Introduction to Video Compression

A. Shoham and J. Bier
Berkeley Design Technology, Inc. (BDTI)
+1 510-665-1600
www.BDTI.com

1. Introduction

Digital video compression/decompression algorithms (codecs) are at the heart of many modern video products, from DVD players to digital video recorders, multimedia jukeboxes, and video-capable cell phones. Understanding the operation of video compression algorithms is essential for developers of embedded systems, processors, and tools targeting video applications. For example, understanding video codecs' processing and memory demands is key to processor selection and software optimization. In this paper, we explain the operation and characteristics of video codecs and the demands codecs make on processors. We also explain how codecs differ from one another and the significance of these differences.

2. Still-Image Compression

Video clips are made up of sequences of individual images, or "frames." Therefore, video compression algorithms share many concepts and techniques with still-image compression algorithms, such as JPEG. In fact, one way to compress video is to ignore the similarities between consecutive video frames, and simply compress each frame independently of other frames. For example, some products employ this approach to compress video streams using the JPEG still-image compression standard. This approach, known as "motion JPEG" or MJPEG, is sometimes used in video production applications. Although modern video compression algorithms go beyond still-image compression schemes and take advantage of the correlation between consecutive video frames using motion estimation and motion compensation, these more advanced algorithms also employ techniques used in still-image compression algorithms. Therefore, we begin our exploration of video compression by discussing the inner workings of transform-based still-image compression algorithms such as JPEG.

2.1 Basic Building Blocks of Digital Image Compression

2.1.1 Block Transform

The image compression techniques used in JPEG and in most video compression algorithms are "lossy." That is, the original uncompressed image can't be perfectly reconstructed from the compressed data, so some information from the original image is lost. Lossy compression algorithms attempt to ensure that the differences between the original uncompressed image and the reconstructed image are not perceptible to the human eye.

The first step in JPEG and similar image compression algorithms is to divide the image into small blocks and transform each block into a frequency-domain representation. Typically, this step uses a discrete cosine transform (DCT) on blocks that are eight pixels wide by eight pixels high. Thus, the DCT operates on 64 input pixels and yields 64 frequency-domain coefficients. The DCT itself preserves all of the information in the eight-by-eight image block. That is, an inverse DCT (IDCT) can be used to perfectly reconstruct the original 64 pixels from the DCT coefficients. However, the human eye is more sensitive to the information contained in DCT coefficients that represent low frequencies (corresponding to large features in the image) than to the information contained in DCT coefficients that represent high frequencies (corresponding to small features). Therefore, the DCT helps separate the more perceptually significant information from less perceptually significant information. Later steps in the compression algorithm encode the low-frequency DCT coefficients with high precision, but use fewer or no bits to encode the high-frequency coefficients, thus discarding information that is less perceptually significant. In the decoding algorithm, an IDCT transforms the imperfectly coded coefficients back into an 8x8 block of pixels.

The computations performed in the IDCT are nearly identical to those performed in the DCT, so these two functions have very similar processing requirements. A single two-dimensional eight-by-eight DCT or IDCT requires a few hundred instruction cycles on a typical DSP. However, video compression algorithms must often perform a vast number of DCTs and/or IDCTs per second. For example, an MPEG-4 video decoder operating at CIF (352x288) resolution and a frame rate of 30 fps may need to perform as many as 71,280 IDCTs per second, depending on the video content. The IDCT function would require over 40 MHz on a Texas Instruments TMS320C55x DSP processor (without the DCT accelerator) under these conditions. IDCT computation can take up as much as 30% of the cycles spent in a video decoder implementation.

Because the DCT and IDCT operate on small image blocks, the memory requirements of these functions are rather small and are typically negligible compared to the size of frame buffers and other data in image and video compression applications. The high computational demand and small memory footprint of the DCT and IDCT functions make them ideal candidates for implementation using dedicated hardware coprocessors.

