Compressed Domain Transcoding of MPEG

Soam Acharya
Department of Electrical Engineering
Ithaca, NY 14853
soam@ee.cornell.edu

Brian Smith
Department of Computer Science
Ithaca, NY 14853
bsmith@cs.cornell.edu

Abstract

Current video compression formats optimize for either compression or editing. For example, motion-JPEG (MJPEG) provides excellent random access and moderate overall compression, while MPEG optimizes for compression at the expense of random access. Converting from one format to another, a process called *transcoding*, is often desirable over the life of a video segment. This paper shows how to transcode MPEG-1 video to motion-JPEG without fully decompressing the MPEG-1 source. The described technique for compressed domain transcoding differs from previous work because it uses a new approximation approach that is optimized for software implementations. This new approach is 1.5 to 3 times faster than spatial domain transcoders and offers an additional degree of freedom: higher transcoding speeds can be obtained at the price of lower picture quality. This speed/quality trade-off is useful in many real-time applications such as off-line editing and video gateways.

1. Introduction

Motion-JPEG (MJPEG) [6], H.261, and MPEG are common video compression schemes. None of these schemes achieve both simple access to individual video frames (i.e., random access) and good compression. Greater compression comes at the expense of complex random access, as shown in Table 1. Hence, a format well suited to one class of video applications (e.g., editing) may not be desirable for a different class (e.g., video on demand). Common video applications and their preferred formats include:

- *Video editing* - random access is important and moderate compression is acceptable. Hence MJPEG is preferred.
- *Video on demand* - compression is crucial to save storage and bandwidth, hence MPEG is often chosen.
- *Video processing* - Video is usually decompressed before it can be processed. However, *compressed domain processing* techniques can operate on video without fully decompressing it. Many of these techniques operate on MJPEG data [5,6,7,8,9].

Thus, no single video format is ideal for all purposes. However, during the course of its lifetime, a typical video segment undergoes editing, processing and storage, often multiple times. This makes the ability to convert between formats highly desirable, a process we call *transcoding*.

Transcoding is useful in other applications. A *video gateway* that connects a fast network to a slower network might transcode video coming in from the fast network to a low bandwidth format better suited for the slow link. For example, Amir, McCanne, and Zhang describe a video gateway that transcodes live MJPEG streams to H.261 streams [4]. Such video gateways require high-speed transcoders.

In this paper, we develop a compressed domain transcoder (*CDTC*) for converting MPEG-1 to MJPEG that is 1.5 to 3 times faster than its spatial domain counterpart. Additional speedup is pos-

	random access	compression
MJPEG	simple	moderate
H.261	difficult	good
MPEG-1	complex	excellent

Table 1: Comparison of Common Compression Schemes

sible at the expense of image quality. We choose to transcode MPEG-1 to MJPEG because of the widespread use of MPEG in storage and MJPEG in non-linear desktop video editing. The recent development of compressed domain processing techniques on MJPEG data provide further incentive. Our techniques are optimized for MPEG-1 but can be adapted for MPEG-2 and H.261.

This paper assumes the reader is familiar with MPEG-1 and JPEG compression [1,2,3]. We introduce basic terminology and concepts by briefly reviewing these algorithms in section 2. To identify the problems associated with transcoding, we describe spatial domain transcoding in section 3. We detail our CDTC in section 4 and also show how we optimize processing while maintaining good picture quality. In sections 5 and 6 we compare the performance of the spatial and compressed domain methods. We outline related work in section 7, and present our conclusions in section 8.

2. A Brief Review of JPEG and MPEG-1

To understand our compressed domain transcoder, a working knowledge of JPEG[1] and MPEG is necessary. This section briefly reviews JPEG and MPEG. Our purpose in this discussion is to remind the reader of the steps in the algorithms and to name these steps. A more detailed discussion of these standards are available elsewhere [1,2,3]. To simplify the discussion, we only consider gray-scale video.

2.1 JPEG

The sequential (baseline) JPEG algorithm, shown in figure 1, consists of the following steps:

1. Decomposition: The original image is divided into 8x8 *blocks*. These blocks are processed in row-major order by the following steps.
2. DCT: Each block is transformed using the Discrete Cosine Transform (DCT). The transformed block has the signal energy concentrated into a few lower order coefficients. The (0,0) element of the transformed block is called the *DC component*, and the other 63 elements are *AC components*.
3. Scaling: Each element in the transformed block is divided by the corresponding elements in an 8x8 *quantization table* (QT).
4. Rounding: The scaled coefficients are rounded to the nearest integer. Steps 3 and 4 together are called *quantization*.
5. Zig-zag mapping: Each quantized 8 x 8 block is converted to a 64 element *vector* using a fixed one to one mapping.
6. Run Length Encoding: Strings of zeros in the AC elements of the quantized vector are run

1. MJPEG applies the JPEG algorithm to each frame in a video sequence.

length encoded. The block, at this stage, is called a *semi-compressed (SC)* block. An SC-block consists of a DC element plus several *(run length, AC value)* pairs.

7. DPCM: The DC coefficient of each SC block is encoded as *difference* from the DC coefficient of the previous block in the sequence.
8. Huffman Coding: The SC block is converted to a bitstream via Huffman encoding.

An important property, for our purposes, is that some of these steps can be transposed. For example, we can swap steps 4 (rounding) and 5 (zig-zag mapping). With this exchange, the first three steps, DCT, scaling and zig-zag mapping, are all linear. Thus, they can be combined into one linear operation, a fact we utilize in section 4. Also, note that SC-blocks are simply sparse representations of the scaled DCT block. Our compressed domain algorithms will operate on SC-blocks, and we represent a JPEG image as an array of SC blocks.

Decompression is the reverse of compression. The bitstream is *entropy decoded* and the DC value of the previous block is added to the recovered difference value to produce an SC block. The SC block is de-*zigzagged* into an 8 x 8 quantized block. This block is multiplied by the QT (the *multiplication* step) and the inverse DCT transform (IDCT) is applied.

2.2 MPEG

MPEG is designed to compress video -- that is, sequences of images (also called *frames*). In MPEG-1 (hereafter called MPEG), each frame is divided into 16 x 16 pixel *macroblocks*. Each macroblock contains six 8 x 8 pixel *blocks*, four from the luminance channel and one from each chrominance channel. As with JPEG, we ignore the chrominance channel for clarity. Each macroblock can be compressed in several ways. Depending on the method used, the macroblock is called an I, P, B, or Bi macroblock[1]. MPEG also defines three types of frames, I, P and B, which we discuss in turn and summarize in table 2.

I frames contain only I macroblocks. The four blocks in an I macroblock are compressed using an algorithm nearly identical[2] to JPEG. In other words, the blocks are converted to SC-blocks and then compressed using Huffman coding.

