DCT Domain Transcoding of H.264/AVC Video into MPEG-2 Video

Vasant Patil[1,2], Tummala Kalyani[1], Atul Bhartia[1], Rajeev Kumar[1], and Jayanta Mukherjee[1]

[1] Computer Science and Engineering Department, Indian Institute of Technology Kharagpur, WB 721 302, India
[2] Institute for Systems Studies and Analyses, Delhi 110 054, India

Abstract. As the number of different video compression standards increase, there is a growing need for conversion between video formats coded in different standards. H.264/AVC is a newly emerging video coding standard which achieves better video quality at reduced bit rate than other standards. The standalone media players that are available in the market do not support H.264 video playback. In this paper, we present novel techniques that can achieve conversion of pre-coded video in H.264/AVC standard to MPEG-2 standard directly in the compressed domain. Experimental results show that the proposed approach can produce transcoded video with quality comparable to the pixel-domain approach at significantly reduced cost.

1 Introduction

Video transcoding deals with converting a previously compressed video signal into another one with different format, such as different bit rate, frame rate, frame size, or even compression standard. Due to the diversity of multimedia applications and present communication infrastructure comprising of different underlying networks and protocols, there has been a growing need for internetwork multimedia communication over heterogeneous networks. Besides the problem of channel characteristics and capacities, different end devices used in today's communication also introduce some problems. For example, people like to use small handheld devices, such as cellular phones, handheld computers, etc., for video communication and Internet access. Most current handheld devices only have limited computing and display capabilities, which are not suitable for high quality video decoding and display. In this case, precoded high quality video may need to be converted into a lower quality one for displaying on handheld devices.

There are applications such as video on demand, video browsing, picture in picture and video conferencing which require video to be transcoded at lower bit rates, reduced frame size and to different codec formats. H.264/AVC is a new generation video codec that has been replacing all previous standards. But it consumes enormous computing and storage resources. So there is a need for transcoding H.264/AVC bitstream to other formats. In this paper, we consider a problem of converting a bitstream coded in H.264/AVC format to MPEG-2 one. A straightforward approach to achieve this is to completely decode the H.264/AVC video into pixel domain and re-encode the decoded frames into MPEG-2 video by performing full-scale motion estimation (FSME) and motion compensation (MC). However, FSME and MC being computationally the most expensive part of the overall encoding process, this approach is not suitable for real time applications. We propose an approach that converts the pre-coded video in H.264/AVC format to MPEG-2 one directly in the DCT domain. The proposed approach obtains the motion compensated residual errors, required to code the transcoded video, directly in the DCT domain and incorporates the motion vector re-estimation techniques to obtain outgoing motion vectors. The experimental results show that the proposed approach significantly reduces the computations while achieving quality comparable to the much costlier pixel-domain approach.

The rest of the paper is organized as follows. Transform domain transcoding of H.264 video is briefly discussed in Section 2. The proposed techniques to obtain motion compensated residual errors in transcoding of I slice and P slice of H.264 video are discussed in Sub-sections 2.1 and 2.2, respectively. Experimental results are presented in Section 3, before we conclude in Section 4.

2 Transcoding in Transform Domain

A picture or frame is a collection of one or more slices in H.264/AVC coding standard. Each slice can be coded using different coding types such as I slice, P slice, B slice, SP slice and SI slice. However, baseline profile uses only two slice coding types, that is, I slice and P slice. In an I slice all macroblocks of the slice are encoded using intra prediction. In a P slice, in addition to the coding types of the I slice, some macroblocks can also be coded using inter prediction with at most one motion vector for prediction macroblock partition [1]; refer [2] for issues in H.264 to MPEG-2 transcoding.

2.1 Transcoding an *I* Slice

Unlike H.264/AVC, the MPEG-2 video does not support intra frame prediction. To transcode H.264 to MPEG-2 the intra predicted macroblocks in I picture must be converted to intra macroblocks without prediction. Conversion of an I frame of H.264 to equivalent I frame in MPEG-2 in the compressed domain is a two step process. First intra prediction is removed and then 8×8 DCT blocks are computed from four adjacent 4×4 integer transform blocks.

