Compressed-Domain Video Processing

Susie Wee, Bo Shen, John Apostolopoulos
Mobile and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2002-282
October 6th , 2002*

E-mail: {swee, boshen, japos}@hpl.hp.com

compressed domain processing, transcoding, video editing, MPEG, splicing, reverse play, frame rate conversion, interlace to progressive, motion vector resampling

Video compression algorithms are being used to compress digital video for a wide variety of applications, including video delivery over the Internet, advanced television broadcasting, video streaming, video conferencing, and video storage and editing. The impressive performance of modern compression algorithms, combined with the growing availability of video encoders and decoders and low-cost computers, storage devices, and networking equipment, makes it evident that between video capture and video playback, video will be handled in compressed video form. The resulting end-to-end compressed digital video systems motivate the need to develop efficient algorithms for handling compressed digital video.

Compute- and memory-efficient, quality-preserving algorithms for handling compressed video streams are called compressed-domain processing (CDP) algorithms. CDP algorithms are useful for a number of applications. For example, a video server transmitting video over the Internet may be restricted by stringent bandwidth requirements. In this scenario, a high-rate compressed bitstream may need to be transcoded to a lower-rate compressed bitstream prior to transmission; this can be achieved by lowering the spatial or temporal resolution of the video or by more coarsely quantizing the MPEG data. Another application may require MPEG video streams to be transcoded into streams that facilitate video editing functionalities such as splicing or fast-forward and reverse play; this may involve removing the temporal dependencies in the coded data stream. Finally, in a video communication system, the transmitted video stream may be subject to harsh channel conditions resulting in data loss; in this instance it may be useful to create a standard-compliant video stream that is more robust to channel errors and network congestion.

This chapter focuses on developing CDP algorithms for bitstreams that are based on video compression algorithms that rely on the block discrete cosine transform (DCT) and motion-compensated prediction, which includes a number of predominant image and video coding standards including JPEG, MPEG-1, MPEG-2, MPEG-4, H.261, H.263, and H.264/MPEG-4 AVC. These CDP algorithms achieve efficiency by using techniques that exploit the coding structures used in the original compression process; these techniques are discussed in detail. Two classes of CDP algorithms are presented–compressed-domain transcoding algorithms that change the video format and compression format of compressed video streams and compressed-domain editing algorithms that perform video processing and editing operations on compressed video streams.

* Internal Accession Date Only
Approved for External Publication

COMPRESSED-DOMAIN VIDEO PROCESSING

Susie Wee, Bo Shen, John Apostolopoulos
Streaming Media Systems Group
Hewlett-Packard Laboratories
Palo Alto, CA, USA
{swee,boshen,japos}@hpl.hp.com

1. INTRODUCTION

Video compression algorithms are being used to compress digital video for a wide variety of applications, including video delivery over the Internet, advanced television broadcasting, video streaming, video conferencing, as well as video storage and editing. The performance of modern compression algorithms such as MPEG-1, MPEG-2, MPEG-4, H.261, H.263, and H.264/MPEG-4 AVC is quite impressive -- raw video data rates often can be reduced by factors of 15-80 or more without considerable loss in reconstructed video quality. This fact, combined with the growing availability of video encoders and decoders and low-cost computers, storage devices, and networking equipment, makes it evident that between video capture and video playback, video will be handled in compressed video form.

End-to-end compressed digital video systems motivate the need to develop algorithms for handling compressed digital video. For example, algorithms are needed to adapt compressed video streams for playback on different devices and for robust delivery over different types of networks. Algorithms are needed for performing video processing and editing operations, including VCR functionalities, on compressed video streams. Many of these algorithms, while simple and straightforward when applied to raw video, are much more complicated and computationally expensive when applied to compressed video streams. This motivates the need for developing efficient algorithms for performing these tasks on compressed video streams.

In this chapter, we describe compute- and memory-efficient, quality-preserving algorithms for handling compressed video streams. These algorithms achieve efficiency by exploiting coding structures used in the original compression process. This class of efficient algorithms for handling compressed video streams are called compressed-domain processing (CDP) algorithms. CDP algorithms that change the video format and compression

format of compressed video streams are called compressed-domain transcoding algorithms, and CDP algorithms that perform video processing and editing operations on compressed video streams are called compressed-domain editing algorithms.

These CDP algorithms are useful for a number of applications. For example, a video server transmitting video over the Internet may be restricted by stringent bandwidth requirements. In this scenario, a high-rate compressed bitstream may need to be transcoded to a lower-rate compressed bitstream prior to transmission; this can be achieved by lowering the spatial or temporal resolution of the video or by more coarsely quantizing the MPEG data. Another application may require MPEG video streams to be transcoded into streams that facilitate video editing functionalities such as splicing or fast-forward and reverse play; this may involve removing the temporal dependencies in the coded data stream. Finally, in a video communication system, the transmitted video stream may be subject to harsh channel conditions resulting in data loss; in this instance it may be useful to create a standard-compliant video stream that is more robust to channel errors and network congestion.

This chapter presents a series of compressed-domain image and video processing algorithms that were designed with the goal of achieving high performance with computational efficiency. It focuses on developing transcoding algorithms for bitstreams that are based on video compression algorithms that rely on the block discrete cosine transform (DCT) and motion-compensated prediction. These algorithms are applicable to a number of predominant image and video coding standards including JPEG, MPEG-1, MPEG-2, MPEG-4, H.261, H.263, and H.264/MPEG-4 AVC. Much of this discussion will focus on MPEG; however, many of these concepts readily apply to the other standards as well.

This chapter proceeds as follows. Section 2 defines the compressed-domain processing problem. Section 3 gives an overview of MPEG basics and it describes the CDP problem in the context of MPEG. Section 4 describes the basic methods used in CDP algorithms. Section 5 describes a series of CDP algorithms that use the basic methods of Section 4. Finally, Section 6 describes some advanced topics in CDP.

2. PROBLEM STATEMENT

Compressed-domain processing performs a user-defined operation on a compressed video stream without going through a complete decompress/process/re-compress cycle; the processed result is a new compressed video stream. In other words, the goal of compressed-domain processing (CDP) algorithms is to efficiently process one standard-compliant compressed video stream into another standard-compliant compressed video stream with a different set of properties. Compressed-domain transcoding algorithms are used to change the video format or compression format of compressed streams, while compressed-domain editing algorithms are used to perform processing operations on compressed streams. CDP differs from the encoding and decoding processes in that both the input and output of the transcoder are compressed video streams.

A conventional solution to the problem of processing compressed video streams, shown in the top path of Figure 1, involves the following steps: first, the input compressed video stream is completely decompressed into its pixel-domain representation; this pixel-domain video is then processed with the appropriate operation; and finally the processed video is recompressed into a new output compressed video stream. Such solutions are computationally expensive and have large memory requirements. In addition, the quality of the coded video can deteriorate with each re-coding cycle.

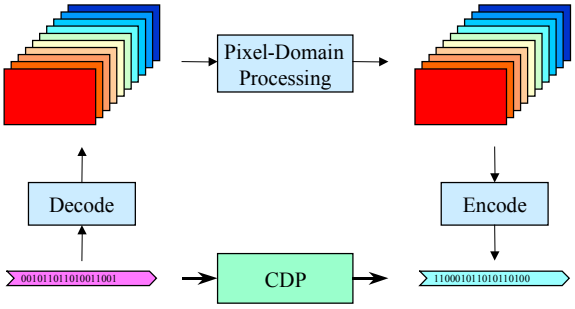


Figure 1. Processing compressed video: the conventional pixel-domain solution (top path) and the compressed-domain processing solution (bottom path).

Compressed-domain processing methods can lead to a more efficient solution by only partially decompressing the bitstream and performing processing directly on the compressed-domain data. The resulting CDP algorithms can have significant savings over their conventional pixel-domain processing counterparts. Roughly speaking, the degree of savings will depend on the particular operation, the desired performance, and the amount of decompression required for the particular operation. This is discussed further in Subsection 3.4 within the context of MPEG compression.

3. MPEG CODING AND COMPRESSED-DOMAIN PROCESSING

3.1 MPEG FRAME CODING

Efficient CDP algorithms are designed to exploit various features of the MPEG video compression standards. Detailed descriptions of the MPEG video compression standards can be found in [1][2]. This section briefly reviews some aspects of the MPEG standards that are relevant to CDP.

MPEG codes video in a hierarchy of units called sequences, groups of pictures (GOPs), pictures, slices, macroblocks, and blocks. 16x16 blocks of pixels in the original video frames are coded as a macroblock, which consists of four 8x8 blocks. The macroblocks are scanned in a left-to-right, top-to-bottom fashion, and series of these macroblocks form a slice. All the slices in a frame comprise a picture, contiguous pictures form a GOP, and all the GOPs form the entire sequence. The MPEG syntax allows a GOP to contain any number of frames, but typical sizes range from 9 to 15 frames. Each GOP refreshes the temporal prediction by coding the first frame in intraframe mode, i.e. without prediction. The remaining frames in the GOP can be coded with intraframe or interframe (predictive) coding techniques.

The MPEG algorithm allows each frame to be coded in one of three modes: intraframe (I), forward prediction (P), and bidirectional prediction (B). A typical IPB pattern in *display order* is:

$B_7\ B_8\ P_9\ B_{10}\ B_{11}\ I_0\ B_1\ B_2\ P_3\ B_4\ B_5\ P_6\ B_7\ B_8\ P_9\ B_{10}\ B_{11}\ I_0\ B_1\ B_2\ P_3$

The subscripts represent the index of the frame within a GOP. I frames are coded independently of other frames. P frames depend on a prediction based on the preceding I or P frame. B frames depend on a prediction based on the preceding and following I or P frames. Notice that each B frame depends on data from a future frame, i.e. future frame must be (de)coded before a current B frame can be (de)coded. For this reason, the coding order is distinguished from the display order. The *coding order* for the sequence shown above is:

$P_9\ B_7\ B_8\ G\ I_0\ B_{10}\ B_{11}\ P_3\ B_1\ B_2\ P_6\ B_4\ B_5\ P_9\ B_7\ B_8\ G\ I_0\ B_{10}\ B_{11}\ P_3\ B_1\ B_2$

MPEG requires the coded video data to be placed in the data stream in coding order. G represents a GOP header that is placed in the compressed bitstream.

A GOP always begins with an I frame. Typically, it includes the following (display order) P and B frames that occur before the next I frame, although the syntax also allows a GOP to contain multiple I frames. The GOP header does not specify the number of I, P, or B frames in the GOP, nor does it specify the structure of the GOP -- these are completely determined by the order of the data in the stream. Thus, there are no rules that restrict the size and structure of the GOP, although care should be taken to ensure that the buffer constraints are satisfied.

MPEG uses block motion-compensated prediction to reduce the temporal redundancies inherent to video. In block motion-compensated prediction, the current frame is divided into 16x16 pixel units called macroblocks. Each macroblock is compared to a number of 16x16 blocks in a previously coded frame. A single motion vector (MV) is used to represent this block with the best match. This block is used as a prediction of the current block, and only the error in the prediction, called the residual, is coded into the data stream.

The frames of a video sequence can be coded as an I, P, or B frame. In I frames, every macroblock must be coded in intraframe mode, i.e. without prediction. In P frames, each macroblock can be coded with forward prediction or in intraframe mode. In B frames, each macroblock can be coded with forward, backward, or bidirectional prediction or in intraframe mode. One MV is specified for each forward- and backward-predicted macroblock while two MVs are specified for each bidirectionally predicted macroblock. Thus, each P frame has a forward motion vector field and one anchor frame, while each B frame has a forward and backward motion vector field and two anchor frames. In some of the following sections, we define B_{for} and B_{back} frames as B frames that use only forward or only backwards prediction. Specifically, B_{for} frames can only have intra and forward-predicted macroblocks while B_{back} frames can only have intra and backward-predicted macroblocks.

