Performance of a Software MPEG Video Decoder*

Ketan Patel, Brian C. Smith, and Lawrence A. Rowe
Computer Science Division-EECS
University of California
Berkeley, CA 94720

Abstract

The design and implementation of a software decoder for MPEG video bitstreams is described. The software has been ported to numerous platforms including PC's, workstations, and mainframe computers. Performance comparisons are given for different bitstreams and platforms including a unique metric devised to compare price/performance across different platforms (percentage of required bit rate per dollar). We also show that memory bandwidth is the primary limitation in performance of the decoder, not the computational complexity of the inverse discrete cosine transform as is commonly thought.

1. Introduction

The CCITT MPEG group was formed in 1988 to develop a standard for storing video and associated audio on digital media. Their goal was to define a standard that required bit rates less than 1.5 Mbits/sec, which is achievable by computing networks and digital storage media available today. A draft proposal was agreed upon in September 1990. Since then, minor changes have been made, and the standard has been released. The work described in this paper is based on the December 1991 committee draft [3].

Many research and commercial groups have developed MPEG decoders. Because of the high stakes involved in commercializing MPEG technology (e.g., HDTV and video conferencing), these groups have been reluctant to release their coders, decoders, or bitstreams. The absence of freely distributable MPEG source code has hindered research on MPEG applications.

We implemented an MPEG video decoder for three reasons. First, we wanted to determine whether MPEG video could be decoded in real-time using a software-only implementation on current generation desktop computers. Second, we needed to develop a portable software decoder for inclusion in the Continuous Media Player being developed at U.C. Berkeley [8]. And third, we wanted to contribute public domain to the research community.

This paper describes the design and implementation of the decoder. A novel feature of our decoder is the use of a dithering algorithm in YCrCb-space. We also report the performance of playing seven anonymous bitstreams that we have acquired on a variety of platforms. Rather than saying "we can play bitstream A on platform P at N frames/second," we devised a metric that compares the relative price/performance of different platforms. In our analysis, we also found that memory bandwidth is the primary limitation in decoder performance, not the computational complexity of the inverse discrete cosine transform (IDCT) as is commonly thought.

The remainder of this paper is organized as follows. Section 2 presents a brief introduction to MPEG video coding. Section 3 describes the implementation of our decoder and presents a time/space performance analysis. Section 4 describes optimizations to improve decoder performance. Section 5 describes the bitstreams used in the dithering and cross-platform analyses presented in sections 6 and 7, respectively. Lastly, we describe the experience of publishing this software on the Internet.

*This research was supported by grants from the National Science Foundation (Grant MIP-90-14940 and Infrastructure Grant CDA-8722788) and Fujitsu Network Transmissions Systems, Inc., and Hewlett-Packard Company.

Ketan Patel: (510) 642-9585, kpatel@cs.berkeley.edu
Brian C. Smith: (510) 642-9585, bsmith@cs.berkeley.edu
Lawrence A. Rowe: (510) 642-5117, larry@cs.berkeley.edu

This paper appeared as part of the ACM Multimedia '93 Conference.

2. The MPEG Video Coding Model

This section briefly describes the MPEG video coding model. More complete descriptions are given in an introductory paper[6] and the ISO standard [3].

I_1 I_2 I_3

Figure 1: Sample sequence.

Video data can be represented as a set of images, I_1, I_2, ..., I_N, that are displayed sequentially. Each image is represented as a two dimensional array of *RGB triplets*, where an RGB triplet is a set of three values that give the red, green and blue levels of a pixel in the image.

MPEG video coding uses three techniques to compress video data. The first technique, called *transform coding*, is very similar to JPEG image compression [2]. Transform coding exploits two facts: 1) the human eye is relatively insensitive to high frequency visual information, and 2) certain mathematical transforms concentrate the energy of an image, which allows the image to be represented by fewer values. The discrete cosine transform (DCT) is one such transform. The DCT also decomposes the image into frequencies, making it straightforward to take advantage of (1).

In MPEG transform coding, each RGB triplet in an image is transformed into a YCrCb triplet. The Y value indicates the *luminance* (black and white) level and Cr/Cb values represent *chrominance* (color information). Since the human eye is less sensitive to chrominance than luminance, the Cr and Cb planes are *subsampled*. In other words, the width and height of the Cr and Cb planes are halved.

