Pure Java-based Streaming MPEG Player

Osama Tolba, Hector Briceño, and Leonard McMillan[*]
Laboratory for Computer Science, MIT

ABSTRACT

We present a pure Java-based streaming MPEG-1 video player. By implementing the player entirely in Java, we guarantee its functionality across platforms within any Java-enabled web browsers, without the need for native libraries. This allows greater use of MPEG video sequences, because the users will no longer need to pre-install any software to display video, beyond Java compatibility. This player features a novel forward-mapping IDCT algorithm that allows it to play locally stored, CIF-sized (352 x 288) video sequences at 11 frames per second, when run on a personal computer with Java "Just-in-Time" compiler. The IDCT algorithm can run with greater speed when the sequence is viewed at reduced size; e.g., performing approximately ¼ the amount of computation when the user resizes the sequence to ½ its original width and height. We are able to play video streams stored anywhere on the Internet with acceptable performance using a proxy server, eliminating the need for large-capacity auxiliary storage. Thus, the player is well suited to small devices, such as digital TV set-top decoders, requiring little more memory than is required for three video frames. Because of our modular design, it is possible to assemble multiple video streams from remote sources and present them simultaneously to the viewers (i.e. picture-in-a-picture style), subject to network and local performance limitations. The same modular system can further provide viewers with their own customized view of each session; e.g., moving and resizing the video display window dynamically, and selecting their preferred set of video controls.

Keywords: Java, MPEG, forward-mapping IDCT, streaming video.

1. INTRODUCTION

Due to the recent explosion of digital video and its applications in the Internet and broadcast media, there is a growing need for Internet-based video players. Some Internet streaming video players already exist, such as RealPlayer by RealNetworks, Inc., but they are platform dependent and require downloading and updating the player code regularly. We believe that players implemented using the new Java Media Framework (JMF) have some limitations too. While JMF implementations can play MPEG-1 video they still rely on native libraries that must be downloaded and installed on the browser's machine.[7] JMF also imposes other limitations; for example, low-level access to picture data is prohibited. We set out to explore whether a player could be implemented entirely in Java. Here are some of the advantages of Java that led us to choosing it for our player, and shortcomings of Java that we have encountered.

Advantages of a Java-based player:

1. Programs written entirely in Java run across platforms within any Java-enabled Web browser or other Java Virtual Machines (JVM' s), without the need for native libraries. This allows greater use of MPEG video sequences, because the users will no longer need to pre-install any software plug-ins to display video. All that is required is Java compatibility, preferably with Just-in-Time (JIT) compiling.
2. Java's small footprint and availability for small devices.
3. Java programs are compact. The size of the Jar file for our minimal MPEG player is 40 kilobytes.
4. Extensive networking capabilities are built into the language, making it easy to write programs that use Internet communication.

Shortcomings of Java:

[*] Address: 545 Technology Square, Cambridge, MA 02139. E-mail: {tolba, hbriceno, mcmillan}@graphics.lcs.mit.edu

1. Programs written in Java—an interpreted language—run slower than ones written in compiled languages, such as C, even with the aid of JIT compilers which translate Java's byte code into native code.
2. Another performance shortcoming of Java lies within its windowing toolkit, the AWT. In our results we show that the video's update rate is greatly influenced by the rate at which the screen component can load and draw the new image.
3. While Java's and Internet browsers' security restrictions are useful for many applications, they create extra work for programmers wishing to establish read-only network connections. Such connections are commonplace in a Web browser but a Java applet is not allowed to perform them unless the browser's security settings are lowered. We have found the task of lowering the security restrictions cumbersome and decided to augment our applications with a server-side proxy server residing on the same host as the applet that makes the necessary network connection and forwards the data to the applet.
4. Java's application programming interface (API) is rapidly evolving. While providing great enhancements, such pace burdens the programmer with compatibility issues. For example, our initial implementation employed Java's older version 1.0 but was extremely slow to update an animated image. Therefore, the final implementation, which uses version 1.1, is not guaranteed to run on every Java machine.
5. Because Java is new and evolving, different Java machines have widely varied performances. This is most evident in the ten-fold increase in performance when using the JIT compilers.

