# The Berkeley Software MPEG-1 Video Decoder

KETAN MAYER-PATEL
University of North Carolina
BRIAN C. SMITH
TippingPoint Technologies, Inc.
and
LAWRENCE A. ROWE
University of California at Berkeley

This article reprises the description of the Berkeley software-only MPEG-1 video decoder originally published in the proceedings of the 1st International ACM Conference on Multimedia in 1993. The software subsequently became widely used in a variety of research systems and commercial products. Its main impact was to provide a platform for experimenting with streaming compressed video and to expose the strengths and weaknesses of software-only video decoding using general purpose computing architectures. This article compares the original performance results with experiments run on a modern processor to demonstrate the gains of processing power in the past ten years relative to this specific application and discusses the history of MPEG-1 video software decoding and the Berkeley MPEG research group.

Categories and Subject Descriptors: H.4.3 [**Information systems applications**]: Communications Applications

General Terms: Performance

Additional Key Words and Phrases: Video compression, MPEG

## 1. INTRODUCTION

The Moving Pictures Experts Group (MPEG) was established in 1988 as an Experts Group of the International Standards Organization (ISO) and the International Electrotechnical Commission (IEC) to develop a standard for storing video and associated audio on digital media. The goal was to define a standard that could be used for playback of audio/video material from a CD player and for digital audio broadcasting. At the time, the bit rate available from a CD player was 1.5 Mbits/sec so that established the target bit rate. The codec design was biased to reduce the computational cost of decoding at the

This article is a revised version of Patel, K., Smith, B., and Rowe, L., Performence of a software MPEG video decoder, In *Proceedings of the 1st ACM International Conference on Multimedia* (New York, N.Y.), ACM, New York, 1993, pp. 75–82.
This research was supported by grants from the National Science Foundation (Grant MIP-9014940 and Infrastructure Grant CDA-8722788) and by donations from Fujitsu Network Transmission Systems, Fuji Xerox Palo Alto Laboratory, Hewlett-Packard Company, NEC Laboratories, Philips Research, and Sony Corporation.
Authors' addresses: K. Mayer-Patel, Department of Computer Science, Campus Box 3175, Siterson Hall, The University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175; email: kmp@email.unc.edu; B. C. Smith, TippingPoint Technologies, Inc., 7501 North Capitol of Texas Highway, Building B, Austin, TX 78731; email: bsmith@tippingpoint.com; L. A. Rowe, Computer Science Division—EECS, University of California, Berkeley, , Berkeley, CA 94720-1776; email: rowe@cs.berkeley.edu.

expense of encoding because the intended applications involved off-line authoring of material, and the business model was based on expensive encoders and inexpensive decoders as found in the audio CD business. A draft proposal was agreed upon in September 1990 and ISO/IEC released a final draft in November 1992 [ISO/IEC 1993]. The standard includes several parts including a video standard, an audio standard, and a systems standard that specifies how audio and video data is interleaved into one bitstream.

We began experimentation with audio/video playback using a video overlay board and a locally connected laserdisc player in early 1991. One experiment was the development of a hypermedia course on semiconductor manufacturing that included video clips from lectures given in a Berkeley class and video tours of semiconductor fabrication equipment and facilities. Evaluation of the courseware showed that short audio/video clips (e.g., less than 5 minutes) were highly valued by students and seemed to positively impact recall and learning [Schank and Rowe 1993]. The problem was that a laserdisc could only hold 60–90 minutes of video material, and we wanted to incorporate 20 hours of lecture material and demonstrations. We briefly considered pressing several laserdiscs and using a shared collection of laserdisc players, but this solution would not scale and did not exploit the emergence of digital media. The obvious solution was to digitize the material and stream it from a file server. So, we began to explore hardware and software decoding on workstations and streaming the media from a shared file server on a network. While hardware decoders could play full-sized audio/video streams, they were expensive (e.g., early boards cost \$10K-\$50K) and were not available for the workstations in the instructional laboratories at Berkeley at that time. Early experiments with Motion JPEG showed that software could decode at most 1–2 frames per second of CIF-sized images (i.e., $352 \times 240$ for NTSC and $352 \times 288$ for PAL).

The MPEG-1 standard was thought to be computational less expensive because it exploited temporal coding. We decided to implement an MPEG-1 video decoder for three reasons. First, we wanted to determine whether MPEG-1 video could be decoded in real-time using a software-only implementation on the then current generation desktop computers. Today we take for granted that software decoding is possible on modern computing devices, whether they are desktop systems, laptops, or even PDAs. No one, as far as we knew, had successfully implemented a software-only decoder for MPEG-1 CIF streams. Second, we needed to develop a portable software decoder for inclusion in the Continuous Media Player being developed at U.C. Berkeley [Rowe and Smith 1993]. And third, we wanted to contribute the source code to the public domain for the research community because many researchers wanted to experiment with MPEG-1 video technology. Several companies participating in the development of the MPEG-1 standard produced software-only encoders and decoders, but they would not make them available to the research community for competitive reasons.