2.1.2 Quantization

As mentioned above, the DCT coefficients for each eight-pixel by eight-pixel block are encoded using more bits for the more perceptually significant low-frequency DCT coefficients and fewer bits for the less significant high-frequency coefficients. This encoding of coefficients is accomplished in two steps: First, quantization is used to discard perceptually insignificant information. Next, statistical methods are used to encode the remaining information using as few bits as possible.

Quantization rounds each DCT coefficient to the nearest of a number of predetermined values. For example, if the DCT coefficient is a real number between -1 and 1, then scaling the coefficient by 20 and rounding to the nearest integer quantizes the coefficient to the nearest of 41 steps, represented by the integers from -20 to +20. Ideally, for each DCT coefficient a scaling factor is chosen so that information contained in the digits to the right of the decimal point of the scaled coefficient may be discarded without introducing perceptible artifacts to the image.

In the image decoder, dequantization performs the inverse of the scaling applied in the encoder. In the example above, the quantized DCT coefficient would be scaled by 1/20, resulting in a dequantized value between -1 and 1. Note that the dequantized coefficients are not equal to the original coefficients, but are close enough so that after the IDCT is applied, the resulting image contains few or no visible artifacts.

Dequantization can require anywhere from about 3% up to about 15% of the processor cycles spent in a video decoding application. Like the DCT and IDCT, the memory requirements of quantization and dequantization are typically negligible.

2.1.3 Coding

The next step in the compression process is to encode the quantized DCT coefficients in the digital bit stream using as few bits as possible. The number of bits used for encoding the quantized DCT coefficients can be minimized by taking advantage of some statistical properties of the quantized coefficients.

After quantization, many of the DCT coefficients have a value of zero. In fact, this is often true for the vast majority of high-frequency DCT coefficients. A technique called "run-length coding" takes advantage of this fact by grouping consecutive zero-valued coefficients (a "run") and encoding the number of coefficients (the "length") instead of encoding the individual zero-valued coefficients.

To maximize the benefit of run-length coding, low-frequency DCT coefficients are encoded first, followed by higher-frequency coefficients, so that the average number of consecutive zero-valued coefficients is as high as possible. This is accomplished by scanning the eight-by-eight-coefficient matrix in a diagonal zig-zag pattern.

Run-length coding is typically followed by variable-length coding (VLC). In variable-length coding, each possible value of an item of data (i.e., each possible run length or each possible value of a quantized DCT coefficient) is called a symbol. Commonly occurring symbols are represented using code words that contain only a few bits, while less common symbols are represented with longer code words. VLC uses fewer bits for the most common symbols compared to fixed-length codes (e.g. directly encoding the quantized DCT coefficients as binary integers) so that on average, VLC requires fewer bits to encode the entire image. Huffman coding is a variable-length coding scheme that optimizes the number of bits in each code word based on the frequency of occurrence of each symbol.

Note that theoretically, VLC is not the most efficient way to encode a sequence of symbols. A technique called "arithmetic coding" can encode a sequence of symbols using fewer bits than VLC. Arithmetic coding is more efficient because it encodes the entire sequence of symbols together, instead of using individual code words whose lengths must each be an integer number of bits. Arithmetic coding is more computationally demanding than VLC and has only recently begun to make its way into commercially available video compression algorithms. Historically, the combination of run-length coding and VLC has provided sufficient coding efficiency with much lower computational requirements than arithmetic coding, so VLC is the coding method used in the vast majority of video compression algorithms available today.

Variable-length coding is implemented by retrieving code words and their lengths from lookup tables, and appending the code word bits to the output bit stream. The corresponding variable-length decoding process (VLD) is much more computationally intensive. Compared to performing a table lookup per symbol in the encoder, the most straightforward implementation of VLD requires a table lookup and some simple decision making to be applied for each bit. VLD requires an average of about 11 operations per input bit. Thus, the processing requirements of VLD are proportional to the video compression codec's selected bit rate. Note that for low image resolutions and frame rates, VLD can sometimes consume as much as 25% of the cycles spent in a video decoder implementation.

