The H.264 Video Compression Standard

by

Andrew Gibson

A project submited to the
Department of Mathematics and Statistics
in conformity with the requirements for
the degree of
Master of Science

Queen's University
Kingston, Ontario, Canada
August, 2002

Abstract

This document will examine the emerging video coding standard, H.264. H.264, originally named H.26L, has been under development by the ITU-T since 1998. Even though the standard has yet to be finalized, development test models have shown remarkable performance gains over the current state of the art video coding standard, MPEG-4 (AS)[1].

The H.264 recommendation is directed at compressing and coding natural video; there are no tools to manipulate individual *objects* represented in the video frames. The compression algorithm follows the same general steps as those from previous H.26X and MPEG standards, however the design philosophy has been updated to reflect recent advances in technology. There is an increased emphasis on fine control over each video frame. The compression tools have been reengineered to operate quickly and efficiently, using a minimum number of computations.

The first chapter provides motivation and background for the study of video compression. The second chapter presents, in a very general form, the tools which are used in H.264. The third and fourth chapters provide an overview of how each specific compression and coding tool is used. Bulky technical specifications, such as bitstream restrictions, have been omitted in favour of providing as concise a document as possible. The fifth chapter contains results from running the most recent test model software for H.264.

This document is not easily read; the compression tools are presented in a multi–layered tree format. For example, there are two main methods of coding the video data, once one mode has been picked, there could be as many as four more options to further specify how the data will be divided; choosing one of these four options results in a further four choices which must be made. If the reader finds themselves getting lost it is best to use the section headings to determine which branch of the tree is currently being discussed. The actual H.264 recommendation does not link these concepts together, the algorithm is presented in a piece–wise manner. Therefore, for

[1] Advanced Simple Profile.

the purpose of this document, the tools have been reorganized in the above mentioned tree structure to provide a logical flow.

Acknowledgements

I would like to take this opportunity to acknowledge the help from my co-supervisors, Dr. Alajaji and Dr. Linder; without their support and guidance I would not have been able to complete this project.

I would also like to thank my friends, Shaun, Jen, Leighton, Michele, Hilton, Matt, Tamy, Jon, Marina, Cindy and Gail for giving me their patient support while I was writing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Video Compression

1.1 Why do we need video compression?

This question would be better asked in two parts. Why would we want to compress anything at all? And why video?

Put succinctly, data compression is needed to achieve the best overall performance from the technology at hand. There are many information sources which can generate an enormous amount of data, enough to exceed the capacities of today's computers and communications systems. It would be advantageous if these data sources could be manipulated to reduce their size. Chapter 2 details many popular data compression techniques.

Video is important because it is a reliable, efficient and effective means of imparting ideas, and on occasion may even be superior to face–to–face human interaction. Video is used to deliver a myriad of information: news, sign–language translations for the hearing impaired, security camera updates, television programs, movies and much more.

Uncompressed video content requires large amounts of bandwidth for transmission, memory for storage and computing power for encoding and decoding. For example, one single frame from a DVD video consists of 720x576 pixels with a colour depth of 24 bits per pixel. If the North American video standard of 29.97 frames per second is used then a bandwidth of approximately 250 Mbs are required to transmit the raw data.

It would seem, therefore, that it would be worthwhile to try and achieve the compression of video with a minimum of computing power while minimizing bandwidth and memory requirements. This is why video compression is needed, or more importantly, why *good* video compression is needed.

1.2 Where is video compression headed?

During the past 10 years, digital video has experienced a rapid growth fueled by expanding computing and communications resources. In fact, it could be said that the desire for higher quality multimedia content is now driving the creation of new technologies[1]. People can now download a trailer of their favorite movie or the latest news broadcast from the Internet. Using video–conferencing, companies can conduct meetings where some of the participants are separated by thousands of kilometers. Complete training videos can be recorded onto a single CD-ROM and easily distributed to employees. People who are hearing impared can obtain sign-language videos to accompany television programs. There are many areas where people are anxious to exploit these advantages of video communications.

1.2.1 Video on Demand

Video on demand represents a shift in thinking about the delivery of multimedia content to consumers. Previously, a small amount of content was pushed at consumers in the hopes of pleasing as many people as possible. Video on demand offers the possibility that people may watch what they want, when they want. In other words, instead of having content pushed at them, they will be pulling it in.

The prospect of being able to download and watch video content on demand is a very appealing idea to consumers. DivX Networks is currently leading the way in providing services which will allow people to download and watch movies over the Internet. Broadway is taking advantage of the services provided by DivX Networks to distribute video versions of plays

[1] For example: the MMX technology released by Intel in 1997, see [11].

and musicals. Other companies such as Pay-Per-View are also exploring this area which would enable them to offer a more diverse range of products and services.

1.2.2 Video Conferencing

Video conferencing is an increasingly popular method for communication. Its growth has been slow over the past 10 years because available computing and network resources have not been sufficient to meet the challenge. With the recent advent of high speed networks and better video compression technology, sales of video conferencing equipment have been increasing dramatically.

Video conferencing is a popular option for many large organizations because it minimizes the need for travel. Instead of continually paying for traveling expenses, a company can either pay to set up a high speed network which has many usages (including video conferencing) or they can rent time on an existing network. Employees will also no longer be burdened with the need to travel.

While video conferencing performance is heavily influenced by the underlying hardware, a good video compression scheme can help make the best use of available resources and provide a high quality picture.

1.2.3 Streaming Wireless Video

Currently deployed wireless technology, such as digital cell phones, can not use enough bandwidth to provide reliable, high–quality video streaming. New standards on the horizon such as 3G wireless phones will change this situation.

Streaming wireless video could be thought of as a subset of video on demand. However, the challenges posed by wireless video are quite unique. Video on demand assumes the existence of a high speed network to smoothly carry heavy data loads, whereas with a wireless network, the amount of bandwidth available is limited and the channel can be very noisy. A video compression scheme for wireless networks would have to be very robust and should be optimized for low resolution video as a reflection of the reality that most wireless devices tend to be small.

There are many planned uses for wireless video. 3G cell phones will soon be sold which will enable people to view news clips, movie trailers or record small video messages and send them to their friends. Other, more practical, uses include security or safety cameras or overseeing troops in a military operation. All this information would need to be encoded, transmitted and decoded in real time. An efficient video compression algorithm is obviously needed to attain these goals.

1.3 Current Video Compression Standards

There are two main standardization bodies for video technology. Firstly, there is a division of the International Telecommunication Union known as the ITU-T. The specific division within the ITU-T which is in charge of multimedia data compression is called SG-16 Video Coding Experts Group (VCEG). They have been responsible for producing many well known and widely used standards such as H.261, H.263 and H.263+. The second organization is the International Organization for Standardization (ISO). Within the ISO, the specific committee which is in charge of multimedia data compression is the JTC1 Moving Pictures Expert Group (MPEG). They have been responsible for producing the MPEG series of standards (MPEG-1, MPEG-2 and MPEG-4). The MPEG standards tend to be much bulkier than those of VCEG. For example the most recent version of H.263, H.263++, is slightly more than 100 pages while the most recent version of MPEG, MPEG-4, is well over 5000 pages. This is because the ITU-T produces standards which are much more modular than MPEG. For example it might take 4 or 5 ITU-T audio and video standards to cover the same range of MPEG-2. Another difference between MPEG and VCEG is that MPEG standards are primarily designed for video storage and streaming while VCEG have produced standards for real-time applications such as video conferencing or video–phones.

1.3.1 H.26X

The first well known standard put out by VCEG was H.261 (*Video codec for audiovisual services at "P x 64 kbit/s"*),[8], in 1990. H.261 has been widely used as the default standard for video conferencing, primarily over ISDN links. H.261 operates with very little encoding delay and minimal computing overhead. In 1996, the first version of H.263 (*Video coding for low bit rate communication*), [8], was released. It was built around the same algorithm as H.261 but with several improvements. More video resolutions were supported, along with error control techniques and an optional arithmetic encoder. Two more versions of H.263 have followed, H.263+ in 1999,[8], and H.263++ in 2001, [8]. While the H.263 series has not been as widely recognized as MPEG, many products use it. For example, many digital cameras use H.263 for capturing video.

In 1998, VCEG started work on a new standard called H.26L, [8], later to be renamed H.264. Development for H.26L is continuing, the first version should be published before the end of 2002. The results so far have been so successful that MPEG has taken notice and now H.26L is being developed under the auspices of both the ITU and MPEG, [7].