Removing intra prediction: The H.264/AVC comprises of two intra coding modes denoted as $Intra_{4\times4}$ or $Intra_{16\times16}$ together with chroma prediction and I_{PCM} prediction modes; see [3] for further details. In $Intra_{4\times4}$ and $Intra_{16\times16}$ macroblock, the predicted block can be obtained in transform domain by using appropriate transformation matrices. For example, the predicted block for modes 0 and 1 is obtained by Eqn. (1) and Eqn. (2) as follows:

Mode 0 (Horizontal prediction):

$$DCT\begin{pmatrix} x & x & x & I \\ x & x & x & J \\ x & x & x & K \\ x & x & x & L \end{pmatrix} DCT\begin{pmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 1&1&1&1 \end{pmatrix} = DCT\begin{pmatrix} I & I & I & I \\ J & J & J & J \\ K & K & K & K \\ L & L & L & L \end{pmatrix} \tag{1}$$

Mode 1 (Vertical prediction):

$$DCT\begin{pmatrix} 0&0&0&1 \\ 0&0&0&1 \\ 0&0&0&1 \\ 0&0&0&1 \end{pmatrix} DCT\begin{pmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ A & B & C & D \end{pmatrix} = DCT\begin{pmatrix} A & B & C & D \\ A & B & C & D \\ A & B & C & D \\ A & B & C & D \end{pmatrix} \tag{2}$$

Mode 2 (DC prediction): It is the average of all the neighboring pixels in upper and left neighboring blocks. To get a predicted block for prediction mode 2 in DCT domain, the upper block is pre-multiplied by $DCT\begin{pmatrix} 0&0&0&1 \\ 0&0&0&1 \\ 0&0&0&1 \\ 0&0&0&1 \end{pmatrix}$ and post-multiplied by $DCT\begin{pmatrix} 1&1&1&1 \\ 1&1&1&1 \\ 1&1&1&1 \\ 1&1&1&1 \end{pmatrix}$. Similarly, the left block is pre-multiplied by $DCT\begin{pmatrix} 1&1&1&1 \\ 1&1&1&1 \\ 1&1&1&1 \\ 1&1&1&1 \end{pmatrix}$ and post-multiplied by $DCT\begin{pmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 1&1&1&1 \end{pmatrix}$. The resultant blocks are then summed up and averaged. Since, the transformation matrices are sparse and their transform is also sparse, intra prediction for these modes in DCT domain requires less computation than pixel domain computation. Similarly, proper transformation matrices for other macroblocks are obtained. The transform domain residual is then added to the predicted blocks so obtained. Table 1 shows the comparison of the intra prediction techniques with pixel domain processing. In this table, a, s and d denote addition, shift and division operations, respectively. It is found that transcoding I picture in transform domain is three times faster as compared to pixel domain transcoding.

Table 1. Computational complexity of transcoding an I slice

Functions	Pixel Domain	Transform Domain		
		Mode 0	Mode 1	Mode 2
IDCT	$16a$+$24s$	0	0	0
Computing a predicted block	0 for mode 0 and 1 $128a$+$16d$ for mode 2	$12a$+$16s$	$12a$+$16s$	$40a$+$40s$+$16d$
Adding residual	$16a$	$16a$	$16a$	$16a$
forward DCT	$16a$+$24s$	0	0	0
Total	$48a$+$32s$+$16s$ for mode 0 and 1 $176a$+$32s$+$16s$+$16d$ for mode 2	$28a$+$16s$	$28a$+$16s$	$56a$+$40s$+$16d$

Transform and block size conversion: Four 4×4 adjacent blocks of H.264 bitstream are converted into one single 8×8 block as follows:

$$X' = \left[S_8\right] \left\{ \begin{bmatrix} I_4^t & 0 \\ 0 & I_4^t \end{bmatrix} \begin{bmatrix} X_1 & X_2 \\ X_3 & X_4 \end{bmatrix} \begin{bmatrix} I_4 & 0 \\ 0 & I_4 \end{bmatrix} \right\} \left[S_8^t\right] \tag{3}$$

where, 't' denotes transposition operation, S_8 is an 8×8 real $2D$ DCT matrix and $X_1 \ldots X_4$ are 4×4 transform coefficient blocks of H.264/AVC bitstream. I_4 is a 4×4 integer transform matrix given as:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$$