In a typical video decoder, the straightforward VLD implementation described above requires several kilobytes of lookup table memory. VLD performance can be greatly improved by operating on multiple bits at a time. However, such optimizations require the use of much larger lookup tables.

One drawback of variable-length codes is that a bit error in the middle of an encoded image or video frame can prevent the decoder from correctly reconstructing the portion of the image that is encoded after the corrupted bit. Upon detection of an error, the decoder can no longer determine the start of the next variable-length code word in the bit stream, because the correct length of the corrupted code word is not known. Thus, the decoder cannot continue decoding the image. One technique that video compression algorithms use to mitigate this problem is to intersperse "resynchronization markers" throughout the encoded bit stream. Resynchronization markers occur at predefined points in the bit stream and provide a known bit pattern that the video decoder can detect. In the event of an error, the decoder is able to search for the next resynchronization marker following the error, and continue decoding the portion of the frame that follows the resynchronization marker.

In addition, the MPEG-4 video compression standard employs "reversible" variable-length codes. Reversible variable-length codes use code words carefully chosen so that they can be uniquely decoded both in the normal forward direction, and also backwards. In the event of an error, the use of reversible codes allows the video decoder to find the resynchronization marker following the error and decode the bit stream in the backward direction from the resynchronization marker toward the error. Thus, the decoder can recover more of the image data than would be possible with resynchronization markers alone.

All of the techniques described so far operate on each eight-pixel by eight-pixel block independently from any other block. Since images typically contain features that are much larger than an eight-by-eight block, more efficient compression can be achieved by taking into account the correlation between adjacent blocks in the image.

To take advantage of inter-block correlation, a prediction step is often added prior to quantization of the DCT coefficients. In this step, the encoder attempts to predict the values of some of the DCT coefficients in each block based on the DCT coefficients of the surrounding blocks. Instead of quantizing and encoding the DCT coefficients directly, the encoder computes, quantizes, and encodes the difference between the actual DCT coefficients and the predicted values of those coefficients. Because the difference between the predicted and actual values of the DCT coefficients tends to be small, this technique reduces the number of bits needed for the DCT coefficients. The

decoder performs the same prediction as the encoder, and adds the differences encoded in the compressed bit stream to the predicted coefficients in order to reconstruct the actual DCT coefficient values. Note that in predicting the DCT coefficient values of a particular block, the decoder has access only to the DCT coefficients of surrounding blocks that have already been decoded. Therefore, the encoder must predict the DCT coefficients of each block based only on the coefficients of previously encoded surrounding blocks.

In the simplest case, only the first DCT coefficient of each block is predicted. This coefficient, called the "DC coefficient," is the lowest frequency DCT coefficient and equals the average of all the pixels in the block. All other coefficients are called "AC coefficients." The simplest way to predict the DC coefficient of an image block is to simply assume that it is equal to the DC coefficient of the adjacent block to the left of the current block. This adjacent block is typically the previously encoded block. In the simplest case, therefore, taking advantage of some of the correlation between image blocks amounts to encoding the difference between the DC coefficient of the current block and the DC coefficient of the previously encoded block instead of encoding the DC coefficient values directly. This practice is referred to as "differential coding of DC coefficients" and is used in the JPEG image compression algorithm.

More sophisticated prediction schemes attempt to predict the first DCT coefficient in each row and each column of the eight-by-eight block. Such schemes are referred to as "AC-DC prediction" and often use more sophisticated prediction methods compared to the differential coding method described above: First, a simple filter may be used to predict each coefficient value instead of assuming that the coefficient is equal to the corresponding coefficient from an adjacent block. Second, the prediction may consider the coefficient values from more than one adjacent block. The prediction can be based on the combined data from several adjacent blocks. Alternatively, the encoder can evaluate all of the previously encoded adjacent blocks and select the one that yields the best predictions on average. In the latter case, the encoder must specify in the bit stream which adjacent block was selected for prediction so that the decoder can perform the same prediction to correctly reconstruct the DCT coefficients.

