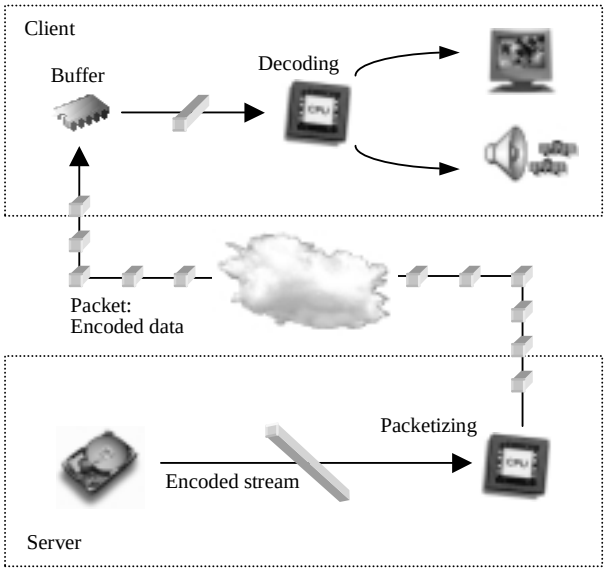
The current streaming technology is based on a model that a server sends encoded streams and that a client does decoding and rendering in real time. In this model, a client must have a powerful hardware and must have specific decoders to handle various compression algorithms (e.g. MPEG, H-263, etc.). This paper starts with a different assumption: network bandwidth is available so that it is no longer a bottleneck. Based on this assumption, we present a new streaming model where both the server and the client participate in the decoding process. The new model reduces the processing requirement at client to the level that a thin device with 486-class CPU (called Sun Ray 1) is able to play the full-size (640X480) NTSC video at 30 frame/sec. Furthermore, the decoding at client becomes independent of any compression algorithm so that Sun Ray 1 can play streams of various compression algorithms without specific decoders.*

1. Introduction

Advances in multimedia technology make it possible to store moving images in digital form. However, even with sophisticated encoding techniques, the storage requirement is too huge. For example, a movie of 15 minutes easily occupies about 150M bytes of disk space. It is difficult to carry such a large file, and it is obviously burdensome to store it in local disks of PC even though disk capacity improves constantly. Media streaming is getting popular and widely used because it removes the storage requirement. As depicted in Fig. 1, with media streaming, a server stores the media files and transmits encoded "streams" to clients that want to view. Then a client decodes the encoded stream and synchronizes the media types (e.g. audio and video) in order to play. The advantage of this streaming model is that a client does not need to store the received media files. It just buffers the media data in memory for a period of time, long enough to avoid network jitter and delay, and throw it away after playing.

1. This research was supported in part by Sun Microsystems Laboratories.
*Sun Ray 1 is the trademark of Sun Microsystems.

The assumption behind the streaming model is that the speed of Internet is slow. The bandwidth that a stream can use is limited to the slowest physical line between the server and client. It can be a part of congested backbone link or a dial-up link. By sending encoded streams, the bandwidth requirement is minimized, which makes the streaming model applicable to the Internet. However, the implication of this model is that the server merely pumps up the compressed files and the responsibility of handling the compressed stream is on clients. We can say that the streaming model is client-centric.



[Figure 1. Current streaming model]

This paper starts with a different assumption: network bandwidth is available so that it is no longer a bottleneck in streaming. As justification of the assumption, nowadays we see dramatic improvement in high-speed network technology. The gigabit and fast Ethernet become a commodity, and the new Internet backbone (such as Internet 2) is promised to be much faster than today's speed. Digital Subscriber Line (DSL) is expected to solve the last mile problem. This trend motivates us to explore a new streaming model, which is the goal of our paper.

Specifically, Fig. 2 is the envisioned model where the server does "intermediate" decoding of compressed

streams and transmits the intermediate data. The client decodes the intermediate data and plays it. The difference from the current streaming model is that the role of client is much simplified. We call this server-centric model. By applying the new model to the streaming of MPEG-1 data, we analyze the trade-off between the bandwidth and processing power in the server-centric model.

This paper is organized as follows. Section 2 describes the limitations of existing streaming software. The server-centric model is explained in details in Section 3. The intermediate data for MPEG-1 is defined, and the synchronization in the server-centric model is presented. Section 4 presents the implementation and the results of experiments. This paper concludes in Section 5.

2. Limitations of existing streaming software

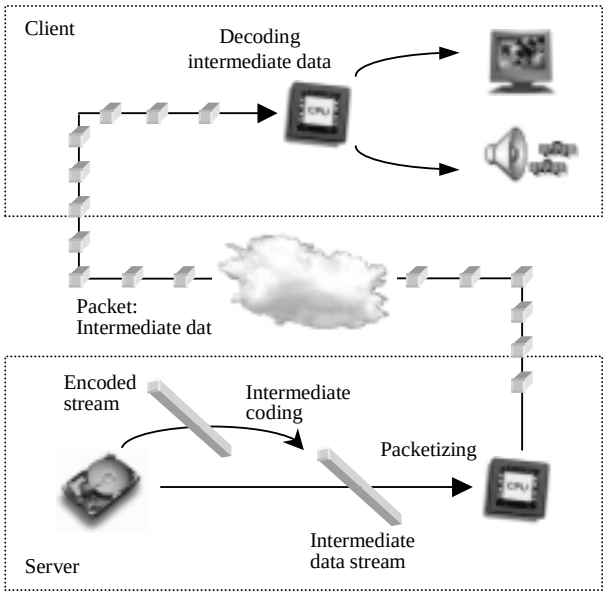
The most famous examples of the current streaming software are the Real Player of Real Networks [3], the Stream Works of Xing Technology [4], and the Media Player of Microsoft [5]. Although they support different encoding formats, the core algorithm is the same because they follow the client-centric model: the server sends encoded data in their own format that is highly compressed, and the client decodes the received data with a decoder that knows how to uncompress the format.