MPEG uses discrete cosine transform (DCT) coding to code the intraframe and residual macroblocks. Specifically, four 8x8 block DCTs are used to encode each macroblock and the resulting DCT coefficients are quantized.

Quantization usually results in a sparse representation of the data, i.e. one in which most of the amplitudes of the quantized DCT coefficients are equal to zero. Then, only the amplitudes and locations of the nonzero coefficients are coded into the compressed data stream.

3.2 MPEG FIELD CODING

While many video compression algorithms, including MPEG-1, H.261, and H.263, are designed for progressive video sequences; MPEG-2 was designed to support both progressive and interlaced video sequences, where two fields, containing the even and odd scanlines, are contained in each frame. MPEG-2 provides a number of coding options to support interlaced video. First, each interlaced video frame can be coded as a frame picture in which the two fields are coded as a single unit or as a field picture in which the fields are coded sequentially. Next, MPEG-2 allows macroblocks to be coded in one of five motion compensation modes: frame prediction for frame pictures, field prediction for frame pictures, field prediction for field pictures, 16x8 prediction for field pictures, and dual prime motion compensation. The frame picture and field picture prediction dependencies are as follows. For frame pictures, the top and bottom reference fields are the top and bottom fields of the previous I or P frame. For field pictures, the top and bottom reference fields are the most recent top and bottom fields. For example, if the top field is specified to be first, then MVs from the top field can point to the top or bottom fields in the previous frame, while MVs from the bottom field can point to the top field of the current frame or the bottom field of the previous frame. Our discussion focuses on P-frame prediction because the transcoder described in Subsection 5.1.5 only processes the MPEG I and P frames. We also focus on field picture coding of interlaced video, and do not discuss dual prime motion compensation.

In MPEG field picture coding, each field is divided into 16x16 macroblocks, each of which can be coded with field prediction or 16x8 motion compensation. In field prediction, the 16x16 field macroblock will contain a field selection bit which indicates whether the prediction is based on the top or bottom reference field and a motion vector which points to the 16x16 region in the appropriate field. In 16x8 prediction, the 16x16 field macroblock is divided into its upper and lower halves, each of which contains 16x8 pixels. Each half has a field selection bit which specifies whether the top or bottom reference field is used and a motion vector which points to the 16x8 pixel region in the appropriate field.

3.3 MPEG BITSTREAM SYNTAX

The syntax of the MPEG-1 data stream has the following structure: A **Sequence header** consists of a *sequence start code* followed by *sequence parameters*. Sequences contain a number of GOPs. Each **GOP header** consists of a *GOP start code* followed by *GOP parameters*. GOPs contain a number of pictures. Each **picture header** consists of a *picture start code* followed by *picture parameters*. Pictures contain a number of slices. Each **slice header** consists of a *slice start code* followed by *slice parameters*. The slice header is followed by **slice data**, which contains the coded macroblocks.

The *sequence header* specifies the picture height, picture width, and sample aspect ratio. In addition, it sets the frame rate, bitrate, and buffer size for the

sequence. If the default quantizers are not used, then the quantizer matrices are also included in the sequence header. The *GOP header* specifies the time code and indicates whether the GOP is open or closed. A GOP is open or closed depending on whether or not the temporal prediction of its frames require data from other GOPs. The *picture header* specifies the temporal reference parameter, the picture type (I, P, or B), and the buffer fullness (via the vbv_delay parameter). If temporal prediction is used, it also describes the motion vector precision (full or half pixel) and the motion vector range. The *slice header* specifies the macroblock row in which slice starts and the initial quantizer scale factor for the DCT coefficients. The *macroblock header* specifies the relative position of the macroblock in relation to the previously coded macroblock. It contains a flag to indicate whether intra or inter-frame coding is used. If inter-frame coding is used, it contains the coded motion vectors, which may be differentially coded with respect to previous motion vectors. The quantizer scale factor may be adjusted at the macroblock level. One bit is used to specify whether the factor is adjusted. If it is, the new scale factor is specified. The macroblock header also specifies a coded block pattern for the macroblock. This describes which of the luminance and chrominance DCT blocks are coded. Finally, the DCT coefficients of the coded blocks are coded into the bitstream. The DC coefficient is coded first, followed by the runlengths and amplitudes of the remaining nonzero coefficients. If it is an intra macroblock, then the DC coefficient is coded differentially.

The sequence, GOP, picture, and slice headers begin with start codes, which are four-byte identifiers that begin with 23 zeros and a one followed by a one byte unique identifier. Start codes are useful because they can be found by examining the bitstream; this facilitates efficient random access into the compressed bitstream. For example, one could find the coded data that corresponds to the 2nd slice of the 2nd picture of the 22nd GOP by simply examining the coded data stream, without parsing and decoding the data. Of course, reconstructing the actual pixels of that slice may require parsing and decoding additional portions of the data stream because of the prediction used in conventional video coding algorithms. However, computational benefits could still be achieved by locating the beginning of the 22nd GOP and parsing and decoding the data from that point on thus exploiting the temporal refresh property inherent to GOPs.

3.4 COMPRESSED-DOMAIN PROCESSING FOR MPEG

The CDP problem statement was described in Section 2. In essence, the goal of CDP is to develop efficient algorithms for performing processing operations on compressed bitstreams. While the conventional approach requires decompressing the bitstream, processing the decoded frames, and re-encoding the result; improved efficiency, with respect to compute and memory requirements, can be achieved by exploiting structures used in the compression algorithms and using this knowledge to avoid the complete decode and re-encode cycle. In the context of MPEG transcoding, improved efficiency can be achieved by exploiting the structures used in MPEG coding. Furthermore, a decode/process/re-encode cycle can lead to significant loss of quality (even if *no processing* is performed besides the decode and re-encode) -- carefully designed CDP algorithms can greatly reduce and in some cases prevent this loss in quality.

MPEG coding uses a number of structures, and different compressed-domain processing operations require processing at different levels of depth. From highest to lowest level, these levels include:

- Sequence-level processing
- GOP-level processing
- Frame-level processing
- Slice-level processing
- Macroblock-level processing
- Block-level processing

Generally speaking, deeper-level operations require more computations. For example, some processing operations in the time domain require less computation if no information below the frame level needs to be adjusted. Operations of this kind include fast forward recoding and cut-and-paste or splicing operations restricted to cut points at GOP boundaries. However, if frame-accurate splicing [3] is required, frame and macroblock level information may need to be adjusted for frames around the splice point, as described in Section 5. In addition, in frame rate reduction transcoding, if the transcoder chooses to only drop non-reference frames such as B frames, a frame-level parsing operation could suffice.

On the other hand, operations related to the modification of content within video frames have to be performed below the frame level. Operations of this kind include spatial resolution reduction transcoding [4], frame-by-frame video reverse play [5] and many video-editing operations such as fading, logo insertion, and video/object overlaying [6][7]. As expected, these operations require significantly more computations, so for these operations efficient compressed-domain methods can lead to significant improvements.

4. COMPRESSED-DOMAIN PROCESSING METHODS

In this section, we examine the basic techniques of compressed-domain processing methods. Since the main techniques used in video compression include spatial to frequency transformation, particularly DCT, and motion-compensated prediction, we focus the investigation on compressed domain methods in these two domains, namely, in the DCT domain and the motion domain.

4.1 DCT-DOMAIN PROCESSING

As described in Section 3, the DCT is the transformation used most often in image and video compression standards. It is therefore important to understand some basic operations that can be performed directly in the DCT domain, i.e. without an inverse DCT/forward DCT cycle.

The earliest work on direct manipulation of compressed image and video data expectedly dealt with point processing, which consists of operations such as contrast manipulation and image subtraction where a pixel value in the output image at position p depends solely on the pixel value at the same position p in the input image. Examples of such work can be found in Chang and Messerschmitt [8], who developed some special functions for video compositing, and in Smith and Rowe [9], who developed a set of algorithms for basic point operations. When viewing compressed domain manipulation as a matrix operation, point processing operations on compressed images

and video can be characterized as inner-block algebra (IBA) operations since the information in the output block, i.e. the manipulated block, comes solely from information in the corresponding input block. These operations are listed in Table 1.

Table 1. Mathematical expression of spatial vs. DCT domain algebraic operations

	Spatial domain signal – x	Transform domain signal – X
Scalar addition	$[f]+\alpha$	$[F]+\begin{bmatrix} 8\alpha/Q_{00} & 0 \\ 0 & 0 \end{bmatrix}$
Scalar Multiplication	$\alpha[f]$	$\alpha[F]$
Pixel Addition	$[f]+[g]$	$[F]+[G]$
Pixel Multiplication	$[f]\bullet[g]$	$[F]\otimes[G]$

In this table, lower case *f* and *g* are used to represent spatial domain signals, while upper case *F* and *G* represent their corresponding DCT domain signals. Since compression standards typically use block-based schemes, each block can be treated as a matrix. Therefore, the operations can be expressed in forms of matrix operations. In general, the relationship holds as:
$DCT(x) = X$,
where *DCT*( ) represents the DCT function.

Because of the large number of zeros in the block in the DCT domain, the data manipulation rate is heavily reduced. The speedup of the first three operations in Table 1 is quite obvious given that the number of non-zero coefficients in *F* and *G* is quite small. As an example of these IBA operations, consider the compositing operation where foreground *f* is combined with background *b* with a factor of α to generate an output *R* in DCT representation. In spatial domain, this operation can be expressed as: $R = DCT(\alpha[f]+(1-\alpha)[b])$. Given the DCT representation of *f* and *b* in the compressed domain, *F* and *B*, the operation can be conveniently performed as: $R = \alpha[F]+(1-\alpha)[B]$. The operation is based on the linearity of the DCT and corresponds to a combination of some of the above-defined image algebra operations; it can be done in DCT domain efficiently with significant speedup. Similar compressed domain algorithms for subtitling and dissolving applications can also be developed based on the above IBA operations with computational speedups of 50 or more over the corresponding processing of the uncompressed data [9].

These methods can also be used for color transformation in the compressed domain. As long as the transformation is linear, it can be derived in the compressed domain using a combination of these IBA operations.

Pixel multiplication can be achieved by a convolution in the DCT domain. Compressed-domain convolution has been derived in [9] by mathematically combining the decompression, manipulation, and re-compression processes

to obtain a single equivalent local linear operation where one can easily take advantage of the energy compaction property in quantized DCT blocks. A similar approach was taken by Smith [10] to extend point processing to global processing of operations where the value of a pixel in the output image is an arbitrary linear combination of pixels in the input image. Shen et al. [11] have studied the theory behind DCT domain convolution based on the orthogonal property of DCT. As a result, an optimized DCT domain convolution algorithm is proposed and applied to the application of DCT domain alpha blending. Specifically, given foreground f to be blended with the background b with an alpha channel a to indicate the transparency of each pixel in f, the operation can be expressed as: $R = DCT([a] \bullet [f] + (1 - [a]) \bullet [b])$. The DCT domain operation is performed as: $R = [A] \otimes [F] + (1 - [A]) \otimes [B]$, where A is the DCT representation of a. A masking operation can also be performed in the same fashion with A representing the mask in the DCT domain. This operation enables the overlay of an object in the DCT domain with arbitrary shape. An important application for this is logo-insertion. Another example where processing of arbitrarily shaped objects arise is discussed in Section 6.1.

Many image manipulation operations are local or neighborhood operations where the pixel value at position p in the output image depends on neighboring pixels of p in the input image. We characterize methods to perform such operations in the compressed domain as inner-block rearrangement or resampling (IBR) methods. These methods are based on the fact that DCT is a unitary orthogonal transform and is distributive to matrix multiplication. It is also distributive to matrix addition, which is actually the case of pixel addition in Table 1. We group these two distributive properties of DCT in Table 2.