AC-DC prediction can take a substantial number of processor cycles when decoding a single image. However, AC-DC prediction cannot be used in conjunction with motion compensation. Therefore, in video compression applications AC-DC prediction is only used a small fraction of the time and usually has a negligible impact on processor load. However, some implementations of AC-DC prediction use large arrays of data. Often it may be possible to overlap these arrays with other memory structures that are not in use during AC-DC prediction to dramatically optimize the video decoder's memory use.

2.2 A Note About Color

Color images are typically represented using several "color planes." For example, an RGB color image contains a red color plane, a green color plane, and a blue color plane. Each plane contains an entire image in a single color (red, green, or blue, respectively). When overlaid and mixed, the three planes make up the full color image. To compress a color image, the still-image compression techniques described here are applied to each color plane in turn.

Video applications often use a color scheme in which the color planes do not correspond to specific colors. Instead, one color plane contains luminance information (the overall brightness of each pixel in the color image) and two more color planes contain color (chrominance) information that when combined with luminance can be used to derive the specific levels of the red, green, and blue components of each image pixel.

Such a color scheme is convenient because the human eye is more sensitive to luminance than to color, so the chrominance planes are often stored and encoded at a lower image resolution than the luminance information. Specifically, video compression algorithms typically encode the chrominance planes with half the horizontal resolution and half the vertical resolution as the luminance plane. Thus, for every 16-pixel by 16-pixel region in the luminance plane, each chrominance plane contains one eight-pixel by eight-pixel block. In typical video compression algorithms, a "macro block" is a 16-pixel by 16-pixel region in the video frame that contains four eight-by-eight luminance blocks and the two corresponding eight-by-eight chrominance blocks. Macro blocks allow motion estimation and compensation, described below, to be used in conjunction with sub-sampling of the chrominance planes as described above.

3. Adding Motion to the Mix

Using the techniques described above, still-image compression algorithms such as JPEG can achieve good image quality at a compression ratio of about 10:1. The most advanced still-image coders may achieve good image quality at compression ratios as high as 30:1. Video compression algorithms, however, employ motion estimation and compensation to take advantage of the similarities between consecutive video frames. This allows video compression algorithms to achieve good video quality at compression ratios up to 200:1.

In some video scenes, such as a news program, little motion occurs. In this case, the majority of the eight-pixel by eight-pixel blocks in each video frame are identical or nearly identical to the corresponding blocks in the previous frame. A compression algorithm can take advantage of this fact by computing the difference between the two frames, and using the still-image compression techniques described above to encode this difference. Because the difference is small for most of the image blocks, it can be encoded with many fewer bits than would be required to encode each frame independently. If the camera pans or large objects in the scene move, however, then each block no longer corresponds to the same block in the previous frame. Instead, each block is similar to an eight-pixel by eight-pixel region in the previous frame that is offset from the block's location by a distance that corresponds to the motion in the image. Note that each video frame is typically composed of a luminance plane and two chrominance planes as described above. Obviously, the motion in each of the three planes is the same. To take advantage of this fact despite the different resolutions of the luminance and chrominance planes, motion is analyzed in terms of macro blocks rather than working with individual eight-by-eight blocks in each of the three planes.

3.1 Motion Estimation and Compensation

Motion estimation attempts to find a region in a previously encoded frame (called a "reference frame") that closely matches each macro block in the current frame. For each macro block, motion estimation results in a "motion vector." The motion vector is

comprised of the horizontal and vertical offsets from the location of the macro block in the current frame to the location in the reference frame of the selected 16-pixel by 16-pixel region. The video encoder typically uses VLC to encode the motion vector in the video bit stream. The selected 16-pixel by 16-pixel region is used as a prediction of the pixels in the current macro block, and the difference between the macro block and the selected region (the "prediction error") is computed and encoded using the still-image compression techniques described above. Most video compression standards allow this prediction to be bypassed if the encoder fails to find a good enough match for the macro block. In this case, the macro block itself is encoded instead of the prediction error.