1.3.2 MPEG

MPEG released their first standard in 1992 which was titled *Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s*, [12]. This title is often shortened to MPEG-1. The motivation behind MPEG-1 was the efficient storage and retrieval of video and audio data from a CD-ROM. Video CD (VCD) is a popular implementation of MPEG-1 where a standard length movie is stored on 2 CD–ROMs. MPEG-1 also provided the well known audio compression format MPEG-1 Layer 3 (MP3).

In 1994, MPEG released their second standard *Generic coding of Moving Pictures and Associated Audio* or MPEG-2, [12]. MPEG-2 contains various improvements and additions over MPEG-1. Protocols were added which supported streaming MPEG-2 data over a range of networks and the remote control of servers containing MPEG-2 data. MPEG-2 Advanced Audio Compression provides superior compression over MPEG-1 as well as support for up to 5 channels, whereas MPEG-1 was limited to two. Since MPEG-2 is the standard which is used for encoding DVDs, it is perhaps the most widely used multimedia data compression standard in the world right now. It is also widely used in Europe and Japan for the delivery of Digital Television.

An interesting side note is that there is no standard entitled MPEG-3. It was started with the view towards providing compression for High Definition Television however it was soon found that MPEG-2 contained all the necessary tools for accomplishing this task.

MPEG-4 or *Coding of audio-visual objects*, [12], is the most recent and most ambitious project MPEG has tackled. The first limited version of MPEG-4 was released in 1998 and it is continuing to evolve. The goal behind MPEG-4 is to divide a video frame into discrete objects which will be coded separately. For example, a video scene depicting a car traveling across the screen could be separated into two objects. The first would be the car and it could be encoded using tools which are suitable for fast moving video. The second object would be the background scenery which is not changing very quickly. This object could be encoded using an array of techniques which would take advantage of redundancy between the frames. Although work on MPEG-4 is continuing there are already limited implementations available. DivX is a codec which is produced by DivX Networks. The DivX standard can compress an 8 GB MPEG-2 video file down to 700MB while still maintaining reasonable quality.

1.3.3 H.264

H.264 has been designed primarily for use in applications using a relatively low bit rate and low picture resolution. The focus is on low delay real–time applications. Its main features are as follows in [5].

- Using a clean slate and back–to–basics design approach that results in an especially simple and efficient design specification
- Having a “network–friendly” structure enabling straightforward adaptation for use over a variety of important commu-

nication network systems.

- Providing sufficient error and packet–loss resilience for use in mobile (e.g. 3G wireless) Internet and other environments with unreliable data delivery
- Having the capability to be used for low–delay real–time applications

In July 2001, a proposal was submitted which would see MPEG and VCEG working together on H.26L. Early test results showed that H.26L was significantly outperforming the *Advanced Simple* profile in MPEG-4. It was during the summer of 2002 when the name was officialy changed to H.264. H.26L is being considered for use in standard video applications along with digital cinema.

Industry has also been rallying behind H.264. Companies such as UB Video, have been quick to produce embedded implementations. Once H.264 becomes part of MPEG, it will gain a much wider audience.

Chapter 2

Standard Video Compression Tools

Video compression uses a combination of techniques to achieve the maximum possible compression while maintaining a desired level of quality. Many techniques are used to gain as much compression as possible by exploiting redundancy in the video. Once as much redundant information as possible has been removed, the encoder starts *losing* information in a controlled manner in order that the video be as close to but still less than the specified bit rate.

The tools are presented below, generally in the same order which they are used in H.264. Rather than presenting specific implementation details about each tool in H.264 (these are held back until the next chapter), they are discussed only in the context of the role they play in most standard video compression schemes (MPEG, H.26X).

The video compression tools fall into four broad categories.

2.1 Temporal Redundancy

Temporal redundancy is the repeated information between successive frames in a video. A newscast for example, typically features the news–reader sitting still with very little movement in the background scenery. It would be a waste to encode all the information about the background for every frame. Thus, it would be very convenient to be able to reference information in previous frames to avoid repeating the same information in the future frames. It should be noted that tools which try and exploit temporal redundancy are less effective in cases where there are frequent scene changes.

2.1.1 Motion Compensation

Motion compensation uses information from previous frames to provide as much information as possible about the current frame. The frame being encoded is divided into rectangular blocks of pixels. For each block in the current frame a search is done in the *reconstruction* of the previous frame to determine the best match. If a suitable match is found, it is then subtracted from the currently regarded block. The hope is that the difference between these two blocks will be mostly zero. If no suitable match can be found, the currently regarded block is transmitted unaltered.

Of course an accompanying *motion vector*[1] must be sent along with the residual block value to enable the decoder to find the appropriate block in the previous reconstructed picture. In the case when no suitable match was found no motion vector data will be sent.

There are two important factors which must be considered in conjunction when performing motion compensation: block size and search radius magnitude. If the block size is made large, the result will give poor prediction, however fewer searches will have to be performed. If the block size is made small, the encoder will be tied down trying to match each of the blocks to somewhere in the previous picture. But smaller block sizes will allow better motion compensation for fine movements within the video sequence. If the search radius is made small, the matching algorithm will have a faster run time; but if there is a lot of movement within the video, the chances of finding the best match from the previous frame are lowered. If the search radius is made large, the matching algorithm will have to examine many pixels and will run slower; however the chances of finding the best match are higher.

[1] The motion vector will be a 2–D integer vector.

2.1.2 Motion Vector Prediction

Since the encoder and decoder both have access to the same information about previous motion vectors, the encoder can take advantage of this to further reduce the dynamic range of the motion vectors it sends. Based on previous motion vectors, a prediction can be produced which can have two uses. Firstly, the searching algorithm can start its search centered on where the prediction points. If the prediction is accurate, this would speed up the search algorithm. Secondly, once the actual motion vector is found, the prediction can be subtracted and the residual motion vector can be transmitted.

2.1.3 Pixel Interpolation

In order to ensure that the best match for each block is found, pixel interpolation is used. New pixels are interpolated by taking weighted sums of the surrounding original pixels. For every layer of interpolated pixels, the image size will be doubled. Therefore, while pixel interpolation does produce significantly better results, it does increase the run–time of the matching algorithm.

Here is an example of a half–pixel precision scheme; the upper case letters correspond to original pixels and the b_i's and c_1 are the new pixels.

A	b_1	B
b_2	c_1	b_3
C	b_4	D

Table 2.1: Half-Pixel Precision Image

The pixels $b_1,\ b_2,\ b_3,\ b_4$ are interpolated using averaging:

$$b_1 = \left\lfloor \frac{A+B}{2} + \frac{1}{2} \right\rfloor \tag{2.1}$$

$$b_2 = \left\lfloor \frac{A+C}{2} + \frac{1}{2} \right\rfloor \tag{2.2}$$

$$b_3 = \left\lfloor \frac{B+D}{2} + \frac{1}{2} \right\rfloor \tag{2.3}$$

$$b_4 = \left\lfloor \frac{C+D}{2} + \frac{1}{2} \right\rfloor, \tag{2.4}$$

Where $\lfloor x \rfloor$ returns the nearest integer, n, less than or euqal to x.

The center pixel c_1 is calculated using the following:

$$c_1 = \left\lfloor \frac{A+B+C+D}{4} + \frac{1}{2} \right\rfloor . \tag{2.5}$$

2.2 Spatial Redundancy

Exploiting spatial redundancy involves taking advantage of correlation between adjoining pixels. Often, pixels which are located close to each other will have either the same value or be very close in value; it would be a waste of resources to continually resend the same pixel values. One of the most common tools for avoiding this waste is to transform the pixel data into another domain where hopefully most of the repeated information could be compacted down into a smaller number of coefficients.

This method is even more effective if it is performed after temporal redundancy has been removed. Once the previous frame has been removed, the current frame would have a very uniform appearance[2]. There would then be a very high correlation between neighboring pixels.

When transform techniques are used on a video frame, the frame is first divided up into rectangular blocks of pixels, usually $N \times N$. The transform then acts on this group of pixels in such a way that most of the magnitude information for the pixels will be mapped to a small number of transform coefficients while the rest of the coefficients would either be very small values or zero. The transform should be reversible so that if the inverse transform is performed the original pixel values are obtained.