P frames exploit the temporal redundancy of video whereby adjacent frames are likely to be similar. A P frame relies on a *reference frame* prior in the sequence called a *past* reference frame. The past reference frame is the most recent I or P frame in the sequence. A P frame consists of either I or P macroblocks. A P macroblock is a *motion vector* and a *residual* macroblock. During decompression, the motion vector is used to extract a *predicted* macroblock from the reference frame, and the residual macroblock is added to the predicted macroblock to produce the macroblock in the output image. The residual macroblock is compressed like the I macroblock, and the motion vectors are also encoded in the bitstream using Huffman coding.

A B frame is similar to a P frame, but uses two reference frames, one from later in the sequence (a *future* reference frame) and one from earlier on in the sequence (a *past* reference frame). As with P frames, these reference frames are the nearest I or P frames in the sequence (see figure 2). A B frame can contain I, P, B, or Bi macroblocks. B macroblocks are identical to P,

1. Two other types of macroblocks, called D and skipped, are also defined by MPEG. We ignore them in this paper for simplicity.
2. The bitstream syntax, entropy encoding technique, and quantization tables are different, as is the encoding order of the image blocks.

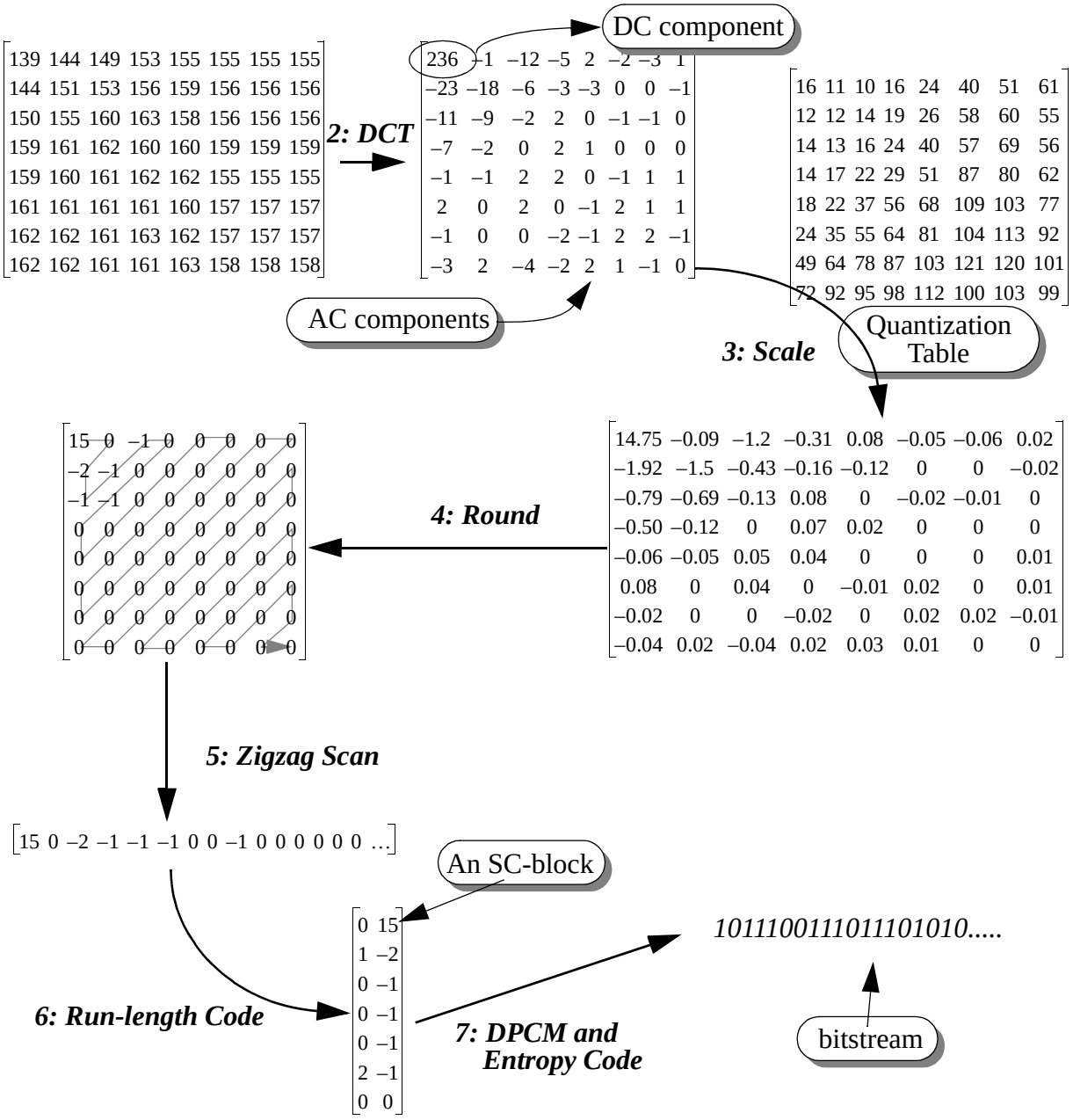

Figure 1: Block Encoding

except they use the future reference frame rather than the past reference frame. Since B macroblocks are so similar to P macroblocks, the techniques we use to deal with the two are identical. Hence, we refrain from discussing B macroblocks for the remainder of the paper.

A Bi macroblock consists of two motion vectors and a residual macroblock. These motion vectors are used to extract the two predicted macroblocks (one from each reference frame),

which are averaged and added to the residual macroblock to produce the result. The residual macroblock is compressed to an SC-block and Huffman coded along with the two motion vectors.

Table 2 summarizes the composition of MPEG frames and macroblocks. If all the blocks in a

Frame Type	Frame Composition		Macroblock Type	Macroblock composition
I frame	I type macroblock		I	4 SC blocks
P frame	I, P		P	4 SC + motion vector (*mv*)
B frame	I, P, B, Bi		B	4 SC + mv
			Bi	4 SC + 2 mv

Table 2: Summary of MPEG frame/macroblock characteristics

macroblock are semi-compressed, we call the macroblock a *SC macroblock*. As with JPEG, a frame can be represented as a two dimensional array of macroblocks. If these macroblocks are semi-compressed, the frame is called a *SC frame*. The use of reference frames in I, P, and B frames leads to the interframe dependencies shown in figure 2. It is these dependencies that complicate random access.

3. Spatial Domain Transcoding

Having sketched MPEG and JPEG, we now turn to the problem of converting MPEG to Motion-JPEG (MJPEG). The most straightforward approach is to decompress the source bitstream completely and compress the result using JPEG. We call this technique method I. It provides a benchmark against which other techniques can be compared.