Processing continues by dividing the image into *macroblocks*. Each macroblock corresponds to a 16 by 16 pixel area of the original image. A macroblock is composed of a set of six 8 by 8 pixel *blocks*, four from the Y plane and one from each of the (subsampled) Cr and Cb planes. Each of these blocks are then processed in the same manner as JPEG: the blocks are transformed using the DCT and the resulting coefficients quantized, run length encoded to remove zeros, and entropy coded. The details can be found in [14], but the important facts for this paper are that 1) the frame is structured as a set of macroblocks, 2) each block in the macroblock is processed using the DCT, and 3) each block, after quantization, contains a large number of zeros.

The second technique MPEG uses to compress video, called *motion compensation*, exploits the fact that a frame I_x is likely to be similar to its predecessor I_{x-1}, and so can be nearly constructed from it. For example, consider the sequence of frames in Figure 1, which might be taken by a camera in a car driving on a country road. Many of the macroblocks in frame I_2 can be approximated by pieces of I_1, which is called the *reference frame*. By *pieces* we mean any 16 by 16 pixel area in the reference frame. Similarly, many macroblocks in I_3 can be approximated by pieces of either I_2 or I_1. The vector indicating the appropriate piece of the reference frame requires fewer bits to encode than the original pixels. This coding results in significant data compression.

Note, however, that the right edge of I_2 (and I_3) can not be obtained from a preceding frame. Nor can the portion of the background blocked by the tree in I_1 because these areas contain new information not present in the reference frame. When such macroblocks are found, they are encoded without motion compensation, using transform coding.

Further compression can be obtained if, at the time I_2 is coded, both I_1 and I_3 are available as reference frames[1]. I_2 can then be built using both I_1 and I_3. When a larger pool of reference frames is available, motion compensation can be used to construct more of the frame being encoded, reducing the number of bits required to encode the frame. A frame built from one reference frame is called a *P* (predicted) frame, and a frame built from both a preceding frame and a subsequent frame is called a *B* (bidirectional) frame. A frame coded without motion compensation, that is, using only transform coding, is called an *I* (intracoded) frame.

Motion compensation in P and B frames is done for each macroblock in the frame. When a macroblock in a P or B frame is encoded, the best matching[2] macroblock in the available reference frames is found, and the amount of x and y translation, called the *motion vector* for the macroblock, is encoded. The motion vector is in units of integral or half integral pixels. When the motion vector is on a half

[1] This strategy will, of course, require buffering the frames and introduce delay in both encoding and decoding.

[2] The criteria for "best matching" is determined by the encoder.