This article describes the MPEG-1 video standard, the implementation of the Berkeley MPEG-1 video software decoder, the performance of the decoder as reported in 1993 and experiments run on a modern processor to demonstrate how processing power has improved over the past ten years, and a discussion of the history of MPEG-1 video software decoding and the Berkeley MPEG research group. It is a revised version of the paper that was originally published at the 1st International ACM Conference on Multimedia [Patel et al. 1993].

## 2. THE MPEG-1 VIDEO CODING MODEL

This section briefly describes the MPEG-1 video coding model. More complete descriptions are given in the standard [ISO/IEC 1993].

Video data can be represented as a set of images, $I_1, I_2, \ldots, I_N$, 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.

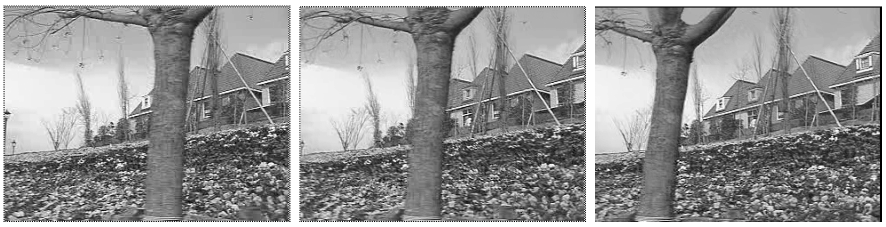

Fig. 1. A sample video sequence.

MPEG-1 video coding uses three techniques to compress video data. The first technique, called *transform coding*, is very similar to JPEG image compression [Pennebaker and Mitchell 1993]. 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 the frequency domain, making it straightforward to take advantage of (1).

In MPEG-1 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 the Cr and Cb values represent *chrominance* (color information). Because 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 (i.e., 4:2:0 sampling).

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 corresponding block from each of the (subsampled) Cr and Cb planes. Each of these blocks is then processed in the same manner as JPEG: the blocks are transformed using the DCT and the resulting coefficients are quantized, run length encoded to remove zeros, and entropy coded. The details can be found in the standard, but the important facts for this paper are: (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-1 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.[1] 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 encoding results in significant data compression.

Note, however, that the right edge of $I_2$ (and $I_3$) cannot 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. This strategy requires buffering the frames and introduces a small time delay in both encoding and decoding. $I_2$ can then be built using both $I_1$ and $I_3$. When a larger pool of reference

[1]This sequence is called the *flower garden* sequence. It was used in the development of the MPEG standards and is still frequently used in testing.

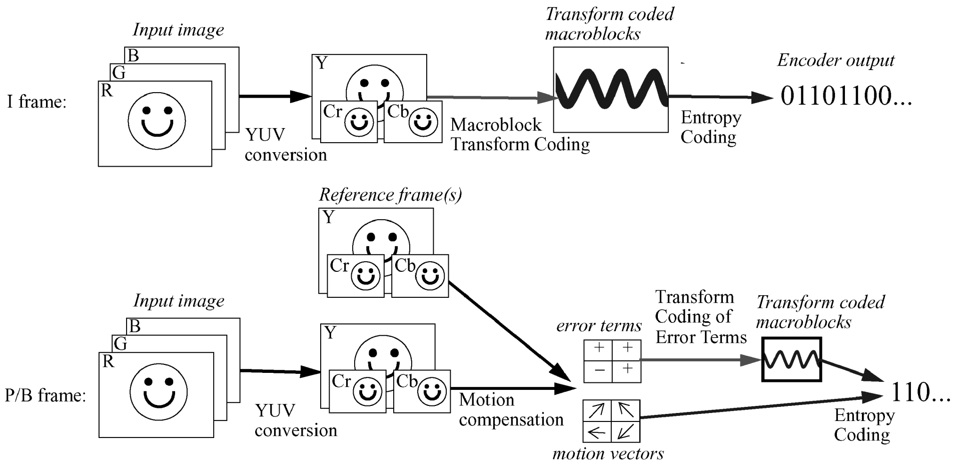

Fig. 2. The MPEG-1 encoding process.

frames is available, motion compensation can be used to construct more of the frame being encoded, thereby reducing the number of bits required to encode the frame. A frame built from one reference frame is called a *P-frame* (predicted), and a frame built from both a preceding frame and a subsequent frame is called a *frameB* (bi-directional). A frame coded without motion compensation, that is, using only transform coding, is called an *I-frame* (intracoded).

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 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 encoder determines the criterion for "best matching," which is typically the candidate motion vector that minimizes the sum of the absolute difference between the macroblock to be coded and the motion vector. The motion vector is represented in units of integral or half-integral pixels. When the motion vector is on a half-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 and transmitted.

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-1 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).
(3) The error terms must be decoded and incorporated (which includes an application of the IDCT).

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

In summary, MPEG-1 uses three techniques (i.e., motion compensation, transform coding, and entropy coding) to compress video data. The standard defines three types of frames, called I-, P-, and B-frames. These frames use zero, one, or 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.

Table I. Decoder Performance by Function

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

## 3. BERKELEY DECODER IMPLEMENTATION AND OPTIMIZATIONS

This section describes the implementation of the decoder and the optimizations implemented in the early 1990s to improve the performance of the decoder.