Method I can be optimized in several ways. The first optimization is to note that I macroblocks can be converted to JPEG blocks by simply requantizing the SC-blocks. Or, better yet, we can redefine the quantization tables in the JPEG bitstream to be the same as used by MPEG and just entropy encode the SC-blocks in the I macroblock directly (if we slightly reorder the blocks). An important detail to remember is that MPEG uses variable quantization and standard JPEG does not, so the coefficients in the SC block may need to be rescaled, but the cost of this rescaling

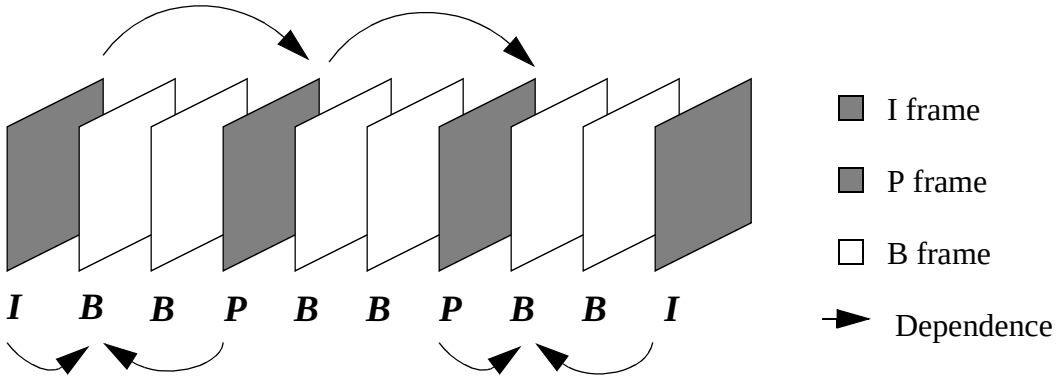

Figure 2: Inter-frame dependencies in MPEG

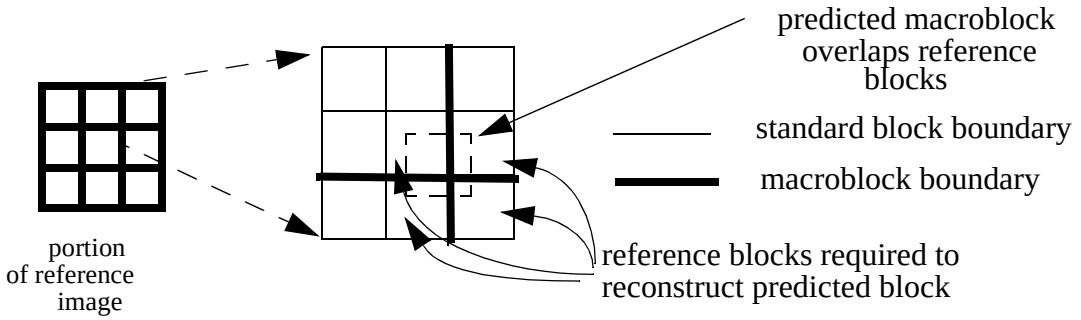

Figure 3: Source Block Extraction Problem

is minimal.

Transcoding P and Bi macroblocks to JPEG blocks is more complex because a predicted macroblock(s) must be extracted from the reference frame(s). The problem is illustrated in Figure 3. As we can see from the figure, up to four reference blocks may be needed to reconstruct a predicted block. These reference blocks, which may be contained in different macroblocks, must be decompressed to construct the predicted block. The residual block must also be decompressed before it is added to the predicted block, and the result compressed.

Extracting the predicted blocks is expensive, since it requires decompressing the relevant macroblocks in the reference frame, but a few optimizations can be made. Experiments reveal that some macroblocks in a reference frame are decompressed multiple times, since they can be referenced in multiple B frames. We therefore decompress the macroblocks in the reference frames as needed and cache the decompressed macroblocks in case we need them later.

Another optimization is to add the predicted and residual macroblocks in the compressed domain [9]. In normal processing, the residual macroblock is decompressed, added to the predicted macroblock, and the result is compressed. But, if we first convert the predicted macroblock to a SC macroblock, we can add the residual SC macroblock directly, avoiding the decompression step.

The modified transcoding method, with these three optimizations (I-frame conversion, caching, and adding the residual in the compressed domain), is called Method II. Method II represents a reasonable effort to improve the performance of a spatial domain transcoder with relatively simple compressed domain techniques. Experiments show that the DCTs and IDCTs required by P/B macroblock extraction are the bottleneck in transcoding speed for Method II. Our next approach, therefore, was to perform predicted block extraction in the compressed domain. The next section describes this approach in detail.

4. Method III - Compressed Domain Transcoding

The bottleneck in Method II is extracting predicted macroblocks from reference images. This section shows how to perform this operation in the compressed domain. We first show how to decompose source block extraction into steps that can be implemented on the SC-blocks.

As evident in figure 3, we can extract a predicted macroblock if we can align the relevant macroblocks in the reference frame to a macroblock boundary. Another way to think of this macroblock alignment is that it involves *translating* the individual blocks and combining them, as shown in figure 4. In this figure, the block f_{00} is translated by (dx, dy). We use the notation

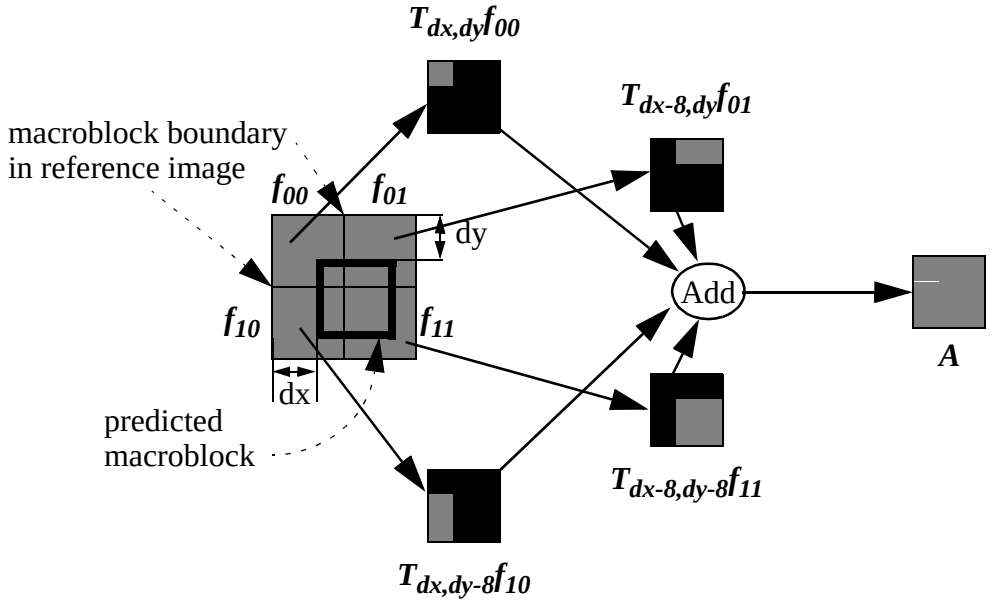

Figure 4: Block alignment using translation

$T_{dx,dy}f_{00}$ to represent this translated block. The black region in $T_{dx,dy}f_{00}$ are pixels not contained in f_{00}, so we set them to zero. If the other three blocks are similarly translated, and the results added, one block in the predicted macroblock is extracted.