However, the client-centric model has a few disadvantages. First, it is difficult to simplify the client. It means that the client should have hardware powerful enough to process the streams in real time because the client has to do decoding and synchronizing and rendering all together. For software decoding, CPU should be fast enough. Even with the latest CPU, however, it is quite easy to see the whole CPU occupied with software-decoding of sophisticated compression algorithms such as MPEG-2. Therefore, most PDA or thin client cannot play the compressed streams due to the lack of CPU power. Second, extensibility - if a client wants to handle both of MPEG and H.263 streams, it should have two separate decoders because the encoding is different so that specific decoders are unavoidable in order to handle different media formats. Third, they are not able to take advantage of emerging high bandwidth networks because the streams are typically encoded at low bit rate (56Kbps), which is targeted for dial-up connection. When a server does streaming with 56K-encoded files, the quality of rendered images is the same regardless of the network bandwidth of the connection. To resolve this problem, a server has several versions of file with different encoding rates or has a layered media file [10]. However, these approaches make it difficult to maintain the server and have difficulties in handling layered media files properly with the dynamically-changing network traffic.

3. Server-centric streaming model

3.1 Intermediate decoding

Our goal is to overcome the disadvantages of the client-centric streaming model. First, we want to have a streaming model that can be applied to "thin" and portable devices as well as "fat" devices. We will see more and more of various types of thin and portable devices, and thus the streaming need for such devices is quite clear. The client-centric streaming model has the orientation for fat devices. Second, even for fat devices, it is more desirable to have a generic decoder than having a decoder for each media data format. There are already many different media data formats and there will be new ones to come. Therefore, we want to have a generic decoder to cover the existing media formats and future ones if possible.

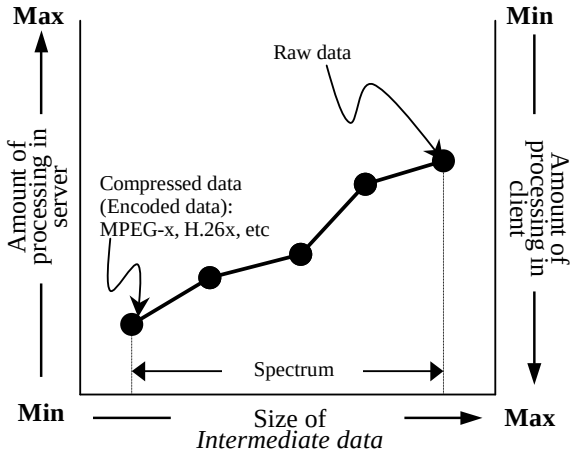


[Figure 2. New streaming model]

The key of the server-centric streaming model is intermediate decoding and intermediate data. The idea of intermediate decoding is to divide the decoding steps into two phases. The server decodes the first phase and the client does the second phase. The intermediate data is the output generated from the first decoding phase in server. In the client-centric model, a client goes through all the decoding steps. But in the server-centric model, the server is involved in decoding. The server does the decoding steps to generate intermediate data and sends the intermediate data. Then the client produces the raw data from the intermediate data. To achieve our goal stated above, decoding of the intermediate data at the client should be simple enough for thin devices, and the intermediate data should be common for as many

compression algorithms as possible.

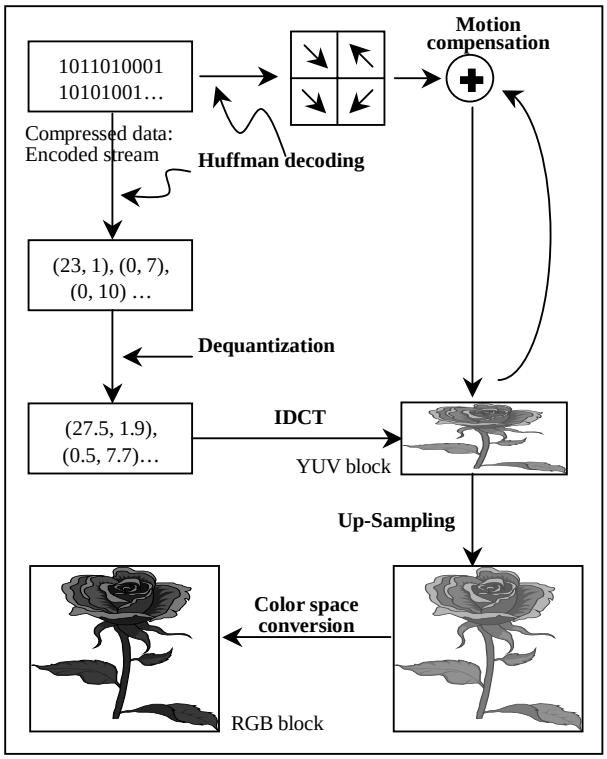
Now question is: how to define intermediate data. Our basic idea is to define the intermediate data in the decoding steps of block-based compressions (where JPEG, MPEG-x, H.26x, etc. belong). The intermediate data can be anywhere between compressed data and RGB raw data. There is a spectrum from compressed data to raw data where intermediate data can be defined as shown in Fig. 3. We believe that nobody has explored the spectrum other than the compressed data for streaming. This paper analyzes the effects of the different intermediate data (points in Fig. 3) in the spectrum.