Let $T = \begin{bmatrix} S_8 \end{bmatrix} \begin{bmatrix} I_4^t & 0 \\ 0 & I_4^t \end{bmatrix}$ and $T^t = \begin{bmatrix} I_4 & 0 \\ 0 & I_4 \end{bmatrix} \begin{bmatrix} S_8^t \end{bmatrix}$. The matrix T is given as:

$$\begin{pmatrix} a & 0 & 0 & 0 & a & 0 & 0 & 0 \\ b & f & -l & p & -b & f & l & p \\ 0 & g & 0 & j & 0 & -g & 0 & -j \\ -c & h & m & -q & c & h & -m & -q \\ 0 & 0 & a & 0 & 0 & 0 & a & 0 \\ d & -i & n & r & -d & -i & -n & r \\ 0 & -j & 0 & g & 0 & j & 0 & -g \\ -e & k & -o & s & e & k & o & s \end{pmatrix} \tag{4}$$

where $a = 1.4142$, $b = 1.2815$, $f = 0.4618$, $l = 0.1056$, $p = 0.0585$, $g = 1.1152$, $j = 0.0793$, $c = 0.4500$, $h = 0.8899$, $m = 0.7259$, $q = 0.0461$, $d = 0.3007$, $i = 0.4319$, $n = 1.0864$, $r = 0.5190$, $e = 0.2549$, $k = 0.2412$, $o = 0.5308$, $s = 0.9875$. It can be pre-computed and stored. Since, this matrix is sparse and symmetric it can be computed as similar to the method suggested by [4]. It needs a total of 704 operations. The pixel domain approach needs 256 multiplications and 416 additions for DCT(S_8). According to [3], each inverse transform (I_4) needs 8 shifts and 32 additions giving a total of 32 shifts and 128 additions for four I_4. The overall computation requirement of the pixel domain processing is 256 multiplications, 32 shifts, 544 additions, for a total of 832 operations. Hence, the DCT domain approach with fast transform implementation saves 128 operations for an 8×8 block, saving about 15% of the computation.

2.2 Transcoding a *P* Slice

Motion estimation (ME) is the most compute intensive process in video encoding. The ME component is more complex in H.264 because it uses motion vectors that can point to areas outside the picture boundary. The H.264 coding also supports a number of different macroblock partition shapes and sizes for each macroblock resulting in a maximum of sixteen motion vectors [3]. It also uses multiple reference frames and quarter-pixel motion vector resolution increasing the search range thereby the complexity.

To transcode H.264 to MPEG-2, the multi-frame references have to be collapsed to a single-frame reference and motion vectors have to be displaced based on the macroblock partition size used in H.264 as MPEG-2 does not support as many macroblock partition sizes for motion compensation. Note that, in this case of transcoding H.264 to MPEG-2, we have considered only baseline profile video which uses single reference frame only. Transcoding inter frames involves three step. First, all the P slices of H.264/AVC are converted into I slices by inverse motion compensation in 4×4 DCT domain. Next, these I slices are converted to an equivalent I frames in MPEG-2 using 8×8 block DCT and then finally, converted to P frames by forward motion compensation in DCT domain.

Converting a *P* frame to an *I* frame: A P frame can be converted to an intra (I) frame by performing the Inverse Motion Compensation (IMC) operation. The DCT domain IMC was first studied in Chang *et al.* [5] and subsequently in Merhav *et al.* [5], Liu *et al.* [6] and Assuncao *et al.* [7]. Their techniques compute a predicted block in DCT domain. A reference block B_{ref} may intersect with four neighboring blocks as shown in Fig. 1. The h and w represent vertical

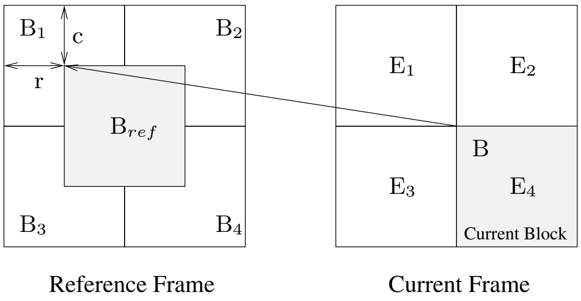


Fig. 1. Single blockwise inverse motion compensation

and horizontal components of the motion vector respectively. If B_1, B_2, B_3, B_4 represent the four neighboring blocks in the spatial domain, then block B_{ref} can be represented by Eqn. (5) as below:

$$B_{ref} = \sum_{i=1}^{4} c_{i1} B_i c_{i2} \tag{5}$$

For a 4×4 block, c_{ij} , $i = 1 \dots 4$ and $j = 1, 2$ are 4×4 sparse matrices of 0 and 1 that perform window and shift operations accordingly. From Eqn. (5), we have

$$DCT(B_{ref}) = S_4 \left(\sum_{i=1}^{4} c_{i1} S_4^t S_4 B_i S_4^t S_4 c_{i2} \right) S_4^t \tag{6}$$