### 3.1 Implementation of the Decoder

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 envisioned using the decoder as a part of a larger system (the CM Player [Rowe and Smith 1993]), 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 such as synchronized audio).

The decoder was implemented in C using the X Windowing System for display. 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. By late 1994, the decoder had been ported to over 10 UNIX platforms, and it had been ported to PC/DOS and the Macintosh.

Preliminary analysis of the run-time performance indicated that dithering accounted for 60% to 80% of the CPU time, depending on the architecture. Consequently, we focused 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 needed 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. Using an ordered dither lead to more significant improvements (over a factor of 15). Dithering is discussed in detail in the performance section.

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. Table I shows the results.

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 [Lane 1992]. 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 operations. 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 to reduce memory traffic in reconstruction and dithering.

## 3.2 Optimizing the Decoder

Three kinds of low-level optimizations were used to improve the decoder: (1) general coding, (2) IDCT, and (3) 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, bounds checking was required because 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). 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 additions), 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 McMillan and Westover [1992]. 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-1 standard specifies that 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.

## 4. PERFORMANCE OF THE SOFTWARE DECODER

This section describes the performance of the decoder as reported in 1993 along with some comparisons using processors today. The last subsection presents a perspective on the history of video decoding

Table II. Sample Bitstreams

| Stream | Stream Size | Frame Size | I-Frame Avg Size | P-Frame Avg Size | B-Frame Avg Size | Frames/ Sec | Bits/ Pixel | Bits/ Second | I:P:B |
|---|---|---|---|---|---|---|---|---|---|
| A | 690K | $320 \times 240$ | 18.9K | 10.6K | 0.8K | 30 | .49 (50:1) | 1.12M | 10:40:98 |
| B | 1102K | $352 \times 240$ | 11.2K | 8.8K | 6.3K | 30 | .70 (34:1) | 1.78M | 11:40:98 |
| C | 736K | $352 \times 288$ | 23.2K | 8.8K | 2.5K | 25 | .47 (51:1) | 1.19M | 11:31:82 |
| D | 559K | $352 \times 240$ | 8.1K | 5.5K | 4.1K | 6 | .45 (54:1) | 0.23M | 6:25:88 |
| E | 884K | $352 \times 240$ | 12.4K | 9.1K | 6.5K | 6 | .70 (34:1) | 0.35M | 6:25:89 |
| F | 315K | $160 \times 128$ | 2.8K | — | — | 30 | 1.1 (20:1) | 0.67M | 113:0:0 |
| G | 1744K | $144 \times 112$ | 2.3K | 1.8K | 0.4K | 30 | .49 (49:1) | 0.24M | 294:293:1171 |

performance over the past ten years including a discussion of the major improvements discovered by other researchers after 1993.

Public domain MPEG-1 data was relatively scarce in 1992 because of the high stakes in commercializing the technology. Companies developing products did not want competitors to know how their encoders worked, so they were reluctant to show other people bitstreams produced by their encoders. We selected seven available bitstreams that we believed constituted a reasonable data set. Table II presents the characteristics of the bitstreams.[2] Five distinct coders were used to generate the data. The same coder generated bitstreams B, D, and E. 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 believed the best metric to judge the performance of the decoder was 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 at the required bit rate.

## 4.1 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 color (24-bit), gray scale and color-mapped display devices. Dithering to full color devices is tantamount to RGB conversion. Dithering to gray scale devices uses 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 Ulichney [1987]. 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.

[2]We did not identify the bitstreams in the original paper, because we wanted to include bitstreams given to us by companies who requested that we keep them private. We would not have been able to publish performance data that depicted realistic performance since the only public bitstreams used I-frame only QCIF encoding.

Table III. Relative Performance of Different Dithering Algorithms

| 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% |

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 reader is referred elsewhere [Ulichney 1987; Foley et al. 1993].

When implementing the decoder, we started with a straightforward implementation of the error diffusion algorithm with propagation of 4 error values (*FS*4). The first improvement we tried was to implement an error diffusion algorithm with only two propagated error values (*FS*2). 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 III shows the relative performance of these dithers along with a grayscale (*GRAY*) and 24-bit color dither (24*BIT*). The table also shows the performance of the decoder without dithering. These tests were run on an Hewlett-Packard (HP) 750, which was the fastest machine available to us at the time. 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 four bitstreams (D, E, F, and G) were playable at the required bit rate (i.e., the percentage was over 100%) using the *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 160 × 128 pixels, and stream G is even smaller at 144 × 128 pixels.

The remaining bitstreams can be played at approximately 50% of the required bit rate, which implied that workstations in 1993 could play around 15 fps. Second, notice that even without dithering, which is shown in the column labeled *NONE*, the decoder achieved only 2/3's of the required bit rate of a full-size, full motion video stream.

Notwithstanding this pessimistic result, private communications at the time with other groups working on decoders optimized for particular platforms told us 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

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

Table IV. Performance/Price Ratios

| 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 |

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.

## 4.2 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 IV shows the PBSD metric for playing the sample bitstreams using ordered dithering on four 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 750 is up to a factor 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 cannot 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. In 1993, we expected the next generation of workstations to be able to support software-only video decoding at the quality of streams A, B and C (i.e., 320 by 240 pixel video at 30 frames per second).