We now show how to perform these operations (translation and addition) directly on the SC-blocks. The SC blocks can be added directly, we can perform this entire operation on SC blocks if we can perform translation on SC blocks.

In previous papers [5,6], we showed how to perform a wide variety of operations, including translation, directly on SC blocks. Briefly, let F_{00} be the SC-block corresponding to f_{00}. Formally, $T_{dx,dy}$ is then a matrix, F_{00} a sparse vector, and translation of F_{00} is accomplished by matrix multiplication. The result of this multiplication is a 64 element vector that represents the translated block after the DCT, scaling, and zig-zag scan steps (steps 2, 3, and 5) of the JPEG algorithm have been performed. This vector is converted to an SC block using steps 4, 6, and 7 (rounding, DPCM, and run-length encoding) of the JPEG algorithm.

Figure 5 shows a slightly different decomposition, where we align macroblocks by translating on each axis separately. The intermediate blocks $\mathbf{G_0}$ and $\mathbf{G_1}$ and the predicted block $\mathbf{A}$ are given by the equations

$$\mathbf{G_0} = \mathbf{T_{dx}F_{00}} + \mathbf{T_{dx-8}F_{01}}$$
$$\mathbf{G_1} = \mathbf{T_{dx}F_{10}} + \mathbf{T_{dx-8}F_{11}}$$
$$\mathbf{A} = \mathbf{T_{dy}G_0} + \mathbf{T_{dy-8}G_1}$$

Analysis shows that computing this decomposition is more efficient than computing the combined operation (figure 4) because the matrices associated with the translation along a single axis, $\mathbf{T_{dx}}$ and $\mathbf{T_{dy}}$, are quite sparse. We therefore focus on this case. In the discussion that follows, we describe the case of translating a block $\mathbf{F}$ by a positive amount dx along the x axis. Translating along the y axis and translating by negative values are similar.

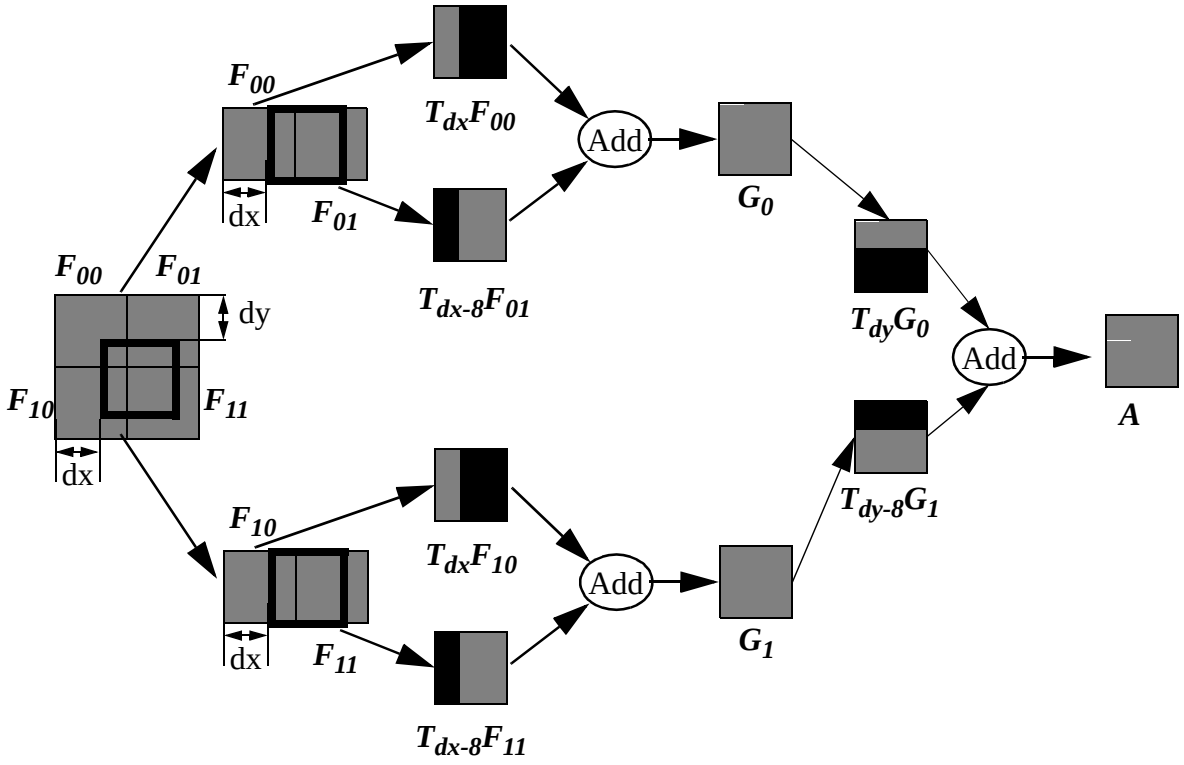

Figure 5: Translating on each axis separately

Our goal is to compute the elements of a vector $\mathbf{G} = \mathbf{T_{dx}F}$:

$$G[i] = \sum_{j=0}^{63} T_{dx}[i,j]F[j] \qquad \text{EQ 1}$$

Following the development in [6], the elements of the matrix $T_{dx}[i,j]$ are given by:

$$T_{dx}[i,j] = \sum_{x}\sum_{v} J(i,x,(y+dx))\bar{J}(x,y,j) \qquad \text{EQ 2}$$

where J is a 3 dimensional matrix that converts an SC block from the source block and $\bar{J}$ performs the reverse computation. Note that scaling, zig-zag encoding, DCT and IDCT are folded into J and $\bar{J}$ and hence into $\mathbf{T_{dx}}$. We present code to compute $\mathbf{T_{dx}[i,j]}$ in the appendix. Since the parameter dx can only assume sixteen discrete values (dx = 0..7 plus half pixels), our task is to efficiently compute equation 1 for these sixteen values. We describe an efficient method in the next section.