[Figure 3. Spectrum of intermediate data for streaming]

As the intermediate data is closer to RGB raw data, the server does more decoding and the intermediate data size gets larger, which lead to higher bandwidth requirement. But the client does less decoding. So there is a trade-off depending on the level of intermediate decoding - how much decoding to be done at the client and the bandwidth to be consumed. To understand the trade-off, we measure the bandwidth and the CPU time in each decoding step of MPEG-1.

As shown in Fig. 4, MPEG-1 decoding steps can be divided into Huffman decoding, Dequantization, IDCT, Motion compensation, Up-sampling, and Color space conversion [1, 2, 7, 11, 12]. The output of each step can be an intermediate data. The data after Huffman decoding is done can be an example of intermediate data. Or the output of IDCT or IDCT and dequantization steps can be an intermediate data respectively. The next Section describes the bandwidth and the CPU time of each decoding step and discusses the pros and cons of the output of each step as the intermediate data.



[Figure 4. Decoding steps of MPEG-1]

3.2 CPU time and bandwidth analysis in MPEG-1

In our analysis, MPEG-1 clips are categorized into two groups (Table 1). One is grouped by frame dimensions (CIF and QCIF), and the other is by compression ratio. We collected clips of high, medium, and low compression ratios for each frame dimension. In Table 1, for example, 121.mpg is a clip of the compression ratio of 121. For each clip, we measure the data size that is essentially the bandwidth consumed.

	320 X 240			160 X 120		
Fr.	121.mpg	56.mpg	27.mpg	31.mpg	12.mpg	7.mpg
I	38	67	79	38	25	80
P	111	201	237	112	73	237
B	282	536	630	297	193	631
Block type						
I	264448	905150	1183859	104093	119365	382202
P	228852	481848	543060	72534	54882	155196
B	341058	773220	419616	41430	47958	117792

[Table 1. MPEG-1 Clips]

Table 1 has the number of frames and blocks for the collected clips because the data size and the CPU time depend on the type of frames in MPEG-1. I frame is composed totally of I blocks. P frame consists of I blocks and P blocks, and B frame contains I blocks, P blocks,

and B blocks. Since I frame does not have any P or B blocks, it doesn't require reference frames and so doesn't need motion compensation. However, P and B frame should have reference frames that are indispensable to motion compensation step. The following system is used to collect all experimental data in this Section: Ultra 30 workstation of Sun Microsystems with 246Mhz UltraSPARC-II CPU and with 512MB Ram.

Table 2 and 3 are the results of measuring the data size for CIF and QCIF frame dimensions respectively. The Data column in the tables means the average size of the encoded frames. For example, the size of encoded I frames of 121.mpg in Table 2 is 196KB on the average. It is decompressed to 4275KB after the IDCT step is done. Note that the IDCT step includes Huffman decoding and Dequantization steps because it is difficult to trace data size up to Dequantization and because the data size does not change over IDCT step. Since there is no motion compensation (MC) step in decoding I frames, so there is no difference in the data size between "After IDCT" and "After MC". The Sum row of each clip is the average data size at each step for all the frames in the clip.

Clip	Frame	Data	After IDCT	After MC	After Color
121.mpg	I	196	4275	4275	8550
	P + B	600	12253	43875	87750
	Sum	796	16528	48150	96300
56.mpg	I	653	7538	7538	15075
	P + B	2614	49034	83250	166500
	Sum	3267	56572	90788	181575
27.mpg	I	772	8888	8888	17775
	P + B	7372	65104	101250	202500
	Sum	8144	73992	110138	220275

*320X240, 30 frame/sec, Unit: KB

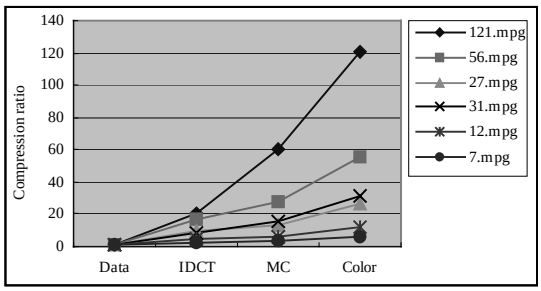
[Table 2. Average data size in CIF]

Clip	Frame	Data	After IDCT	After MC	After Color
31.mpg	I	130	1069	1069	2138
	P + B	669	5437	11503	23006
	Sum	799	6506	12572	25144
12.mpg	I	171	703	703	1406
	P + B	1198	6757	7481	14963
	Sum	1369	7460	8184	16369
7.mpg	I	711	2250	2250	4500
	P + B	7351	21638	24413	48825
	Sum	8062	23888	26663	53325

*160X120, 30 frame/sec, Unit: KB

[Table 3. Average data size in QCIF]

Fig. 5 shows the data size increase ratio after each step. The y-axis is the ratio, and the value in the Data column in Table 2 and 3 is the basis for the ratio. The size of After Color step is always twice the size of After MC because of 4:2:0 sub-sampling in MPEG-1. In other steps, the data size increase ratio cannot be predicted. Clips containing a large number of P or B blocks have high compression ratio that leads to the stiff slope between IDCT and MC in Fig. 5. The reason is that the P and B type block is highly compressed, and MC expands the blocks.