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 affect the results.

## 4.3 Performance Today

This section updates the performance experiments reported in the original paper with new experiments designed to highlight the advances in processors in the last ten years. As expected, software decoding performance exhibits almost linear speed up relative to processor speed. More surprisingly,

Table V. Bitstreams Used for New Experiments

| Stream | Stream Size | Frame Size | I-Frame Avg Size | P-Frame Avg Size | B-Frame Avg Size | Frames/ Sec | Bits/ Pixel | Bits/ Second | I:P:B |
|---|---|---|---|---|---|---|---|---|---|
| flower | 690K | $320 \times 240$ | 18.9K | 10.6K | 0.8K | 30 | .49 (50:1) | 1.12M | 10:40:98 |
| canyon | 315K | $160 \times 128$ | 2.8K | — | — | 30 | 1.1 (20:1) | 0.67M | 113:0:0 |
| mobile | 718K | $352 \times 240$ | 19.2K | 7.4K | 2.2K | 30 | .45 (54:1) | 1.14M | 10:40:100 |
| ballet | 719K | $360 \times 240$ | 19.9K | 11.3K | 3.8K | 30 | .65 (37:1) | 1.72M | 9:25:66 |
| brunner | 1.1M | $368 \times 160$ | 2.7K | 0.5K | – | 30 | .27 (89:1) | 478K | 363:181:0 |
| hlander | 3.2M | $384 \times 288$ | 7.4K | 3.8K | — | 30 | .45 (54:1) | 1.5M | 343:171:0 |

we find a larger range of performance on modern processors between streams of different characteristics suggesting a cost difference between different encoding methods within the standard.

Table V lists the names and characteristics of the bitstreams used for the new set of experiments. We were able to identify and obtain two of the seven bitstreams used in the original set of experiments. The streams listed as “flower” and “canyon” correspond to “Stream A” and “Stream F” respectively in Table II. Additionally, we were able to obtain another bitstream (listed as “mobile”) encoded in 1993 that exhibits characteristics that closely matches “Stream D” from the original bitstream set. We also included three more recently encoded bitstreams named “ballet,” “brunner,” and “hlander.”

For the two original streams used (i.e., “flower” and “canyon”), we are able to include the original performance measurements from three of the platforms used in the original paper. These include the Sun Sparc 1+, the HP 750, and the DEC 5000 running at clock speeds of 36 MHz, 66 MHz, and 150 MHz, respectively. We added to this set the results of new experiments run on six Pentium platforms purchased over the course of the last seven years running at clock speeds of 400 MHz, 700 MHz, 900 MHz, 1.0 GHz, 1.8 GHz, and 2.9 GHz. The experiments were run on these machines using the FreeBSD 2.4 operating system.

Figures 3, 4, and 5 show the results of the new experiments. In all three graphs, the x-axis is the processor speed in MHz. The legend of all three graphs is ordered from top to bottom to reflect the relative order of the lines on the graph.

Figure 3 shows performance in terms of frames per second. At the time of the original experiments, we estimated that processors of the day allowed software decoding of small-sized video (i.e., $320 \times 240$) that was a factor of two from real-time. This prediction is born out by the almost perfectly linear speed up in decoding performance as processors became faster. The difference in decoding performance relative to different bitstreams is due to the fact that these bitstreams do not all have the same frame size or compression rate. Thus, “canyon” which is quite small in frame size achieves an astronomical frame rate on our fastest processor relative to the other streams.

Figure 4 normalizes for frame size by showing performance in terms of macroblocks per second. The horizontal guidelines indicate the necessary rate of macroblocks per second decoded required to achieve a particular frame geometry at 30 frames per second. Each horizontal guideline is labeled with its corresponding frame geometry on the right. Interestingly, the performance required to decode MPEG-1 video streams for a high-resolution frame geometry in line with current HDTV standards (i.e., $1280 \times 720$) is only just now becoming feasible given the original decoder. We believe that optimizing the code to exploit special-purpose instructions and GPU’s would allow the code to decode an HDTV-sized image in real-time.

Figure 5 normalizes for both frame size and compression ratio by showing decoding speed in terms of bits per second decoded. One interesting aspect of this graph is that there is still a fairly wide variation in decoding performance for different bitstreams on a particular processor. These differences can be attributed to the fact that different streams employ MPEG-1 coding techniques in different ways.

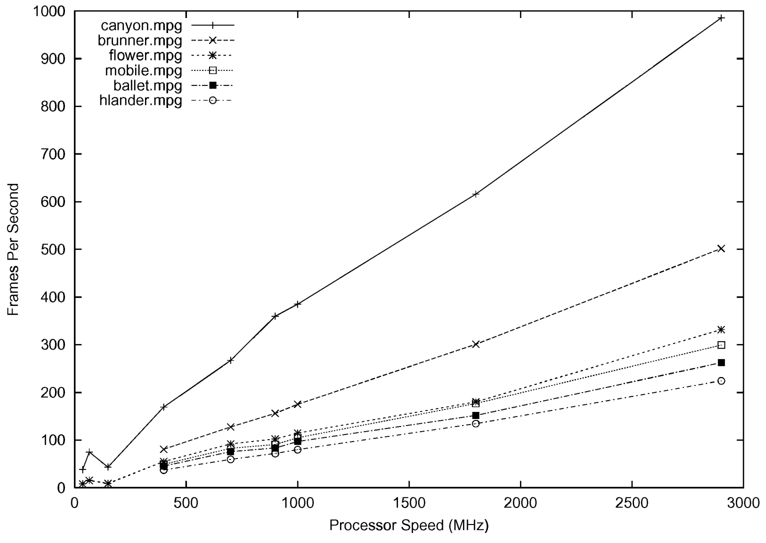

Fig. 3. Frames Decoded Per Second.