where, S_4 represents a 4-point DCT matrix. Since, DCT is an unitary orthogonal transformation and is guaranteed to be distributive to matrix multiplications, above equation can be re-written as:

$$DCT(B_{ref}) = \sum_{i=1}^{4} DCT(c_{i1}) DCT(B_i) DCT(c_{i2}) \tag{7}$$

The DCT of the inverse motion compensated block from the current error residual block E is then given as:

$$DCT(B) = DCT(B_{ref}) + DCT(E) \tag{8}$$

Using the approach presented in [5] and [6] to compute the 4×4 DCT block from the reference frame, we require sixteen iterations to completely predict a macroblock of partition size 16×16. In the proposed approach, we extend these schemes to compute the macroblock partition block in one step. We also extend it to include half-pixel interpolation using 6-tap FIR filter. We present IMC for an 8×8 macroblock partition. The extension to the other partitions is straightforward.

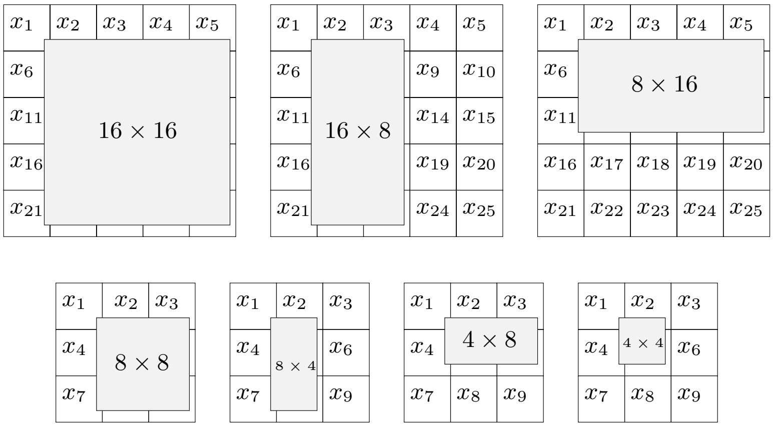


Fig. 2. Macroblock partition wise inverse motion compensation

As shown in Fig. 2, X' is the predicted macroblock in the reference frame which starts from the location (r, c) with reference to the first block in the array of adjacent blocks. If $x_1 \ldots x_9$ are the adjacent blocks in spatial domain then an 8×8 macroblock partition block from the 12×12 block can be extracted as follows:

$$x' = L_r \begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix} R_c \tag{9}$$

where, x' is the predicted macroblock in spatial domain, L_r is an 8×12 matrix and R_c is a 12×8 matrix. These matrices are different for different values of r and c (refer Fig. 1). Since $1 \le r \le 4$ and $1 \le c \le 4$, there can be four different L_r and R_c matrices which can be pre-computed and stored. The structure of L_r matrix is given as:

$$L_r = \left[0_{8 \times r-1} \; I_8 \; 0_{8 \times 4-r+1} \right]_{8 \times 12}$$

where, I_8 is an identity matrix of length 8 and '0' represents a matrix of zero elements. Similarly, we can derive R_c matrices. Let us define 12×12 matrices $\overline{S}$, $\overline{S}^t$ and A as follows:

$$\overline{S} = \begin{pmatrix} I_4 & 0 & 0 \\ 0 & I_4 & 0 \\ 0 & 0 & I_4 \end{pmatrix}, \quad \overline{S}^t = \begin{pmatrix} I_4^t & 0 & 0 \\ 0 & I_4^t & 0 \\ 0 & 0 & I_4^t \end{pmatrix} \text{ and } A = \begin{pmatrix} X_1 & X_2 & X_3 \\ X_4 & X_5 & X_6 \\ X_7 & X_8 & X_9 \end{pmatrix}$$

where, I_4 is a 4×4 forward integer transform matrix, 't' denotes matrix transposition. I_4^t is a 4×4 inverse integer transform matrix of H.264/AVC and $X_i = DCT(x_i)$. Assuming that we have obtained X_1 to X_9 by partial decoding, Eqn. (9) can be re-written to extract macroblock X' in DCT domain as:

$$X' = \begin{pmatrix} S_4 & 0 \\ 0 & S_4 \end{pmatrix} \left\{ L_r \times \overline{S}^t \times A \times \overline{S} \times R_c \right\} \begin{pmatrix} S_4^t & 0 \\ 0 & S_4^t \end{pmatrix} \tag{10}$$