Many video sources in the Internet can be extremely long and in some cases arbitrarily long as in the case of live video. In the past, players downloaded the entire video sequence before playing it. This initial waiting period limited the broad utilization of video. Recently video players have adopted the strategy of decoding video sequences as they are received; a technique referred to as *streaming*. Streaming has the side-benefit of constant storage requirements at the receiver. Our video player is capable of playing streaming video from either local files or the Internet. Currently we are using the HTTP protocol[3] that is compatible with all web-servers to fetch video from the Internet, but other protocols can be incorporated into our modular framework.

In section 2 we give a description of the system architecture we adopted for the player. Specific details about the inverse discrete cosine transform (IDCT) algorithm employed in this player are given in section 3, while Sections 4 and 5 include the results and discussion. This player was implemented as part of the ongoing *Computational Video* project at MIT.[8]

2. SYSTEM ARCHITECTURE

We have adopted a modular design of the video player based on source-player (producer-consumer) architecture. In this architecture, a single Java canvas or panel is overloaded to act as the player associated with a source object that reads bits from the video stream and decodes the individual frames. Any number of these players can be embedded in a Java applet or application. The applet may also host a panel of video controls resembling a video cassette player that communicates user interaction to all the players in the applet. The player panel has a similar set of controls in the form of a popup menu with additional controls that allow the viewer to change the size and appearance of the video component. Such modular design allows the users to assemble their own customized sessions including any number and shape of video windows and controls. We envision a scenario where a variety of video controls with different styles and functionality is available, perhaps via the Internet. The session can be configured dynamically as well, for example to move or reduce the footprint of a player to expose the video players behind it.

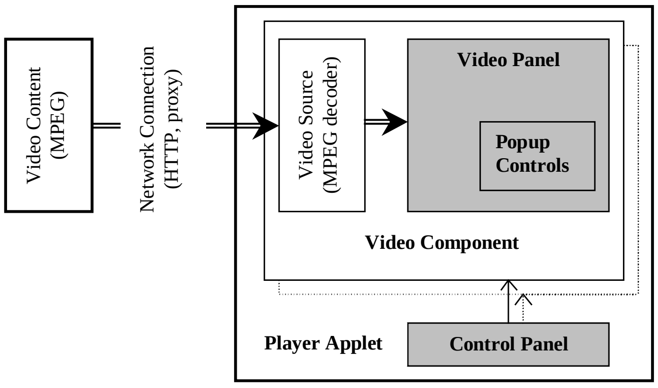

Figure 1. Diagram showing the source-player architecture of the video player, with visual parts shaded.

Under this architecture the video producer can implement any video CODEC, in this case is an MPEG decoder. The decoder reads its bits from an input stream residing on the applet's HTTP server, or any other server via a proxy server as discussed later. Instead of downloading the entire file beforehand, the producer begins decoding the video stream immediately, thereby eliminating the need for large amounts of memory. However, because of the structure of the MPEG stream, which includes forward and backward prediction video frames, the decoder must store two reference frames in addition to the one currently being decoded. Hence, the overall memory requirements of this decoder consist of three YUV frames, double-buffered display, room for lookup tables, and limited input buffer.

The need for a proxy server arises from Java's and Web browsers' security limitations. Most browsers, in their default settings, do not permit Java applets to make network connections, such as the ones needed to play video sequences from other Internet locations. A proxy is a server that acts as an intermediary, forwarding bits from a source server to a destination client that cannot make the direct connection itself. Naturally, the proxy server has a limited buffer in order to reduce network communication and enhance performance. We implemented this server using the C language for efficiency and because it does not affect the player's portability.

3. FORWARD-MAPPING IDCT

MPEG decoding involves many steps, as shown in Figure 2. A straightforward implementation of these procedures in Java would prove to be too slow for practical use. In order to optimize this process it is necessary to identify its bottlenecks. We have found the IDCT computation to be a major one.[†] In this section we describe a novel IDCT algorithm we have developed based on McMillan and Westover's forward-mapping IDCT (FMIDCT) and the table lookup technique that combines the inverse quantizer and the IDCT. The performance gains allow us to decode MPEG streams in Java software without native libraries or hardware acceleration.