4.1 Performance Optimization I - Dynamic Thresholding

The vector **G** in equation 1 is the product of a matrix $\mathbf{T_{dx}}$ and the vector corresponding to the SC block **F**. In the calculation of **G**, we can tolerate some error in the result. We can afford some error because the elements of **G** will be rounded off to the nearest integer (quantized) after the computation, as part of converting **G** to an SC block. Thus, introducing errors of, say, +/- 0.5

should not unduly affect the quality of the output. Suppose we fix an allowable error *maxerr* for the computation of each element of **G**, and that **F** has *n* non-zero elements. Computing an element *G[i]* requires *n* multiplies (and *n-1* adds) of the form T_{dx}*[i,j]*F[j]*. We call a product term *insignificant* if:

$$\left|T_{dx}[i,j]F[j]\right| < \frac{maxerr}{n} \qquad \text{EQ 3}$$

or

$$\left|T_{dx}[i,j]\right| < \left|\frac{maxerr}{nF[j]}\right| \qquad \text{EQ 4}$$

Intuitively, insignificant terms are terms in equation 1 that we can throw away without introducing too much error (i.e., more than *maxerr*). The central idea of our optimization is that we compute the product T_{dx}*[i,j]*F[j]* only if equation 4 holds. Conceptually, we must check this condition before every multiply. However, since T_{dx} is a constant, it turns out there is an elegant way of doing this efficiently.

Consider the calculation of **G** in equation 1. Using the fact that **F** is typically sparse, we can calculate **G** using the following code fragment:

```
zero G[]
for each non-zero F[i] do
        for all j do
                G[j] += T[i,j]*F[i];
        endfor
endfor
```

If we unroll the inner loop using a switch statement, our code fragment looks like this:

```
zero G[]
for each non-zero F[i] do
        v = F[i];
        case i in
         0:
                G[0] += T[0,0]*v;
                G[1] += T[0,1]*v;
                G[2] += T[0,2]*v;
                ....
         1:
                ....
```

For a fixed translation *dx*, **T** is constant, so we can substitute T[0,0], T[0,1] by precomputed values. For example, suppose T[0,0]=0.3214, T[0,1]=-0.0027, T[0,2]=-0.271, T[0,3]=-0.027, and the remaining T[0,j] are zero. Our code fragment becomes:

```
zero G[] vector
for each non-zero F[i] do
        v = F[i];
        case i in
         0:
                G[0] += 0.3214*v;
                G[1] += -0.0027*v;
                G[2] += -0.271*v;
                G[3] += 0.1045*v;
                ....
```

Finally, we sort each case statement by the absolute value of the constant, smallest value at the top:

```
zero G[] vector
for each non-zero F[i] do
        v = F[i];
        case i in
         0:
                G[1] += -0.0027*v;
                G[3] += 0.1045*v;
                G[2] += -0.271*v;
                G[0] += 0.3214*v;
                ....
```

For a given value of *F[i]*, only some of the terms *T[i,j]*F[i]* are significant. For example, if *F[i]=1*, *maxerr=0.5*, and *n=3*, then by equation 4, any term with a coefficient *T[i,j]* less than *0.5/3*1 = 0.167* is insignificant (i.e., the first two terms in case 0 of the example above). Since the values are sorted, we know that all the preceding terms are insignificant as well. Thus, if we can jump to the first significant coefficient in the case statement, we will only evaluate the product terms required. We call this technique *dynamic thresholding*.

We can implement dynamic thresholding in this case by computing an integral threshold, *intThresh*, which is a fixed point representation of the coefficient threshold (equation 4):

$$intThresh = \left\| \left\lceil \frac{maxerr}{nF[i]} \times 16 \right\rceil \right\| \qquad \text{EQ 5}$$

We use *intThresh* as an index into a jump table that skips insignificant coefficients *T[i,j]*. Implementing this idea results in the following code:

```
zero G[]
n = number of non-zero elements in F[]
for each non-zero F[i] do
        v = F[i];
        intThresh = abs(ceiling(16*maxerr/(F[i]*n)));
        case i in
         0:
                case intThresh in
                 0, 1:
                        G[1] += -0.0027*v;
                 2:
                        G[3] += 0.1045*v;
                 3:
                        G[2] += -0.271*v;
                 default:
                        out[0] += 0.3214*v;
                ...
```

Note that this code can execute very efficiently. Once the initial jump is performed, the minimal number of product terms are computed as a sequence of multiply/add instructions. Experiments show that 20%-30% of the multiplies are avoided for typical values of *maxerr.* Since there is no interdependency of terms, these instructions can be pipelined resulting in a throughput of one cycle per term.

Since writing code of this nature is tedious, we developed a code-generator to write the required functions. Our generator produces a procedure for each offset *dx* and *dy,* resulting in 32 procedures, 16 each for horizontal and vertical translations. Half pixel values were ignored, and each procedure uses fixed point arithmetic with a 22 bit fractional part. In addition, for product term *T[i,j]*F[i]* in the generated code, we approximated the fixed point coefficient *T[i,j]* by zeroing the lower eight bits, which allowed the compiler to replace the multiply with several shifts and adds.

However, our initial experiments showed that dynamic thresholding, by itself, did not provide either sufficient speedup or satisfactory image quality. We needed to use caching (section 4.3) and temporary requantization (section 4.4) to improve speed and quality further.

4.2 Performance Optimization II - Caching

Caching is a technique that reduces the number of translation computations required. Consider the process of computing two adjacent blocks $\mathbf{B_0}$ and $\mathbf{B_1}$ from reference blocks $\mathbf{F_0}$...$\mathbf{F_5}$, as shown in figure 6. $\mathbf{B_0}$ and $\mathbf{B_1}$ can be derived as follows:

$$\mathbf{G_0} = \mathbf{T_{dx}F_0} + \mathbf{T_{dx\text{-}8}F_1}$$
$$\mathbf{G_1} = \mathbf{T_{dx}F_2} + \mathbf{T_{dx\text{-}8}F_3}$$
$$\mathbf{G_2} = \mathbf{T_{dx}F_4} + \mathbf{T_{dx\text{-}8}F_5}$$
$$\mathbf{B_0} = \mathbf{T_{dy}G_0} + \mathbf{T_{dy\text{-}8}G_1}$$
$$\mathbf{B_1} = \mathbf{T_{dy}G_1} + \mathbf{T_{dy\text{-}8}G_2} \qquad \text{EQ 6}$$