[Figure 5. Compression ratio]

The CPU time analysis is shown in Table 4 and 5. The Parsing column includes Huffman decoding and dequantization. The reason is that Huffman decoding is just table look-up so that Huffman decoding is absorbed into the dequantization step. Each column in the tables is the average CPU time for each decoding step. The Color column means the time to do color space conversion and up-sampling, and Render is the time taken for rendering. The Avg. row of a clip is the average decoding time at each step for all frames. In 121.mpg, the Parsing step for I frames takes 1ms of CPU time, and 36ms (1+21+0+8+6) is the average CPU time to go through all the decoding steps of I frames – let's call it full decoding time.

Clip	Frame	Parsing	IDCT	MC	Color	Render
121.mpg	I	1	21	0	8	6
	P + B	2	5	6	8	6
	Avg.	1.9	6.4	5.5	8.0	6.0
56.mpg	I	1	22	0	8	6
	P + B	2	10	9	8	6
	Avg.	1.9	11.0	8.3	8.0	6.0
27.mpg	I	1	24	0	8	6
	P + B	2	15	6	8	6
	Avg.	1.9	15.8	5.5	8.0	6.0

*320X240, 30 frame/sec, Unit: ms

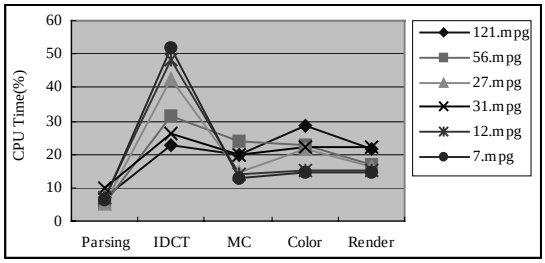
[Table 4. CPU Time in CIF]

Clip	Frame	Parsing	IDCT	MC	Color	Render
31.mpg	I	0	7	0	2	2
	P + B	1	2	2	2	2
	Avg.	0.9	2.4	1.8	2.0	2.0
12.mpg	I	0	10	0	2	2
	P + B	1	6	2	2	2
	Avg.	0.9	6.3	1.8	2.0	2.0
7.mpg	I	0	11	0	2	2
	P + B	1	7	2	2	2
	Avg.	0.9	7.3	1.8	2.0	2.0

*160X120, 30 frame/sec, Unit: ms

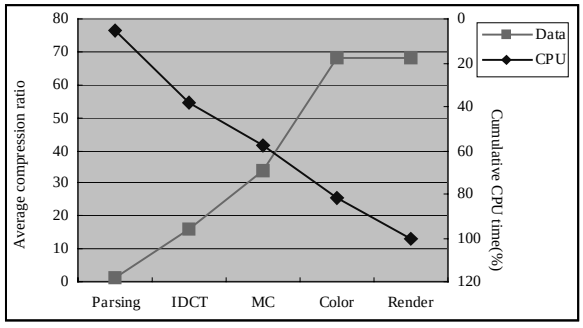
[Table 5. CPU Time in QCIF]

Fig. 6 shows the percentage of the CPU time taken at each step over the full decoding time. The IDCT step is the most CPU-intensive. Also, the less a clip is compressed, the higher percentage the CPU time of IDCT reaches.

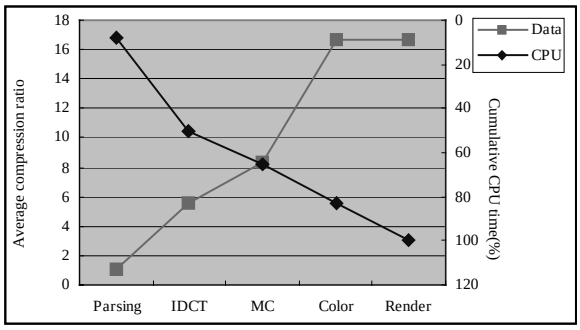


[Figure 6. CPU percentage in each step]

Fig. 7 is the summary of our bandwidth and CPU time trade-off analysis. The left y-axis represents the average compression ratio for each step and the right y-axis represents the percentage of cumulative CPU time to decode. For instance, 15 of the left y-axis is the average of IDCT values of Fig 5, and 35 of the right y-axis is the percentage of the sum of Parsing and IDCT CPU time over the full decoding time.



[Figure 7-a. CPU time & Data size in CIF]



[Figure 7-b. CPU time & Data size in QCIF]

Fig. 7a and 7b suggest two candidates for the reasonable trade-off between the bandwidth and the CPU time: IDCT and MC. The outputs of the Parsing and Color steps are either too CPU-intensive or bandwidth-hogging. If the output of IDCT is the intermediate data, for the CIF frame dimension, the client need to do 62 percent of the full decoding time, and the bandwidth consumption is 23 percent of the full bandwidth. For QCIF, decoding the output of IDCT consumes 33 percent of the full bandwidth and the half of the full decoding time. However, a concern is that the client should buffer several frames for motion compensation. In addition, to do motion compensation, the server should send reference frames in advance. This requirement makes difficult synchronization mechanism (synchronization mechanism will be explained in the next Section).