Table 2. Mathematical expression of distributiveness of DCT

	Spatial domain signal – x	Transform domain signal – X
Matrix Addition	$[f]+[g]$	$[F]+[G]$
Matrix Multiplication	$[f][g]$	$[F][G]$

Based on above, Chang and Messerschmitt [8] developed a set of algorithms to manipulate images directly in the compressed domain. Some of the interesting algorithms they developed include the translation of images by arbitrary amounts, linear filtering, and scaling. In general, a manipulation requiring uniform and integer scaling, i.e. certain forms of filtering, is easy to implement in the DCT domain using the resampling matrix. Since each block can use the same resampling matrix in space invariant filtering, these kinds of manipulations require little overhead in the DCT domain. In addition, translation of images by arbitrary amounts represents a shifting operation that is often used in video coding. We defer a detailed discussion of this particular method to Section 4.3.

Another set of algorithm has also been introduced to manipulate the orientation of DCT blocks [12]. These methods can be employed to flip-flop a DCT frame as well as rotate a DCT frame at multiples of 90 degree, simply by switching the location and/or signs of certain DCT coefficients in the DCT

blocks. For example, the DCT transform result of a transposed pixel block f is equivalent of the transpose of the corresponding DCT block. This operation is expressed mathematically as:

$$DCT([f]^t) = [F]^t .$$

A horizontal flip of a pixel block ($[f]^h$) can be achieved in the DCT domain by performing an element-by-element multiplication with a matrix composed of only two values: 1 or –1. The operation is therefore just sign reversal on some non-zero coefficients. Mathematically, this operation is expressed as:

$$DCT([f]^h) = [F] \bullet [H],$$

where H is defined as follows assuming an 8x8 block operation,

$$H_{ij} = \begin{cases} -1 & j = 1,3,5,7 \\ 1 & j = 0,2,4,6 \end{cases}.$$

For the full set of operations of this type, please refer to [12]. Note that for all the cases, the DC coefficient remains unchanged because of the fact that each pixel maintains its gray level while its location within the block is changed. These flip-flop and special angle rotation methods are very useful in applications such as image orientation manipulation that is used often in copy machines, printers and scanners.

4.2 MOTION VECTOR PROCESSING (MV RESAMPLING)

From a video coding perspective, motion vectors are estimated through block matching in a reference frame. This process is often compute intensive. The key of compressed-domain manipulation of motion vectors is to derive new motion vectors out of existing motion vector information contained in the input compressed bitstream.

Consider a motion vector processing scenario that arises in a spatial resolution reduction transcoder. Given the motion vectors for a group of four 16x16 macroblocks of the original video (NxM), how does one estimate the motion vectors for the 16x16 macroblocks in the downscaled video (e.g., N/2xM/2)? Consider forward-predicted macroblocks in a forward-predicted (P) frame, wherein each macroblock is associated with a motion vector and four 8x8 DCT blocks that represent the motion-compensated prediction residual information. The downscale-by-two operation requires four input macroblocks to form a single new output macroblock. In this case, it is necessary to estimate a single motion vector for the new macroblock from the motion vectors associated with the four input macroblocks.

The question asked above can be viewed as a motion vector resampling problem. Specifically, given a set of motion vectors MV in the input compressed bitstream, how does one compute the motion vectors MV* of the output compressed bitstream? Motion vector resampling algorithms can be classified into 5 classes as shown in Figure 2 [5]. The most accurate, but least efficient algorithm is Class V, in which one decompresses the original frames into their full pixel representation; and then one performs full search motion estimation on the decompressed frames. Since motion estimation is by far the most compute-intensive part of the transcoding operation, this is a very expensive solution. Simpler motion vector resampling algorithms are

given in classes I through IV in order of increasing computational complexity, where increased complexity typically results in more accurate motion vectors. Class I MV resampling algorithms calculate each output motion vector based on its corresponding input motion vector. Class II algorithms calculate each output motion vector based on a neighbourhood of input motion vectors. Class III algorithms also use a neighbourhood of input motion vectors, but also consider other parameters from the input bitstream such as quantization parameters and coding modes when processing them. Class IV algorithms use a neighbourhood of motion vectors and other input bitstream parameters, but also use the decompressed frames. For example, the input motion vectors may be used to narrow the search range used when estimating the output motion vectors. Finally, Class V corresponds to full search motion estimation on the decompressed frames.

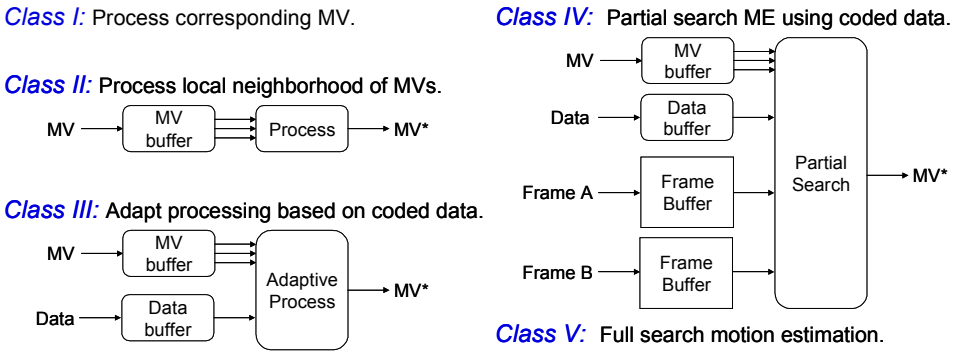


Figure 2. Classes of motion vector resampling methods.

The conventional spatial-domain approach of estimating the motion vectors for the downscaled video is to first decompress the video, downscale the video in the spatial domain then use one of the several widely known spatial-domain motion-estimation techniques (e.g., [13]) to recompute the motion vectors. This is computationally intensive. A class II approach might be to simply take the average of the four motion vectors associated with the four macroblocks and divide it by two so that the resulting motion vector can be associated with the 16x16 macroblock of the downscaled-by-two video. While this operation requires little processing, the motion vectors obtained in this manner are not optimal in most cases.

Adaptive motion vector resampling (AMVR) is a class III approach proposed in [4] to estimate the output motion vectors using the original motion information from the MPEG or H.26x bitstream of the original NxN video sequence. This method uses the DCT blocks to derive the block-activity information for the motion-vector estimation. When comparing the compressed-domain AMVR method to the conventional spatial-domain method, the results suggest that AMVR generates, with significantly less computation, motion vectors for the N/2xM/2 downscaled video that are very close to the optimal motion vector field that would be derived from an N/2xM/2 version of the original video sequence.

This weighted average motion vector scheme can also be extended to motion vector downsampling by arbitrary factors. In this operation, the number of participating macroblocks is not an integer. Therefore, the portion of the area

of the participating macroblock is used to weight the contributions of the existing motion vectors.

A class IV method for performing motion vector estimation out of existing motion vectors can be found in [14]. In frame rate reduction transcoding, if a P-picture is to be dropped, the motion vectors of macroblocks on the next P-picture should be adjusted since the reference frame is now different. Youn et al. [14] proposed a motion vector composition method to compute a motion vector from the incoming motion vectors. In this method, the derived motion vector can be refined by performing partial search motion estimation within a narrow search range.

The MV resampling problem for the compressed-domain reverse play operation was examined in [5]. In this application, the goal was to compute the "backward" motion vectors between two frames of a sequence when given the "forward" motion vectors between two frames. Perhaps contrary to intuition, the resulting forward and backwards motion vector fields are not simply inverted versions of one another because of the block-based motion-compensated processing used in typical compression algorithms. A variety of MV resampling algorithms is presented in [5], and experimental results are given that illustrate the tradeoffs in complexity and performance.

4.3 MC+DCT PROCESSING

The previous section introduced methodologies for deriving output motion vectors from existing input motion vectors in the compressed domain. However, if the newly derived motion vectors are used in conjunction with the original residual data, the result is imperfect and will result in a drift error. To avoid drift error, it is important to reconstruct the original reference frame and re-compute the residual data. This renders the IDCT process as the next computation bottleneck since the residual data is in the DCT domain. Alternatively, DCT domain motion compensation methods, such as the one introduced in [8], can be employed where the reference frames are converted to a DCT representation so that no IDCT is needed.

Inverse motion compensation is the process of extracting a 16x16 block given a motion vector in the reference frame. It can be characterised by a group matrix multiplications. Due to the distributive property of the DCT, this operation can be achieved by matrix multiplications of DCT blocks. Mathematically, consider a block g of size 8x8 in a reference frame pointed by a motion vector (x,y). Block g may lie in an area covered by a 2x2 array of blocks (f_1, f_2, f_3, f_4) in the reference frame. g can then be calculated as:

$$g = \sum_{i=1}^{4} m_{xi} f_i m_{yi} \text{ , where } 0 \le x, y \le 7 .$$

The *shifting* matrices m_{xi} and m_{yi} are defined as:

$$m_{xi} = \begin{cases} \begin{bmatrix} 0 & I_x \\ 0 & 0 \end{bmatrix} & i = 1,2 \\ \begin{bmatrix} 0 & I_{8-x} \\ 0 & 0 \end{bmatrix} & i = 3,4 \end{cases}, \text{ and } m_{yi} = \begin{cases} \begin{bmatrix} 0 & 0 \\ I_y & 0 \end{bmatrix} & i = 1,3 \\ \begin{bmatrix} 0 & 0 \\ I_{8-y} & 0 \end{bmatrix} & i = 2,4 \end{cases},$$

where I_z are identity matrices of size 8x8. In the DCT domain, this operation can be expressed as:

$$G = \sum_{i=1}^{4} M_{xi} F_i M_{yi} \ , \qquad (1)$$

where M_{xi} and M_{yi} are the DCT representations of m_{xi} and m_{yi} respectively. Since these shifting matrices are constant, they can be pre-computed and stored in the memory. However, the computing of Eq (1) may still be CPU-intensive since the shifting matrices may not be sparse enough. To this end, various authors have proposed different methods to combat this problem.

Merhav and Bhaskaran [15] proposed to decompose the DCT domain shifting matrices. Matrix decomposition methods are based on the sparseness of the factorized DCT transform matrices. Factorization of DCT transform matrix is introduced in a fast DCT algorithm [16]. The goal of the decomposition is to replace the matrix multiplication with a product of diagonal matrices, simple permutation matrices and more sparse matrices. The multiplication with a diagonal matrix can be absorbed in the quantization process. The multiplication with a permutation matrix can be performed by coefficient permutation. And finally, the multiplication with a sparse matrix requires fewer multiplications. Effectively, the matrix multiplication is achieved with less computation.

In an alternative approach, the coefficients in the shifting matrix can be approximated so that floating point multiplication can be replaced by integer shift and add operation. Work of this kind is introduced in [17]. Effectively, fewer basic CPU operations are needed since multiplication operations are avoided. A similar method is also used in [18] for DCT domain downsampling of images by employing the approximated downsampling matrices.

To further reduce the computation complexity of the DCT domain motion compensation process, a look-up-table (LUT) based method [19] is proposed by modelling the statistical distribution of DCT coefficients in compressed images and video sequences and precomputing all possible combinations of $M_{xi} F_i M_{yi}$ as in Eq (1). As a result, the matrix multiplications are reduced to simple table look-ups. Using around 800KB of memory, the LUT-based method can save more than 50% of computing time.

4.4 RATE CONTROL/BUFFER REQUIREMENTS

Rate control is another important issue in video coding. For compressed domain processing, the output of the process should also be confined to a certain bitrate so that it can be delivered in a constant transmission rate. Eleftheriadis and Anastassiou [20] have considered rate reduction by an optimal truncation or selection of DCT coefficients. Since fewer coefficients are coded, a lower number of bits are spent in coding them. Nakajima et al [21] achieve the similar rate reduction by re-quantization using a larger quantization step size.