The matrix multiplication inside the curly braces results in an 8×8 spatial domain block. The pre-multiplication of $\begin{pmatrix} S_4 & 0 \\ 0 & S_4 \end{pmatrix}$ and post-multiplication of $\begin{pmatrix} S_4^t & 0 \\ 0 & S_4^t \end{pmatrix}$ result in an 8×8 macroblock partition. With the above procedure, macroblock partitions of size 16×16, 16×8, 8×16 etc., can also be computed. Since, H.264 uses motion vectors that can point to areas outside the picture boundary, all adjacent blocks required to compute the predicted block may not be available. In that case, the reference frame is extrapolated beyond the image boundaries by repeating the edge samples before interpolation. For example, Fig. 3 illustrates the need for expansion. The blocks outside the picture boundary are obtained by

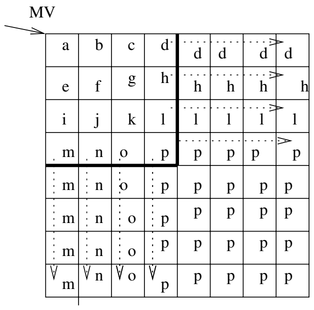


Fig. 3. Motion vectors pointing outside object boundary

copying the boundary row or column pixels. This is achieved by pre-multiplying the collected adjacent block matrices x_1 to x_9 in Eqn. (9) for up and down directions and post-multiplying them for left and right directions with proper matrices. It is found that total sixteen type of expansion matrices are required. Eqn. (10) may be re-written to consider expansion matrices as follows:

$$X' = \begin{pmatrix} S_4 & 0 \\ 0 & S_4 \end{pmatrix} \left\{ L_r \times e_r \times \overline{S}^t \times A \times \overline{S} \times e_c \times R_c \right\} \begin{pmatrix} S_4^t & 0 \\ 0 & S_4^t \end{pmatrix} \tag{11}$$

where, e_r and e_c are row and column wise expansion matrices.

Half-pixel and quarter-pixel inverse motion compensation: H.264/AVC uses quarter-pixel accurate motion vectors with 6-tap FIR filter. The motion vectors in MPEG-2 are half-pixel accurate and the half-pixel samples are obtained by bilinear interpolation of the neighboring four samples. In inverse motion compensation using fractional sample accuracy, the 6-tap FIR filter should be used to obtain luma half-pixel samples, bilinear interpolation to obtain quarter-pixel luma samples and weighted bilinear interpolation to obtain $\frac{1}{8}^{th}$ pixel accurate chroma samples. The fractional pixel predicted luma block is computed by applying 6-tap FIR filter in horizontal direction only, vertical direction only, horizontal first and then vertical direction, vertical first and then horizontal direction. This is achieved by modifying the L_r and R_c matrices in Eqn. (9) and Eqn. (10) to include the 6-tap FIR filter. For example, L_r and R_c matrices for 4×4 macroblock partition using horizontal direction only half samples when $r = 3$ and $c = 3$ are given below:

$$L_r = \left[\begin{array}{cccccccccccc} 0 & 0 & 1 & -5 & 20 & 20 & -5 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -5 & 20 & 20 & -5 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -5 & 20 & 20 & -5 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -5 & 20 & 20 & -5 & 1 & 0 \\ \multicolumn{2}{c}{2\ column} & \multicolumn{9}{c}{9\ column} & 1\ column \end{array}\right]$$

and

$$R_c = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^t$$

Similarly, L_r and R_c matrices for vertical direction only half samples are obtained.

Chroma sub-pixel interpolation in transform domain: Chroma $\frac{1}{8}^{th}$ pixel samples are obtained using L_r and R_c matrices. For $r = 3$ and $c = 3$, we have

$$L_r = \left[\begin{array}{cccccccccccc} 0 & 0 & jf0 & jf1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & jf0 & jf1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & jf0 & jf1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & jf0 & jf1 & 0 & 0 & 0 & 0 & 0 \\ \multicolumn{2}{c}{2\ column} & \multicolumn{5}{c}{5\ column} & \multicolumn{5}{c}{5\ column} \end{array}\right]$$

and

$$R_c = \begin{bmatrix} 0 & 0 & if0 & if1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & if0 & if1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & if0 & if1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & if0 & if1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^t$$

where, $if0 = xFrac$, $if1 = 8 - xFrac$, $jf0 = yFrac$, $jf1 = 8 - yFrac$. The $xFrac$ and $yFrac$ denote fractional part of x and y component of a motion vector, respectively. Similar, matrices can be derived for other values of r and c.