Note that the reference frame isn't always the previously displayed frame in the sequence of video frames. Video compression algorithms commonly encode frames in a different order from the order in which they are displayed. The encoder may skip several frames ahead and encode a future video frame, then skip backward and encode the next frame in the display sequence. This is done so that motion estimation can be performed backward in time, using the encoded future frame as a reference frame. Video compression algorithms also commonly allow the use of two reference frames—one previously displayed frame and one previously encoded future frame. This allows the encoder to select a 16-pixel by 16-pixel region from either reference frame, or to predict a macro block by interpolating between a 16-pixel by 16-pixel region in the previously displayed frame and a 16-pixel by 16-pixel region in the future frame.

One drawback of relying on previously encoded frames for correct decoding of each new frame is that errors in the transmission of a frame make every subsequent frame impossible to reconstruct. To alleviate this problem, video compression standards occasionally encode one video frame using still-image coding techniques only, without relying on previously encoded frames. These frames are called "intra frames" or "I-frames." If a frame in the compressed bit stream is corrupted by errors the video decoder must wait until the next I-frame, which doesn't require a reference frame for reconstruction.

Frames that are encoded using only a previously displayed reference frame are called "P-frames," and frames that are encoded using both future and previously displayed reference frames are called "B-frames." In a typical scenario, the codec encodes an I-frame, skips several frames ahead and encodes a future P-frame using the encoded I-frame as a reference frame, then skips back to the next frame following the I-frame. The frames between the encoded I- and P-frames are encoded as B-frames. Next, the encoder skips several frames again, encoding another P-frame using the first P-frame as a reference frame, then once again skips back to fill in the gap in the display sequence with B-frames. This process continues, with a new I-frame inserted for every 12 to 15 P- and B-frames. For example, a typical sequence of frames is illustrated in Figure 1.

	Display Order (left to right)													
Encoding Order (top to bottom)	I													
					P									
		B												
			B											
				B										
									P					
						B								

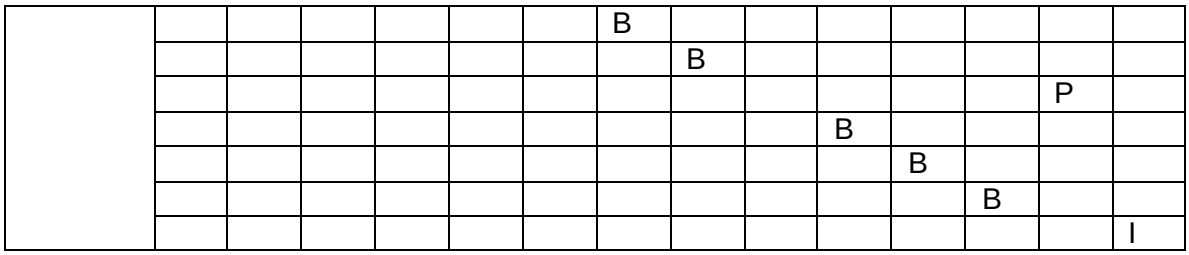

Figure 1. A typical sequence of I, P, and B frames.

Video compression standards sometimes restrict the size of the horizontal and vertical components of a motion vector so that the maximum possible distance between each macro block and the 16-pixel by 16-pixel region selected during motion estimation is much smaller than the width or height of the frame. This restriction slightly reduces the number of bits needed to encode motion vectors, and can also reduce the amount of computation required to perform motion estimation. The portion of the reference frame that contains all possible 16-by-16 regions that are within the reach of the allowable motion vectors is called the "search area."

In addition, modern video compression standards allow motion vectors to have non-integer values. That is, the encoder may estimate that the motion between the reference frame and current frame for a given macro block is not an integer number of pixels. Motion vector resolutions of one-half or one-quarter of a pixel are common. Thus, to predict the pixels in the current macro block, the corresponding region in the reference frame must be interpolated to estimate the pixel values at non-integer pixel locations. The difference between this prediction and the actual pixel values is computed and encoded as described above.