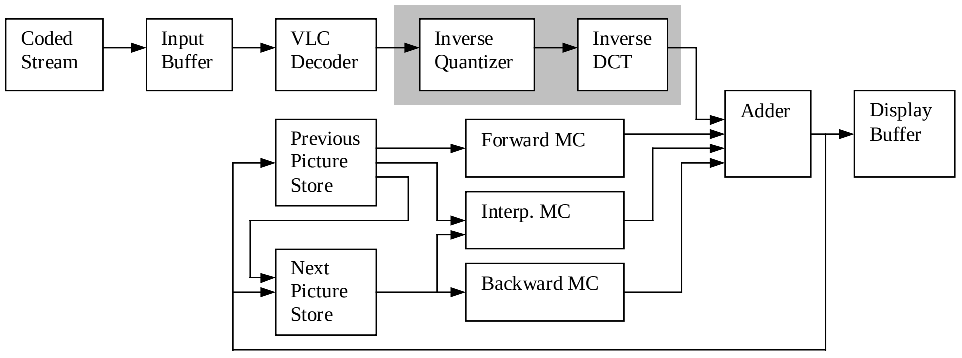

Figure 2. MPEG decoder block-diagram, with shaded area showing where the FMIDCT optimizes the process.

3.1 Background

McMillan and Westover presented an algorithm for computing the IDCT that exploits the sparseness of the typical IDCT input sequence.[4] The procedure involves computing the individual contributions of each input vector element and accumulating its contribution into the final result. This style of computation is commonly referred to as *forward-mapping* evaluation. This technique for the IDCT can be easily derived from the definition of the type II two-dimensional NxN IDCT given below:

$$O_{x,y} = \frac{2}{N}\sum_{v=0}^{N-1}\sum_{u=0}^{N-1} f_u f_v \cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right] I_{u,v}\,, \tag{1}$$

[†] If this were not true, there would be no significant speedup in decoding predicted (P or B) frames, as opposed to intra-coded (I) frames. In our experiments, I-frames take twice as long to decode as other predicted frames on the average.

$$\text{where } f_i = \begin{cases} \frac{\sqrt{2}}{2} & i = 0 \\ 1 & otherwise \end{cases}$$

This equation can be expressed as a matrix product as follows:

$$\mathbf{O} = \mathbf{C}\,\mathbf{I}\,, \tag{2}$$

where **O** and **I** are column vectors resulting from the row enumeration of $O_{x,y}$ and $I_{u,v}$, and **C** is a N^2xN^2 matrix.

The contribution of each element $i_{u,v}$ of the input vector **I** is the product of $i_{u,v}$ and a particular column within matrix **C**. This column is the unit-valued basis vector $\mathbf{K}_{u,v}$, unique to input $i_{u,v}$. One can easily observe that zero-valued elements of the input vector make no contribution to the output sequence. Furthermore, while each unit-valued basis vector $\mathbf{K}_{u,v}$ has N^2 coefficients there are at most N(N+2)/8 unique coefficients if the sign is ignored. For an 8x8 transform the number of unique coefficients for each of the basis vectors is summarized in Table 1.

v\u	0	1	2	3	4	5	6	7
0	**1**	**4**	**2**	**4**	**1**	**4**	**2**	**4**
1	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**
2	**2**	**8**	**3**	**8**	**2**	**8**	**3**	**8**
3	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**
4	**1**	**4**	**2**	**4**	**1**	**4**	**2**	**4**
5	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**
6	**2**	**8**	**3**	**8**	**2**	**8**	**3**	**8**
7	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**

Table 1. Number of unique coefficients for the basis vectors in an 8x8 transform.

The forward-mapping approach takes advantage of this repetition by performing the minimum number of multiplies, corresponding to the number of unique coefficients within the basis functions, and then performing the appropriate add or subtract into the output vector.

In practice the input sequence presented to the IDCT is generated by a dequantization step. This dequantization requires a unique multiplication for each element in the input sequence, and potentially other non-linear operations. In the FMIDCT the dequantization and the basis function scaling for each element of the input sequence are combined into a single table lookup. For an 8x8 transform this requires an M entry table where each table entry is composed of K elements, where Table 1 determines K. The contents of a table entry reflect both the dequantization and subsequent scaling of the unique coefficients of basis function. McMillan and Westover also described a technique for reducing the number of table elements and deferring their generation. As a result, the direct computation of the two-dimensional IDCT can typically be accomplished with fewer than 10 adds or subtracts and virtually no multiplies per output element, which compares favorably to any other fast IDCT implementation.