For compressed domain processing, it is important for the rate control module to use compressed domain information existing in the original stream. This is a challenging problem, since the compressed bitstream lacks information that was available to the original encoder. To illustrate this

problem, consider TM5 rate control, which is used in many video coding standards. This rate controller begins by estimating the number of bits available to code the picture, and computes a reference value of the quantization parameter based on the buffer fullness and target bitrate. It then adaptly raises or lowers the quantization parameter for each macroblock based on the spatial activity of that macroblock. The spatial activity measure as defined in TM5 as the variance of each block:

$$V = \frac{1}{64}\sum_{i=1}^{64}(P_i - P_{mean})^2 \quad \text{, where } P_{mean} = \frac{1}{64}\sum_{i=1}^{64} P_i \text{ .}$$

However, the pixel domain information P_i may not be available in the compressed domain processing. In this case, the activity measure has to be derived from the DCT coefficients instead of the pixel domain frames. For example, the energy of quantized AC coefficients in the DCT block can be used as a measure of the variance. It has been shown in [4] that this approach achieves satisfactory rate control. In addition, the target bit budget for a particular frame can be derived from the bitrate reduction factor and the number of bits spent for the corresponding original frame, which is directly available from the original video stream.

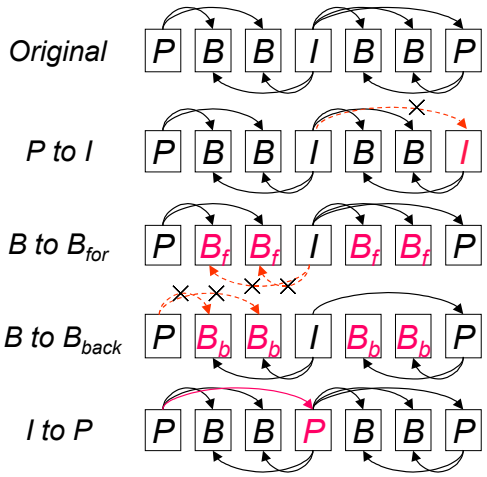
4.5 FRAME CONVERSIONS

Frame conversions are another basic tool that can be used in compressed-domain video processing operations [22]. They are especially useful in frame-level processing applications such as splicing and reverse play. Frame conversions are used to convert coded frames from one prediction mode to another to change the prediction dependencies in coded video streams. For example, an original video stream coded with I, P, and B frames may be temporarily converted to a stream coded with all I frames, i.e. a stream without temporal prediction, to facilitate pixel-level editing operations. Also, an IPB sequence may be converted to an IB sequence in which P frames are converted to I frames to facilitate random access into the stream. Also, when splicing two video sequences together, frame conversions can be used to remove prediction dependencies from video frames that are not included in the final spliced sequence. Furthermore, one may wish to use frame conversions to add prediction dependencies to a stream, for example to convert from an all I-frame compressed video stream to an I and P frame compressed stream to achieve a higher compression rate.

A number of frame conversion examples are shown in Figure 3. The original IPB sequence is shown in the top. Examples of frame conversions that remove temporal dependencies between frames are given: specifically P-to-I frame, B-to-B_{for} conversion, and B-to-B_{back} conversion. These operations are useful for editing operations such as splicing. Finally, an example of I-to-P conversion is shown in which prediction dependencies are added between frames. This is useful in applications that require further compression of a pre-compressed video stream.

Frame conversions require macroblock-level and block-level processing because they modify the motion vector and DCT coefficients of the compressed stream. Specifically, frame conversions require examining each macroblock of the compressed frame, and when necessary changing its coding mode to an appropriate dependency. Depending on the conversion, some, but not all, macroblocks may need to be processed. An example in

which a macroblock may not need to be processed is in a P-to-I frame conversion. Since P frames contain i- and p- type macroblocks and I frames contain only i-type macroblocks; a P-to-I conversion requires converting all p-type input macroblocks to i-type output macroblocks; however, note that i-type input macroblocks do not need to be converted. The list of frame types and allowed macroblock coding modes are shown in the upper right table in Figure 3. The lower right table shows macroblock conversions needed for some frame conversion operations. These conversions will be used in the splicing and reverse play applications described in Section 5. The conversion of a macroblock from p-type to i-type can be performed with the inverse motion compensation process introduced in Subsection 4.3.



Frame Type	MB Type
I	i
P	i, p
B	i, b_{forw}, b_{back}, b
B_{intra}	i
B_{forw}	i, b_{forw}
B_{back}	i, b_{back}

Frame Conversion	MB Conversion
P $\rightarrow$ I	p $\rightarrow$ i
B $\rightarrow$ B_{forw}	b $\rightarrow$ b_{forw} b_{back} $\rightarrow$ i
B $\rightarrow$ B_{back}	b $\rightarrow$ b_{back} b_{forw} $\rightarrow$ i
B $\rightarrow$ B_{intra}	b $\rightarrow$ i b_{back} $\rightarrow$ i b_{forw} $\rightarrow$ i

Figure 3. Frame conversions and required macroblock conversions.

One should note that in standards like MPEG, frame conversions performed on one frame may affect the prediction used in other frames because of the prediction rules specified by I, P, and B frames. Specifically, I-to-P and P-to-I frame conversions do not affect other coded frames. However, I-to-B, B-to-I, P-to-B, and B-to-P frame conversions do affect other coded frames. This can be understood by considering the prediction dependency rules of MPEG. Specifically, since P frames are specified to depend on the nearest preceding I or P frame and B frames are specified to depend on the nearest surrounding I or P frames, it is understandable that frame conversions of certain types will affect the prediction dependency tree inferred from the frame coding types.

5. APPLICATIONS

This section shows how the compressed domain processing methods described in Section 4 can be applied to video transcoding and video processing/editing applications. Algorithms and architectures are described for a number of CDP operations.

5.1 COMPRESSED-DOMAIN TRANSCODING APPLICATIONS

With the introduction of the next generation wireless networks, mobile devices will access an increasing amount of media-rich content. However, a mobile device may not have enough display space to render content that was originally created for desktop clients. Moreover, wireless networks typically support lower bandwidths than wired networks, and may not be able to carry media content made for higher-bandwidth wired networks. In these cases, transcoders can be used to transform multimedia content to an appropriate video format and bandwidth for wireless mobile streaming media systems.

A conceptually simple and straightforward method to perform this transcoding is to decode the original video stream, downsample the decoded frames to a smaller size, and re-encode the downsampled frames at a lower bitrate. However, a typical CCIR601 MPEG-2 video requires almost all the cycles of a 300Mhz CPU to perform real-time decoding. Encoding is significantly more complex and usually cannot be accomplished in real time without the help of dedicated hardware or a high-end PC. These factors render the conceptually simple and straightforward transcoding method impractical. Furthermore, this simple approach can lead to significant loss in video quality. In addition, if transcoding is provided as a network service in the path between the content provider and content consumer, it is highly desirable for the transcoding unit to handle as many concurrent sessions as possible. This scalability is critical to enable wireless networks to handle user requests that may be very intense at high load times. Therefore, it is very important to develop fast algorithms to reduce the compute and memory loads for transcoding sessions.

5.1.1 Compressed-Domain Transcoding Architectures

Video processing applications often involve a combination of spatial and temporal processing. For example, one may wish to downscale the spatial resolution and lower the frame rate of a video sequence. When these video processing applications are performed on compressed video streams, a number of additional requirements may arise. For example, in addition to performing the specified video processing task, the output compressed video stream may need to satisfy additional requirements such as maximum bitrate, buffer size, or particular compression format (e.g. MPEG-4 or H.263). While conventional approaches to applying traditional video processing operations on compressed video streams generally have high compute and memory requirements, the algorithmic optimizations described in Section 4 can be used to design efficient compressed-domain transcoding algorithms with significantly reduced compute and memory requirements. A number of transcoding architectures were discussed in [23][24][25][26].

Figure 4 shows a progression of architectures that reduce the compute and memory requirements of such applications. These architectures are discussed in the context of lowering the spatial and temporal resolution of the video from S_0,T_0 to S_1,T_1 and lowering the bitrate of the bitstream from R_0 to R_1. The top diagram shows the conventional approach to processing the compressed video stream. First the input compressed bitstream with bitrate R_0 is decoded into its decompressed video frames, which have a spatial resolution and temporal frame rate of S_0 and T_0. These frames are then processed temporally to a lower frame rate $T_1 < T_0$ by dropping appropriate

frames. The spatial resolution is then reduced to $S_1 < S_0$ by spatially downsampling the remaining frames. The resulting frames with resolution S_1,T_1 are then re-encoded into a compressed bitstream with a final bitrate of $R_1<R_0$. The memory requirements of this approach are high because of the frame stores required to store the decompressed video frames at resolution S_0,T_0. The computational requirements are high because of the operations needed to decode, process, and re-encode the frames; in particular, motion estimation performed during re-encoding can be quite compute intensive.

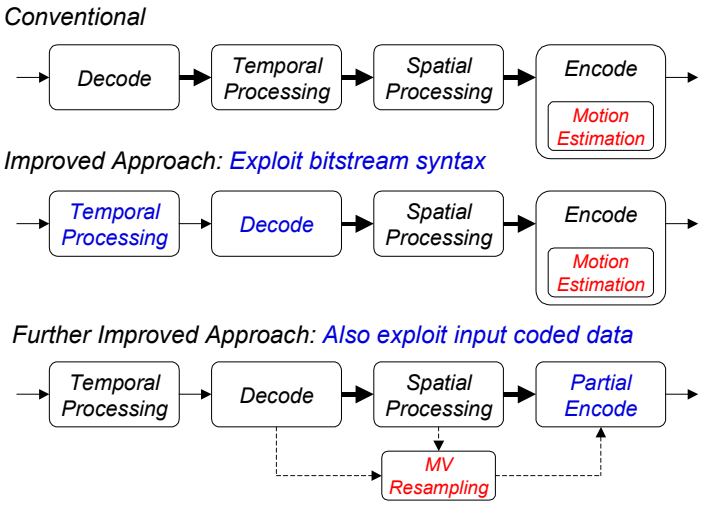


Figure 4. Architectural development of CDP algorithms.

The middle diagram shows an improved approach to the problem. By exploiting the picture start codes and frame prediction types used in the input compressed bitstream, the frame rate of the input bitstream can be reduced directly at the bitstream level prior to decompression. Specifically, in order to reduce the temporal frame rate, rather than decoding the entire bitstream and subsequently dropping frames, one may instead examine the bitstream for picture startcodes, determine the picture type from the picture header, and then selectively discard the bits that correspond to B pictures. The resulting lower-rate $R' < R_0$ bitstream can be decoded into video frames with resolution S_0,T_1. The limitation is that the temporal frame rate can only be reduced by restricted factors because of the prediction dependencies used in the input bitstream, e.g. in the case where two B frames are used between the I and P frames, the temporal frame rate can only be reduced by a factor of 3. The advantages are the reduced processing requirements needed for MPEG decoding and the reduced memory requirements achieved by eliminating the need to store the higher frame rate sequence. In this approach, the computational requirements are still high due to the motion estimation that must be performed in the encoder.

The bottom diagram shows an improved approach for this transcoding operation. Once again, the temporal frame rate is reduced at the bitstream layer by exploiting the picture start codes and picture headers. Furthermore, deriving the output coding parameters from those given in the input bitstream can significantly reduce the compute requirements of the final encode operation. This is advantageous because some of the computations that need to be performed in the encoder, such as motion

estimation, may have already been performed by the original encoder and may be represented by coding parameters, such as motion vectors, given in the input bitstream. Rather than blindly recomputing this information from the decoded, downsampled video frames, the encoder can exploit the information contained in the input bitstream. In other words, much of the information that is derived in the original encoder can be reused in the transcoder. Specifically, the motion vectors, quantization parameters, and prediction modes contained in the input compressed bitstream can be used to calculate the motion vectors, quantization parameters, and prediction modes used in the encoder, thus largely bypassing the expensive operations performed in the conventional encoder.

Also, when transcoding to reduce the spatial resolution, the number of macroblocks in the input and output frames can differ; the bottom architecture can be further improved to consider this difference and achieve a better tradeoff in complexity and quality [23]. Note that the DCT-domain methods discussed in Section 4 can be used for further improvements.