Motion estimation is the most computationally demanding task in image compression applications, and can require as much as 80% of the processor cycles spent in the video encoder. The simplest and most thorough way to perform motion estimation is to evaluate every possible 16-by-16 region in the search area, and select the best match. Typically, a "sum of absolute differences" (SAD) or "sum of squared differences" (SSD) computation is used to determine how closely a 16-pixel by 16-pixel region matches a macro block. The SAD or SSD is often computed for the luminance plane only, but can also include the chrominance planes. A relatively small search area of 48 pixels by 24 pixels, for example, contains 1024 possible 16-by-16 regions at ½pixel resolution. Performing an SAD on the luminance plane only for one such region requires 256 subtractions, 256 absolute value operations, and 255 additions. Thus, not including the interpolation required for non-integer motion vectors, the SAD computations needed to exhaustively scan this search area for the best match require a total of 785,408 arithmetic operations per macro block, which equates to over 4.6 billion arithmetic operations per second at CIF (352 by 288 pixels) video resolution and a modest frame rate of 15 frames per second.

Because of this high computational load, practical implementations of motion estimation do not use an exhaustive search. Instead, motion estimation algorithms use various methods to select a limited number of promising candidate motion vectors (roughly 10 to 100 vectors in most cases) and evaluate only the 16-pixel by 16-pixel regions corresponding to these candidate vectors. One approach is to select the candidate motion vectors in several stages. For example, five initial candidate vectors may be

selected and evaluated. The results are used to eliminate unlikely portions of the search area and hone in on the most promising portion of the search area. Five new vectors are selected and the process is repeated. After a few such stages, the best motion vector found so far is selected.

Another approach analyzes the motion vectors selected for surrounding macro blocks in the current and previous frames in the video sequence in an effort to predict the motion in the current macro block. A handful of candidate motion vectors are selected based on this analysis, and only these vectors are evaluated.

By selecting a small number of candidate vectors instead of scanning the search area exhaustively, the computational demand of motion estimation can be reduced considerably, sometimes by over two orders of magnitude. Note that there is a tradeoff between computational demand and image quality and/or compression efficiency: using a larger number of candidate motion vectors allows the encoder to find a 16-pixel by 16-pixel region in the reference frame that better matches each macro block, thus reducing the prediction error. Therefore, increasing the number of candidate vectors on average allows the prediction error to be encoded with fewer bits or higher precision, at the cost of performing more SAD (or SSD) computations.

In addition to the two approaches describe above, many other methods for selecting appropriate candidate motion vectors exist, including a wide variety of proprietary solutions. Most video compression standards specify only the format of the compressed video bit stream and the decoding steps, and leave the encoding process undefined so that encoders can employ a variety of approaches to motion estimation. The approach to motion estimation is the largest differentiator among video encoder implementations that comply with a common standard. The choice of motion estimation technique significantly impacts computational requirements and video quality; therefore, details of the approach to motion estimation in commercially available encoders are often closely guarded trade secrets.

Many processors targeting multimedia applications provide a specialized instruction to accelerate SAD computations, or a dedicated SAD coprocessor to offload this computationally demanding task from the CPU.

Note that in order to perform motion estimation, the encoder must keep one or two reference frames in memory in addition to the current frame. The required frame buffers are very often larger than the available on-chip memory, requiring additional memory chips in many applications. Keeping reference frames in off-chip memory results in very high external memory bandwidth in the encoder, although large on-chip caches can help reduce the required bandwidth considerably.

Some video compression standards allow each macro block to be divided into two or four subsections, and a separate motion vector is found for each subsection. This option requires more bits to encode the two or four motion vectors for the macro block compared to the default of one motion vector. However, this may be a worthwhile tradeoff if the additional motion vectors provide a better prediction of the macro block pixels and results in fewer bits used for the encoding of the prediction error.