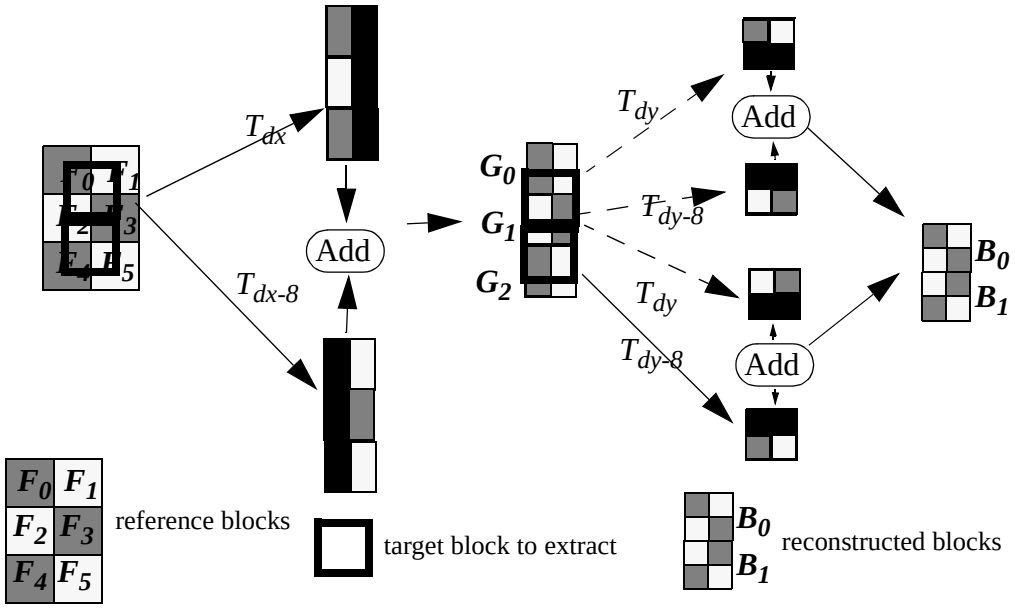

Figure 6: Detailed Predicted Vector Extraction Example

Since the intermediate block $\mathbf{G_1}$ is required in the computation of both $\mathbf{B_0}$ and $\mathbf{B_1}$, computing $\mathbf{G_1}$ once for both $\mathbf{B_0}$ and $\mathbf{B_1}$ is advantageous. By doing so, we save a total of 4 out of 24 translation calculations per macroblock.

We generalized this idea further by caching the result of *every* translation of a reference block, since P or B blocks in other frames (or in the same frame but adjacent macroblocks) might require the reference block to be translated by exactly the same offset. We used a directly associative cache model: each reference block had dedicated cache entries for every potential translation in the x and y direction. This required a space allocation of 16 additional blocks per reference block. However, actual memory requirements were more reasonable because of the following factors:

- typically, less than half the frames in an MPEG sequence are reference frames.
- the mpeg decoder only decodes a small subset of all available frames at a time. Therefore, it allocates sufficient space in its data structures only for that small subset. Hence, allocating extra space for caching would not significantly augment the transcoder primary memory requirements.

4.3 Quality Optimization - Temporary Requantization

Dynamic thresholding is only advantageous when SC blocks are sparse (i.e., *n* in equation 4 is small). To achieve low coefficient density, we initially quantized the input reference SC blocks (e.g. $\mathbf{F_0..F_5}$ in equation 6) using the standard MPEG quantization table, *prior* to performing the translation operations. Unfortunately, we found that this table scaled the coefficients too coarsely resulting in loss of detail in the final image. Hence, we used an intermediate table with smaller values[1] to quantize the input SC blocks. This resulted in sparser intermediate translated blocks

1. We used the MPEG quantization table scaled by 2

$\mathbf{T_{dx}F}$ and $\mathbf{T_{dx}G}$ and lower transcoding speed. The performance drop, however, was offset by a significant improvement in final image quality.

5. Experiments

To compare their performance, we implemented all 3 methods (spatial domain methods I and II and compressed domain method III). We used the Berkeley MPEG decoder and Independent JPEG group encoder to build the transcoders. Method I was straightforward to implement, methods II and III were more involved. We modified the decoder to store sparse 8x8 blocks as *RLE vectors*. Each RLE vector has an array of *(index, value)* pairs and a field indicating the size of the array. The translation transforms for the frequency domain approach converted RLE vectors to RLE vectors via procedures produced using a code-generator. Our transcoder is publicly available [11].

All experiments were run on an HP 735. We measured application performance by the number of frames transcoded per second. All data was read from, and written to, memory, so I/O was not a factor.

We measured the quality of the output using the PSNR metric:

$$PSNR = 20\log\left[\frac{255}{\sqrt{\frac{rms(C-O)}{imsize}}}\right]$$

where *imsize* is the image size, *rms* is the root mean squared over the image, *C* is the image produced by the transcoder and *O* is the corresponding image produced by decoding the MPEG stream to a sequence of gray-scale images.

We tested all the methods on a selection of MPEG streams whose properties are summarized in table 3.

Clip Name	Frame Size	Avg I (bytes)	Avg. P size	Avg. B size	Frame Pattern	Viewing fps
alesi.mpg	240x192	5043	4418	2243	IBBPBBPBBP BB	3
bike.mpg	352x240	11756	7106	1606	IBBPBB	5
bus.mpg	352x240	13512	7657	2723	IBBPBBPBBP BBPBB	4
cannon.mpg	192x144	7792	5918	1381	IBBPBB	5
us.mpg	352x240	6479	4660	1465	IBBPBB	5
raiders.mpg	160x128	1953			I	9

Table 3: Properties of the test MPEG sequences

6. Results

Table 4 shows the transcoding performance and image quality (PSNR) for methods I, II, and

III on a typical stream ("bike.mpg"). As we can see from the table, the PSNR for I-frames remains constant, regardless of method used. This is expected since the processing of I frames is almost identical in all methods. Method 1 gives the best picture quality but is also the slowest. Method II provides almost the same quality, but the speedup from method II is usually not significant. Method III provides significant performance improvement at the cost of slightly degraded picture quality, depending on the setting of *maxerr.* Performance leveled out for values of *maxerr* larger than about 15. Although method III avoids performing many operations in the translation procedures for large *maxerr*, the overhead of entropy encoding and decoding limit the speedup obtained. Profiling reveals about 50% of the code is spent in the translation procedures, with the remaining time spent in other procedures. This leads us to conclude that the maximum speedup possible through compressed domain transcoding is about 3.6, and is limited by Huffman coding.

Method	frames/sec	PSNR for I	PSNR for P	PSNR for B	Speedup
Method I	5.7	36.3	37.5	37.8	1.0
Method II	5.9	36.3	37.4	37.7	1.0
Method III *maxerr*=0	6.5	36.3	35.3	35.7	1.1
maxerr=5	8.1	36.3	35.0	35.3	1.4
maxerr=10	9.6	36.3	34.5	34.7	1.7
maxerr=15	10.5	36.3	33.5	33.6	1.8

Table 4: Speed and Quality of Various Transcoders On "bike.mpg" Sequence

It is interesting to note that the PSNR of the B frames is higher than that of the P frames. This property is due to the averaging techniques used in B block reconstruction, which act as a filter that reduces errors.

The image quality of MPEG sequences with a long frame pattern of P and B frames between successive I frames (such as "alesi.mpg") tend to degrade on the order of 0.1-1dB between successive P and B frames *within* the group of pictures. This is due to the error generation in the transcoding of a P frame which then propagates to subsequent P and B frames. Long frame sequences also impair transcoding performance because of the presence of proportionately more non-reference frames and, thus, the need for more translation operations necessary to restore these frames.

Figure 7 shows a sample B frame from "bus.mpg". Figures 8 and 9 present the same still but generated via method III with *maxerr* 0 and 10 respectively. All three images are available on-line [10].