3.2 Improvements to the FMIDCT

In the remainder of this section we describe new improvements to the FMIDCT, achieved by exploiting the symmetries of the basis functions. These improvements reduce the required number of additions by a factor of four. It uses a smaller accumulator array that can potentially fit in registers, which greatly enhances the performance. Also, we reduced the size of the lookup tables, improving their hit-rate, and share them between dequantizers.

Each of the two-dimensional basis functions associated with a given input can be classified into one of four symmetries as shown below:

$$\text{Type A:} \begin{bmatrix} Q & H \\ V & D \end{bmatrix}, \text{ Type B:} \begin{bmatrix} Q & -H \\ V & -D \end{bmatrix}, \text{ Type C:} \begin{bmatrix} Q & H \\ -V & -D \end{bmatrix}, \text{ and Type D:} \begin{bmatrix} Q & -H \\ -V & D \end{bmatrix}, \tag{3}$$

where Q is the upper-left quadrant of the basis function, $H = Q\,R$, $V = R\,Q$, $D = R\,Q\,R$, and $R = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$.

In other words, *H* is the horizontal mirror image of *Q, V* is its vertical mirror image, and *D* is mirrored in both dimensions. The recognition of these basis function symmetries allows the number of accumulators to be reduced by a factor of four. Table 2 classifies the symmetries of the 64 basis functions in an 8x8 transform.

V \ U	0	1	2	3	4	5	6	7
0	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
1	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**
2	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
3	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**
4	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
5	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**
6	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
7	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**

Table 2. Symmetry types for the basis functions in an 8x8 transform.

The improved FMIDCT algorithm requires four $N^2/4$ accumulator arrays, Q_A, Q_B, Q_C, and Q_D, one for each symmetry type. Only the upper left quadrant of the basis vector is computed for any symmetry, thereby reducing the required computation by a factor of four. As each element of the input array is processed its contribution is accumulated into the appropriate array according to Table 2. Finally, the four accumulator arrays are merged to form the intensity values as follows:

$$\begin{aligned} Q_F &= Q_A + Q_B + Q_C + Q_D \\ H_F &= (Q_A - Q_B + Q_C - Q_D)\,R \\ V_F &= R\,(Q_A + Q_B - Q_C - Q_D) \\ D_F &= R\,(Q_A - Q_B - Q_C + Q_D)\,R \end{aligned} \tag{4}$$

Where Q_F, H_F, V_F, and D_F are the four quadrants in a final block of luminance or chroma values. Computing the quadrants in two butterfly stages as shown below can reduce the three operations per output element implied in equation 4:

$$\begin{aligned} B_{1a} &= Q_A + Q_B \quad B_{1b} = Q_A - Q_B \\ B_{2a} &= Q_C + Q_D \quad B_{2b} = Q_C - Q_D \\ Q_F &= B_{1a} + B_{2a} \\ H_F &= (B_{1b} + B_{2b})\,R \\ V_F &= R\,(B_{1a} - B_{2b}) \\ D_F &= R\,(B_{1b} - B_{2a})\,R \end{aligned} \tag{5}$$

Thus the total number of operations required for the improved FMIDCT is $(L/4 + 2)\,N^2$, where L is the number of non-zero coefficients, versus LN^2 for the original implementation. For all values of L greater than 2 the new formulation requires fewer operations. When L equals 2 no more that 2 of the quadrant accumulator arrays will contain nonzero values, thus equation 6 can be simplified such that no more than one operation is required to unfold the symmetries. When L equals 1 the merger requires no accumulation.

3.3 Lookup Tables and Caching Techniques

In the original FMIDCT implementation the dequantization process was assumed to have only two independent variables, the quantized symbol and the quantizer scale. Since the quantizer scale typically varies from element to element of the input vector, the original FMIDCT required N^2 separate tables. The dequantization process, however, is often more complex—containing as many as four independent variables. For instance, in MPEG-1 the dequantizer is defined by four parameters, a 31-level quantizer scale defined for each macro-block, a coefficient quantizer value for each element of the input vector, a binary variable based on whether the block is intra-frame or inter-frame encoded, and the quantized symbol. Implementing the table lookup function as described in the original FMIDCT would result in a factor of 62 increase in table size. In addition to the dramatic increase in storage requirements the hit-rate of the hashing-based lazy-evaluation technique is considerably reduced, thereby increasing the number of multiplies.