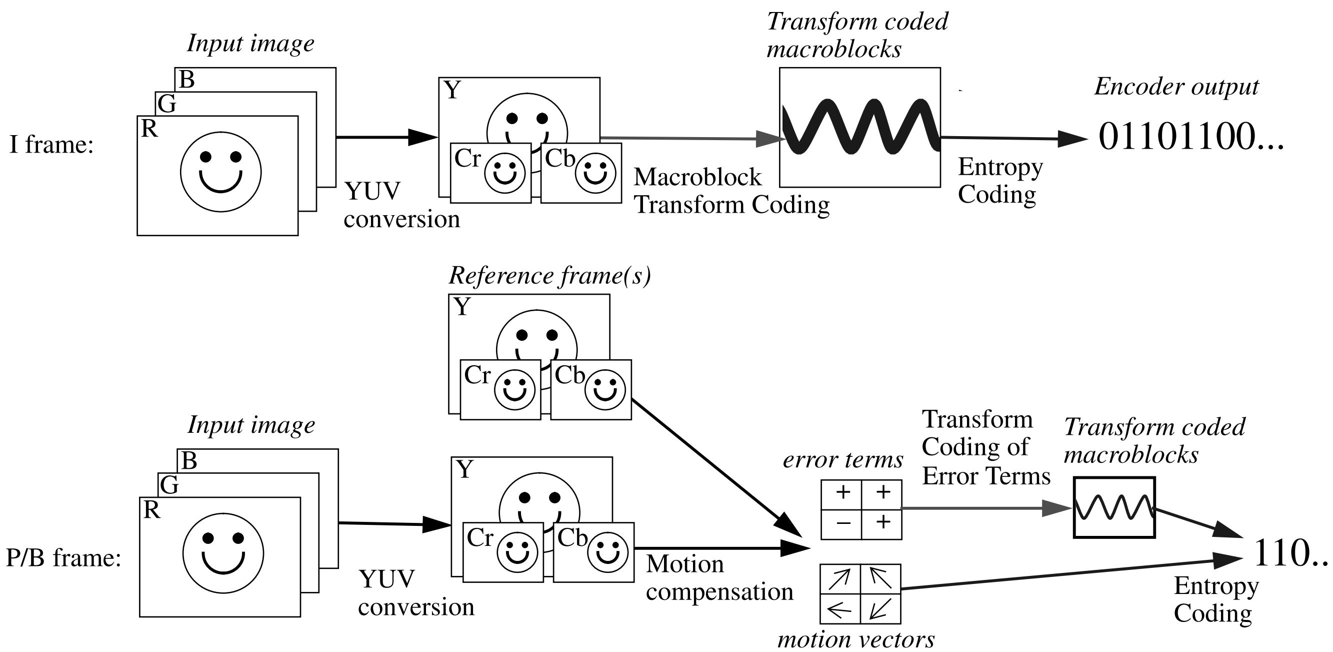

Figure 2: MPEG video coding procedure.

pixel boundary, the nearest pixels are averaged. The match between the predicted and actual macroblocks is often not exact, so the difference between the macroblocks, called the *error term*, is encoded using transform coding.

After motion compensation and transform coding, a final pass is made over the data using Huffman coding (i.e., entropy coding). Figure 2 summarizes the MPEG video coding process.

To rebuild the YCrCb frame, the following operations are needed:

(1) the Huffman coding must be inverted
(2) the motion vectors must be reconstructed and the appropriate parts of the reference frame copied (in the case of P or B frames), and
(3) the error terms must be decoded and incorporated (which includes an application of the IDCT).

Once the YCrCb frame has been built, the frame is converted to a representation appropriate for display. This last step is called *dithering*.

In summary, MPEG uses three techniques, that is motion compensation, transform coding, and entropy coding, to compress video data. MPEG defines three types of frames, called I, P and B frames. These frames use zero, one, and two reference frames for motion compensation, respectively. Frames are represented as an array of macroblocks, and both motion compensation and transform coding operate on macroblocks.

3. Implementation

The decoder is structured to process a small, implementation dependent *quantum* of macroblocks at a time so it can be suspended while processing a frame. We envision using the decoder as a part of a larger system (the CM Player [8]), for delivering video data over local networks. This architecture is required by the player to service other tasks required in a multimedia system (e.g., handling user input or playing other media).

The decoder was implemented in C using the X Windowing System. It is composed of 12K lines of code. Our intent was to create a program that would be portable across a variety of UNIX platforms. To date, the decoder has been ported to over 10 platforms. Ports have also been completed to PC's and Macintosh's.

Preliminary analysis of the run-time performance indicated that dithering accounted for 60% to 80% of the time, depending on the architecture. Consequently, we focussed our attention on speeding up this part of the code. We also optimized other decoding procedures using standard optimization tricks: 1) in-line procedure expansion, 2) caching frequently accessed values, and 3) custom coding frequent bit twiddling operations. Altogether, these changes to the decoder reduced the time by 50%. The specific optimizations are discussed in the next section. More significant improvements (over a factor of 15) were made by using an ordered dither. Dithering is discussed in detail in section 5.

The fastest color dithering algorithm with reasonable quality is an ordered dither that maps a 24-bit YCrCb image to a 7-bit color space (i.e., 128 colors) using a fixed color map. We analyzed the performance of the decoder using this technique. The following table shows the results:

Function	% Time
Parsing	17.4%
IDCT	14.2%
Reconstruction	31.5%
Dithering	24.3%
Misc. Arithmetic	9.9%
Misc.	2.7%

Parsing includes all functions involved in bitstream parsing, entropy and motion vector decoding, and coefficient reconstruction. The IDCT code, which is called up to six times per macroblock, is a modified version of the fastest public domain IDCT available [5]. The algorithm applies a 1 dimensional IDCT to each row and then to each column. Zero coefficients are detected and used to avoid unnecessary calculation. Functions that perform predictive pixel reconstruction, including copying and averaging relevant pixels from reference frames, are grouped under the category *reconstruction*. Finally, dithering converts the reconstructed YCrCb image into a representation appropriate for display.