5.1.2 Intra-Frame Transcoding

Images and video frames coded with intraframe methods are represented by sets of block DCT coefficients. When using intraframe DCT coding, the original video frame is divided into 8x8 blocks, each of which is independently transformed with an 8x8 DCT. This imposes an artificial block structure that complicates a number of spatial processing operations, such as translation, downscaling, and filtering, that were considered straightforward in the pixel domain.

For spatial downsampling or resolution reduction on an intra-coded frame, one 8x8 DCT block of the downscaled image is determined from multiple 8x8 DCT blocks of the original image. Efficient downsampling algorithms can be derived in the DCT domain. Based on the distributed property of the DCT discussed in Subsection 4.1, DCT-domain downsampling can be achieved by matrix multiplication. Merhav and Bhaskaran [28] have developed an efficient matrix multiplication for downscale of DCT blocks. Natarajan and Bhaskaran [18] also used approximated DCT matrices to achieve the same goal. The approximated DCT matrices contain only elements of value 0, 1, or a power of ½. Effectively, the matrix multiplication can be achieved by integer shifts and additions, leading to a multiplication free implementation.

Efficient algorithms have also been developed for filtering images in the DCT domain. For example, [29] proposes a method to apply two-dimensional symmetric, separable filters to DCT-coded images.

5.1.3 Inter-Frame Transcoding

Video frames coded with interframe coding techniques are represented with motion vectors and residual DCT coefficients. These frames are coded based on a prediction from one or more previously coded frames; thus, properly decoding one frame requires first decoding one or more other frames. This temporal dependence among frames severely complicates a number of spatial and temporal processing techniques such as translation, downscaling, and splicing.

To facilitate efficient transcoding in the compressed domain, one wants to reuse as much information as possible in the origin video bitstream. The motion vector information of the transcoded video can be derived using the motion vector processing method introduced in Subsection 4.2. The computing of the residual DCT data can follow the guidelines provided in Subsection 4.3. Specifically, an interframe representation can be transcoded to an intraframe representation in the DCT domain. Subsequently, the DCT domain residual data can be obtained based on the derived motion vector information.

5.1.4 Format Conversion: Video Downscaling

Downscaling, or reducing the spatial resolution, of compressed video streams is an operation that benefits from the compressed-domain methods described in Section 4 and the compressed-domain transcoding architectures presented in Subsection 5.1.1. A block diagram of the compressed-domain downscaling algorithm is shown in Figure 5. The input bitstream is partially decoded into its motion vector and DCT domain representation. The motion vectors are resampled with the MV resampling methods described in Subsection 4.2. The DCT coefficients are processed with the DCT-domain processing techniques described in Subsections 4.1 and 4.3. A number of coding parameters from the input bitstream are extracted and used in the MV resampling and partial encoding steps of the transcoder. Rate control techniques, like those described in Section 4.4, are used to adapt the bitrate of the output stream. This is discussed in more detail below.

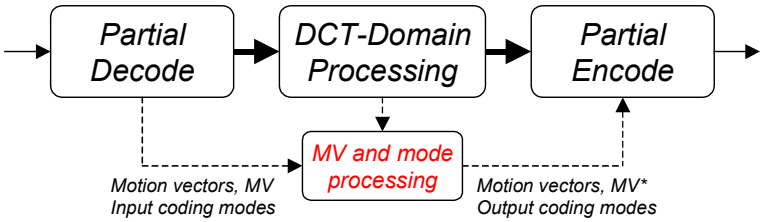


Figure 5. Compressed-domain downscaling algorithm.

The compressed-domain downscaling operation is complicated by the prediction dependencies used between frames during compression. Specifically, there are two tracks of dependencies in such a transcoding session. The first dependency is among frames in the original input video stream, while the second is among frames in the output downsampled video stream. The motion vectors for the down-sampled version can be estimated based on the motion vectors in the original video. However, even when the motion information in the original video is reused, it is necessary to reconstruct the reference frames to avoid drift error due to imperfect motion vector estimation. As described in Subsection 4.3, the reconstruction may be performed using a DCT domain motion compensation method.

The selection of coding type for macroblock in the interframes is also an important issue. In the downsampling-by-two case, there may be four macroblocks each with a different coding type involved in the creation of each output macroblock; the transcoder may choose the dominant coding type as the coding type for the output macroblock. In addition, rate control must be used to control the bitrate of the transcoding result.

5.1.5 Format Conversion: Field-to-Frame Transcoding

This section focuses on the problem of transcoding a field-coded compressed bitstream to a lower-rate, lower-resolution frame-coded compressed bitstream [26]. For example, conversions between interlaced MPEG-2 sequences to progressive MPEG-1, H.261, H.263, or MPEG-4 simple profile streams lie within this space. To simplify discussion, this section focuses on transcoding a given MPEG-2 bitstream to a lower-rate H.263 or MPEG-4 simple profile bitstream [26][30][31]. This is a practically important transcoding problem for converting MPEG-2 coded DVD and Digital TV video, which is often interlaced, to H.263 or MPEG-4 video for streaming over the Internet or over wireless links (e.g. 3G cellular) to PCs, PDAs and cell phones that usually have progressive displays. For brevity, we refer to the output format as H.263, however it can be H.261, H.263, MPEG-1, or MPEG-4.

The conventional approach to the problem is as follows. An MPEG bitstream is first decoded into its decompressed interlaced video frames. These high-resolution interlaced video frames are then downsampled to form a progressive video sequence with a lower spatial resolution and frame rate. This sequence is then re-encoded into a lower-rate H.263 bitstream. This conventional approach to transcoding is inefficient in its use of computational and memory resources. It is desirable to have computation- and memory-efficient algorithms that achieve MPEG-2 to H.263 transcoding with minimal loss in picture quality.

A number of issues arise when designing MPEG-2 to H.263 transcoding algorithms. While both standards are based on block motion compensation and the block DCT, there are many differences that must be addressed. A few of these differences are listed below:

- *Interlaced vs. progressive video format:* MPEG-2 allows interlaced video formats for applications including digital television and DVD. H.263 only supports progressive formats.
- *Number of I frames:* MPEG uses more frequent I frames to enable random access into compressed bitstreams. H.263 uses fewer I frames to achieve better compression.
- *Frame coding types:* MPEG allows pictures to be coded as I, P, or B frames. H.263 has some modes that allow pictures to be coded as I, P, or B frames; but has other modes that only allow pictures to be coded as I, P, or optionally PB frames. Traditional I, P, B frame coding allows any number of B frames to be included between a pair of I or P frames, while H.263 I, P, PB frame coding allows at most one.
- *Prediction modes:* In support of interlaced video formats, MPEG-2 allows field-based prediction, frame-based prediction, and 16x8 field-based prediction. H.263 only supports frame-based prediction but optionally allows an advanced prediction mode in which four motion vectors are allowed per macroblock.
- *Motion vector restrictions:* MPEG motion vectors must point inside the picture, while H.263 has an unrestricted motion vector mode that allows motion vectors to point outside the picture. The benefits of this mode can be significant, especially for lower-resolution sequences where the boundary macroblocks account for a larger percentage of the video.

A block diagram of the MPEG-2 to H.263 transcoder [26][30] is shown in Figure 6. The transcoder accepts an MPEG IPB bitstream as input. The bitstream is scanned for picture start codes and the picture headers are examined to determine the frame type. The bits corresponding to B frames are discarded, while the remaining bits are passed on to the MPEG IP decoder. The decoded frames are downsampled to the appropriate spatial resolution and then passed to the modified H.263 IP encoder.

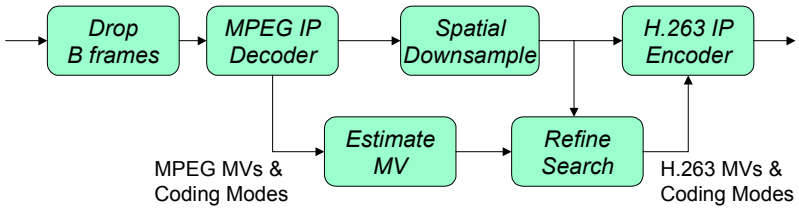


Figure 6. MPEG-2 to H.263 transcoder block diagram.

This encoder differs from a conventional H.263 encoder in that it does not perform conventional motion estimation; rather, it uses motion vectors and coding modes computed from the MPEG motion vectors and coding modes and the decoded, downsampled frames. There are a number of ways that this motion vector resampling can be done [4][5]. The class IV partial search method described in Subsection 4.2 was chosen. Specifically, the MPEG motion vectors and coding modes are used to form one or more initial estimates for each H.263 motion vector. A set of candidate motion vectors is generated; this set may include each initial estimate and its neighbouring vectors, where the size of the neighbourhood can vary depending on the available computational resources. The set of candidate motion vectors is tested on the decoded, downsampled frames and the best vector is chosen based on a criteria such as residual energy. A half-pixel refinement may be performed and the final mode decision (inter or intra) is then made.

Design considerations

Many degrees of freedom exist when designing an MPEG-2 to H.263 transcoder. For instance, a designer can make different choices in the mapping of input and output frame types; and the designer can choose how to vary the temporal frame rate and spatial resolution. Each of these decisions has different impact on the computational and memory requirements and performance of the final algorithm. This section presents a very simple algorithm that makes design choices that naturally match the characteristics of the input and output bitstreams.

The target format of the transcoder can be chosen based on the format of the input source bitstream. A careful choice of source and target formats can greatly reduce the computational and memory requirements of the transcoding operation.

Spatial and temporal resolutions: The chosen correspondence between the input and output coded video frames is shown in Figure 7. The horizontal and vertical spatial resolutions are reduced by factors of two because the MPEG-2 interlaced field format provides a natural factor of two reduction in the vertical spatial resolution. Thus, the spatial downsampling is performed by simply extracting the top field of the MPEG-2 interlaced video frame and horizontally downsampling it by a factor of two. This simple spatial

downsampling method allows the algorithm to avoid the difficulties associated with interlaced to progressive conversions. The temporal resolution is reduced by a factor of three, because MPEG-2 picture start codes, picture headers, and prediction rules make it possible to efficiently discard B-frame data from the bitstream without impacting the remaining I and P frames. Note that even though only the top fields of the MPEG I and P frames are used in the H.263 encoder, both the top and bottom fields must be decoded because of the prediction dependencies that result from the MPEG-2 interlaced field coding modes.

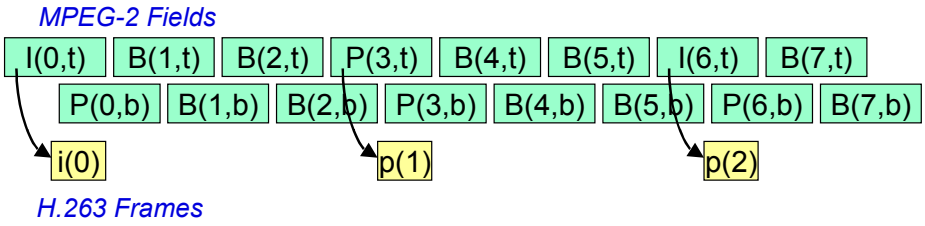


Figure 7. Video formats for MPEG-2 to H.263 tranascoding.

Frame coding types: MPEG-2 allows I, P, and B frames while H.263 allows I and P frames and optionally PB frames. With sufficient memory and computational capabilities, an algorithm can be designed to transcode from any input MPEG coding pattern to any output H.263 coding pattern as in [31]. Alternatively, one may take the simpler approach of determining the coding pattern of the target H.263 bitstream based on the coding pattern of the source MPEG-2 bitstream. By aligning the coding patterns of the input and output bitstreams and allowing temporal downsampling, a significant improvement in computational efficiency can be achieved.

Specifically, a natural alignment between the two standards can be obtained by dropping the MPEG B frames and converting the remaining MPEG I and P frames to H.263 I and P frames, thus exploiting the similar roles of P frames in the two standards and exploiting the ease in which B frame data can be discarded from an MPEG-2 bitstream without affecting the remaining I and P frames. Since MPEG-2 sequences typically use an IBBPBBPBB structure, dropping the B frames results in a factor of three reduction in frame rate. While H.263 allows an advanced coding mode of PB pictures, it is not used in this algorithm because it does not align well with MPEG's IBBPBBPBB structure.