Another important property of the underlying basis vectors is there are only S unique sets. This reflects the fact that each of the basis functions with U unique coefficients shares the same U coefficients. So in the case of an 8x8 transform, all the basis functions with ten unique coefficients share in common the same coefficient values as all other basis vectors with ten coefficients. Therefore, it is conceivable that several different combinations of dequantization parameters might resolve into the same value. The improved version of the FMIDCT uses a two-level table lookup for dequantization and the scaling of basis functions. This permits the tables' contents to be shared between dequantizers.

The technique is described as follows. A hashing function, $H(\mathbf{p})$, is applied to the parameter vector **p** containing the four dequantizer parameters. The result is used to index the first level table, called the *dequantizer cache*. Each entry of the dequantizer cache is composed of a key parameter vector **k** and an index i_d into the second table, which stores the scaled basis vectors. If **k** matches **p** then it is used to index one of S tables, one for each of the unique basis vector sets and the scaled basis vector is constructed from the values stored there. If **k** does not match **p** then the appropriate dequantizer is invoked with parameter vector **p**, the dequantized value replaces i_d and **k** replaced by **p**. The dequantized value i_d is then used as an index into the second level table as before. The key values in the dequantization cache should be first initialized to an invalid state (we have used the fact that the quantizer symbols cannot have a value of zero). The process is shown with sample code in the Appendix.

4. IMPLEMENTATION & RESULTS

We have implemented an MPEG-1 video player in as a Java applet that utilizes our improved FMIDCT algorithm. This applet is capable of decoding and displaying 176 x 144 video sequences at rates of approximately 28 frames per second, and 352 x 240 sequences at 11 frames per second using Internet Explorer 4.0 on a Pentium computer with PII 266 MHz processor. Table 3 gives the results for sample sequences gathered from various Internet locations. While the player is capable of playing them directly from their respective sources, the results shown here are for sequences that have been downloaded to local storage before being played. Had they been played directly across the Internet, the frame rates would have been slower reflecting network bottlenecks rather than decoding time. The average performance of decoding and displaying an MPEG video sequence seems to be on the order of 670,000 pixels per second; with a three-fold increase in the performance when the pictures are decoded but not displayed, achieving 1.8 million pixels per second. This illustrates our previous suggestion that Java's display technology hinders the performance of the player, consuming two-thirds of the total time needed to decode and display the pictures. Future releases of Java promise to reduce this problem. In practice, Internet broadcasts take network bandwidth limitations into consideration, limiting the dimensions and frame rates to those manageable by our player. For instance, most Internet broadcast video has an image size of 176 x 144 and frame rates of 15 frames per second or less. It is worth noting that we have encountered other Java applets that play MPEG video via the Internet, to which our player compares favorably.[11,12]

Video Clip	Width	Height	Pixels	Decode rate	Pixels/sec	Display rate	Pixels/sec
30way	160	112	17,920	115	2,060,800	36.88	660,890
benylin	176	144	25,344	99.5	2,521,728	28.71	727,626
cart	240	180	43,200	28.7	1,239,840	16.25	702,000
daimler	368	272	100,096	14	1,401,344	6.05	605,581
glacier	200	100	20,000	90.4	1,808,000	38.2	764,000
hill	240	180	43,200	40.2	1,736,640	16.75	723,600
ligther	176	144	25,344	73.1	1,852,646	25.46	645,258
llupanav	160	128	20,480	79.7	1,632,256	24.86	509,133
mjackson	160	120	19,200	110.3	2,117,760	32.88	631,296
ts	352	240	84,480	15.7	1,326,336	7.7	650,496
wg_wt_1	304	224	68,096	29.8	2,029,261	10.9	742,246
Average					1,793,328		669,284

Table 3. Frame rates achieved with various sample video sequences.