The best transform to use for energy compaction is the Karhunen–Loeve Transform (KLT), see [4]. However, the KLT is image dependent which makes it too computationally intensive for video compression.

2.2.1 Discrete Cosine Transform (DCT)

The discrete cosine transform (DCT), [4], is a very popular transform for exploiting spatial redundancy. It has been used in JPEG, H.263, MPEG-1,

[2]In the extreme case where two successive frames are identical, after subtracting off the previous frame, the current frame would be entirely devoid of any colour.

MPEG-2 and MPEG-4. It was dropped from JPEG-2000 in favour of wavelet transforms.

The DCT is popular because it achieves significant energy compaction (or coefficient decorrelation) relative to the amount of computation required. It is not data dependent which means it does not need to be recalculated for every new set of data to which it is applied.

The DCT is an orthonormal transform which means it is energy preserving. It captures frequency information in pictures by mapping it to a finite number of basis functions. These orthonormal bases span a frequency space and range from DC to higher frequency functions. The DCT will map pixel information so that smooth regions with little colour change will be represented by mostly low frequency bases and abrupt colour changes will represented by mostly high frequency bases.

The transform matrix for the DCT is calculated as follows:

$$[\mathbf{C_{i,j}}] = \begin{cases} \sqrt{\frac{1}{N}} & \text{for} \quad i = 0, j = 0 \ldots, N-1 \\ \sqrt{\frac{2}{N}} \cos \frac{(2j+1)i\pi}{2N} & \text{for} \quad i = 1 \ldots, N-1, j = 0, 1, \ldots, N-1. \end{cases}$$

The DCT is often implemented in 8x8 or 4x4 blocks. However, the problem with the DCT is that it introduces floating point numbers. Any scheme which uses floating point numbers risks incurring the errors of rounding mismatch between the encoder and decoder. A transform which is image independent and performs just as well as the DCT and uses only integer arithmetic such as addition or bit shifting would be ideal.

2.2.2 Discrete Walsh-Hadamad Transform

The discrete Walsh-Hadamad transform (DWT), [4], is very simple to implement. The operations consist entirely of addition and subtraction. The simplicity of operation obviously comes at a price which is a considerably lower degree of pixel decorrelation. A 4x4 DWHT has the following form:

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

2.2.3 Run–Length Coding

Run–Length coding is used in situations where the same value is known to be repeating in long sequences. It will be assumed that this value is zero[3]. Being able to take advantage of run–length coding is one of the reasons why transform coding is used on images. Once the pixel information has been mapped to the basis functions, the coefficients which correspond to high frequency components can be set to zero[4].

The sequence of data will be broken up into code words. These code words usually have 2 or 3 components. The first component will represent the length of the run of zeros. The second component will be the magnitude of the non-zero coefficient which ends the run of zeros. If a 2–D Variable Length Code (VLC) table is being used then there will be a special symbol to represent the end of the block (EOB). If a 3–D table is used then each code word will have a third component which indicates whether or not the end of the data has been reached.

Here is a simple example of the effectiveness of run–length coding:

The sequence S:

0,0,0,0,0,0,4,0,0,5,9,0,0,0,0,0,0,0,0,0,0,3

will be represented by:

(6,4),(2,5),(0,9),(10,3),EOB.

Instead of sending 21 code words to represent S only 5 codewords are used. The count is 5 and not 9 because the each codeword will signal the magnitude of the coefficient as well as the size of the next run of 0's.

There is a trade off to be made with run–length coding. The longer the run of zeros, the larger the code table grows. If a run of k zeros is allowed and an upper bound of N is placed on the number of distinct possible coefficients which will appear in the sequence, the code table will have a size of kN. Therefore if k is made too large the codewords will become very large and possibly outweigh the benefits of run–length coding.

[3] In practice this is the usual state of affairs.

[4] This is actually part of *quantization* which will be covered below shortly.

2.3 Quantization

Thus far all the methods presented have fallen into the category of *lossless* coding. Schemes were presented which took advantage of empirical results and the correlation between neighboring pixels and successive frames. All of these methods leave the original signal completely unchanged which is highly desirable. However they all have their limits and this is when *lossy coding* is used. In the context of video compression, lossy coding is used to make up the difference between the lower limit of lossless encoding and the upper limit of the available bit rate.

Most lossy encoding schemes work by making a trade off between the signal distortion and the signal rate. By lowering the rate (by dropping information) the signal quality is degraded, however video is one of the applications where loss is tolerable. Of course a signal which has been encoded using lossy techniques can not be recovered exactly at the decoder

There are many ways to perform lossy coding but quantization is one of the most commonly used compression schemes. The goal of quantization is to represent a very large, possibly infinite set of values, with a much smaller set. It is usually used in video compression schemes for two purposes. The first reason has already been stated; by mapping the source data to a smaller set, fewer bits will be required.

A quantizer can be modeled as a many to 1 function, f. The range, H, of f contains all the possible values which the source picture can take. The domain of f, R, is divided into quantization regions R_i $i = 1, 2, \ldots$. For each of the regions, R_i, there exists a distinct value $f(x)$ in the domain

R to which every $x \in R_i$ is mapped. The value $f(x)$ does not even have to belong to H. R is populated by values which are the best approximations for their corresponding regions. The assumption is that the human eye will not miss the loss of precision. The effect is to reduce the number of possible colours. The impact can be most easily noticed in areas of a picture where there should be a fine gradation of colour change.

Here is a simple example of quantization. Let the data source $R = (0, 10)$. There are an infinite number of values in R. The goal is to efficiently represent values from this source. They could be rounded to the nearest decimal which would result in 100 codewords. If the source produces the value π then it would be mapped to 3.1.

The effects of quantization are very complex and there are many possible ways of designing a quantizer. For example, the statistical properties of the source could be taken into account so that the approximating value for each region is the region's centroid. This would result in a lower signal distortion and a correspondingly higher SNR value.

Another method for increasing the effectiveness of the quantizer is to take advantage of correlation between subsequent samples. This is accomplished by grouping samples together into vectors. Vector quantization is rarely employed in video compression because of the additional complexity[5] incurred by its use. Scalar quantization is the preferred method.

The second, less obvious benefit of quantization is the manner in which it complements run–length coding. Coefficients with small magnitudes (such as high frequency transform coefficients) can be decimated to zero. This paves the way for run–length coding to further compress the signal.

2.4 Entropy Encoding

Once the video information has been squeezed into the run–length symbols there is still further lossless compression which can be obtained. Instead of assigning every symbol a codeword of equal length, a probability model can be used to assign code words based on the *expected* occurrence of symbols or runs of symbols. Outcomes which are assigned a higher probability (usually based on empirical tests) are given shorter code words so that the expected code rate is lower than if every outcome had equi–length codewords.

Entropy encoding can also be done with adaptive models. As more data is encoded the coder can learn which symbols are occurring most often and shift the model accordingly.

[5] As the number of dimensions is increased this complexity usually increases exponentially.

2.4.1 Huffman Encoding

Huffman codes are a well known, optimal entropy encoding scheme, see [1]. They rely on two important observations about optimal codes:

1 In an optimum code, the symbols with the highest probability of occurring will have the shortest codewords.

2 In an optimum code, the two symbols which occurs the least frequently will have equal length codewords.

Using these two facts, the algorithm for building a Huffman binary code is relatively simple. The easiest way to present the Huffman algorithm is through an example. Table 2.2 details a five word alphabet with accompanying probabilities. One possible Huffman code to compress this source will be presented next.

Symbol	Probability P(symbol)	Codeword
a_1	0.15	$c(a_1)$
a_2	0.2	$c(a_2)$
a_3	0.2	$c(a_3)$
a_4	0.4	$c(a_4)$
a_5	0.05	$c(a_5)$

Table 2.2: Source Alphabet A

The two least likely symbols are a_1 and a_5. They will therefore have the longest codewords; $c(a_1) = \alpha_1 * 0$[6] and $c(a_5) = \alpha_1 * 1$.

Now define a new alphabet A' with four letters $\{a_2, a_3, a_4, a'_5\}$. Here, a'_5 is composed of a_5 and a_1 from A and has probability equal to $p(a_5)+p(a_1) = 0.2$. The two symbols with the lowest probability are a_2 and a_3. They will have codewords $c(a_2) = \alpha_2 * 0$ and $c(a_3) = \alpha_2 * 1$.