The problem that remains is to convert the MPEG-coded interlaced I and P frames to the spatially downsampled H.263-coded progressive I and P frames. The problem of frame conversions can be thought of as manipulating prediction dependencies in the compressed data; this topic was addressed in [22] and in Subsection 4.5 for MPEG progressive frame conversions. This MPEG-2 to H.263 transcoding algorithm requires three types of frame conversions: (1) MPEG I field to H.263 I frame, (2) MPEG I field to H.263 P frame, and (3) MPEG P field to H.263 P frame. The first is straightforward. The latter two require the transcoder to efficiently calculate the H.263 motion vectors and coding modes from those given in the MPEG-2 bitstream. When using the partial search method described in Subsection 4.3, the first step is to create one or more initial estimates of each H.263 motion vector from the MPEG-2 motion vectors. In the following two

sections, we discuss the methods used to accomplish this for MPEG I field to H.263 P frame conversions and for MPEG P field to H.263 P frame conversions. Further details of the MPEG-2 to H.263 transcoder, including the progressive to interlace frame conversions, are given in [26][30]. These conversions address the differences between the MPEG-2 and H.263 standards described at the beginning of the section, and exploit the information in the input video stream to greatly reduce the computational and memory requirements of the transcoder with little loss in video quality.

5.2 EDITING

This section describes a series of compressed-domain editing applications. It begins with temporal mode conversion, which can be used to transcode an MPEG sequence into a format that facilitates video editing operations. It then describes two frame-level processing operations, frame-accurate splicing and frame-by-frame reverse play. All these operations use the frame conversion methods described in Subsection 4.5 to manipulate the prediction dependencies of compressed frames [22].

5.2.1 Temporal Mode Conversion

The ability to transcode between arbitrary temporal modes adds a great deal of flexibility and power to compressed-domain video processing. In addition, it provides a method of trading off parameters to achieve various rate/robustness profiles. For example, an MPEG sequence consisting of all I frames, while least efficient from a compression viewpoint, is most robust to channel impairments in a video communication system. In addition, the all I-frame MPEG video stream best facilitates many video-editing operations such as splicing, downscaling, and reverse play. Finally, once an I-frame representation is available, the intraframe transcoding algorithms described in Subsection 5.1 can be applied to each frame of the sequence to achieve the same effect on the entire sequence.

In general, temporal mode conversions can be performed with the frame conversion method described in Subsection 4.5. For frames that need to be converted to different prediction modes, macroblock and block level processing can be used to convert the appropriate macroblocks between different types.

The following steps describe a DCT-domain approach to transcoding an MPEG video stream containing I, P, and B frames into an MPEG video stream containing only I frames. This processing must be performed for the appropriate macroblocks of the converted frames.

1. *Calculate the DCT coefficients of the motion-compensated prediction.* This can be calculated from the intraframe coefficients of the previously coded frames by using the compressed-domain inverse motion compensation routine described in Subsection 4.3.
2. *Form the intraframe DCT representation of each frame.* This step simply involves adding the predicted DCT coefficients to the residual DCT coefficients.
3. *Requantize the intraframe DCT coefficients.* This step must be performed to ensure that the buffer constraints of the new stream are satisfied. Requantization may be used to control the rate of the new stream.

4. *Reorder the coded data and update the relevant header information.* If B-frames are used, the coding order of the IPB MPEG stream will differ from the coding order of the I-only MPEG stream. Thus, the coded data for each frame must be shuffled appropriately. In addition, the appropriate parameters of the header data must be updated.

5.2.1 Frame-Accurate Splicing

The goal of the splicing operation is to form a video data stream that contains the first N_{head} frames of one video sequence and the last N_{tail} frames of another video sequence. For uncoded video, the solution is obvious: simply discard the unused frames and concatenate the remaining data. Two properties make this solution obvious: (1) the data needed to represent each frame is self-contained, i.e. it is independent of the data from other frames; and (2) the uncoded video data has the desirable property of original ordering, i.e. the order of the video data corresponds to the display order of the video frames. MPEG-coded video data does not necessarily retain these properties of temporal independence or original ordering (although it can be forced to do so at the expense of compression efficiency). This complicates the task of splicing two MPEG-coded data streams.

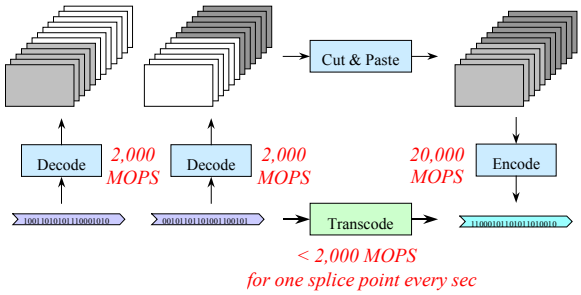


Figure 8. Splicing operation.

This section describes a flexible algorithm that splices two streams directly in the compressed domain [3]. The algorithm allows a natural tradeoff between computational complexity and compression efficiency, thus it can be tailored to the requirements of a particular system. This algorithm possesses a number of attributes. A minimal number of frames are decoded and processed, thus leading to low computational requirements while preserving compression efficiency. In addition, the head and tail data streams can be processed separately. Finally, if desired, the processing can be performed so that the final spliced data stream is a simple concatenation of the two streams and so that the order of the coded video data remains intact.

The conventional splicing solution is to completely decompress the video, splice the decoded video frames, and recompress the result. With this method, every frame in the spliced video sequence must be recompressed. This method has a number of disadvantages, including high computational requirements, high memory requirements, and low performance, since each recoding cycle can deteriorate the video data.

An improved compressed-domain splicing algorithm is shown in Figure 9. The computational requirements are reduced by only processing the frames affected by the splice, and by only decoding the frames needed for that processing. This is also shown in Figure 9. Specifically, the only frames that

need to be recoded are within in the GOPs affected by the head and tail cut points; at most, there will be one such GOP in the head data stream and one in the tail data stream. Furthermore, the only additional frames that need to be decoded are the I and P frames in the two GOPs affected by the splice.

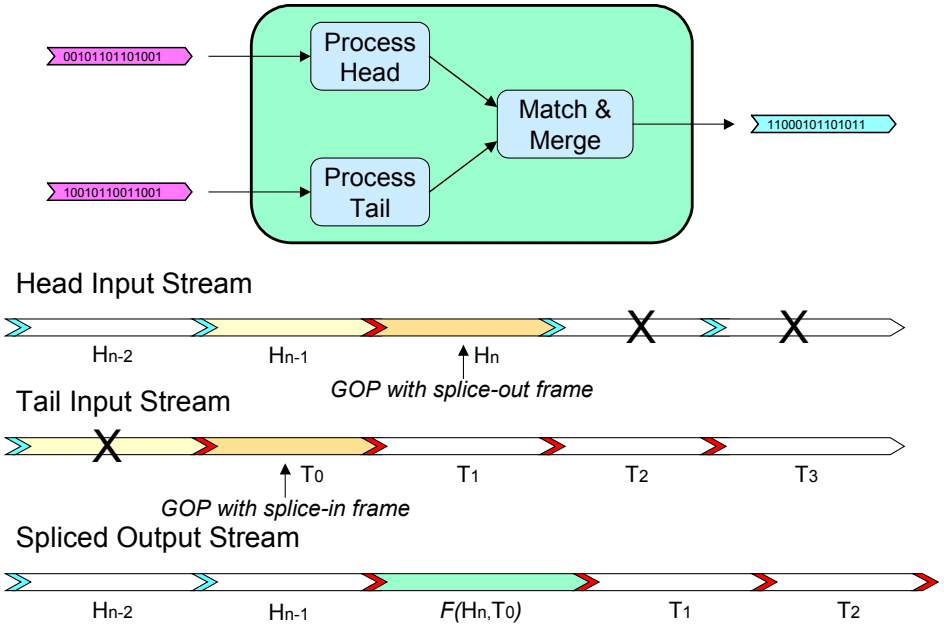


Figure 9. Compressed-domain splicing and processed bitstreams.

The algorithm results in an MPEG-compliant data stream with variable-sized GOPs. This exploits the fact that the GOP header does not specify the number of frames in the GOP or its structure; rather these are fully specified by the order of the data in the coded data stream.

Each step of the splicing operation is described below. Further discussion is included in [3].

1. Process the head data stream. This step involves removing any backward prediction dependencies on frames not included in the splice. The simplest case occurs when the cut for the head data occurs immediately after an I or P frame. When this occurs, there are no prediction dependencies on cut frames and all the relevant video data is contained in one contiguous portion of the data stream. The irrelevant portion of the data stream can simply be discarded, and the remaining relevant portion does not need to be processed. When the cut occurs immediately after a B frame, some extra processing is required because one or more B-frame predictions will be based on an anchor frame that is not included in the final spliced video sequence. In this case, the leading portion of the data stream is extracted up to the last I or P frame included in the splice, then the remaining B frames should be converted to B_{for} frames or P frames.

2. Form the tail data stream. This step involves removing any forward prediction dependencies on frames not included in the splice. The simplest case occurs when the cut occurs immediately before an I frame. When this occurs, the video data preceding this frame may be discarded and the remaining portion does not need to be processed. When the cut occurs

before a P frame, the P frame must be converted to an I frame and the remaining data remains in tact. When the cut occurs before a B frame, extra processing is required because one of the anchor frames is not included in the spliced sequence. In this case, if the first non-B frame is a P frame, it must be converted to an I frame. Then, each of the first consecutive B frames must be converted to B_{back} frames.

3. Match and merge the head and tail data streams. The IPB structure and the buffer parameters of the head and tail data streams determine the complexity of the matching operation. This step requires concatenating the two streams and then processing the frames near the splice point to ensure that the buffer constraints are satisfied. This requires *matching* the buffer parameters of the pictures surrounding the splice point. In the simplest case, a simple requantization will suffice. However, in more difficult cases, a frame conversion will also be required to prevent decoder buffer underflow. Furthermore, since prediction dependencies are inferred from the coding order of the compressed stream, when the merging step is performed the coded frames must be interleaved appropriately. The correct ordering will depend on the particular frame conversions used to remove the dependencies on cut frames.

The first two steps may require converting frames between the I, P, and B prediction modes. Converting P or B frames to I frames is quite straightforward as is B-to-B_{for} conversion and B-to-B_{back} conversion, however, conversion between any other set of prediction modes can require more computations to compute new motion vectors. Exact algorithms involve performing motion estimation on the decoded video -- this process can dominate the computational requirements of the algorithm. Approximate algorithms such as motion vector resampling can significantly reduce the computations required for these conversions.

Results of a spliced video sequence are shown in Figure 10. The right side of the figure plots the frame quality (in peak signal-to-noise ratio) for original compressed football and cheerleader sequences, and the spliced result when splicing between the two sequences every twenty frames. In the spliced result, the solid line contains the original quality values from the corresponding frames in the original coded football and cheerleader sequences, while the dotted line represents the quality of the sequence resulting from the compressed-domain splicing operation. Note that the spliced sequence has a slight degradation in quality at the splice points. This slight loss in quality is due to the removal of prediction dependencies in

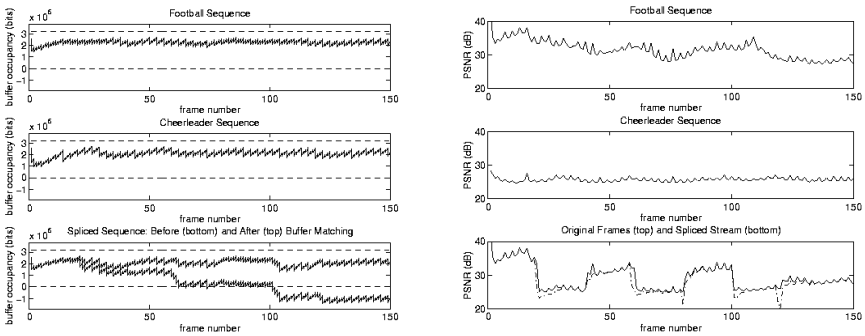


Figure 10. Performance of compressed-domain splicing algorithm.

the compressed video in conjunction with the rate matching needed to satisfy buffer requirements. However, note that it returns to full quality a few frames after the splice point (within one GOP). The plots on the left show the buffer occupancy for the original input sequences and the output spliced sequence. In the bottom plot, the bottom line shows the buffer usage if the rate matching operation is not performed; this results in an eventual decoder buffer underflow. The top line shows the result of the compressed-domain splicing algorithm with appropriate rate matching. In this case, the buffer occupancy levels stay consistent with the original streams except in small areas surrounding the splice points. However, as we saw in the quality plots, the quality and buffer occupancy levels match those of the input sequences within a few frames.