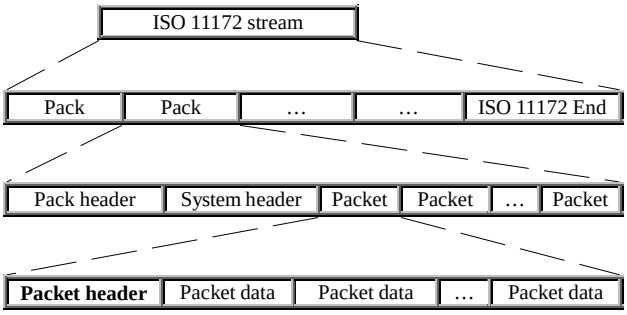
The other candidate is the output of MC. It is interesting that the cross-over point in both Fig. 7a and 7b is near MC. For CIF, decoding the output of MC consumes a little less than the half of the full decoding time and the half of the full bandwidth. For QCIF, 34 percent of the full decoding time is needed at the client, and the half of the full bandwidth is consumed. An advantage of the output of MC is that it does not have dependency on reference frames, which makes the buffering and synchronization of streaming easier. Another advantage is that the output of MC is common to any block-based compression algorithms, which means that one decoder at the client can play all the streams of current and future algorithms as long as they are block-based. Therefore, we chose to use the output of MC as the intermediate data.

3.3 Synchronization

A difficult problem we have to address in the server-centric model is the synchronization of media types (e.g. video, audio, etc). In the traditional model, because a server sends encoded media files in the form of so-called system stream where various media types are multiplexed with synchronization information, client can do decode and play them with the synchronization information in the system stream. In contrast, in the new streaming model, the server does decoding and sends a stream of intermediate data generated from each media type (e.g. video, audio, etc.). In order to generate intermediate data, the server should demultiplex the multiplexed media files so that synchronization information is removed from intermediate data. So clients have no idea about how to synchronize various intermediate data (e.g. video intermediate data, audio intermediate data, etc).

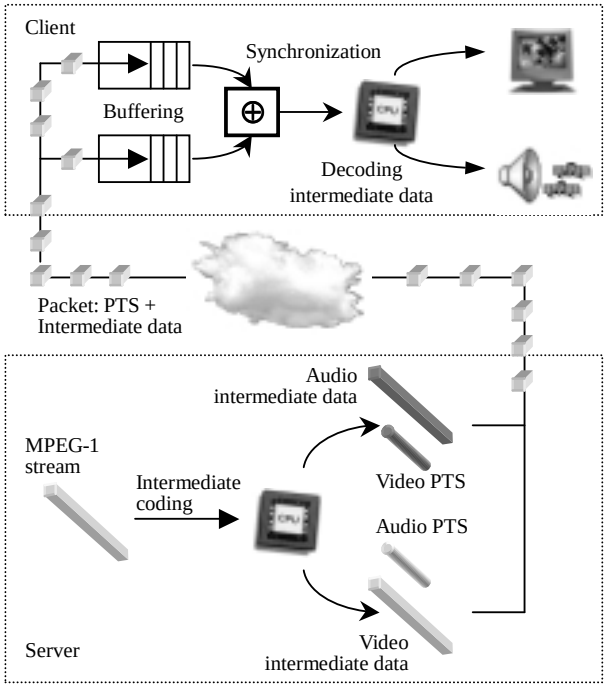
For example, consider the synchronization in MPEG-1. Fig. 8 depicts the structure of the MPEG-1 system stream. The only synchronization information is PTS (Present Time Stamp) in **Packet header** of Packet layer. PTS is a 33 bit integer data. Its unit is SCF (System Clock Frequency: 90KHz). PTS is converted into time information for synchronization. Since every Packet layer

has a PTS value, synchronization of video and audio streams is done per Packet. Inside the Packet structure, video data or audio data is stored in Packet data. With the new streaming model, the server generates the intermediate data from each Packet data. Therefore, PTS is not included in the intermediate data. We need solutions that allow the client to synchronize the packets of the intermediate data.



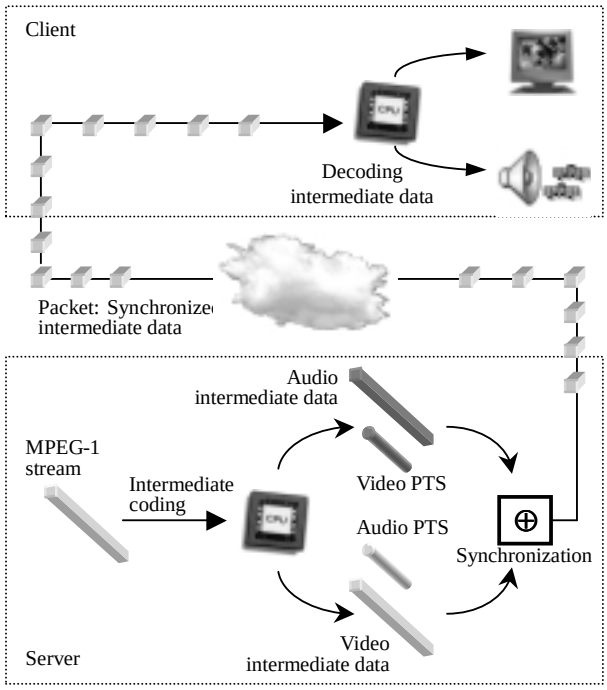
[Figure 8. MPEG-1 System Layer]

One solution is to put the synchronization hint into the protocol header like RTP and to have clients synchronize the packets as shown in Fig. 9. Clients need to buffer the packets and check whether it is time to start decoding.