Now define A'' with three letters $\{a'_3, a_4, a'_5\}$ where a'_3 has probability equal to $p(a_2) + p(a_3)$=0.4. The two symbols with the lowest probability are a'_3[7] and a'_5. They will have codewords $c(a'_3) = \alpha_3 * 0$ and $c(a'_5) = \alpha_3 * 1$.

[6] $*$ denotes concatenation.

[7] The choice between a'_3 and a_4 is purely arbitrary.

Therefore, $\alpha_1 = \alpha_3 * 1$ and $\alpha_2 = \alpha_3 * 0$ which results in:

$$\begin{aligned}
c(a_1) &= \alpha_3 * 1 * 0 \\
c(a_5) &= \alpha_3 * 1 * 1 \\
c(a_2) &= \alpha_3 * 0 * 0 \\
c(a_3) &= \alpha_3 * 0 * 1.
\end{aligned}$$

Continuing in this manner the Huffman code for A will be:

$$\begin{aligned}
c(a_1) &= 010 \\
c(a_5) &= 011 \\
c(a_2) &= 000 \\
c(a_3) &= 001 \\
c(a_4) &= 1.
\end{aligned}$$

Basic Huffman codes such as in the example above are not as practical as other schemes because a stationary probability model is assumed, which does not reflect reality in many cases. There are many variations on Huffman encoding such as minimum variance Huffman codes, extended Huffman codes and adaptive Huffman codes.

2.4.2 Arithmetic Encoding

An increasingly popular method of lossless data compression is arithmetic encoding, [4]. The concept involved is very simple, however, implementation details are often very intricate.

Given a source alphabet $A = \{x_1, x_2, \ldots, x_n\}$ with a cumulative distribution function $F(x)$, the sequence of symbols generated is represented by a single **tag** . A tag is used to identify the entire sequence of symbols.

The region [0,1), which corresponds to the range of $F(x)$, is the initial region for generating the tag. This region contains an infinite number of values, which should suffice for representing all possible symbol sequences with unique tags. As more symbols are received the size of the interval containing the tag is shrunk. Given the nature of real numbers, this can be done indefinitely.

The interval [0,1) is partitioned according to $F(x)$. If symbol x_i occurs, then the new interval is $[F(x_{i-1}), F(x_i))$. The new interval is partitioned in the same proportions as [0,1) using a scaled version of $F(x)$, $\frac{F(x)}{p(x_i)} + F(x_{i-1})$.

This process will continue until it is time to send a tag. The tag will be any number which is inside the current interval.

Implementing this scheme using real numbers is impossible, therefore implementations of arithmetic encoding can use a fixed precision floating point implementation where the tag must be sent after a certain number of symbols has been sent. This prevents any underflow errors. There is also an integer implementation which uses scaling to prevent any possibility of generating a region which is too small.

2.5 Deblocking Filters

Deblocking filters do not compress the video sequence. Rather, when they are applied to a video stream which has been compresed, they mitigate some of the unwelcome side-effects of compression.

A picture is essentially a 2–D array filled with pixel values. In order to accommodate a wide range of image sizes there should be a general way of sub–dividing it into manageable chunks. The most common way of dividing an image is into smaller 2–D arrays. This allows each sub–array to be compressed using the surrounding local statistics.

The drawback to using 2–D arrays is **blocking artifacts**. Most image depict scenes with arbitrary shapes which are not composed of squares and rectangles. Approximating everything in the image as a rectangle invariably results in visual distortion. Instances of this visual distortion are called blocking artifacts.

A deblocking filter is applied to the pixel values on either side of a block division. The filter uses averaging to smooth over any sharp discontinuties between adjacent blocks. The effect is to soften the impact of using the rectangle approximation.

2.6 Summary

All these tools can be used in a video compression scheme. Only the very basic elements of each tool have been presented. In the case of H.264, to save computation, the implementation of some tools is restricted to a very basic level[8].

Other tools[9] are implemented using very complex rules which often obscure the actual concepts involved.

Every effort will be made in the following two chapters to present the relevant details in as clear a manner as possible.

[8] Quantization is kept to the simple case of scalar quantization.

[9] Namely, arithmetic coding and deblocking filter.

Chapter 3

H.264 – Baseline Profile

The H.264 standard is currently divided into 3 sections, a **baseline** profile which all standard compliant decoders must support, an optional **main** profile of which the baseline profile is a subset, and a collection of supplemental features which are part of neither the baseline or main profiles. This chapter will examine the following features:

- I and P picture types
- In–loop deblocking filter
- Progressive pictures and Interlaced pictures
- $\frac{1}{4}$–sample motion compensation
- Block segmentation down to 4x4 block size
- VLC–based entropy coding
- Chrominance format 4:2:0

These features make up the baseline profile. The additional material which makes up the main profile will be presented in Chapter 4.

Superficially, H.264 is similar to MPEG-2 and the H.263 series. Motion compensation, motion vector prediction and pixel interpolation are used for reducing temporal redundancy between successive pictures. Transform and run–length coding are used to reduce spatial redundancy within each picture. Quantization and entropy coding are used to obtain further bandwidth reduction.

However, all of these features have been re–engineered in H.264 to streamline calculations and provide optimal video quality. Almost all the calculations can be performed using addition and bit–shifting[1]. Picture quality is improved by the in–loop deblocking filter and the option of using block sizes as small as 4 pixels by 4 lines.

There are also some entirely new features in H.264 such as multiple picture referencing. Another of the new features in H.264 is the network adaptation layer (NAL). The NAL was added because of the growing need for interaction between software layers and network layers. The NAL was designed to be independent from the video coding layer. Since we are focusing on the video coding layer, no further mention will be made of the NAL.

Many aspects of the encoder are not specified by the H.264 standard. Rather, a thorough description for a legal bitstream and decoder are given. Any encoder implementation can be used, provided the bitstream it produces can be decoded by

a standard H.264 compliant decoder.

3.1 The Basic Algorithm

The basic communication model, Figure 3.1, will be assumed. An encoder outputs to a channel. This could be a channel in space, such as an office network or a channel in time, such as a computer hard drive. The channel outputs to a decoder which will in turn output to a video display unit.

Figure 3.1: Basic Communications Model

Figure 3.2 details the basic H.264 algorithm where:

A is the input for *inter* mode (motion compensation).

B is the input for emphintra mode (no motion compensation).

[1] Bit–shifting has the effect of multiplication or division by a power of 2.

C is the bitstream containing quantized transform coefficients as well as encoder parameters.

D is the residual motion vector used in *inter* mode.

E is the input to the channel, the bitstream is either entropy encoded using the H.264 universal variable length coding table or the Context Adaptive Binary Arithmetic Coder (CABAC).

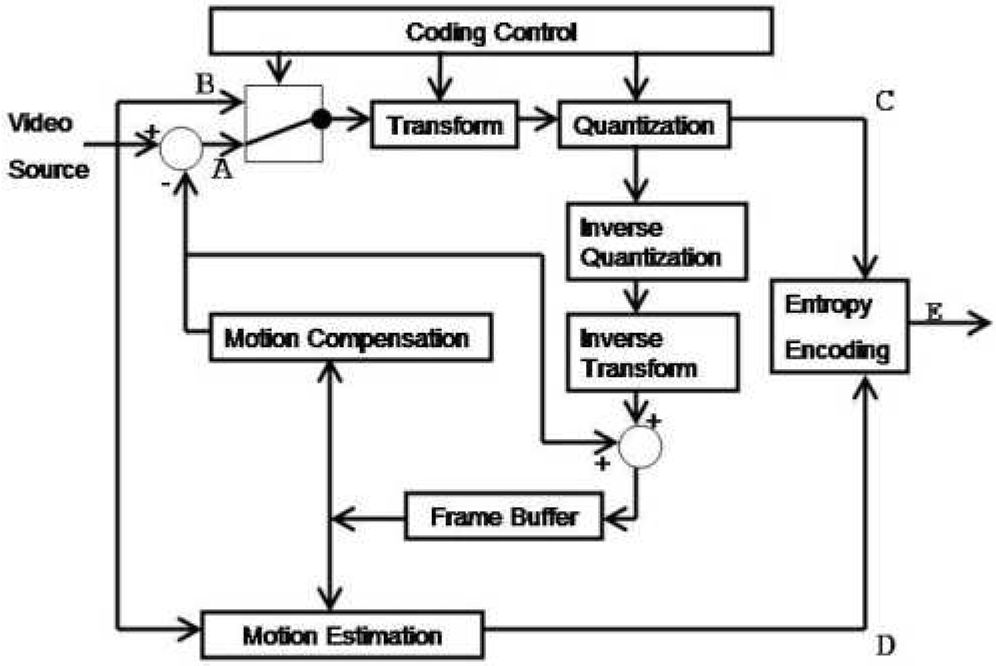

Figure 3.2: H.264 Encoder

The incoming picture is first divided into blocks of 16 pixels by 16 lines. A block size of 16x16 is called a **macroblock**. A very general version of the algorithm follows:

1. The macroblock is sent to be transformed (the transform divides the macroblocks into 16 4x4 blocks). The macroblock is also sent to the motion compensation module to find the best matching blocks in the picture buffer (also called the frame buffer).
2. A motion vector is formed, which points to the best match in one of the previous **reconstructed** pictures. A prediction for the motion

vector is also formed. The difference between the motion vector and its predicted value is sent over the channel. The best match in the picture buffer is sent up to the waiting macroblock and subtracted to form a **residual** macroblock. If a close match is found, the hope is that the residual macroblock will have many pixel values close to 0. If a good match is not found, nothing is subtracted from the macroblock and there will be significantly more information to encode and transmit.