5.2.2 Frame-by-Frame Reverse Play

The goal of the compressed-domain reverse-play operation is to create a new MPEG data stream that, when decoded, displays the video frames in the reverse order from the original MPEG data stream. For uncoded video the solution is simple: reorder the video frame data in reverse order. The simplicity of this solution relies on two properties: the data for each video frame is self-contained and it is independent of its placement in the data stream. These properties typically do not hold true for MPEG-coded video data.

Compressed-domain reverse-play is difficult because MPEG compression is not invariant to changes in frame order, e.g. reversing the order of the input frames will not simply reverse the order of the output MPEG stream. Furthermore, reversing the order of the input video frames does not result in a ”reversed” motion vector field. However, if the processing is performed carefully, much of the motion vector information contained in the original MPEG video stream can be reused to save a significant amount of computations.

This section describes a reverse-play transcoding algorithm that operates directly on the compressed-domain data [32][33]. This algorithm is simple and achieves high performance with low computational and memory requirements. This algorithm only decodes the following data from the original MPEG data stream: I frames must be partially decompressed into their DCT representation and P frames must be partially decompressed to their MV/DCT representation, while for B frames only the forward and

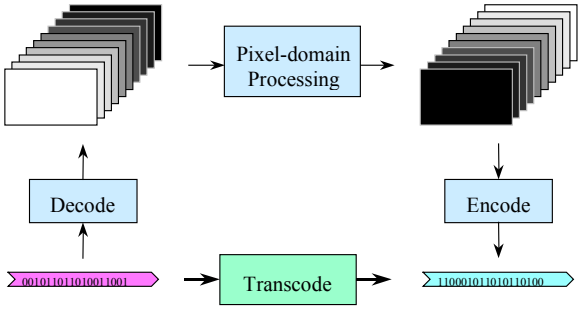


Figure 11. Reverse play operation.

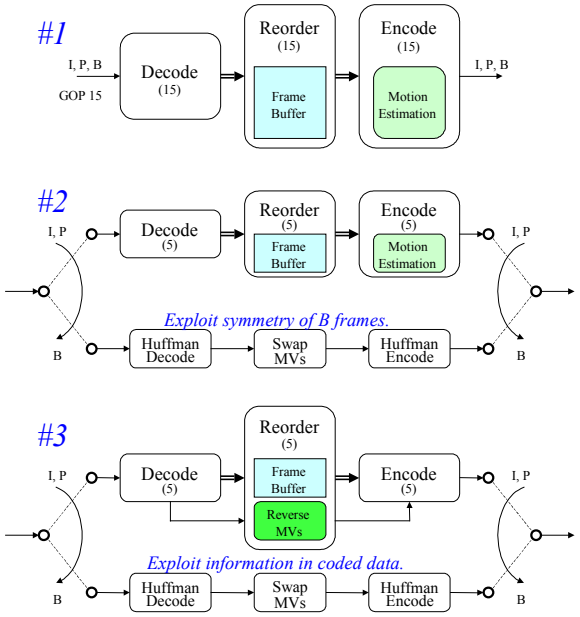


Figure 12. Architectures for compressed-domain reverse play.

backward motion vector fields need to be decoded, i.e. only bitstream processing is needed.

The development of the compressed-domain reverse play algorithm is shown in Figure 12. In the conventional approach shown in the top of the figure, each GOP in the MPEG stream, starting from the end of the sequence, is completely decoded into uncompressed frames and stored in a frame buffer. The uncompressed frames are reordered, and the resulting frames are re-encoded into an output MPEG stream that contains the original frames in reverse order.

The middle figure shows an improved approach to the algorithm. This improvement results from exploiting the symmetry of B frames. Specifically, it uses the fact that the coding of the reverse-ordered sequence can be performed so that the same frames are coded as B frames and thus will have the same surrounding anchor frames. The one difference will be that the forward and backward anchors will be reversed. In this case, major computational savings can be achieved by performing simplified processing on the B frames. Specifically, for B frames only a bitstream-level decoding is used to efficiently decode the motion vectors and coding modes, swap them between forward and backward modes, and repackage the results. This greatly reduces the computational requirements because 2/3 of the frames are B frames and because typically the processing required for B frames is greater than that required for P frames, which in turn is much greater than that required for I frames. Also, note that the frame buffer requirements are reduced by a factor of three because the B frames are not decoded.

The bottom figure shows a further improvement that can be had by using motion vector resampling, as described in Subsection 4.2, on the I and P frames. In this architecture, the motion vectors given in the input bitstream

are used to compute the motion vectors for the output bitstream, thereby avoiding the computationally expensive motion estimation process in the re-encoding process. The computational and performance tradeoffs of these architectures are discussed in detail in [5].

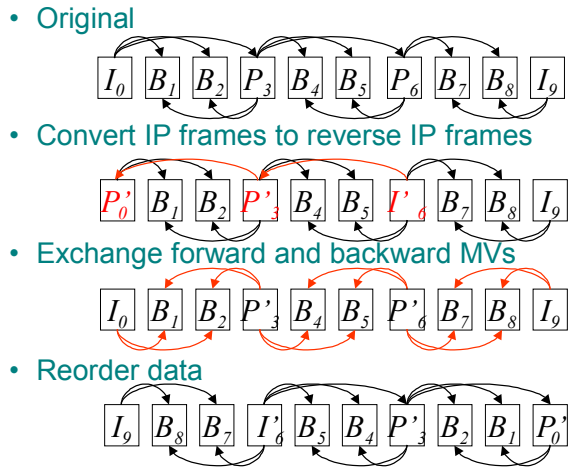


Figure 13. MPEG compressed-domain reverse play algorithm.

The resulting compressed-domain reverse-play algorithm shown in Figure 13 has the following steps:

1. *Convert the IP frames to reverse IP frames.* While the input motion vectors were originally computed for forward prediction between the I and P frames, the reverse IP frames require output motion vectors to be converted in the reverse order. Motion vector resampling methods described in Subsection 4.2 and in [5] can be used to calculate the new reversed motion vectors. Once the motion vectors are computed, the new output DCT coefficients can be computed directly in the DCT-domain by using the compressed-domain inverse motion compensation algorithm described in Subsection 4.3.
2. *Exchange the forward and backward motion vector fields used in each B frame.* This step exploits the symmetry of the B frame prediction process. In the reversed stream, the B frames will have the same two anchor frames, but in the reverse order. Thus, the forward prediction field can simply be exchanged with the backward prediction field, resulting in significant computational savings. Notice that only the motion vector fields need to be decoded for the B frames.
3. *Requantize the DCT coefficients.* This step must be performed to ensure that the buffer constraints of the new stream are satisfied. Requantization may be used to control the rate of the new stream.
4. *Properly reorder the frame data and update the relevant header information.* If no B frames are used, then the reordering process is quite straightforward. However, when B frames are used, care must be taken to properly reorder the data from the original coding order to the appropriate reverse coding order. In addition, the parameters in the header data must be updated appropriately.

6. ADVANCED TOPICS

6.1 OBJECT-BASED TO BLOCK-BASED TRANSCODING

This chapter focused on compressed-domain processing and transcoding algorithms for block-based compression schemes such as MPEG-1, MPEG-2, MPEG-4 simple profile, H.261, H.263, and H.264/MPEG-4 AVC. These compression standards represent each video frame as a rectangular array of pixels, and perform compression based on block-based processing, e.g. the block DCT and block-based motion estimation and motion compensated prediction. These compression algorithms are referred to as block- or frame-based schemes. Recently, object-based representations and compression algorithms have been developed -- the object-based coding part of MPEG-4 is the most well known example. These object-based representations decompose the image or video into arbitrarily shaped (non-rectangular) objects, unlike the block-based representations discussed above.

Object-based representations provide a more natural representation than square blocks, and can facilitate a number of new functionalities such as interactivity with objects in the video and greater content-creation flexibility. The object-based profiles of MPEG-4 are especially appealing for content creation and editing. For example, it may be useful to separately represent and encode different objects, such as different people or foreground or background objects, in a video scene in order to simplify manipulation of the scene. Therefore, object-based coding, such as MPEG-4, may become a natural approach to create, manipulate, and distribute new content. On the other hand, most clients may have block-based decoders, especially thin clients such as PDAs or cell phones. Therefore, it may become important to be able to efficiently transcode from object-based coding to block-based coding, e.g. from object-based MPEG-4 to block-based MPEG-2 or MPEG-4 simple profile. Efficient object-based to block-based transcoding algorithms were developed for intraframe (image) and interframe (video) compression in [7]. These efficient transcoding algorithms use many of the compressed-domain methods described in Section 4.

At each time instance (or frame), a video object has a shape, an amplitude (texture) within the shape, and a motion from frame to frame. In object-based coding, the shape (or support region) of the arbitrarily shaped object is often represented by a binary mask, and the texture of the object is represented by DCT transform coefficients. The object-based coding tools are often designed based on block-based coding tools. Typically in object-based image coding, such as in MPEG-4, a bounding box is placed around the object and the box is divided into blocks. The resulting blocks are classified as interior, boundary, or exterior blocks based on whether the block is completely within, partially within, or completely outside the object's support. For intraframe coding, a conventional block-DCT is applied to interior blocks and a modified block transform is applied to boundary blocks. For interframe coding, a macroblock and transform block structure similar to block-based video coding is used, where motion vectors are computed for macroblocks and conventional or modified block transforms are applied to interior and boundary blocks.

Many of the issues that arise in intraframe object-based to block-based transcoding algorithms can be understood by considering the simplified problem of overlaying an arbitrarily shaped object onto a fixed rectangular image, and producing the output compressed image that contains the rectangular image with the arbitrarily shaped overlay.

The simplest case occurs when the block boundaries of the fixed rectangular image and of the overlaid object are aligned. In this case, the output blocks can be computed in one of three cases. First, output image blocks that do not contain any portion of the overlay object may be simply copied from the corresponding block in the fixed rectangular image. Second, output image blocks that are completely covered by the overlaid object are replaced with the object's corresponding interior block. Finally, output image blocks that partially contain pixels from the rectangular image and the overlaid object are computed from the corresponding block from the fixed rectangular image and the corresponding boundary block from the overlaid object. Specifically, the new output coded block can be computed by properly masking the two blocks according to the object's segmentation mask. This can be computed in the spatial domain by inverse transforming the corresponding blocks in the background image and object, appropriately combining the two blocks with a spatial-domain masking operation, and transforming the result. Alternatively, it can be computed with compressed-domain masking operations, as described in Subsections 4.1, to reduce the computational requirements of the operation.

If the block boundaries of the object are not aligned with the block boundaries of the fixed rectangular image, then the affected blocks need additional processing. In this scenario, a shifting operation and a combined shifting/masking operation are needed for the unaligned block boundaries. Once again, output blocks that do not contain any portion of the overlaid object are copied from the corresponding input block in the rectangular image. Each remaining output block in the original image will overlap with 2 to 4 of the overlaid object's coded blocks (depending on whether one or both of the horizontal and vertical axes are misaligned). For image blocks with full coverage of the object and for which all the overlapping object's blocks are interior blocks, a shifting operation can be used to compute the new output "shifted" block. For the remaining blocks, a combined shifting/masking operation can be used to compute the new output block. As in the previous example, these computations can be performed in the spatial domain, or possibly more efficiently in the transform domain using the operations described in Subsections 4.1 and 4.3.