[Figure 9. Client side synchronization]

The other one is that the server synchronizes the packets before sending them as shown in Fig. 10. The server does synchronization using the synchronization hint. All the clients have to do is just to process all packets without worrying about synchronization as they receive the packets. It is questionable whether this solution would be effective in the WAN environment, but it would work in the LAN environment where the packet order is likely to be preserved. This solution allows clients to process packets with minimal buffering. We used this solution in the paper.

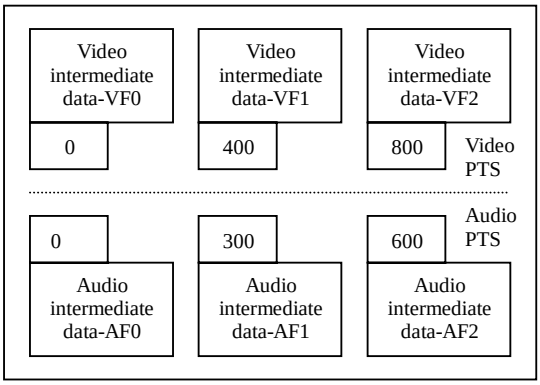


[Figure 10. Server side synchronization]

A caution is that the synchronization hint (e.g. PTS in MPEG-1) indicates the timing of display, not timing of transmission. In other words, the synchronization hint is the information to synchronize video and audio streams only for local display, and so even if the server sends the synchronized video and audio packets according to the hint, it is possible that video and audio data are not synchronized properly at the client due to the network delay. It is because the audio packet is typically much smaller than the video packet so that it takes less time to send an audio packet than a video packet. Therefore, the server must consider the network delay in addition to the hint. The factors affecting the network delay are network bandwidth, network congestion, network transmission delay, etc. To achieve the precise synchronization at the client, a server should calculate the expected network delay and send the packets of intermediate data accordingly.

For example, suppose that a server has video and audio intermediate data in Fig. 11. For local display, based on PTS values, the play order is: VF0, AF0, AF1, VF1, AF2, and VF2. But if a server considers network delay, the

sending order may be changed. In other words, since playing is done in client, the intermediate data have to be sent in advance in order to allow the client to play the data according to the PTS values. Therefore, the sending time is not based on PTS. Suppose that the network delay of a video intermediate data and an audio intermediate data takes 350 SCF and 30 SCF respectively. Then the sending order is: VF0, AF0, VF1, AF1, VF2, and AF2.



[Figure 11. Snapshot of video and audio intermediate data]

We can formularize the sending time as follows.

Video Sending PTS = Video PTS – Delay (Video size)
Audio Sending PTS = Audio PTS – Delay (Audio size)
Delay (X) = (X * 8 / N (b/s)) * 90 * 10^3 PTS/s
♦N: NIC bandwidth

[Equation 1. Sending PTS]

So, a server should send video and audio intermediate data synchronously with Sending PTS, not with PTS.

4. Implementation and Performance

4.1 Evaluation Methodology

We implemented an MPEG-1 player based on the new streaming model. The basis of our implementation is the public domain Berkeley video decoder [8] and maplay [9] of Berlin University. We had to make numerous changes because the Berkeley decoder does not understand the system layer and both public domain software include several bugs. We made the Berkeley decoder and the maplay to play MPEG-1 system streams with and without streaming. Our streaming model is incorporated into Sun Ray 1. Sun Ray 1 is a stateless thin client machines that does not have enough processing power for handling MPEG-1 streams. Sun Ray 1 consists of four major hardware components: 100MHz microSPARC-IIep processor that is equivalent to 486-class CPU, 10/100Mbps ethernet device, frame buffer controller with ATI Rage 128 chip, and 8MB of memory. Sun Ray 1 does not run any operating system so that no application runs on Sun Ray 1. For more information on Sun Ray 1, refer to Sun's web page [6].

	Server system	Sun Ray 1	PC
CPU	Ultra SPARC-II 246Mhz	microSPARC-IIep	486DX2-80Mhz
Ram	128MB	8MB	8MB
VGA	Elite3D m6	ATI Rage 128	PCI VGA
NIC	100 Mbps	100 Mbps	N/A
OS	Solaris 2.6	Embedded	Linux

[Table 6. Server/Client system configuration]

In streaming, the server software is responsible for three things: decoding MPEG-1 files, generating video and audio intermediate data, and synchronizing them. The Berkeley video decoder and the maplay are modified to generate and synchronize audio and video streams of intermediate data. The streams are packetized and sent via UDP for efficient communication.

The client software running in Sun Ray 1 is to receive the datagrams and decode video and audio intermediate data into "raw" data (e.g. RGB signals and PCM samples) out of intermediate data. Remember that the client is not responsible for synchronization. The decoding that it has to do (from intermediate data to raw data) is very light, which greatly simplifies the client software. The decoder in Sun Ray 1 occupies only a couple of kilobytes of memory. We believe that it is very possible to build a client in a very inexpensive (low powered) hardware that plays the MPEG-1 stream.

To characterize the performance of our streaming model, we analyze the CPU utilization of server software and network utilization. We want to emphasize that our implementation is based on relatively "unoptimized" public domain software so that the absolute numbers are not important. Our goal is to see the ratio of decoding time versus network transmission time and rendering time.