3. The residual macroblock is then transformed, quantized and run–length coded. The resulting sequence is sent to the channel input. It is also sent to the motion compensation module for reconstruction. The reconstructed macroblock is used for compressing the current frame and possibly, any subsequent picture.
4. At the channel input, picture data and encoder parameters are entropy coded before transmission.

The encoder has to operate its own decoder in order to ensure that the values which are subtracted from the macroblock are the same values which are added on at the decoder side.

3.2 Terminology and Concepts

Interlaced vs. Progressive Video

There are two types of video supported in H.264; interlaced and progressive. Interlaced video is a hold–over from the days of broadcast television. It was originally introduced to reduce the flicker on black and white televisions. Each video frame is divided up into two fields, where the second field is delivered half a frame period after the first.

At the receiver end (the television), the two fields are interlaced together to form the original frame. The first field consists of odd numbered picture lines (1,3,5,. . .) and the second field is formed by taking the even lines (2,4,6,. . .). A progressive video frame is not divided into any sections, the entire picture is coded and transmitted as one unit. It reflects the change towards video display units which do not use a cathode ray tube.

In order to be as general as possible, the term picture will be used instead of frame or field. This is the notation which is used in [6], where a picture is defined as:

> Source, coded or reconstructed image data. A source or reconstructed picture consists of three rectangular arrays of 8–bit numbers representing the luma and two chroma signals. A coded picture is made of a picture header, the optional extensions immediately following it and the following picture data. A coded picture may be a coded frame of a coded field. For progressive video, a picture is identical to a frame, while for a interlaced video a picture can refer to a frame or the top field or the bottom field of the frame depending on the context.

Intra vs. ***Inter***

When picture data is coded without reference to other pictures, it is said to be ***intra*** coded. This means there is no motion vector produced.

In the case of H.264, prediction is still used but only using data from within the current picture. Depending on which *intra* mode is being used, there could be as many as 8 prediction modes. Therefore, instead of sending a motion vector, a **prediction mode** parameter must be sent. A picture which has been *intra* coded is called an I picture.

Inter coding implies that previous pictures are being referenced to compensate for temporal redundancy. In fact, H.264 allows a picture to reference pictures other than the one most recently decoded. This is a new feature from H.263. A picture which has been coded in *inter* mode is called a P picture.

Colour Components

By convention, a colour signal is divided up into 3 components, a **luminance** and 2 **chrominance** components. The components are a linear combination of the red, green and blue colours which make up the colour of each pixel. The luminance component contains all the data needed to form the grey–scale version of the picture. The chrominance data adds the missing colour[2]. Since

[2]Chrominance and colour will be used interchangeably in this document.

the luminance data contains a significant portion of the picture information, it is assigned a higher importance than the chrominance components. Video formats are defined by the way in which chrominance data is treated. A very common video format is 4:2:0, in which, for every 4 samples of luminance data only 1 of each chrominance component will be sampled.

Notation

In order to present equations in as simple a form as possible, bit–shifting operations will often be represented using multiplication and division operators. Therefore, any multiplication or division by a power of 2 is accomplished using bit shifting. Any division by a number which is not a power of 2 can be considered as only producing an integer, the remainder is dropped.

Conversely, sometimes using the division or multiplication operators obscure how the calculation takes place. In these instances $a >> b$ and $a << b$ will be used to indicate a right or left shift, respectively, of a by b bits.

3.3 Operation

H.264 will operate on any picture size with dimensions which are multiples of 16. Both progressive and interlaced formats are supported; in fact, they may even be mixed together in the same video stream. Only the 4:2:0 chrominance sub–sampling format is supported.

An interlaced frame, which consists of two fields separated by half the frame period, can be coded as two separate **field pictures** or together as a **frame picture**. A progressive frame is coded as a single frame picture. A progressive frame can still be divided into two fields (top and bottom) if a subsequent field needs to reference a sub–field of that frame.

H.264 subdivides each picture into increasingly smaller units. The first division is the **slice**, then **macroblock**, then **block** and finally **pixel**. Pixels are never coded individually, however, individual pixel values are used in calculations such as motion prediction.

The slice is an independent coding unit. Within each picture there will be at least one slice. The maximum number of slices is the number of macroblocks in the picture. For example, if the QCIF[3] resolution is used then there will be 99 macroblocks; therefore, there can be as many as 99 slices. Each slice is completely self contained, so should a slice be lost in transit then all the received slices can still be fully decoded. More slice divisions will increase the number of parameters which need to be signaled, but the result will increase error resilience. Slices are an error control tool and they have little purpose beyond providing protection against channel errors. As such, slices will not be discussed in much detail.

The format of a macroblock is defined by the video format which is being used. Since the only currently supported format for H.264 is 4:2:0, the colour components are sub-sampled by 2 in the horizontal and vertical axes. Therefore, a macroblock for the 4:2:0 format contains a single 16 pixel by 16 line array of luminance data and two 8 pixel by 8 line arrays of chrominance data.

A block has a fairly loss structural definition. It is an N pixel by M line array of pixel values where N and M can equal 4 or 8. The largest block is 8x8 and the smallest is 4x4. Depending on the coding mode, a macroblock could contain 4 8x8 blocks of luminance data, 16 4x4 blocks or 6 8x4 blocks.

3.4 Macroblocks and Blocks

Even though the macroblocks are subdivided into smaller blocks, they are the basic unit the encoder uses for processing pictures.

3.4.1 Macroblock Conventions

If a macroblock in the current picture is skipped, the corresponding macroblock[4] from the preceding picture is copied into its place. Both *intra* and *inter* coded macroblocks can be skipped. If the current picture is a field from an interlaced picture then the macroblock is copied from the preceding field

[3] QCIF stands for Quarter Common Interchange Format, there are 144 pixels by 176 lines in a QCIF picture.

[4] The same (x,y) position.

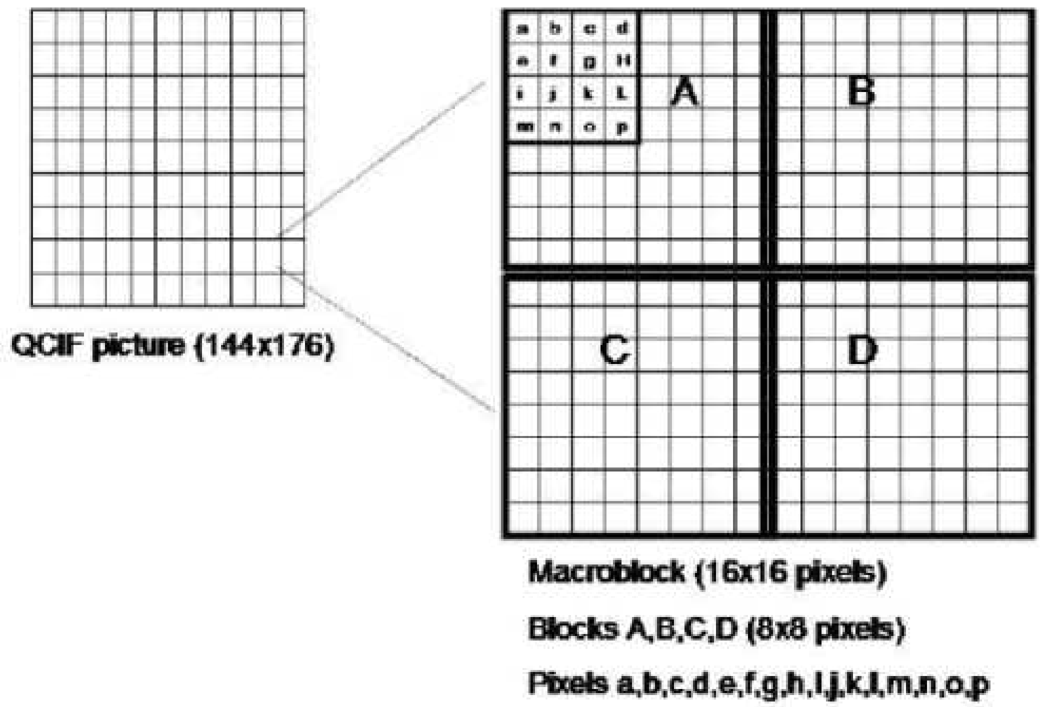

Figure 3.3: Example of a picture breakdown into 4x4 blocks

of the same parity (either top or bottom).