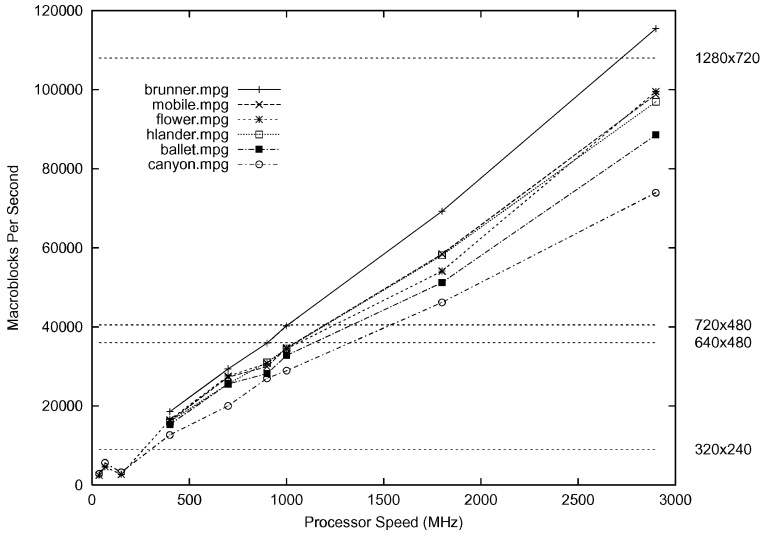

Fig. 4. Macroblocks Decoded Per Second.

For example, unlike the other streams “canyon” does not make use of motion compensation and thus does not contain P- or B-frames. Similarly, each stream has some set of features (e.g., frame geometry, proportion of different frame types, etc.) that distinguish it from the other streams. The relative cost of these different techniques is what accounts for the variation in decoding speed on a specific processor.

Finally, it is important to note that these experiments were run using the public domain code released in late 1995. It has not been optimized to use the special-purpose instructions or GPU services provided in a modern computing system.

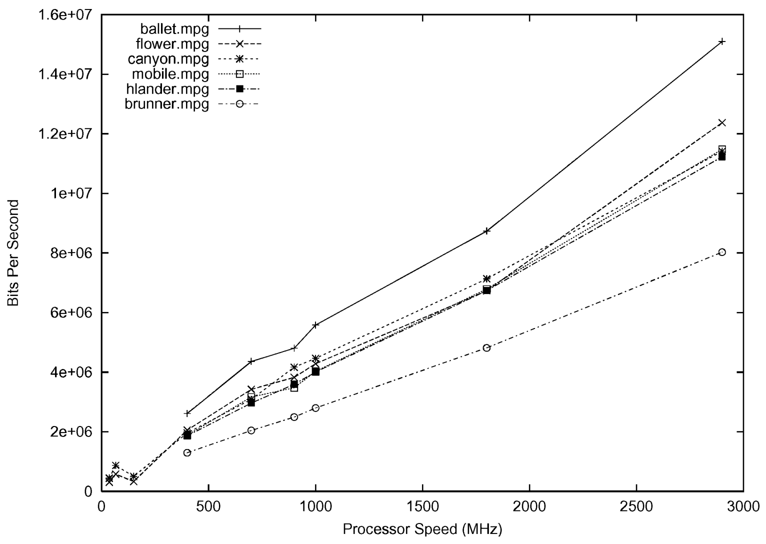

Fig. 5. Bits Decoded Per Second.

## 4.4 Historical Perspective on Performance

The Berkeley MPEG-1 video decoder was the first widely distributed public domain system for decoding video streams on a desktop computer. Performance was within a factor of two of real-time performance on workstations at the time. Almost all of the performance optimizations and implementation short cuts that we used to achieve this basic level of performance involved reducing the amount of memory touched by the decoder. It quickly became clear that memory performance and the effects of cache structure and organization mattered as much if not more than raw CPU speed. Although our goal was never to produce the highest performance code, other researchers continued to explore ways to speed-up MPEG-1 decoding. In this section, we review some of these other efforts. A common theme of these efforts is highly optimized use of memory management facilities to further reduce the number of memory copies required. Ultimately, these efforts led to the development of special-purpose instruction sets which exploited subword-level parallelism. These instructions effectively quadruple both arithmetic performance and memory bandwidth to the CPU for 8-bit operands.