The object-to-block based interframe (video) transcoding algorithm share the issues that arise in the intraframe (image) transcoding algorithm with regard to the alignment of macroblock boundaries between the rectangular video and overlaid video object, or between multiple arbitrarily shaped video objects. Furthermore, a number of important problems arise because of the different prediction dependencies that exist for the multiple objects in the object-coded video and the desired single dependency tree for the block-based coded video. This requires significant manipulation of the temporal dependencies in the coded video. Briefly speaking, given multiple arbitrarily shaped objects described by shape parameters and motion and DCT

coefficients, the transcoding algorithm requires the computation of output block-based motion vectors and DCT coefficients. The solution presented computes output motion vectors with motion vector resampling techniques and computes output DCT coefficients with efficient transform-domain processing algorithms for combinations of the shifting, masking, and inverse motion compensation operations. Furthermore, the algorithm uses macroblock mode conversions, similar to those described in Subsection 4.5 and [22], to appropriately compensate for prediction dependencies that originally may have relied upon areas now covered by the overlaid object. The reader is referred to [7] for a detailed description of the transcoding algorithm.

6.2 SECURE SCALABLE STREAMING

It should now be obvious that transcoding is a useful capability in streaming media and media communication applications, because it allows intermediate network nodes to adapt compressed media streams for downstream client capabilities and time-varying network conditions. An additional issue that arises in some streaming media and media communication applications is security, in that an application may require the transported media stream to remain encrypted at all times. In applications where this type of security is required, the transcoding algorithms described earlier in this chapter can only be applied by decrypting the stream, transcoding the decrypted stream, and encrypting the result. By requiring decryption at transcoding nodes, this solution breaks the end-to-end security of the system.

Secure Scalable Streaming (SSS) is a solution that achieves the challenge of simultaneously enabling security and transcoding, specifically it enables transcoding without decryption [34][35]. SSS uses jointly designed scalable coding and progressive encryption techniques to encode and encrypt video into secure scalable packets that are transmitted across the network. The joint encoding and encryption is performed such that these resulting secure scalable packets can be transcoded at intermediate, possibly untrusted, network nodes by simply truncating or discarding packets and without compromising the end-to-end security of the system. The secure scalable packets may have unencrypted headers that provide hints, such as optimal truncation points, which the downstream transcoders use to achieve rate-distortion (R-D) optimal fine-grain transcoding across the encrypted packets.

The transcoding methods presented in this chapter are very powerful in that they can operate on most standard-compliant streams. However, in applications that require end-to-end security (where the transcoder is not allowed to see the bits), SSS can be used with certain types of scalable image and video compression algorithms to simultaneously provide security and scalability by enabling transcoding without decryption.

6.3 APPLICATIONS TO MOBILE STREAMING MEDIA SYSTEMS

The increased bandwidth of next-generation wireless systems will make streaming media a critical component of future wireless services. The network infrastructure will need to be able to handle the demands of mobility and streaming media, in a manner that scales to large numbers of users. Mobile streaming media (MSM) systems can be used to enable media delivery

over next-generation mobile networks. For example, a mobile streaming media content delivery network (MSM-CDN) can be used to efficiently distribute and deliver media content to large numbers of mobile users [36]. These MSM systems need to handle large numbers of compressed media streams; the CDP methods presented in this chapter can be used to do so in an efficient and scalable manner. For example, compressed-domain transcoding can be used to adapt media streams originally made for high-resolution display devices such as DVDs into media streams made for lower-resolution portable devices [26], and to adapt streams for different types of portable devices. Furthermore, transcoding can be used to adaptively stream content over error-prone, time-varying wireless links by adapting the error-resilience based on channel conditions [37]. When using transcoding sessions in mobile environments, a number of system-level technical challenges arise. For example, user mobility may cause a server handoff in an MSM-CDN, which in turn may require the midstream handoff of a transcoding session [38]. CDP is likely to play a critical and enabling role in next-generation MSM systems that require scalability and performance, and in many cases CDP will enable next-generation wireless, media-rich services.

Acknowledgment
The authors gratefully acknowledge Dr. Fred Kitson and Dr. Mark Smith of HP Labs for their support of this technical work throughout the years.

REFERENCES

[1] J. Mitchell, W. Pennebaker, C. Fogg, and D. LeGall, "MPEG Video Compression Standard", Digital Multimedia Standards Series, Chapman and Hall, 1997.

[2] V. Bhaskaran and K. Konstantinides, "Image and Video Compression Standards: Algorithms and Architectures", Kluwer Academic Publishers, Second Edition, June 1997.

[3] S. Wee and V. Bhaskaran, "Splicing MPEG video streams in the compressed-domain", *IEEE Workshop on Multimedia Signal Processing,* June 1997.

[4] B. Shen, I.K. Sethi, and V. Bhaskaran, “Adaptive Motion Vector Resampling for Compressed Video Down-scaling”, *IEEE Transactions on Circuits and Systems for Video Technology*, vol.9, no.6, pp.929-936, Sept. 1999.

[5] S. Wee, "Reversing motion vector fields", *IEEE International Conference on Image Processing,* Oct. 1998.

[6] B. Shen, I.K.Sethi and V. Bhaskaran, “Closed-Loop MPEG Video Rendering,” *International Conference on Multimedia Computing and Systems*, pp. 286-293, Ottawa, Canada, June1997.

[7] J.G. Apostolopoulos, “Transcoding between object-based and block-based image/video representations, e.g. MPEG-4 to MPEG-2 transcoding”, *HP Labs technical report,* May 1998.

[8] S.-F. Chang and D. Messerschmitt, "Manipulation and compositing of MC-DCT compressed video", IEEE Journal on Selected Areas in Communications, vol. 13, Jan. 1995.

[9] B.C. Smith and L. Rowe “Algorithms for Manipulating Compressed Images,” *IEEE Computer Graphics and Application*s, Sept. 1993.

[10] B.C. Smith, “Fast Software Processing of Motion JPEG Video,” in *Proc. of the Second ACM International Conference on Multimedi*a, ACM Press, pp. 77-88, San Francisco, Oct. 1994.

[11] B. Shen, I.K.Sethi and V.Bhaskaran, "DCT convolution and its applications in compressed video editing", *IEEE Trans. Circuits and Systems for Video Technology*, vol.8, no.8, pp.947-952, Dec. 1998.

[12] B. Shen, "Block Based Manipulations on Transform Compressed Images and Video", *Multimedia Systems Journal*, Vol. 6, No. 1, March 1998.

[13] Jaswant R. Jain and Anil K. Jain, "Displacement Measurement and Its Application in Interframe Image Coding," IEEE Trans. Communications, vol. com-29, no. 12, pp. 1799-1808, Dec. 1981.

[14] Jeongnam Youn, Ming-Ting Sun, Chia-Wen Lin, "Motion vector refinement for high-performance transcoding," *IEEE Transactions on Multimedia*, vol.1, no.1, pp. 30-40, March 1999.

[15] N. Merhav and V. Bhaskaran, "Fast algorithms for DCT-domain image downsampling and for inverse motion compensation", *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 7, June 1997.

[16] Y. Arai, T. Agui and M. Nakajima, "A Fast DCT-SQ Scheme for Images," *Trans. of The IEICE*, vol. E71, no. 11, Nov. 1988.

[17] P. A. Assuncao and M. Ghanbari, "A frequency-domain video transcoder for dynamic bit-rate reduction of MPEG-2 bit streams," *IEEE Trans. On Circuits and Systems for Video Technology*, vol. 8, no. 8, pp. 953-967, Dec. 1998.

[18] B. Natarajan and V. Bhaskaran, "A fast approximate algorithm for scaling down digital images in the DCT domain," *IEEE International. Conference On Image Processing*, Washington DC. Oct. 1995.

[19] S. Liu, A.C. Bovik, "Local Bandwidth Constrained Fast Inverse Motion Compensation for DCT-Domain Video Transcoding," IEEE Trans. On Circuits and Systems for Video Technology, vol. 12, no. 5, May 2002.

[20] A. Eleftheriadis and D. Anastassiou, "Constrained and general dynamic rate shaping of compressed digital video," *IEEE International Conference on Image Processing*, Washington, D.C., 1995.

[21] Y. Nakajima, H, Hori and T. Kanoh, " Rate conversion of MPEG coded video by re-quantization process," *IEEE International Conference on Image Processing,* Washington, D.C., 1995.

[22] S.J. Wee, "Manipulating temporal dependencies in compressed video data with applications to compressed-domain processing of MPEG video", *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Phoenix, Arizona, March 1999.

[23] H. Sun, W. Kwok, J. Zdepski, "Architectures for MPEG compressed bitstream scaling", *IEEE Transactions on Circuits Systems and Video Technology,* April 1996.

[24] G. Keesman, R. Hellinghuizen, F. Hoeksema, and G. Heideman, "Transcoding MPEG bitstreams", *Signal Processing: Image Communication*, September 1996.

[25] N. Bjork and C. Christopoulos, "Transcoder architectures for video coding", *IEEE International Conference on Image Processing*, May 1998.

[26] S.J. Wee, J.G. Apostolopoulos, and N. Feamster, "Field-to-Frame Transcoding with Temporal and Spatial Downsampling", *IEEE International Conference on Image Processing*, Kobe, Japan, October 1999.

[27] B. Shen and S. Roy, “A Very Fast Video Spatial Resolution Reduction Transcoder”, *International Conference On Acoustics, Speech, and Signal Processing*, May2002.

[28] N. Merhav and V. Bhaskaran, “A fast algorithm of DCT-domain image downscaling,” *International Conference On Acoustics, Speech, and Signal Processing*, Atlanta GA, May 1996.

[29] N. Merhav and R. Kresch, "Approximate convolution using DCT coefficient multipliers", *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 8, Aug. 1998.

[30] N. Feamster and S. Wee, “An MPEG-2 to H.263 transcoder”, *SPIE Voice, Video, and Data Communications Conference*, September 1999.

[31] T. Shanableh and M. Ghanbari, “Heterogeneous video transcoding MPEG:1,2 to H.263”, *Packet Video Workshop,* April 1999.

[32] S.J. Wee and B. Vasudev, “Compressed-Domain Reverse Play of MPEG Video Streams”, *SPIE Voice, Video, and Data Communications Conference*, Boston, MA, November 1998.

[33] M.-S. Chen and D. Kandlur, "Downloading and stream conversion: supporting interactive playout of videos in a client station", *International Conference on Multimedia Computing,* May 1995.

[34] S.J. Wee and J.G. Apostolopoulos, “Secure Scalable Video Streaming for Wireless Networks”, *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Salt Lake City, Utah, May 2001.

[35] S.J. Wee and J.G. Apostolopoulos, “Secure Scalable Streaming Enabling Transcoding without Decryption”, *IEEE International Conference on Image Processing,* Thessaloniki, Greece, October 2001.

[36] T. Yoshimura, Y. Yonemoto, T. Ohya, M. Etoh, S. Wee, “Mobile Streaming Media CDN enabled by Dynamic SMIL”, *Eleventh International World Wide Web Conference,* May 2002.

[37] G. De Los Reyes, A.R. Reibman, S.-F. Chang, J.C.-I Chuang, “Error-resilient transcoding for video over wireless channels”, *IEEE Journal on Selected Areas in Communications*, June 2000.

[38] S. Roy, B. Shen, V. Sundaram, R. Kumar, “Application Level Hand-Off Support for Mobile Media Transcoding Sessions”, *Workshop on Network and Operating System Support for Digital Audio and Video,* Miami Beach, Florida, May 2002.

INDEX