One quantizer is chosen for an entire slice. The quantizer is signaled by the QP value. QP can be changed on a per macroblock basis by an optional macroblock header parameter, delta–QP, which either increments or decrements QP. The advantage of assuming one quantizer for an entire slice is clear.

The decision as to whether to use *inter* or *intra* coding can be changed as often as every macroblock.

An *intra* macroblock can be skipped or coded in either *16x16 mode*, where the macroblock is not sub–divided into smaller blocks, or *intra 4x4 mode*, where the macroblock is sub-divided into 16 4x4 blocks.

An *inter* macroblock, which has not been skipped, will have **at least** one motion vector sent with it. Only one motion is sent if the macroblock is not further subdivided. If a macroblock is subdivided into two blocks of size 16x8 or 8x16 then two motion vectors will be sent, one for each sub–block. The macroblock header will indicate in which way it is divided.

If the *inter* macroblock is divided into 8x8 segments then a new situation arises. A further set of four codewords have to be sent, signaling the decoder how each of the 8x8 blocks are subdivided. An 8x8 block can be divided into an 8x8 block, two 8x4 blocks, two 4x8 blocks or four 4x4 blocks. If a macroblock is divided into 4 8x8 blocks and then each of those blocks is divided into four 4x4 sub–blocks then sixteen motion vectors would be required. Just to make things complicated, any 8x8 block in an *inter* macroblock can be individually coded in *intra* mode.

3.4.2 *intra* Mode

Intra coding uses pixel values from the current picture (or slice) to minimize the amount of information necessary to represent the block. There are two sets of prediction modes for *intra* coding. One set is *16x16 mode*, where there are four methods to choose from, while for the *4x4 mode*, there are nine available methods.

3.4.2.1 *Intra 4x4 mode* prediction for Luminance

H.264 provides nine prediction modes for *intra 4x4 mode*. Only three of the modes will actually be presented, the first two are unique enough to require individual presentation while the third mode is the most complex and all other prediction modes can be considered simple cases of this one.

Figure 3.4 provides the notation for the prediction equations. Values A–Q represent pixel values from decoded macroblocks on top and to the left[5] of the macroblock currently being decoded. Sometimes these values will not be accessible by the decoder. For example, if the current macroblock is on the top border of a picture then Q and A–H will not exist. These situations are dealt with as special cases in all the prediction modes. Values a–p represent the 16 pixel values in one of the 16 4x4 blocks in the current macroblock.

[5] Top and left are used because of the scanning order.

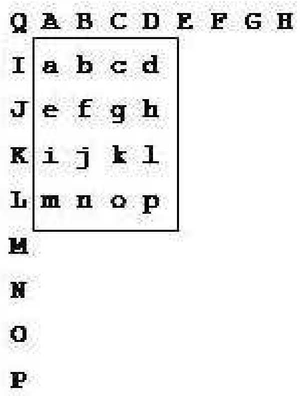

Figure 3.4: Pixel Notation

3.4.2.2 Prediction Mode 0: DC prediction

If all the samples A, B, C, D, I, J, K, L exist in the current picture (or slice), then every sample in the 4x4 block is predicted using the same value. The average of the surrounding samples is taken. The predictor P is calculated:

$$P = \frac{A + B + C + D + I + J + K + L + 4}{8}. \tag{3.1}$$

Note the division is accomplished by left–shifting the bits of P by 3. If A, B, C, D are outside the picture and I, J, K, L are not or if I, J, K, L are outside the picture and A, B, C, D are not then:

$$P = \frac{I + J + K + L + 2}{4} \quad \text{or} \quad P = \frac{A + B + C + D + 2}{4}. \tag{3.2}$$

If all A, B, C, D, I, J, K, L samples are outside the picture or slice, then all samples in the block are predicted by the value 128. This situation corresponds to the top–left macroblock of a picture.

3.4.2.3 Prediction Modes 1 & 2: Vertical and Horizontal Prediction

In vertical prediction, the top samples A, B, C, D are used to predict the columns beneath. For example, a, e, i, m are predicted by A. The standard does not specify what happens if mode 1 is used and A, B, C, D are not in the field. Hopefully an encoder would be implemented so that this situation would not happen or the pixel values would be predicted by 128.

There is a matching Mode 2: Horizontal Prediction where the I, J, K, L pixel values are used to predict rows of pixels.

3.4.2.4 Prediction Mode 7: Horizontal–Up prediction

This mode can only be picked when $A, B, C, D, I, J, K, L, Q$ are accessible.

$$\begin{aligned}
\tilde{a} &= \frac{B + 2C + D + 2I + 2J + 4}{8} \\
\tilde{b} &= \frac{C + 2D + E + I + 2J + K + 4}{8} \\
\tilde{c}, \tilde{e} &= \frac{D + 2E + F + 2J + 2K + 4}{8} \\
\tilde{d}, \tilde{f} &= \frac{E + 2f + G + J + 2K + L + 4}{8} \\
\tilde{g}, \tilde{i} &= \frac{F + 2G + H + 2K + 2L + 4}{8} \\
\tilde{h}, \tilde{j} &= \frac{G + 3H + K + 3L + 4}{8} \\
\tilde{l}, \tilde{n} &= \frac{L + 2M + N + 2}{8} \\
\tilde{k}, \tilde{m} &= \frac{G + H + L + M + 2}{4} \\
\tilde{o} &= \frac{M + N + 1}{2} \\
\tilde{p} &= \frac{M + 2N + Q + 2}{4}.
\end{aligned}$$

The pixel values used in the prediction equations lie roughly in line with the pixel which is being predicted. Having modes such as this allows the encoder to adapt to pictures with different visual characteristics. There are no specifications in the standard for choosing particular prediction modes. This is left to the implementors of the encoder, who could in fact, choose to ignore all the prediction modes except DC prediction for the sake of simplicity. There are explicit rules for signaling which prediction mode has been chosen.

3.4.2.5 *Intra 4x4 mode* Luminance Prediction Coding

Each of the 4x4 blocks will have its own prediction mode which needs to be sent to the decoder. This will require a relatively large number of bits. Each macroblock will have 16 4x4 blocks. If a fixed length code is used, 5 bits will be needed to signal which of the 9 modes are being used. That is 80 extra bits per macroblock which need to be sent. The solution to this problem is to use the observation that the prediction modes of adjacent blocks are highly correlated.

Using blocks A and B (see Figure 3.5–a), a sorted list of prediction modes can be produced for block C. The list is sorted in descending order by which prediction mode is the most probable. Instead of actually sending the prediction mode, the location of the mode in the sorted list is sent instead. At the decoder end, a table lookup is performed based on the prediction modes of A and B to obtain the correct sorted list for block C. In order to exploit as much correlation as possible between adjacent 4x4 blocks, the prediction modes of 2 4x4 blocks are coded together. Figure 3.5–b illustrates how the 4x4 blocks are grouped together.