The first software-only decoder to play an MPEG-1 *constrained parameters bitrate* (CBP) video stream was produced at Digital Equipment Corporation by Sam Ho [1993, Personal communication]. CBP is the baseline for MPEG-1 video that is 24- or 30-frames per second and CIF-sized images at 1.5 Mbs. This decoder ran on the Alpha RISC processor, which had an on-chip memory cache and pipelined architecture. The software was carefully optimized to avoid cache misses. In essence, every memory copy in the code was optimized so that data in the cache would be accessed before off-chip data because the performance of the Alpha was severely impacted when off-chip data was needed.

The impact of data cache performance on software decoding was later studied by Soderquist and Leeser [1997]. They showed that conventional cache-memory designs are inefficient and suggested several changes to the memory architecture to improve memory performance.

Another research group at Digital Equipment Corporation developed improved dithers for displaying decompressed video images [Bahl et al. 1995]. In retrospect, we spent more time and effort on dithering and color space conversion than we did on other aspects of the code. Some of the effort was required because different computer systems used different color models. Today, almost all computers use 24-bit color because video memory and graphics processors (GPU) are inexpensive. The other reasons we spent so much time on dithers was that users wanted different features optimized (e.g., decode performance or decoded image quality) which required different routines.

Researchers at Hewlett-Packard produced the first MPEG-1 decoder that could play an MPEG-1 CBP stream including both audio and video in real-time. The code was carefully optimized to run on the HP Precision Architecture (PA) RISC processor Bhaskaran et al. [1995] and to exploit system parallelism. The code took advantage of two significant performance optimizations. First, color space conversion was moved to the GPU, which significantly reduces CPU resource requirements, as shown above and in experiments conducted by Lee [1995]. And second, the PA-RISC architecture had subword parallelism instructions (i.e., a single-instruction that operated on multiple data values stored in a 32-bit word). The architecture also supported saturation arithmetic operations on the two 8-bit values simultaneously. Saturation arithmetic implements the underflow and overflow checking and clipping discussed above. Consequently, operations on pixel values could be implemented in parallel in one instruction rather than sequentially in multiple instructions. This architecture improved the performance of many operations including the IDCT and inverse quantization. This small architectural change to the PA-RISC processor was an early example of special-purpose multimedia instructions that are now provided on all modern processors (e.g., INTEL MMX, SPARC VIS, etc.). While this MPEG-1 decoder used special-purpose hardware, it was primarily a software decoder that exploited hardware parallelism in the system.

In modern systems, hardware support for video decoding often exists in three different components. First, single-instruction multiple-data (SIMD) instruction set extensions such as MMX are widely exploited. Many useful instructions are added including saturation arithmetic, a multiple/accumulate operation common to signal processing applications, and operations to manipulate and convert between planar and packed data representations. Second, increasingly the GPU is being used as a general computing resource. In particular, because the graphics pipeline lends itself to a parallel architecture in order to simultaneously process different portions of the screen (or of the 3D model being rendered), an argument can be made that incorporating multimedia-specific SIMD instructions is better suited to the GPU than the CPU. Some groups are beginning to explore adding new features using these additional processing elements (e.g., traditional video effects such as titling, blue screen compositing, etc.). Finally, some vendors have added hardware to a DVD-ROM player to accelerate decoding and playback. Similar attempts were made with memory architectures in the late 1990s (e.g., Huffman decoding), although as far as we know, they have not been widely adopted.

Regardless of hardware advances, software-only decoding of full-sized MPEG video, including HDTV, is possible on modern processors. Some observers argue that current processors are fast enough, so higher speeds and optimized software decoders are good enough. Our view is that improved performance is still needed so that systems can support multiple simultaneous streams and larger image sizes (e.g., high-definition image sizes). The next generation distributed collaboration systems must include viewing several people at different remote locations, which implies multiple streams. Moreover, high-bandwidth applications like tele-immersion are likely to employ tens if not hundreds of video streams.

## 5. BERKELEY MPEG-1 VIDEO TOOLS HISTORY

The MPEG-1 video decoder (`mpeg_play`) was released on the Internet in October 1992. Within six weeks of announcing the availability of the software on our FTP site, over 500 copies had been downloaded, and people were asking for sample MPEG-1 video files to play. We published the sample files we had on the FTP site and over the next six months over 10,000 copies of the video files were downloaded. These numbers sound small today, but remember this was before widespread use of the World Wide Web. In fact, NCSA released Mosaic in early 1993 and encouraged users to install `mpeg_play` as an external application to play MPEG-1 video files. Needless to say, the number of `mpeg_play` downloads increased dramatically. The last time we estimated the number of downloads was 1996 when the number had reached 20M copies.

We developed the code on Unix, ported it to a variety of available workstations, and distributed it as open source. Many people contributed performance optimizations and bug fixes. Other people developed callable libraries so the decoder could be used by other applications, and they developed GUI interfaces to select the file to be played and control the decoder. Eventually the code was ported to the Macintosh and PC/DOS. The widely distributed Macintosh Sparkle is based on the Berkeley decoder. Sadly, the person who ported the code to the PC redistributed it as a shareware binary and did not maintain the code.