Table 5 compares the performance of all three techniques on a variety of streams. These measurements show that the speedup from method II is usually small. The compressed domain approach typically improves performance by 1.5 to 3 times. Notable exceptions are "bus.mpg" and "raiders.mpg". The former is encoded via intra- and non intra-block quantization tables that are not the default values recommended by the official MPEG-1 standard. Since the translation procedures are optimized for the standard tables, both transcoding performance and image quality

suffer. However, this is not a serious concern since the majority of MPEGs are encoded using the standard table set. The long frame pattern length of “bus.mpg” is also another reason for its poor transcoding performance. “Raiders.mpg” consists of nothing but I-frames, which explains why the frame rate of “raiders.mpg” remains the same for all values of *maxerr* in method III. The dynamic thresholding routines only work for P and B motion compensated frames.

Table 5 also shows that despite being similar in picture size, there is a noticeable performance

Figure 7: B frame decoded directly to gray-scale

Figure 8: Same frame decoded with *maxerr*=0

MPEG Clip	Method I	Method II	Method III maxerr=0	Method III maxerr=1	Method III maxerr=5	Method III maxerr=10
alesi.mpg	9.7	11.2	9.7	10.0	11.8	14.3
bike.mpg	5.7	5.9	6.5	6.6	8.1	9.6
bus.mpg	5.3	5.1	3.8	3.8	4.4	5.4
cannon.mpg	15.7	16.1	28.0	28.6	31.4	34.4
us.mpg	6.0	6.3	13.6	14.1	15.9	17.1
raiders.mpg	20.5	90.0	143.8	143.8	143.8	143.8

Table 5: Speed of Various Transcoders (frames transcoded/second)

difference between “bike.mpg” and “us.mpg”. This is explained by the frame sizes of the MPEG bitstreams. On average, each compressed I frame from “bike.mpg” is 11756 bytes in size. P and B frames are 7106 and 1606 bytes respectively. In contrast, each I frame from “us.mpg” is 6479 bytes. P and B frames are 4660 and 1465 bytes. The smaller data sizes for “us.mpg” implies sparser blocks resulting in a greater speedup for the compressed domain processing. Another reason is that “bike.mpg” has a longer frame pattern than “us.mpg”. Hence, the transcoder has to perform relatively more block extraction operations in the former sequence than the latter resulting in poorer performance for “bike.mpg”.

Caching saved about 37%-40% of all potential translation operations in all our test streams except “cannon.mpg” where a high cache hit rate of 46% and a small frame sequence greatly offset the transcoding costs of a stream with a relatively high bit rate.

Overall, the performance of compressed domain transcoding depends on the bitrate of the streams, the length of the frame pattern and the degree of caching achieved.

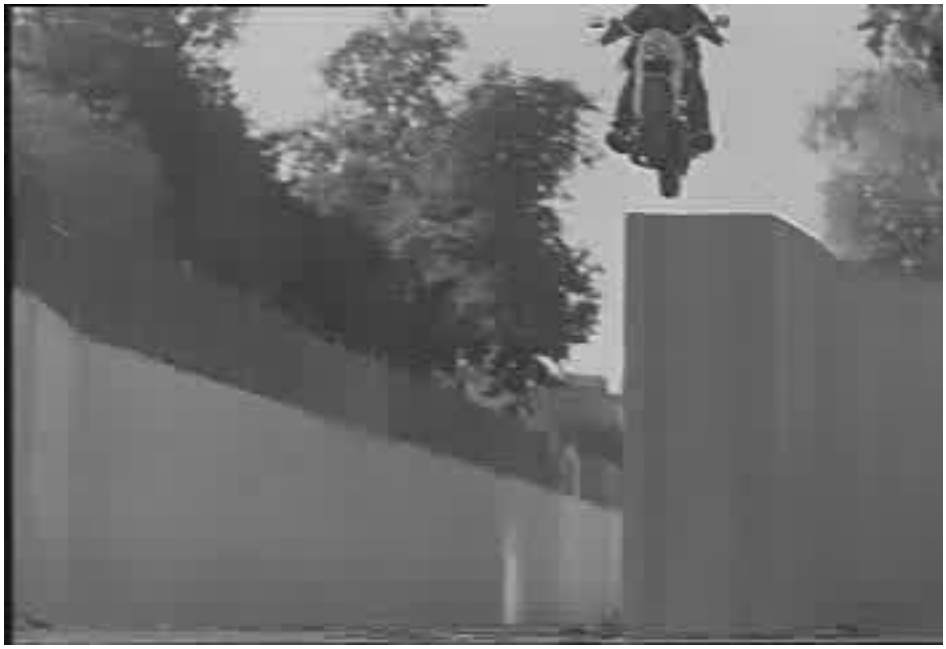

Figure 9: Same frame decoded with *maxerr*=10

7. Related Work

The work by Chang and Messerschmitt [7,8,9] is closest to ours. They developed the first techniques for solving the macroblock alignment problem in the compressed domain. Their approach is to pre- and post-multiply the reference macroblocks by appropriate matrices. They report on the performance of software simulations for video compositing in the compressed domain for motion compensated video in [9], a process that involves block translation as well as scaling, compositing and motion vector search in the compressed domain. Additionally, they mention the need for intermediate quantization to preserve image quality. Our work explores more of the design space by comparing spatial and compressed domain transcoding, and is more systems-oriented.

Other compressed domain processing techniques typically focus on specialized (but important) problems. Yeo and Liu have studied the problem of cut-detection on MPEG data [12]. They approximate Chang's techniques to compute differences between successive frames. Natarajan and Bhaskaran [13] show that the operation of shrinking an image by a factor of two can be implemented in the compressed domain using only shifts and adds. Their method approximates the transform matrices as powers of 2. Sethi and Shen [15] examine the special case of inner block transforms (IBTs), which are form of factoring compressed domain operations. IBTs, together with pixel addition, can be used to implement a wide variety of image operations. Sethi and Shen also [16] examine the decomposition of complex affine transforms, such as shearing and perspective mapping, into multipass operations. Content-based search on images and video is another related area of research. Seales [14] has done work on object recognition using an eigenspace method in the compressed domain, and Arman[17] has shown how to perform cut detection in motion-JPEG video data.

8. Conclusions

In this paper, we have evaluated the performance of software implementations of compressed domain processing for the problem of MPEG to JPEG transcoding. We have also developed a novel practical method of implementing compressed domain processing based on code generation and dynamic thresholding. We found that transcoding speed is proportional to the bitrate of the compressed video stream, the length of the frame pattern and the degree of caching possible. Performance on current generation workstations indicate 352x240 video sequences can be transcoded at around 10-15 frames/second. In other words, we are roughly a factor or two away from real time performance. Transcoding speed can also be dynamically varied, a useful feature in real-time systems where time is critical.