Our modular source-player design enabled us to implement a picture-in-a-picture player. Figure 3 shows a screen capture of an applet playing two video streams simultaneously. The user can resize and move each window dynamically via mouse interaction. This applet and other demonstrations are available at our Internet site.[8]

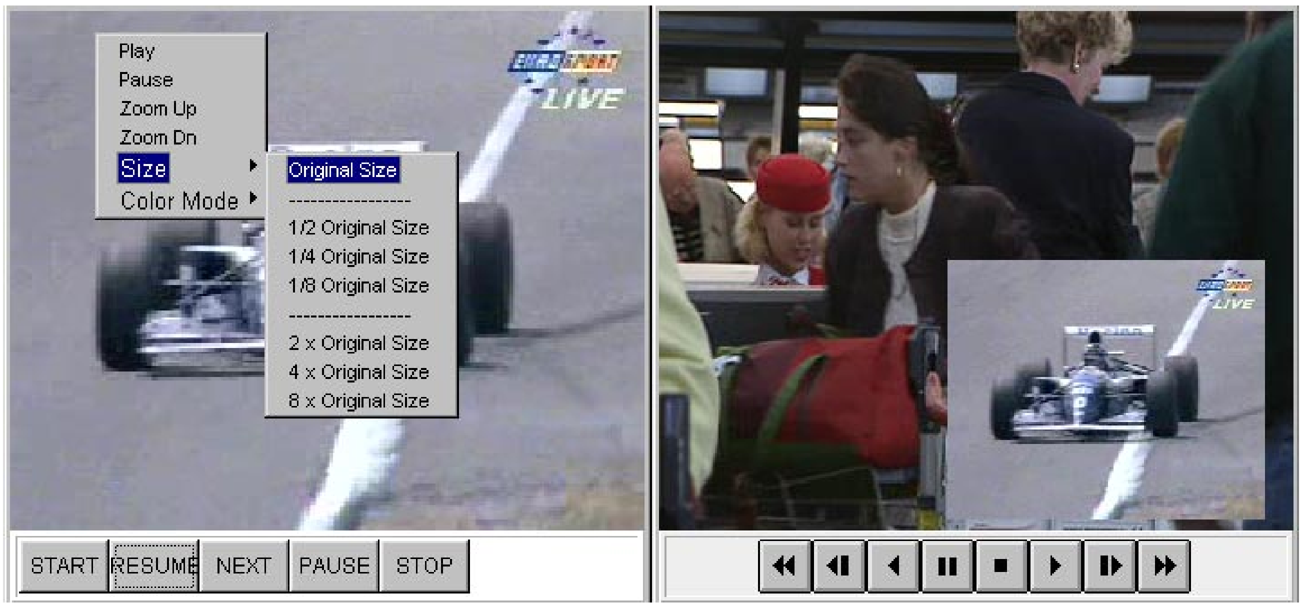

Figure 3. Single player with textual video controls (left) and a picture-in-a-picture example with visual controls (right).

5. CONCLUSIONS

We have presented an efficient streaming MPEG-1 video player implemented entirely in Java, which eliminates the need to pre-install native software and is well suited to small devices. The player owes its efficiency to an improved forward-mapping IDCT algorithm described here as well. Our modular design provides viewers with their own customized session, which we demonstrated in the picture-in-a-picture example.

Further tests and enhancements to this player include the implementation of the trade-off between quality and speed feature, described by McMillan and Westover. For example, the IDCT algorithm can run with greater speed when the video is viewed at reduced size, performing approximately ¼ the amount of computation when the user resizes the video to ½ its original width and height. It is also possible to use this feature to improve the frame rates of video with large pictures by playing them at reduced size or reduced quality. However, this feature was not envisioned with MPEG video in mind and some difficulties exist in using it here. A crude application of this feature to intra-coded frames would result in compounded degradation of predicted frames, introducing new artifacts into the decoded stream. Another feature that we would like to investigate is rate control. Currently, we are playing the sequences at the fastest achievable rates, which are generally below the sequences' original frame rates. It is possible to drop frames by avoiding decoding them, but the structure of MPEG video makes this feature problematic because of inter-frame dependencies. For example, if we decided to decode all intra-coded frames in anticipation that other frames will depend on them, we may encounter sequences made up entirely of I-frames, for which this technique will fail to speed the play back. Alternatively, we could defer decoding P or I-frames that we decide to skip until we encounter other frames that depend on them. Such method is complicated, demands more memory, and does not guarantee speed either. Finally, we are currently using this framework to design and implement other players in the context of the Computational Video project, such as a players for motion-JPEG video and other special purpose video.