For the performance characterization, three sets of experiments are done. The first experiment is to run the video player on 486-based PC. It is to simulate a case in which the player runs on Sun Ray 1 without streaming. The reason why we need the simulation is because the players cannot run on Sun Ray. The second experiment is to stream the video and audio without using intermediate data. The purpose is to establish the basis for the comparison. The last experiment is to apply the new streaming model. The machine configurations used in the experiments are shown in Table 6.

We used six MPEG-1 files shown in Table 7. The first letter means compression ratio and stands for "high", "medium", and "low", and the second for "big" and

"small". The property of "big" is 320X240-video frame dimension, 30 frames per second, stereo mode, and 44.1KHz frequency. "small" means 160X120-video frame dimension, 30 frames per second, stereo mode, and 44.1KHz frequency.

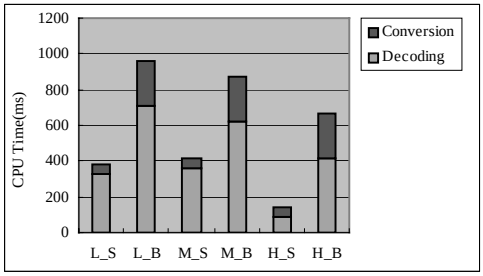
	L_S	L_B	M_S	M_B	H_S	H_B
Ratio	3:1	24:1	10:1	43:1	24:1	99:1
Frame	952	952	617	452	456	447
Time(s)	32	32	21	15	15	15

[Table 7. MPEG-1 clips properties]

4.2 Evaluation

4.2.1 486DX2 based PC

We ported the video player to Linux and measured the CPU time on a 486 PC. For the fair comparison with intermediate data, we measured the time for Parsing, IDCT, and MC ("Decoding" in Fig. 8) and the time for color space conversion ("Conversion ") separately.



[Figure 12. 486DX2 CPU utilization]

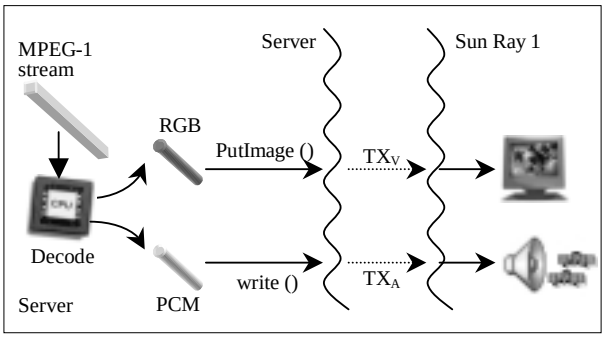
The x-axis in Fig. 12 is the different clips in Table 7. Fig. 12 shows that the best case (H_S clip) decodes about 6 frames/sec and the worst (L_B clip) 1 frame/sec. The results indicate that Sun Ray 1 would be very sluggish to play encoded MPEG-1 streams.

4.2.2 Sun Ray 1 with raw data

For normal window rendering, Sun Ray 1 receives pixels. When a video application runs without intermediate data, the server emits the raw data (RGB values) to Sun Ray 1. The X Window sever and its device driver (part of Sun Ray 1 server software) open a socket and send the raw data [9].

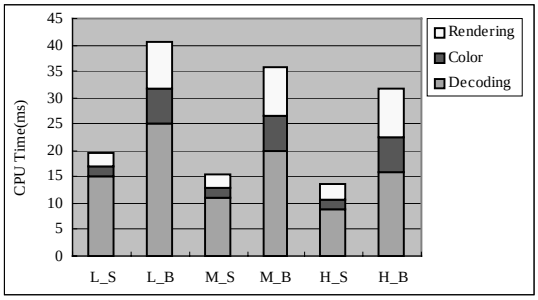
Fig. 13 depicts how the video and audio are played without intermediate data. The MPEG-1 player decodes a clip, and it issues two calls: write() for audio, PutImage() for video. At this time, the server device drivers understand that the destination is Sun Ray 1 and convert the requests to send RGB and PCM data. The network bandwidth can be calculated as follows:

Video (Bytes/sec): Width * Height * 3 * Frame rate
Audio (Bytes/sec): Frequency * Mode * 2 (16bit)



[Figure 13. MPEG-1 player execution with raw data]

The CPU time at the server is shown in Fig. 14. Rendering in the figure means the CPU time to send the raw data to Sun Ray 1. In this experiment, the synchronization problem described in Section 3.3 happens. It is because the size of video data (RGB) is larger than audio data (PCM). It means that TX_V takes more time than TX_A. Therefore even though MPEG-1 player plays video and audio data synchronously at the server, the synchronization at Sun Ray 1 may not be correct.

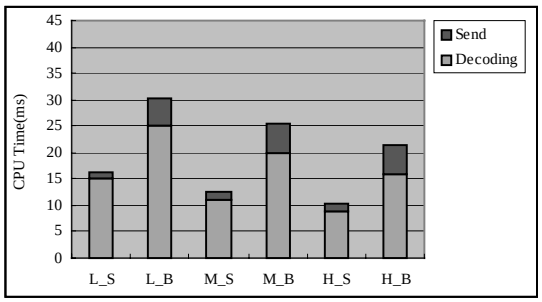


[Figure 14. Server CPU time with raw data]

Nevertheless, Sun Ray 1 shows very good performance compared with 486DX2 based PC. Sun Ray 1 plays "small" MPEG-1 clips at full frame rate and decodes L_B about 25 frames/sec.