The table shows that the majority of time is spent in reconstruction and dithering. Parsing and IDCT each require approximately 15% of the time. The reason reconstruction and dithering are so expensive is that they are memory intensive operation. On general purpose RISC computers, memory references take significantly longer than arithmetic operations, since the arithmetic operations are performed on registers. Even though steps such as parsing and IDCT are CPU intensive, their operands stay in registers, and are therefore faster than the memory intensive operations of reconstruction and dithering. While improving the IDCT is important, our decoder could be sped up most significantly by finding a scheme that would reduce memory traffic in reconstruction and dithering.

4. Optimizations

This section describes some of the low-level optimizations used to improve the basic decoder. Three kinds of improvements are discussed: general coding, IDCT, and average cheating.

Numerous coding optimizations were applied throughout the code. One strategy was to use local copies of variables to avoid memory references. For example, since the addition of the error term to a pixel value often causes underflow or overflow (i.e., values less than 0 or greater than 255), bounds checking was required. This implementation results in three operations: the addition of the error term to the pixel, an underflow check, and an overflow check. Instead of accessing the pixel in memory three times, a local copy is made, the three operations are performed, and the result is stored back into memory. Since the compiler allocates the local copy to a register, we found the operations themselves to be about four times faster.

We also applied this technique to the bit parsing operations by keeping a copy of the next 32 bits in a global variable. The actual input bitstream is only accessed when the number of bits required is greater than the number of bits left in the copy. In critical segments of the code, particularly macroblock parsing, the global copy is again copied into a local register. These optimizations resulted in 10-15% increases in performance.

Other optimizations applied included: 1) loop unrolling, 2) math optimizations (i.e., strength reductions such as replacing multiplications and divisions with shifts and adds), and 3) in-line expansion of bit parsing and Huffman decoding functions.

The IDCT code was heavily optimized. The input array to the IDCT is typically very sparse. Analysis showed that 30%-40% of the blocks contained less than five coefficients in our sample data, and frequently only one coefficient exists. These special cases are detected during macroblock parsing and passed to the IDCT code that is optimized for them. We implemented the forward-mapped IDCT optimization suggested by X and Y [7]. Surprisingly, it did not speed-up the code. We are not sure why this optimization did not work. Perhaps the strength reductions performed automatically by the compiler already reduced the number of multiplies. Or, the additional memory required for the cache destroyed the on-chip memory cache reference pattern established by pixel reconstruction.

Finally, we found a way to cheat on the computation of pixel averaging in interframes (i.e., P- and B-frames). The MPEG standard specifies that predictive pixel values for interframes are constructed from copying areas in past or future frames based on a transmitted set of motion vectors. These motion vectors can be in half-pixel increments which means pixel values must be averaged.

The worst case occurs when both the horizontal and vertical vectors lie on half-pixel boundaries. In this case, each result pixel is an average of four pixels. The increased precision achieved by doing the pixel averaging is lost, however, in the dithering process. We optimize these functions in three ways. First, if both horizontal and vertical vectors lie on whole pixel boundaries, no averaging is required and the reconstruction is implemented as a memory copy.

Second, if only one motion vector lies on a half-pixel boundary, the average is done correctly. And finally, if both vectors lie on half-pixel boundaries, the average is computed with only 2 of the 4 values. We average the value in the upper left quadrant with the value in the lower right quadrant, rather than averaging all four values. Although this method produces pixels that are not exactly correct, dithering makes the error unnoticeable.

5. Sample Bitstreams

This section describes the bitstreams used for the performance comparisons presented in the next two sections.

Public domain MPEG data is relatively scarce. We selected seven bitstreams available to us that we believe constitute a reasonable data set. Table 1 presents the characteristics of the bitstreams. Five distinct coders were used to generate the data. Bitstreams B, D, and E were generated by the same coder. The video sequences are completely different except the sequences encoded in bitstreams B and C which use selections from the same raw footage.