In this way, a P frame in H.264 is converted to an I frame without intra prediction. This I frame consists of DCT blocks of size 4×4. The transform block size and kernel is converted by using the conversion transform kernel in Eqn. (4). The I frames are then converted back to the P frames in MPEG-2 as discussed in the following section.

Conversion of an *I* frame in MPEG-2 into a *P* frame in MPEG-2: A P frame is obtained by performing motion estimation and then motion compensation. In this case, the motion vectors can be re-estimated by AMVR method [8]. Motion compensation in DCT domain is done using the similar approach discussed in Section 2.2. The predicted macroblock (16×16) is obtained by applying Merhav's scheme [5] for the whole macroblock at once as follows:

$$X' = \begin{pmatrix} S_8 & 0 \\ 0 & S_8 \end{pmatrix} \Big\{ L_r \times \widehat{S}^t \times A \times \widehat{S} \times R_c \Big\} \begin{pmatrix} S_8^t & 0 \\ 0 & S_8^t \end{pmatrix} \tag{12}$$

where, S_8 is forward DCT matrix of size 8×8. L_r and R_c are row and column transformation matrices, respectively, as explained in Sect. 2.2. $\widehat{S}$, $\widehat{S}^t$ and A denote 32×32 matrices given as $\widehat{S} = \begin{pmatrix} S_8 & 0 & 0 \\ 0 & S_8 & 0 \\ 0 & 0 & S_8 \end{pmatrix}$, $\widehat{S}^t = \begin{pmatrix} S_8^t & 0 & 0 \\ 0 & S_8^t & 0 \\ 0 & 0 & S_8^t \end{pmatrix}$ and $A = \begin{pmatrix} X_1 & X_2 & X_3 \\ X_4 & X_5 & X_6 \\ X_7 & X_8 & X_9 \end{pmatrix}$. An 8-point DCT matrix is factorized as $S_8 = DPB_1B_2MA_1A_2A_3$ where, D is an 8×8 diagonal matrix and P is an 8×8 permutation matrix. B_1, B_2, A_1, A_2, A_3 are 8×8 sparse matrices of 1, 0 and -1. M is an 8×8 sparse matrix of real numbers. Refer [5] for exact entries of D, P, B_1, B_2, A_1, A_2, A_3 and M matrices.

Then, $\widehat{S}^t$ can be re-written using the above factorization as follows:

$$\widehat{S}^t = \underbrace{\begin{bmatrix} Q^t & 0 & 0 \\ 0 & Q^t & 0 \\ 0 & 0 & Q^t \end{bmatrix}}_{\widehat{Q}^t} \underbrace{\begin{bmatrix} B_2^t & 0 & 0 \\ 0 & B_2^t & 0 \\ 0 & 0 & B_2^t \end{bmatrix}}_{\widehat{B}_2^t} \underbrace{\begin{bmatrix} B_1^t & 0 & 0 \\ 0 & B_1^t & 0 \\ 0 & 0 & B_1^t \end{bmatrix}}_{\widehat{B}_1^t} \underbrace{\begin{bmatrix} P^t & 0 & 0 \\ 0 & P^t & 0 \\ 0 & 0 & P^t \end{bmatrix}}_{\widehat{P}^t} \underbrace{\begin{bmatrix} D^t & 0 & 0 \\ 0 & D^t & 0 \\ 0 & 0 & D^t \end{bmatrix}}_{\widehat{D}^t} \tag{13}$$

where $Q^t = (MA_1A_2A_3)^t$ given as:

$$Q^t = \begin{bmatrix} 1 & 1 & a & 1 & -c & 0 & b & 1 \\ 1 & -1 & a & 0 & -c & a & b & 0 \\ 1 & -1 & -a & 0 & b & a & c & 0 \\ 1 & 1 & -a & -1 & b & 0 & c & 0 \\ 1 & 1 & -a & -1 & -b & 0 & -c & 0 \\ 1 & -1 & -a & 0 & -b & -a & -c & 0 \\ 1 & -1 & a & 0 & c & -a & -b & 0 \\ 1 & 1 & a & 1 & c & 0 & -b & 1 \end{bmatrix}$$