6. ACKNOWLEDGEMENTS

This work is supported by DARPA agreement number F30602-97-1-0283. The video sequences used in our demonstrations and statistics were downloaded from various Internet sources reached via *MPEG.ORG*'s Web site.[10]

7. REFERENCES

1. Kamanagar, F. A. and K. R. Rao, "Fast Algorithms for the 2-D Discrete Cosine Transform", *IEEE Trans. On Computers*, vol. C-31, no. 9, pp. 899-906, Sep 1982.
2. LeGall, D. "MPEG: A Video Compression Standard for Multimedia Applications", *Com. of the ACM*, vol. 34, no. 4, pp. 46-58, April 1991.
3. Fielding R. et al, "*Hypertext Transfer Protocol - HTTP/1.1*", Internet RFC 2068, January 1997.
4. McMillan, Leonard and Lee Westover, "A Forward-Mapping Realization of the Inverse Discrete Cosine Transform", *Proceedings of the Data Compression Conference (DCC '92)*, IEEE Computer Society Press, March 24-27, pp. 219-228, 1992.
5. Rao, K.R. and P. Yip. *Discrete Cosine Transform: algorithms, advantages, applications*. Academic Press, Boston, 1990.
6. Vetterli, M. and H. Nussbaumer, "Simple FFT and DCT Algorithms with Reduced Number of Operations", *Signal Processing*, vol. 6, pp. 267-278, Aug 1984.
7. http://java.sun.com
8. http://compvid.lcs.mit.edu/cv/
9. http://bmrc.berkeley.edu/projects/mpeg/
10. http://www.mpeg.org
11. http://www.dcc.uchile.cl/~chasan/MPEGPlayer.zip
12. http://rnvs.informatik.tu-chemnitz.de/~ja/MPEG/JITVERS/MPEG_Play.html

APPENDIX

The following Java code fragment demonstrates the first stage of table lookup associated with the dequantizer cache:

```
// Dequantizer cache
private int cacheKey[] = new int[4096];
private int cacheEntry[] = new int[4096];

// Each entry in icoeff packs two values: position in the 8x8 block and quantized DCT coefficient
private void dequantize( int qscale, int []icoeff, byte []qmatrix, int type) {
    // Store output in place. We will use the results in the IDCT method
    // (not shown here) to look up the scaled basis vectors
    int []ocoeff = icoeff;

    int i, j, k, index, key, level, qval;

    // DC of intra-coded block
    j = k = 0;
    if (type == 0) {
        level = icoeff[j++];
        level >>= 8;
        ocoeff[k++] = (level + 2048) * 32 << 8;
    }

    // Process all input coefficients
    while ((level = icoeff[j++]) != 0) {
        i = level & 63;                 // position in 8x8 block
        level >>= 8;                    // quantized DCT coefficient
        qval = qmatrix[i];              // quantization value for this position

        // Pack the four-parameter vector p into a single integer
        key = (level << 16) | (qval << 8) | (type << 5) | qscale;
        // Determine hashed index H(p)
        index = ( level + ((qval - 16) << 4) + ((qscale - 8) << 8) + (type << 11)) & 0xfff;

        if (cacheKey[index] == key) {   // k stored in cache matches p
            level = cacheEntry[index];  // retreive dequantized coefficient from cache
        } else {
            // Invoke appropriate dequantizer
            if (type == 0) level = (level * qscale * qval) >> 3;
            else level = ((2 * level + ((level >> 31) | 1)) * qscale * qval) >> 4;
            if (level == 0) continue;
            if ((level & 1) == 0) level -= (level >> 31) | 1;

            // Clamp dequantized coefficient to interval [-2048, 2047]
            if (level > 2047) level = 2047;
            else if (level < -2048) level = -2048;

            // Convert dequantized coefficient into index of scaled basis vector
            // in the second table (we add 2048 to avoid negative indices)
            level = (level + 2048) * 32;

            // Store entry in dequantizer cache
            cacheKey[index] = key;      // replace k by p
            cacheEntry[index] = level;  // replace i_d with dequantized coefficient
        }
        ocoeff[k++] = (level << 8) | i; // pack level and index into output coeff
    } // No more input coefficients
    ocoeff[k] = level;
}
```