3.2 Motion Compensation

In the video decoder, motion compensation uses the motion vectors encoded in the video bit stream to predict the pixels in each macro block. If the horizontal and vertical components of the motion vector are both integer values, then the predicted macro block is simply a copy of the 16-pixel by 16-pixel region of the reference frame. If either component of the motion vector has a non-integer value, interpolation is used to estimate the image at non-integer pixel locations. Next, the prediction error is decoded and added to the predicted macro block in order to reconstruct the actual macro block pixels.

Compared to motion estimation, motion compensation is much less computationally demanding. While motion estimation must perform SAD or SSD computation on a number of 16-pixel by 16-pixel regions per macro block, motion compensation simply copies or interpolates one such region. Because of this important difference, video decoding is much less computationally demanding than encoding. Nevertheless, motion compensation can still take up as much as 40% of the processor cycles in a video decoder, although this number varies greatly depending on the content of a video sequence, the video compression standard, and the decoder implementation. For example, the motion compensation workload can comprise as little as 5% of the processor cycles spent in the decoder for a frame that makes little use of interpolation.

Like motion estimation, motion compensation requires the video decoder to keep one or two reference frames in memory, often requiring external memory chips for this purpose. However, motion compensation makes fewer accesses to reference frame buffers than does motion estimation. Therefore, memory bandwidth requirements are less stringent for motion compensation compared to motion estimation, although high memory bandwidth is still desirable for best processor performance in motion compensation functions.

4. Reducing Artifacts

4.1 Blocking and Ringing Artifacts

Ideally, lossy image and video compression algorithms discard only perceptually insignificant information, so that to the human eye the reconstructed image or video sequence appears identical to the original uncompressed image or video. In practice, however, some visible artifacts may occur. This can happen due to a poor encoder implementation, video content that is particularly challenging to encode, or a selected bit rate that is too low for the video sequence resolution and frame rate. The latter case is particularly common, since many applications trade off video quality for a reduction in storage and/or bandwidth requirements.

Two types of artifacts, “blocking” and “ringing,” are particularly common in video compression. Blocking artifacts are due to the fact that compression algorithms divide each frame into eight-pixel by eight-pixel blocks. Each block is reconstructed with some small errors, and the errors at the edges of a block often contrast with the errors at the edges of neighboring blocks, making block boundaries visible. Ringing artifacts are due to the encoder discarding too much information in quantizing the high-frequency DCT coefficients. Ringing artifacts appear as distortions around the edges of image features.

4.2 Deblocking and Deringing Image Filters

Video compression applications often employ filters following decompression to reduce blocking and ringing artifacts. These filtering steps are known as “deblocking” and “deringing,” respectively. Both deblocking and deringing utilize low-pass FIR (finite impulse response) filters to hide these visible artifacts. Deblocking filters are applied at the edges of image blocks, blending the edges of each block with those of its neighbors to hide blocking artifacts. Deringing often uses an adaptive filter. The deringing filter first detects the edges of image features. A low-pass filter is then applied to the areas near the detected edges to smooth away ringing artifacts, but the edge pixels themselves are left unfiltered or weakly filtered in order to avoid blurring.

Deblocking and deringing filters are fairly computationally demanding. Combined, these filters can easily consume more processor cycles than the video decoder itself. For example, an MPEG-4 simple-profile, level 1 (176x144 pixel, 15 fps) decoder optimized for the ARM9E general-purpose processor core requires that the processor be run at an instruction cycle rate of about 14 MHz when decoding a moderately complex video stream. If deblocking is added, the processor must be run at 33 MHz. If deringing and deblocking are both added, the processor must be run at about 39 MHz—nearly three times the clock rate required for the video decompression algorithm alone.

4.3 Post-processing vs. In-line Implementation

Deblocking and deringing filters can be applied to video frames as a separate post-processing step that is independent of video decompression. This approach provides system designers the flexibility to select the best deblocking and/or deringing filters for their application, or to forego these filters entirely in order to reduce computational demands. With this approach, the video decoder uses each unfiltered reconstructed frame as a reference frame for decoding future video frames, and an additional frame buffer is required for the final filtered video output.