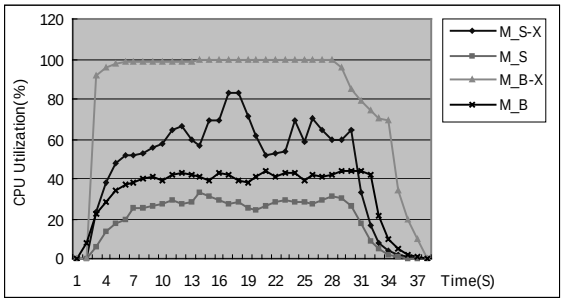
4.2.3 Sun Ray 1 with intermediate data

We used the output of MC as the intermediate data. Fig. 15 is the server CPU utilization to generate intermediate data. Compared with Fig. 14, it shows even better performance. For example, L_B consumes about 16ms for Color and Rendering in Fig. 14, but L_B in Fig. 15 needs 6ms. Similarly, H_S in Fig. 14 takes 4ms while H_S in Fig. 11 takes about 2ms. The saving of using intermediate data increases with the bigger frame dimension.



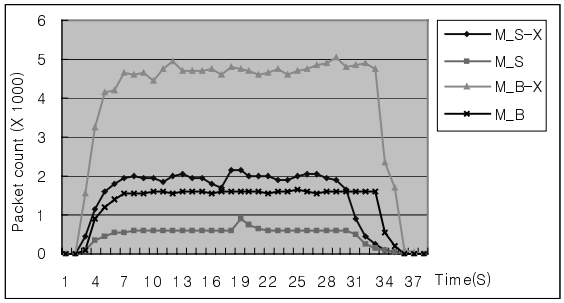
[Figure 15. Server CPU time with intermediate data]

To see another benefit of the intermediate data, the system-wide CPU utilization have been measured, and the results are in Fig. 16. We used the performance meter tool in Solaris. We ran 6 clips in Table 7, and the performance is quite similar to each other. So the results with two clips (M_S and M_B) are presented in Fig. 16. M_S-X and M_B-X are the results with raw data described in Section 4.2.2.



[Figure 16. Server CPU utilization comparisons]

As Fig.16 shows, our streaming model reduces the CPU utilization at the server by 60 percent of that with raw data. One reason we found is that M_S-X and M_B-X consume kernel resources very heavily. This result means that intermediate coding is much cheaper than the full decoding. In addition to the CPU time, the network utilization has also been measured. The packet counts were collected and the results are in Fig. 17.



[Figure 17. Packet comparisons]

Fig. 17 shows that for M_B, the total number of packets (Fig. 17) is over three times of that with raw data Note that the result is higher than expectation because the data size of the intermediate data is the half of raw data (as Fig. 7a and 7b show).

With the intermediate data, we found that Sun Ray 1 can decode MPEG-1 streams that is 352X280 with 48Khz stereo and can display 1056X840 size (scale up 3 times) at full frame rate. Without scaling, Sun Ray 1 plays up to 3 clips at full frame rate simultaneously.

5. Conclusion

In this paper, we propose a new streaming model. The main difference between our model and the traditional model is the data format used in streaming. While the traditional model transfers encoded media streams, our streaming model uses the intermediate data. An advantage of our model is that the CPU power required in the client is significantly low and that buffering is minimized. In addition, because a server can transform a variety of the encoded media files into the intermediate data, a client can play them without any special decoder or any modification. That is, whatever encoded media files the server can decode, the client can play them by decoding the intermediate data.

We apply this model to MPEG-1 and analyze the CPU time and the bandwidth of each decoding step to show the trade-off in different intermediate decoding levels. The server and client software have been implemented on top of UDP. We ran numerous MPEG-1 clips on a thin client called Sun Ray 1 and observed that Sun Ray 1 is able to play MPEG-1 streams of the full-size display at 30 frames per sec.

In the future, we plan to investigate mechanisms to reduce the network bandwidth requirement and apply our model to other standards such as MPEG-2.

Acknowledgements

We are grateful to Duane Northcutt at Sun Microsystems for his support and encouragement throughout the project. Jim Hanko, Jerry Wall, and Alan Ruberg implemented the client software for Sun Ray 1.

6. References

[1]. D. Legall, “MPEG – A Video Compression Standard For Multimedia Application,” *Communications of the ACM*, April 1991, Vol 34, Num 4, pp 46-58.

[2]. Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J. LeGall, *MPEG video*

compression standard, Chapman and Hall, 1996.
[3]. URL http://www.real.com
[4]. URL http://www.xingtech.com
[5]. URL http://www.microsoft.com
[6]. URL http://www.sun.com
[7]. FTP archive at URL ftp://ftp.cs.tu-berlin.de/pub/av/maplay1.2/
[8]. FTP archive at URL ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg/play
[9]. Brian K. Schmidt, Monica S. Lam, J. Duane Northcutt, "The interactive performance of SLIM: a stateless, thin-client architecture," ACM Symposium on Operating Systems Principles, December 1999, pp 32-47.
[10]. Reza Rejanie, Mark Hendley, Deborah Estrin, "Quality Adaptation for Congestion Controlled Video Playback over the Internet," Proceedings of ACM SIGCOMM '99, Cambridge, MA., September 1999.
[11]. L.A. Rowe, K. Patel and B.C. Smith, "Performance of a Software MPEG Video Decoder," Proc. *ACM Multimedia 93*, Anaheim, CA, August 1993.
[12]. Willian B Pennebaker, *JPEG still data compression standard*, Van Nostrand Reinhold, New Youk, 1992.