Over the next five years we developed several more MPEG-1 applications, now called the *Berkeley MPEG-1 Video Tools* [Rowe et al. 1995], and continued to experiment with various streaming media projects using the code. We implemented an encoder because people kept requesting MPEG-1 video files, and we needed an encoder for research we were doing on video segmentation and video-on-demand. An undergraduate student (Dan Wallach) and one of the co-authors of this paper (Brian Smith) implemented a simple I-frame encoder that was later extended to a full-function MPEG-1 video encoder [Gong and Rowe 1994]. The encoder was modified by another undergraduate (Eugene Hung) to read input frames from output produced by another program, which makes it very useful for producing short movies directly from simulation programs. It was also used in several experiments on parallel encoding [Shen et al. 1995].

Several tools were developed to analyze MPEG-1 video streams so we could debug our software and determine the coding tactics used by other systems. These tools included:

(1) `mpeg_stat`—A tool for analyzing the bit rate, frame size and pattern, block coding distribution, and other useful information.
(2) `mpeg_blocks`—A tool for displaying how individual blocks are encoded in a frame. It displays macroblocks using a single color representing how the block was encoded (e.g., red is an I-block, blue is a P-block, etc.). This tool allowed us to examine the tactics used by an encoder alongside the image itself.
(3) `mpeg_bits`—A tool for displaying how many bits are used to encode each block. A color selected from a sequence of colors ranging from white, meaning no bits, to blue, meaning many bits, is displayed for each macroblock. This tool allowed us to debug a simple bit rate control algorithm implemented in the encoder and to analyze rate control by other encoders.

The `mpeg_stat` program output was text printed to a shell or console window so it was more portable than the other two programs that used an X Windows graphical user interface. To this day, many people request PC or Macintosh versions of the graphical analysis tools.

The decoder was also integrated into the Continuous Media Toolkit (CMT) being developing at the time [Mayer-Patel and Rowe 1997]. The CMT media player, which supported synchronized audio and video playback from media files distributed on different hosts on the Internet, supported full-function VCR commands, by which we mean pausing, stopping, rewinding, jumping to a random position, and playing forward and backward at variable speeds. Supporting this functionality for MPEG-1 video was challenging because the number of frames that had to be decoded before the next frame can be displayed is unbounded when jumping to a random position or when playing backwards and because audio synchronization and variable-rate playback required intelligent heuristics to drop frames [Rowe et al. 1994]. The key idea to solving this problem was to allow the decoder to tell the file server which frame to send next, which necessitated building a frame index in the file representation of the video, and to monitor the decoding rate of the client processor (i.e., macroblocks decoded per second). The decoding rate allows the client software to estimate whether the next frame to be decoded will be done in time. The performance objective was to minimize jitter between frames so playback appears smooth even if frames are being dropped.

The Berkeley MPEG-1 video tools were the defacto standard for video research for several years. Many researchers and companies redistributed the code in products and used it in their own work. Because we developed the code when processors were too slow to decode a stream in real-time, `mpeg_play` did not include logic for synchronization or control of frame presentation. Instead, the decoder simply went as fast as it could, assuming that it would never go fast enough. So, we were somewhat surprised (and delighted) when a bug report was received saying that the decoder was playing the sequence too fast. Clearly, Moore's law was catching up with our code. On the other hand, another interesting comment came from a developer at a commercial MPEG hardware encoder company who said they loved the Berkeley decoder because it was so slow! It allowed them to watch encoded streams more closely when debugging their encoder. We were surprised to receive a complement for writing slow software since software developers are almost always criticized for producing software with poor performance.

Several companies used the code to test architectural design alternatives for next generation processors and systems. They shared with us their studies of the performance of the decoder on various architectures and proposed architectures. In some cases they provided suggestions that significantly improved the performance of the public domain code.

And, one day in the spring of 1992, we received our first video e-mail. A researcher in Europe had encoded a short video showing his car and apartment. He converted it to ASCII and sent it to us in a message. It was a simple matter to convert the file back to binary and play it using our decoder. As we played the video, we were convinced again of the impact that audio and video has on communication. We later added a MIME-type to our mail system so we could easily send video files as e-mail attachments.

Development on the software stopped in mid-1996 because our intention was never to produce a product or spend a career optimizing video codecs. And, by that time, Stefan Eckert's encoder and decoder optimized for the PC and later extended to handle MPEG-2 video had been released [Eckert 1995]. The research question was answered so we moved on to other topics including webcast production, distributed collaboration, lecture capture, and video processing algorithms.

## 6. CONCLUSIONS

This article describes the software developed by the the Berkeley MPEG Research Group including the first widely distributed public domain MPEG-1 video decoder. Results published at ACM Multimedia 1993 were presented along with performance experiments run on modern computer systems which dramatically show the effect of computer performance over the past decade. These new experiments shows that full-sized 30-fps video can be decoded in real-time and that with modest changes HDTV-sized images can be decoded too. Nevertheless, we still believe further performance improvements are needed in video decoding so that many streams can be decoded simultaneously and larger image sizes can be used.

The article also presented an historical perspective on the development of software decoding of video over the past decade and our experience distributing an extremely popular open source software package.