The variation in frame rates, frame size, and compression ratios makes analysis with these bitstreams difficult to compare. We believe the best metric to judge the performance of the decoder is to measure the percentage of the required bit rate achieved by the decoder. For example, if a bitstream must be decoded at a rate of 1 Mbit/sec to play it at the appropriate frame rate, a decoder that plays at a rate of 0.5 Mbit/sec is able to achieve 50% of the required bit rate. Given a set of bitstreams, two decoders running on the same platform can be compared by calculating the percentage of bitstreams each decoder can play in real-time (i.e., at the required bit rate).

6. Dithering Performance

This section describes the performance improvements made to the dithering algorithm(s) used in the decoder. In this context, dithering is the process of converting a 24-bit YCrCb image into a representation appropriate for display. In principle, the YCrCb image is first converted to an RGB representation and the dithering algorithm is applied. Virtually all dithering algorithms, however, can be applied directly to the YCrCb image. This strategy avoids the memory traffic and arithmetic associated with RGB conversion, and it further reduces memory accesses since the Cr and Cb planes are subsampled.

The decoder supports monochrome, full (24 bit) color, gray scale and color mapped display devices. Dithering to full color devices is tantamount to RGB conversion. Dithering to gray scale devices is done by using only the luminance plane of the image. For color mapped devices, two dithering techniques are used: error diffusion (sometime called Floyd-Steinberg) and ordered dither. Both are discussed in [9]. Dithering to monochrome devices can be done using either thresholding or error diffusion.

In error diffusion dithering, each image pixel is mapped to the closest pixel in a fixed size color map. In our decoder, the color map has 128 entries, with 3 bits allocated for luminance, and 2 bits each for Cr and Cb chrominance. The difference, expressed as a YCrCb triplet, between the image pixel and the colormap pixel is called the *error*. The error is then distributed to neighboring pixels. For example, half the error might be added to the pixel below and half to the pixel to the right of the current pixel. The next (possibly modified) pixel is then processed. Processing is often done in a serpentine scan order, that is odd number rows are processed left to right and even number rows are processed right to left.

In threshold dithering, any pixel below a certain *threshold* of luminance is mapped to black, and all other values are mapped to white. Thresholding can be extended to color devices by dividing the color space into a fixed number of regions. Each pixel is mapped to a value inside the region.

Ordered dithering is similar to threshold dithering, except the pixel's (x,y) coordinate in the image is used to determine the threshold value. An N by N *dithering matrix* D(i,j) is used to determine the threshold: D(x mod N, y mod N) is the threshold at position (x,y). A four by four dithering matrix is used in our decoder. The matrix is chosen so that, over any N by N region of the image with the same pixel value, the mean of the dithered pixels in the region is equal to the original pixel value. For further details, the interested

Stream	Stream Size	Frame Size	Avg. Size I Frame	Avg. Size P Frame	Avg. Size B Frame	Frames/ Second	Bits/ Pixel	Bits/ Second	I:P:B
A	690K	320x240	18.9K	10.6K	0.8K	30	.488 (50:1)	1.12M	10:40:98
B	1102K	352x240	11.2K	8.8K	6.3K	30	.701 (34:1)	1.78M	11:40:98
C	736K	352x288	23.2K	8.8K	2.5K	25	.469 (51:1)	1.19M	11:31:82
D	559K	352x240	8.1K	5.5K	4.1K	6	.445 (54:1)	0.23M	6:25:88
E	884K	352x240	12.4K	9.1K	6.5K	6	.698 (34:1)	0.35M	6:25:89
F	315K	160x128	2.8K	N/A	N/A	30	1.09 (20:1)	0.67M	113:0:0
G	1744K	144x112	2.3K	1.8K	0.4K	30	.492 (49:1)	0.24M	294:293:1171

Table 1: Sample Bitstreams

Stream	FS4	FS2	ORDERED	GRAY	24BIT	NONE
A	12.64%	27.4%	51.4%	62.4%	35.7%	66.6%
B	11.2%	23.7%	42.5%	52.3%	30.3%	53.4%
C	11.6%	25.6%	48.6%	61.2%	33.5%	62.8%
D	56.7%	120.9%	225.3%	279.3%	156.1%	287.4%
E	56.0%	117.7%	210.5%	266.7%	151.5%	259.7%
F	43.7%	86.4%	146.2%	200.0%	111.8%	172.7%
G	60.5%	133.5%	247.2%	322.0%	180.9%	327.4%

Table 2: Relative performance of different dithering algorithms.

reader is referred to [9] and [1].