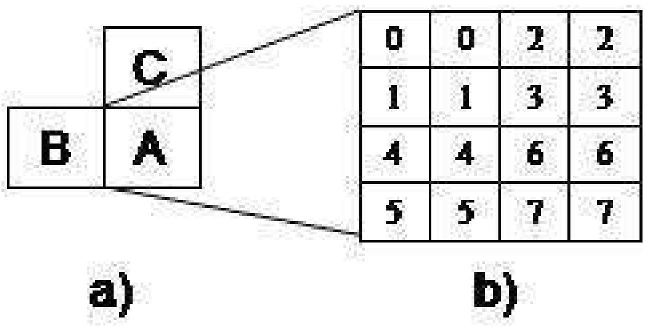

Figure 3.5: *Intra 4x4 mode* Prediction

3.4.2.6 *Intra 16x16 mode* Luminance Prediction

Recall that there are two ways of predicting an *intra* block, *4x4 mode* and *16x16 mode*. The prediction modes and coding rules have just been presented for *4x4 mode*.

When a macroblock is coded in *16x16 mode*, there are 4 available prediction modes. Three of the prediction modes are similar to *4x4 mode* prediction modes, vertical (Mode 0), horizontal (Mode 1) and DC prediction (Mode 2).

Prediction mode 3, Plane Prediction, will be presented next.

Let $P(i,j)$ denote the pixel values where $i,j = -1 \ldots 15$. Nonnegative indices are used to reference the pixels inside the 16x16 macroblock. Negative indices are used to reference pixels bordering the 16x16 block on the top and left sides. Therefore $P(-1,j)$ are the neighboring samples to the left and $P(i,-1)$ are the neighboring samples on top.

$$\text{let } G = (a + b(i-7) + c(j-7) + 16) >> 5 \tag{3.3}$$

where:

$$a = 16(P(-1,15) + P(15,-1)) \tag{3.4}$$

$$b = (5(\sum_{i=1}^{8} i(P(7+i,-1) - P(7-i,-1))) + 32) >> 6 \tag{3.5}$$

$$c = (5(\sum_{j=1}^{8} j(P(-1,7+j) - P(-1,7-j))) + 32) >> 6 \tag{3.6}$$

$$\tilde{\mathbf{P}}(\mathbf{i},\mathbf{j}) = \begin{cases} 0 & \text{if } G < 0 \\ G & \text{otherwise} \\ 255 & \text{if } G > 255. \end{cases}$$

The values b and c are a measure of how the border pixel intensities are changing along the top and left respectively. If one of these values is positive then that indicates there is a general trend towards an increase in pixel intensity from left to right or top to bottom. If b or c is negative then it would indicate the opposite. Pixels inside the block are then correspondingly given lower or higher values depending on their position. The values b and c will be 0 in two cases, if the bordering pixels are all the same intensity or if a decrease or increase in pixel intensity is mirrored about

the seventh top or left border pixel. If either of these cases occur for both b and c, the a value is used to predict all the pixel values. a is the mean of the top right and bottom left bordering pixels.

3.4.2.7 *Intra 16x16 Mode* Luminance Prediction Coding

The rules for coding the *16x16 mode* prediction modes are very different to those for coding the prediction modes for *intra 4x4 mode*. Three parameters are grouped together, prediction mode, AC, and the chrominance coded block pattern (*cbp*). Coding the prediction mode requires 2 bits to represent the 4 possible modes. AC is a flag which indicates whether or not there is at least one AC coefficient in one of the 16 4x4 blocks[6]. Only 1 bit is required for the AC flag. The chrominance *cbp* is a slightly more informative version of the AC flag, it can take on 3 possible values. It can indicate if there are no chrominance coefficients at all, if there are non-zero DC coefficients but all the AC coefficients are zero or if there may be non–zero DC coefficients but there is at least one non-zero AC coefficient.

3.4.2.8 *Intra* Coding for Chrominance

There is only one way of dividing and coding chrominance information which saves the need for signaling modes. Both of the 8x8 chrominance blocks in a macroblock are divided into four 4x4 blocks. A DC style prediction mode is used where adjacent samples on the left and top side of the chrominance block are used for the calculation. As usual, if the adjacent values are inaccessible, then 128 is used as the default prediction value.

3.4.3 *Inter* Mode

H.264 has the capability, in *inter* mode, to reference more than the last picture for motion prediction. This adds to the complexity, but increases the chance of finding a good match.

If the header information signals a frame picture[7], then the motion vector points to the corresponding frame in the reconstruction buffer which is either a frame or two successive fields reconstructed as a single frame.

If the header signals a field picture, the motion vector points to a corresponding field **of the same parity** which was either encoded on its own or with another field to form a frame.

The reference index parameter is signaled for every *inter* coded 16x16, 16x8 or 8x16 block. If 8x8 mode is being used, each **non *intra* coded** 8x8

[6]Even though the macroblock is coded in *16x16 mode* it is still divided into 4x4 blocks for the purposes of transforming the residual pixel values

[7]See section 3.2 for the distinction between frames and fields.

partition will have its own reference index parameter.

3.4.3.1 Motion Vectors

For each of the motion vectors $\vec{v}$ which are sent, a prediction $\tilde{p_i}$ $i = \{0, 1\}$ is made independently for both the horizontal and vertical components. The residual, $\vec{r} = \vec{v} - (\tilde{p_0}, \tilde{p_1})$, is the value which is sent across the channel. Motion vectors are allowed to point outside the field[8]. When a motion vector references pixels outside the picture (or slice), the nearest pixel on the border is substituted.

It should be noted that the motion vector predictions are formed independently of the algorithm which searches for the best match. This results in a lower complexity for the decoder. Once a motion vector has been calculated using any algorithm, the prediction is subtracted. At the decoder, the only calculation needed is to add the prediction back to the residual value. The vector will then point to the relevant (x, y) position in the appropriate picture. H.264 does not specify which algorithm should be used for searching for the best match. Since this portion of the standard is computationally intensive, it allows developers the freedom to use the best algorithm which suits the available hardware.

3.4.3.2 Motion Vector Prediction

All but the motion vectors for 8x16 and 16x8 are predicted, one component at a time, using **median prediction**. The prediction is formed by taking the median of the surrounding motion vector components A, B, C and D (see Figure 3.6). The prediction is calculated one vector component at a time. A, B, C, D are motion vector components which **could** be referencing different pictures in the picture buffer. For example, the reference index for E could be pointing to the third picture back and A, B, C and D could all point to the most recent picture. In this case A, B, C and D cannot be used and the value of E will be predicted by a default value. If only one of A, B, C or D point to the same reference picture as E then it is used to predict E.

[8] This was an optional annex in H.263 called **Unrestricted Motion Vector Mode.**

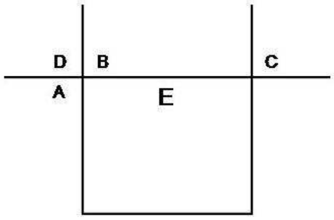

Figure 3.6: Median Prediction Notation

If A and D are outside the picture, their values are replaced by 0 and they are marked as having a different reference picture than E. If D, C, B are outside the picture, A is used as the predictor. If C is outside the picture, it is replaced by D in the median calculation. If any of A, B, C or D are *intra* coded, they are marked as having a different reference picture as E.

For predicting 8x16 and 16x8 blocks, the following rules are used. For 8x16 blocks, the left side is predicted by A, if A refers to the same picture, otherwise median prediction is used. For the right side, C is used, if it refers to same picture, otherwise median prediction is used. The same rule is applied to 16x8 blocks substituting B for predicting the upper block and A for the lower block.

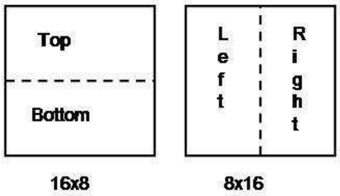

Figure 3.7: 16x8 and 8x16 Block Division

3.4.3.3 *Inter* Mode Coding

A coded bit pattern is also sent for each *inter* coded macroblock. It contains information on which of the luminance and chrominance 8x8 blocks[9] contain

[9] This is an independent division of the macroblock from the N, M division which has already been chosen.

non-zero transform coefficients. Four bits are used to signal which of the four luminance blocks contain at least one 4x4 block with at least one non-zero coefficients. A 0 in one of the bit positions would indicate that a particular 8x8 block has no non-zero coefficient. For the chrominance data, a flag, *nc*, which can take on three values is used.

If *nc*=0 – There are no non–zero chrominance coefficients.

If *nc*=1 – There is at least one non-zero DC chrominance coefficient but all AC coefficients are still zero.