ACKNOWLEDGMENTS

The MPEG video research at Berkeley would not have been possible without the generous financial support provided by our corporate sponsors Fujitsu, FX PAL, HP, NEC, Philips, Sony, and by The National Science Foundation. Toshihiko Kawai from Sony generously provided our first MPEG-1 encoded bitstream that allowed us to debug the first version of the decoder. We are also grateful to Chad

Fogg of Chromatic Research who answered many questions about the standard and discussed our work with the MPEG standards committee. Many other researchers generously provided advice and patches to fix bugs and optimize performance including Todd Brunhoff of North Valley Research, Reid Judd of Sun Microsystems, Tom Lane of the Independent JPEG Group, Arian Koster of Philips Research, Paulo Villegas Nunez of Telefonico, Jef Poskanzer of NETPBM fame, Eiichi Kowashi of Intel, Avideh Zakhor of U.C. Berkeley, Rainer Menes of the Technical University of Munich, Robert Safranek of ATT, and Jim Boucher of Boston University. Over the years, many students contributed to the Berkeley MPEG research group including Steve Smoot, Kevin Gong, Eugene Hung, Radhika Malpani, Doug Banks, Sam Tze-San Fung, Darryl Brown, Kim Liu, and Dan Wallach.

## REFERENCES

Bahl, P., Gauthier, P., and Ulichney, R. 1995. Software-only compression, rendering, and playback of digital video. *Digi. Tech. J. 7*, 4, 52–75.

Bhaskaran, V., Konstantinides, K., Lee, R., and Beck, J. 1995. Algorithmic and architectural enhancements for real time MPEG-1 decoding on a general purpose RISC workstation. *IEEE Trans. Circ. Syst. Video Tech. 5*, 5 (Oct.), 380–386.

Eckert, S. 1995. High performance software MEPG video playback on PCS. In *Proceedings of the SPIE Conference on Digital Video Compression: Algorithms and Technologies* (Bellingham, Wash.), Vol. 2419. 446–473.

Foley, J., Dam, A. V., Feiner, S., and Hughes, J. 1993. *Introduction to Computer Graphics*. Addison-Wesley, Reading, Mass.

Gong, K. and Rowe, L. 1994. Parallel MPEG-1 video encoding. In *Proceedings of the International Picture Coding Symposium (PCS'93)* (Sacramento, Calif.).

ISO/IEC 1993. *ISO/IEC 11172-1993: MPEG-1 coding of moving pictures and associated audio at up to about 1.5 Mbits/second*.

Lane, T. 1992. *JPEG software*. Independent JPEG Group. http://www.ijg.org/.

Lee, R. B. 1995. Realtime MPEG video via software decompression on a PA-RISC processor. In *Proceedings of IEEE COMPCON*. IEEE Computer Society Press, Los Alamitos, Calif., 186–192.

Mayer-Patel, K. and Rowe, L. 1997. Design and performance of the berkeley continuous media toolkit. In *Proceedings of the SPIE Conference on Multimedia Computing and Networking*. Vol. 3020. 194–206.

McMillan, L. and Westover, L. 1992. A forward-mapping realization of the inverse discrete cosine transform. In *Proceedings of Data Compression Conference*. IEEE Computer Society Press, Los Alatimos, CA.

Patel, K., Smith, B., and Rowe, L. 1993. Performance of a software MPEG video decoder. In *Proceedings of the 1st ACM International Conference on Multimedia* (New York, N.Y.), J. Garcia-Luna and P. Venkatrangan, Eds. ACM, New York, 75–82.

Pennebaker, W. and Mitchell, J. 1993. *JPEG—Sill image data compression standard*. Van Nostrand Reinhold, New York, N.Y.

Rowe, L. A., Smoot, S., Patel, K., Smith, B., Gong, K., Hung, E., Banks, D., Fung, S. T.-S., Brown, D., and Wallach, D. 1995. *Berkeley MPEG-1 Video Tools*, Version 1, Release 2. http://www.bmrc.berkeley.edu/mpeg.

Rowe, L., Patel, K., Smith, B., and Liu, K. 1994. MPEG video in software: Representation, transmission and playback. In *Proceedings of the SPIE Conference on High-Speed Networking and Multimedia Computing* (Bellingham, Wash.), A. Rodriguez, M.-S. Chen, and J. Maitan, Eds. Vol. 2188. 134–144.

Rowe, L. and Smith, B. 1993. A continuous media player. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (La Jolla, Calif.). Springer-Verlag, Lecture Notes in Computer Science, vol. 712. New York, 376–386.

Schank, P. and Rowe, L. 1993. The design and assessment of a hypermedia course on semiconductor manufacturing. *J. Educat. Multimed. Hypermed. 2*, 3, 299–320.

Shen, K., Rowe, L., and Delp, E. 1995. A parallel implementation of an MPEG-1 encoder: Faster than real time! In *Proceedings of the SPIE Conference on Digital Video Compression: Algorithms and Technologies* (Bellingham, Wash.). A. Rodriguez, R. Safranek, and E. J. Delp, Eds. Vol. 2419. 407–418.

Soderquist, P. and Leeser, M. 1997. Optimizing the data cache performance of a software MPEG-2 video decoder. In *Proceedings of the 5th ACM International Conference on Multimedia*. ACM, New York, 291–301.

Ulichney, R. 1987. *Digital Halftoning*. MIT Press, Cambridge, Mass.

Received November 2004; accepted November 2004