When implementing the decoder, we started with a straightforward implementation of the error diffusion algorithm with propagation of 4 error values (*FS4*). The first improvement we tried was to implement an error diffusion algorithm with only 2 propagated error values (*FS2*). This change improved run-time performance at a small, and essentially insignificant, reduction in quality.

The second improvement we implemented was to use an ordered dither. We map directly from YCrCb space to a 7-bit color map value by using the pixel position, 3 bits of luminance, and 2 bits each of Cr and Cb chrominance. We call this dither *ORDERED*.

Table 2 shows the relative performance of these dithers along with a grayscale (*GRAY*) and 24-bit color dither (*24BIT*). The table also shows the performance of the decoder without dithering. These tests were run on an HP 750 which is the fastest machine currently available to us. The results are expressed as percentages of required bit rates to factor out the differences in bitstreams.

Several observations can be made. First, notice that only 4 bitstreams (D, E, F, and G) were playable at the required bit rate (i.e., the percentage was over 100%) using an ORDERED dither. These bitstreams have low required bit rates because streams D and E were coded at 6 frames per second (*fps*), stream F is only 160x128 pixels, and stream G is even smaller at 144x128 pixels.

The remaining bitstreams can be played at approximately 50% of the required bit rate which implies that the current generation workstations can play around 15 fps.

Second, notice that even without dithering, which is shown in the column labeled NONE, our decoder can achieve only 2/3's of the required bit rate of a full-size, full-motion video stream.

Notwithstanding this pessimistic result, private communications with other groups working on decoders optimized for particular platforms say that their decoders operate 2-3 times faster than our portable implementation.[3] In addition, faster machines (e.g., DEC Alpha) are reported to play CIF video (i.e., 360x240) at 28 fps. The implication is that we are very close to being able to decode and play reasonable-sized videos. Indeed, video with small images and low frame rates can be played on PC's and Macintosh's.

7. Cross-Platform Performance

In evaluating the decoder on different platforms, we cannot use the percentage of required bit rate metric to rate the platform because price/performance is important. For example, running a decoder on two platforms where one platform is 4 times more expensive does not really tell you much. A better metric would factor in the cost of the hardware.

The metric we propose is the *percentage of required bit rate per second per thousand dollars*. We will call this metric *PBSD*. For example, suppose two machines M1 and M2 that cost \$15K and \$12K respectively play a bitstream at 100% and 50% of the required bit rate. The PBSD metrics for the two machines are 6.7 and 4.2. Higher numbers are better, so machine M1 is more cost efficient than M2.

On the other hand, suppose that M1 played only 60% of the required bit rate. The PBSD metrics would be 4.0 and 4.2, which implies that M2 has better price performance. Finally, suppose both machines can play the bitstream at 100% of the required bit rate. In this case, M2 is clearly better since it is less expensive, and the metrics confirm this comparison because they are 6.7 and 8.3.

Table 3 shows the PBSD metric for playing the sample bitstreams using ordered dithering on three workstations in our research group. The tests were run with image display accomplished using shared memory between the decoder and the frame buffer in the X server.

From the table we conclude that the HP is up to a factor

[3] For example, machines with graphics pipelines or parallel ALU's can overlap multiply-add and matrix operations.

of three less efficient on a price/performance basis. This example reveals a premium paid to achieve higher bit rates. Even though the SPARC 1+ achieves a PBSD metric of 4.84 for bitstream F, this represents only 34% of the bitrate. To achieve 100% of the bitrate for stream F, an HP is required, since multiple Sparc 1+'s can not be combined to achieve the bitrate.

On the whole, the HP 750 is between 4-5 times faster than the Sparc 1+ and nearly 2 times faster than the Sparc 10. If we extrapolate from these data points, we can expect the next generation of workstations to be able to support video at the quality of streams A, B and C (320 by 240 pixel video at 30 frames per second) using a software only solution.