$a = 0.7071$, $b = 0.9239$, and $c = 0.3827$. Note that, $\widehat{D}^t$, $\widehat{P}^t$, $\widehat{B}_1^t$, $\widehat{B}_2^t$ and $\widehat{Q}^t$ denote matrices of size 24×24. Similarly, $\widehat{S}$ can also be factorized as $\widehat{S} = \widehat{D}\widehat{P}\widehat{B}_1\widehat{B}_2\widehat{Q}$. The Eqn. (12) can be re-written as:

$$M' = \begin{pmatrix} S_8 & 0 \\ 0 & S_8 \end{pmatrix} \Big\{ L_r \times \widehat{Q}^t\widehat{B}_2^t\widehat{B}_1^t\widehat{P}^t\widehat{D}^t \times A \times \widehat{D}\widehat{P}\widehat{B}_1\widehat{B}_2\widehat{Q} \times R_c \Big\} \begin{pmatrix} S_8^t & 0 \\ 0 & S_8^t \end{pmatrix} \tag{14}$$

The multiplication by $\widehat{Q}^t(\widehat{Q})$, $\widehat{B}_2^t(\widehat{B}_2)$, $\widehat{B}_1^t(\widehat{B}_1)$, $\widehat{P}^t(\widehat{P})$ and $\widehat{D}^t(\widehat{D})$ can be realized by performing multiplication with corresponding 8×8 component matrices

$Q^t(Q)$, $B_2^t(B_2)$, $B_1^t(B_1)$, $P^t(P)$ and $D^t(D)$, respectively. When counting the operations, multiplication by P^t and P can be ignored as they cause only changes in the order of the components. The multiplications by D^t and D can also be ignored while counting the operations because these can be absorbed in the quantizer and dequantizer [5]. The multiplication of $\widehat{B}_1(\widehat{B}_1^t)$ and $\widehat{B}_2(\widehat{B}_2^t)$ matrices with another 24×24 arbitrary matrix requires 288 addition operations. Let $J_r = L_r \times Q^t$ and $K_c = Q \times R_c$. The J_r and K_c matrices are sparse having similar kind of structure. We adopt a similar strategy as suggested in [5] to perform multiplication with J_r and K_c matrices. This in worst case ($r = c = 5$) requires $880m + 5248a$ operations, where 'a' denotes addition and 'm' denotes multiplication operation. This means $3.59m + 23.06a$ operations per pixel to extract a 16×16 macroblock. By assuming one multiplication to be equivalent to three machine instructions and one addition to be equivalent to one machine instruction this is 23.52% improvement over 8×8 block based approach of Merhav *et al.* [5].

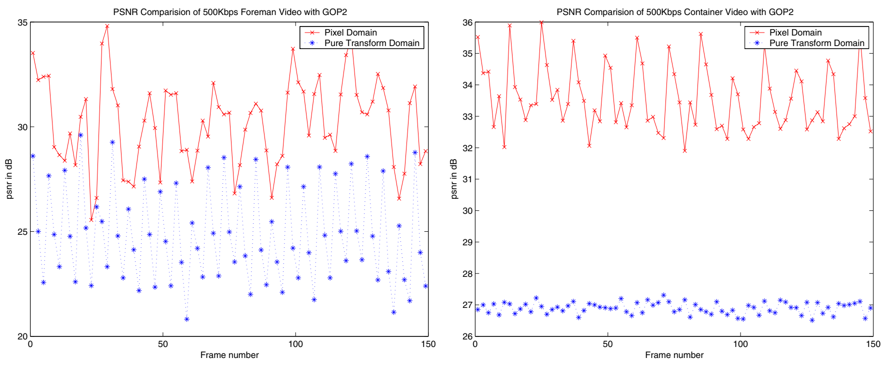
Half-precision motion vectors: With half-pixel precision motion vectors, either two or four pixels are needed to calculate the actual prediction of single pixel. This means, in worst case, we need to apply Eqn. (14) four times to extract M' with half-pixel precision motion vectors along both the directions as:

$$M' = \begin{pmatrix} S_8 & 0 \\ 0 & S_8 \end{pmatrix} \left\{ \overline{L}_r \times \widehat{Q}^t \widehat{B}_2^t \widehat{B}_1^t \widehat{P}^t \widehat{D}^t \times A \times \widehat{D}\widehat{P}\widehat{B}_1\widehat{B}_2\widehat{Q} \times \overline{R}_c \right\} \begin{pmatrix} S_8^t & 0 \\ 0 & S_8^t \end{pmatrix} \quad (15)$$

where, $\overline{L}_r = \frac{1}{2}(L_r + L_{r+1})$ and $\overline{R}_c = \frac{1}{2}(R_c + R_{c+1})$. Multiplication by $\overline{L}_r$ and $\overline{R}_c$ require 384 multiplications and 384 addition operations each. This means $6.09m + 19.38a$ operations per pixel to extract a 16×16 macroblock with half-pixel precision motion vectors along both the directions. By assuming one multiplication to be equivalent to three machine instructions and one addition to be equivalent to one machine instruction this is 79.42% improvement over the brute-force approach of Merhav *et al.* [5].