The MPEG to JPEG transcoding speed obtained by almost any imaginable compressed domain technique is ultimately limited by the entropy coding methods used in MPEG and JPEG, which are not well suited for software implementation. This observation applies to other types of compressed domain processing as well. We therefore conclude that next generation compression standards should consider alternative entropy coding techniques that are more well suited for software implementations. Given the complexity of compression standard like MPEG-4, such a change would cost very little in complexity, but improve our ability to process the compressed data.

9. Acknowledgments

We would like to thank Larry Rowe and Shih-Fu Chang for their comments and suggestions on an earlier draft of this paper.

References

1. Gregory K. Wallace, *The JPEG Still Picture Compression Standard*, Communications of the ACM, Volume 34, No 4, pp. 30-44, April 1991.
2. D. Le Gall, *MPEG: A Video Compression Standard for Multimedia Applications*, Communications of the ACM, Volume 34, No 4, pp. 46-58, April 1991
3. *Information Technology--Generic Coding of Moving Pictures and Associated Audio*, ISO/IEC 13818-2 Committee Draft (MPEG-2).
4. Amir, E., McCanne, S., and Zhang, H., An Application-level Video Gateway, Proc. of the Third ACM International Conference on Multimedia, November 1995, San Francisco, CA, pp. 511-522.
5. Brian C. Smith, Lawrence A. Rowe, *Algorithms for Manipulating Compressed Images*, IEEE Computer Graphics and Applications, Volume 13, No 5, pp 34-42, Sept. 1993
6. Brian C. Smith, Lawrence A. Rowe, *Compressed Domain Processing of JPEG-Encoded Images*, Real-Time Imaging, volume 1, number 2, July, 1996, pp. 3-17
7. Shih-Fu Chang, Wen-Lung Chen and David G. Messerschmitt, *Video Compositing in the DCT Domain*, IEEE Workshop on Visual Signal Processing and Communications, Raleigh, NC, pp. 138-143, Sep. 1992.
8. Shih-Fu Chang and David G. Messerschmitt, *A New Approach to Decoding and Compositing Motion Compensated DCT-Based Images*, IEEE Intern. Conf. on Acoustics, Speech, and Signal Processing, Minneapolis, Minnesota, pp. V421-V424, April, 1993
9. Shih-Fu Chang and D.G. Messerschmitt, *Manipulation and Compositing of MC-DCT Compressed Video*, IEEE Journal of Selected Areas in Communications (JSAC), Special Issue on Intelligent Signal Processing, pp. 1-11, Jan. 1995.
10. http://www.cs.cornell.edu/Info/Projects/zeno/Papers/Transcode/index.html
11. ftp://ftp.cs.cornell.edu/pub/multimed/m2j.tar.gz
12. Boon-Lock Yeo and Bede Liu, *Rapid Scene Analysis on Compressed Video*, IEEE Transactions on Circuit and Systems for Video Technology, Vol. 5, No. 6, pp. 533-544, Dec. 1995.u
13. B. Natarajan, V. Bhaskaran, *A Fast Approximate Algorithm for Scaling Down Digital Images in the DCT Domain*, IEEE International Conference on Image Processing, Oct. 23-26, 1995, Washington D.C.
14. W. Brent Seales, Matthew D. Cutts, *Vision and Multimedia: Object Recognition in the Compressed Domain*, University of Kentucky Technical Report, Oct 1995,
15. Bo Shen and Ishwar K. Sethi, *Inner-block operations on compressed images*, Proc. of the Third ACM International Conference on Multimedia, November 1995, San Francisco, CA, 1995
16. Bo Shen and Ishwar K. Sethi, *Scanline Algorithms in the JPEG Discrete Cosine Transform Compressed Domain*, Journal of Electronic Imaging, pp 182-190, April, 1996.
17. Farshid Arman, Arding Hsu, Ming-Yee Chiu, *Image Processing on Compressed Data for Large Video Databases*, Proceedings of the First ACM International Conference on Multimedia, Anaheim, CA, August 1993.

Appendix: Computing the J and $\bar{J}$ operators

```
#define U       0
#define V       1

// Lookup table to encode the zig-zag scan order
static int zz[64][2] = {
    {1, 1},
    {2, 1}, {1, 2},
    {1, 3}, {2, 2}, {3, 1},
    {4, 1}, {3, 2}, {2, 3}, {1, 4},
    {1, 5}, {2, 4}, {3, 3}, {4, 2}, {5, 1},
    {6, 1}, {5, 2}, {4, 3}, {3, 4}, {2, 5}, {1, 6},
    {1, 7}, {2, 6}, {3, 5}, {4, 4}, {5, 3}, {6, 2}, {7, 1},
    {8, 1}, {7, 2}, {6, 3}, {5, 4}, {4, 5}, {3, 6}, {2, 7}, {1, 8},
    {2, 8}, {3, 7}, {4, 6}, {5, 5}, {6, 4}, {7, 3}, {8, 2},
    {8, 3}, {7, 4}, {6, 5}, {5, 6}, {4, 7}, {3, 8},
    {4, 8}, {5, 7}, {6, 6}, {7, 5}, {8, 4},
    {8, 5}, {7, 6}, {6, 7}, {5, 8},
    {6, 8}, {7, 7}, {8, 6},
    {8, 7}, {7, 8},
    {8, 8}};
```

```
// The default quantization table
static double q[8][8] = {
   { 16,  11,  12,  14,  12,  10,  16,  14},
   { 13,  14,  18,  17,  16,  19,  24,  40},
   { 26,  24,  22,  22,  24,  49,  35,  37},
   { 29,  40,  58,  51,  61,  60,  57,  51},
   { 56,  55,  64,  72,  92,  78,  64,  68},
   { 87,  69,  55,  56,  80, 109,  81,  87},
   { 95,  98, 103, 104, 103,  62,  77, 113},
   {121, 112, 100, 120,  92, 101, 103,  99}};

// The following functions compute the compressed domain operators, as desribed by Smith [6]

#define A(u)    ((u)? 0.5 : 0.5/SQRT2)
double C(int i,u) {
   return A(u)*cos((2*i+1)*u*PI/16.0);
}

double J(int i,j,k) {
   int u = zz[k][U]-1;
   int v = zz[k][V]-1;
   return C(i,u)*C(j,v)/q[u][v];
}

double Jhat(int k,i,j) {
   int u = zz[k][U]-1;
   int v = zz[k][V]-1;
   return C(i,u)*C(j,v)*q[u][v];
}

{
   double sum, T[64][64];
     for (k=0; k<64; k++) {
          for (l=0; l<64; l++) {
             sum = 0.0;
             for (i=0; i<8; i++)
                for (j=0; j<8-dx; j++)
                   sum += J(i, j+dx, l)*Jhat(k,i,j);
             T[k][l] = sum;
          }
}
```

Code To Compute $\mathbf{T_{dx}}$