Many factors will influence this comparison, including whether the X shared memory option is available to reduce copies between the decoder and the X server, whether the X server is local or across a network, and whether the file containing the compressed data is local or NSF mounted. All these changes can significantly effect the results.

8. Internet Distribution

We were amazed at the response we received when we distributed this code on the internet.[4] Within 6 weeks of announcing the availability of a portable software decoder for MPEG video on several newsgroups (e.g., alt.graphics.pixutils and comp.compression), over 500 people had FTP'd the software. Since then it has been retrieved by several thousand people. They have reported numerous bugs and suggestions for improvement, and they have contributed code to add features, fix bugs, and support new platforms.

We also received our first video mail when a user sent us an MPEG bitstream in a message.[5] We played the mail message with our decoder. We added definitions to play MPEG components to an extensible mail system using our decoder[4].

[4] You can FTP the software from toe.cs.berkeley.edu [128.32.149.117] in the directory pub/multimedia/mpeg.

9. Conclusions

Several conclusions can be drawn from this work. First, while IDCT performance is important, it is not the most critical process in a software decoder. Data structure organization and bit-level manipulations are critical.

Second, memory bandwidth is important on RISC processors. We suspect hardware implementations of MPEG will use fast static RAM and pipeline key operations (i.e., parsing, IDCT, reconstruction, etc.) to avoid this memory bandwidth problem.

Lastly, current generation workstations, like the HP 750, can decode 320 by 240 video sequences at 10-15 frames per second, within a factor of two of real-time performance. We anticipate real-time decoding with the new generation of workstations that will soon be available.

Acknowledgments

We want to thank the numerous people who ported the decoder to new platforms, supplied ideas and code to improve performance of the software, and provided bug fixes and extensions. While we do not have room to name all the people who have helped, important contributions were made by Todd Brunhoff of North Valley Research, Reid Judd of Sun Microsystems, Toshihiko Kawai of Sony, Tom Lane of the Independent JPEG Group, Arian Koster of Phil-

[5] The message was *uuencode*'d which converts a binary file to ASCII. *Uudecode* is a companion program that converts the ASCII back to binary. It works well with saved mail messages because it ignores message headers.

Stream	HP 750	SUN Sparc 1+	Sun Sparc 10	DECstation 5000/125
Cost	$43K	$7K	$22K	$10K
A	1.19	1.71	1.20	1.31
B	0.99	1.43	0.98	1.15
C	1.13	1.65	1.14	1.29
D	2.32	7.72	4.55	5.26
E	2.32	7.14	4.55	5.60
F	2.32	4.84	3.32	4.11
G	2.32	8.08	4.55	6.37

Table 3: Performance/Price ratios.

lips Research, and Paulo Villegas Nuñez of Telefonico, Madrid, Spain.

References

[1] Foley, James D. et al., *Computer Graphics: Principles and Practice*, 2nd edition. Addison-Wesley, Reading, Mass., 1990.

[2] ISO/IEC JTC1/SC2/WG10, "Digital Compression and Coding of Continuous-Tone Still Images", *ISO/IEC Draft International Standard 10918-1*, January 10, 1992.

[3] ISO/IEC JTC/SC29, "Coded Representation of Picture, Audio and Multimedia/Hypermedia Information", *Committee Draft of Standard ISO/IEC 11172*, December 6, 1991.

[4] Knack, K., "MIME silences multimedia critics," *LAN Computing*, Vol. 3, No. 5, May, 1992: pp 3.

[5] Lane, Tom, "JPEG Software," Independent JPEG Group, unpublished paper, December 1992.

[6] LeGall, Didier, "MPEG - A Video Compression Standard For Multimedia Applications," *Communications of the ACM*, April 1991, Vol 34, Num 4, pp 46-58.

[7] McMillan, Leonard and Lee Westover, "A Forward-Mapping Realization of the Inverse Discrete Cosine Transform," *Data Compression Conference '92*, IEEE Computer Society Press, Los Alamitos, CA., 1992.

[8] Rowe, Lawrence A. and Brian C. Smith, "A Continuous Media Player," *Proc. 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, San Diego, CA (Nov. 1992).

[9] Ulichney, Robert, *Digital Halftoning*,MIT Press, Cambridge, Mass. 1987.