3 Experimental Results

The experimental results are based on our transcoding implementation using JM reference software version 10.2. To present the results we use *Foreman* and *Container* test sequences. The first 150 frames of these sequences in SIF (352×288) format are encoded using the baseline profile with I and P frames. Table 2 shows the computation comparison of the proposed DCT domain approach with the pixel domain approach. To obtain the outgoing motion vectors in transform domain approach, we have used AMVR method [8]. It is assumed that only 50% of the 4×4 transform block has non zero coefficients. It is observed that about 80% of the inter frame blocks have diagonal mode of interpolation. Fig. 4(a) and (b), show the PSNRs (dB) for individual frames of *Foreman* and *Container* sequences, respectively. As it can be seen, the proposed DCT domain approach produces the transcoded video with quality comparable with the pixel-domain approach at substantially reduced computations.



(a) (b)

Fig. 4. Experimental results: (a) *Foreman* (b) *Container*

Table 2. Computational complexity

Approach	Functions	Complexities		
		Mults.	Adds.	Shifts
Pixel-Domain	IDCT ($32a$+$8s$ per $4x4$ block)		512	128
	IMC interpolation ($192m$+$192a$ per $4x4$ block)	3072	3072	
	AMVR ($36m$+$50a$ per Macroblock)	36	50	
	FDCT ($256m$+$461a$ per $8x8$ block)	1024	1844	
	Total	4132	5478	128
DCT-Domain	MPIMC interpolation($24a$+$94s$ per $4x4$ block)		384	1504
	8×8 DCT conversion ($352m$+$352a$ per $8x8$ block)	1408	1408	
	AMVR ($36m$+$50a$ per Macroblock)	36	50	
	MC ($3.59m$+$23.06a$ or $6.09m$+$19.38a$ per pixel)	879	5248	
	Total	2323	7090	1504

4 Conclusions

We have presented a transform domain approach to convert the H.264/AVC video to MPEG-2 video. In this, we have presented novel techniques to convert *I* and *P* slice in H.264/AVC video to MPEG-2 frames, directly in the DCT domain. As compared with the pixel domain approach, the proposed approach significantly reduces the computational requirement. Our experimental results using baseline profile show that the proposed approach produces MPEG-2 video with PSNR comparable to the pixel domain approach.

References

1. Sullivan, G., Topiwala, P., Luthra, A.: The H.264/AVC advanced video coding standard: Overview and introduction to fidelity range extensions. SPIE Conference on Applications of Digital Image Processing XXVII Special Session on Advances in the New Emerging Standard: H.264/AVC (2004)

2. Kalva, H.: Issues in H.254/MPEG-2 video transcoding. In: First IEEE Consumer Communications and Networking Conference. (2004) 657–659
3. Weigand, T., Sullivan, T.: Draft ITU-T recommendation and final draft international standard of joint video specification. ITU-T Rec. H.264 — ISO/IEC 14496-10 AVC (2003)
4. Xin, J., Vetro, A., Sun, H.: Converting DCT coefficients to H.264/AVC transform coefficients. In: IEEE Pacific-Rim Conference on Multimedia (PCM), Lecure Notes in Computer Science. (2004)
5. Merhav, N., Bhaskaran, V.: Fast algorithms for DCT-Domain image down-sampling and for inverse motion compensation. IEEE Transactions on Circuits and Systems for Video Technology **7** (1997) 468–474
6. Koc, U.V., Liu, K.J.R.: Motion compensation on DCT domain. EURASIP Journal on Applied Signal Processing (2001) 147–162
7. Assuncao, P.A.A., Ghanbari, M.: A frequency-domain video transcoder for dynamic bitrate reduction of MPEG-2 bit streams. IEEE Transactions on Circuits and Systems for Video Technology **8** (1998) 953–967
8. Shen, B., Ishwar, I.K., Bhaskaran, V.: Adaptive motion-vector re-sampling for compressed video downscaling. IEEE Transactions on Circuits and Systems for Video Technology **9** (1999) 929–936