Alternatively, deblocking and/or deringing can be integrated into the video decompression algorithm. This approach, sometimes referred to as “loop filtering,” uses the filtered reconstructed frame as the reference frame for decoding future video frames. This approach requires the video encoder to perform the same deblocking and/or deringing filtering steps as the decoder, in order to keep each reference frame used in encoding identical to that used in decoding. The need to perform filtering in the encoder increases processor performance requirements for encoding, but can improve image quality, especially for very low bit rates. In addition, the extra frame buffer that is required when deblocking and/or deringing are implemented as a separate post-processing step is not needed when deblocking and deringing are integrated into the decompression algorithm .

5. Color Space Conversion

As noted above, video compression algorithms typically represent color images using luminance and chrominance planes. In contrast, video cameras and displays typically mix red, green, and blue light to represent different colors. Therefore, the red, green, and blue pixels captured by a camera must be converted into luminance and

chrominance values for video encoding, and the luminance and chrominance pixels output by the video decoder must be converted to specific levels of red, green, and blue for display. The equations for this conversion require about 12 arithmetic operations per image pixel, not including the interpolation needed to compensate for the fact that the chrominance planes have a lower resolution than the luminance plane at the video compression algorithm's input and output. For a CIF (352 by 288 pixel) image resolution at 15 frames per second, conversion (without any interpolation) requires over 18 million operations per second. This computational load can be significant; when implemented in software, color conversion requires roughly one-third to two-thirds as many processor cycles as the video decoder.

6. Summary and Conclusions

Video compression algorithms employ a variety of signal-processing tasks such as motion estimation, transforms, and variable-length coding. Although most modern video compression algorithms share these basic tasks, there is enormous variation among algorithms and implementation techniques. For example, the algorithmic approaches and implementation techniques used for performing motion estimation can vary among video encoders even when the encoders comply with the same compression standard. In addition, the most efficient implementation approach for a given signal-processing task can differ considerably from one processor to another, even when a similar algorithmic approach is used on each processor. Finally, the computational load of some tasks, such as motion compensation, can fluctuate wildly depending on the video program content. Therefore, the computational load of a video encoder or decoder on a particular processor can be difficult to predict.

Despite this variability, a few trends can readily be observed:

- Motion estimation is by far the most computationally demanding task in the video compression process, often making the computational load of the encoder several times greater than that of the decoder.

- The computational load of the decoder is typically dominated by the variable-length decoding, inverse transform, and motion compensation functions.

- The computational load of motion estimation, motion compensation, transform, and quantization/dequantization tasks is generally proportional to the number of pixels per frame and to the frame rate. In contrast, the computational load of the variable-length decoding function is proportional to the bit rate of the compressed video bit stream.

- Post-processing steps applied to the video stream after decoding—namely, deblocking, deringing, and color space conversion—contribute considerably to the computational load of video decoding applications. The computational load of these functions can easily exceed that of the video decompression step, and is proportional to the number of pixels per frame and to the frame rate.

The memory requirements of a video compression application are much easier to predict than its computational load: in video compression applications memory use is dominated

by the large buffers used to store the current and reference video frames. Only two frame buffers are needed if the compression scheme supports only I- and P-frames; three frame buffers are needed if B-frames are also supported. Post-processing steps such as deblocking, deringing, and color space conversion may require an additional output buffer. The size of these buffers is proportional to the number of pixels per frame.

Combined, other factors such as program memory, lookup tables, and intermediate data comprise a significant portion of a typical video application's memory use, although this portion is often still several times smaller than the frame buffer memory.

Implementing highly optimized video encoding and decoding software requires a thorough understanding of the signal-processing concepts introduced in this paper and of the target processor. Most video compression standards do not specify the method for performing motion estimation. Although reference encoder implementations are provided for most standards, in-depth understanding of video compression algorithms often allows designers to utilize more sophisticated motion estimation methods and obtain better results. In addition, a thorough understanding of signal-processing principles, practical implementations of signal-processing functions, and the details of the target processor are crucial in order to efficiently map the varied tasks in a video compression algorithm to the processor's architectural resources.