If *nc*=3 – There may be non–zero DC coefficients but there is at least one non–zero AC coefficient.

3.4.3.4 Predicting *Inter* Chrominance

There is no independent process for determining the chrominance vector. Rather, once the luminance motion vector has been determined the chrominance vector can be determined by dividing the vector components by two. This is done because the chrominance samples are sub-sampled by 2. The advantage of this system is the chrominance vector does not need to be transmitted.

In summary, Table 3.1 lists block types with the maximum number of coefficients which can be stored in each.

3.5 Decoding Process

3.5.1 Slice Decoding

If the header of the slice has a different picture number from the previously decoded slice, the decoding of a new picture is started, otherwise the slice is added to the current picture. All the default values for the slice are loaded into memory, such as the slice quantizer pointer.

Block Type	max_coeff
luminance DC blocks for *intra 16x16 mode*	16
luminance AC blocks for *intra 16x16 mode*	15
luminance DC block for *intra 4x4 mode*	16
luminance AC block for *inter 4x4 mode*	16
U-Chrominance DC block for *intra*	4
V-Chrominance DC block for *intra*	4
U-Chrominance DC block for *inter*	4
V-Chrominance DC block for *inter*	4
U-Chrominance AC block for *intra*	15
V-Chrominance AC block for *intra*	15
U-Chrominance AC block for *inter*	15
V-Chrominance AC block for *inter*	15

Table 3.1: All block types

3.5.2 Multi-Picture Handling

In order to accommodate multiple picture referencing, the decoder operates a multi-picture buffer (MPB). This buffer is partitioned into two parts. The first partition holds **short–term pictures** and the second partition holds **long–term picture**. Pictures coming into the encoder are automatically put into the short–term buffer. By default the size of the long–term buffer is 0. Only by explicit signaling can the long–term buffer be opened and assigned a size and only by explicit signaling can a picture be assigned long–term status.

The size of the long–term buffer can be changed with any frequency. The standard specifically states it has no recommendations on how often this should be done. When the long–term buffer size is changed all long–term pictures with long–term indices greater than the new size of the buffer are marked as unused (they are deleted). If the encoder wishes to ensure a long–term picture will not be deleted when a change of buffer size is signaled it should assign the picture a small index.

All incoming pictures have a **picture number** PN which is assigned by the encoder. For each successive picture, the encoder increments PN by one and copies that value into the picture's header. This operation is done modulo MAX_PN[10]. The decoder can use this value to check if any pictures have been lost in transmission. If an error is detected, the decoder could possibly ask for retransmission depending on the application. PN will be the unique identifier of the picture within the MPB. Therefore, when a picture arrives with a PN value the same as another in the buffer[11] the older picture is removed.

Once a picture arrives in the multi–picture buffer it is assigned another index, its short–term relative index or MPB–index. This number is used to determine the age of the picture. By default the oldest picture is discarded in favour of the new pictures. When a picture is too old, the space it is occupying in the buffer is marked as unused for incoming pictures to use. The default MPB–index for every new picture is 0. MPB–indices are only changed when pictures are added or removed from the MPB. Pictures are removed from the MPB if they are too old or if instructions have been signaled to have it moved to the long–term buffer.

When a new picture comes into the decoder there are two ways of removing short–term pictures to make space. These two methods can be switched as often as every picture.

The first method, **Sliding Window**, has the buffer acting as a first–in–first–out queue. The oldest picture is removed if the encoder determines there is not enough room in the buffer for the incoming picture which is assigned the short–term index of 0 and all other indices are incremented by one.

The second method, **Adaptive Memory Control**, allows the use of commands which are signaled from the encoder with explicit instructions specifying:

1. which short–term pictures are to be removed from the buffer,
2. which pictures are to be assigned long–term status,
3. how many long–term pictures are allowed,

[10] There are no recommendations for setting the size of MAX_PN or if its value can be changed.

[11] Because of the modulo nature of PN.

4. which pictures should be marked as unused,
5. if the entire buffer should be cleared out.

Once the appropriate reference block has been found in the buffer, by adding on the prediction values to the residual motion vector, the process of reconstructing an **approximation** of the original block begins.

3.5.3 Pixel interpolation

The decoder first has to check to see what level of pixel interpolation was used, quarter or eighth. The referenced block in the multi–picture buffer is then expanded using interpolation filters and averaging. The filters and averaging calculations have been chosen to limit calculations to addition and bit–shifting. There is no option to use only half–pixel precision.

3.5.3.1 Quarter–Pixel Interpolation

If quarter pixel interpolation is used, the half–pixel samples are calculated by applying a 6–tap filter F (1,−5,20,20,−5,1)[12] to original pixel values. The quarter–pixel values are then calculated by averaging the half–pixel and original pixel values.

In Figure 3.8, A refers to the original pixels, all lower case values refer to the newer interpolated pixels which are obtained in the following manner:

b^h is calculated by applying F to the nearest 6 original pixel values in the horizontal direction. The result of this calculation, b, is then normalized in the following manner:

$$b^h = \frac{b+5}{32}. \tag{3.7}$$

b^v is calculated by applying F to the nearest 6 original pixel values in the vertical direction. The result of this calculation, b is then normalized in the same way as b^h.

[12] Observe: the sum of these taps is 32, therefore normalization consists of a right bit shift by 5.

c^m is calculated by applying F to the 6 nearest intermediate values b. The result, c, is then normalized in the following manner:

$$c^m = \frac{c + 512}{1024}. \tag{3.8}$$

d,g,e are quarter precision pixels which are obtained by using bilinear interpolation on the nearest values in integer or half sample positions.

$$d = \frac{A + b^h}{2} \quad g = \frac{b^v + c}{2} \quad e = \frac{b^h + c^m}{2}. \tag{3.9}$$

h is calculated by averaging the nearest b^h and b^v values in diagonal directions.

$$h = \frac{b^h + h^v}{2}. \tag{3.10}$$

i is calculated by averaging the nearest four integer pixel values.

A	d	b^h	d	A
e	h	f	h	e
b^v	g	c^m	g	b^v
e	h	f	i	e
A	d	b^h	d	A

Figure 3.8: Quarter Pixel Resolution

3.5.3.2 Eighth–Pixel Interpolation

Although eighth–pixel interpolation is not part of either the baseline or main profiles it will be briefly presented next to give an idea of the additional precision which can be obtained. Eighth–pixel interpolation is performed in the same manner as quarter–pixel. The quarter pixel values are calculated using 8–tap filters where:

$\frac{1}{4}$ **position** $(-3, 12, -37, 229, 71, 21, 6, -1)$,

$\frac{2}{4}$ **position** $(-3, 12, -37, 158, 158, -39, 12, -3)$,

$\frac{3}{4}$ **position** $(-1, 6, -21, 71, 229, -37, 12, -3)$.

The eighth pixel values are calculated using averaging of integer, half and quarter pixel values. Figure 3.9 illustrates how many new pixels are added when eighth–pixel interpolation is used.

A	d	b^h	d	b^h	d	b^h	d	A
d	e	d	f^h	d	f^h	d	e	d
b^v	d	c^q	d	c^q	d	c^q	d	b^v
d	f^v	d	g	d	g	d	f^v	d
b^v	d	c^q	d	c^m	d	c^q	d	b^v
d	f^v	d	g	d	g	d	f^v	d
b^v	d	c^q	d	c^q	d	c^q	d	b^v
d	e	d	f^h	d	f^h	d	e	d
A	d	b^h	d	b^h	d	b^h	d	A

Figure 3.9: Eighth Pixel Resolution

3.5.3.3 Chrominance Interpolation

Quarter and eighth precision chrominance pixels are interpolated by taking weighted averages of the distance from the new pixel to four surrounding original pixels. Equation (3.11) is used for calculating every interpolated chrominance pixel, the notation can found in Figure 3.10.

$$v = \frac{(s-d^x)(s-d^y)A + d^x(s-d^y)B + (s-d^x)d^yC + d^xd^yD + \frac{s^2}{2}}{s^2}. \quad (3.11)$$

3.5.4 Scanning Order

The residual coefficients are variable length coded using a 2–D run–length coding scheme. The first component is the number of zeros encountered before the first non–zero coefficient and the second component is the magnitude of this coefficient. An additional, non 2–D, symbol is also sent which represents an End Of Block (EOB). If EOB is encountered then the decoder knows there are no more non–zero coefficients in the current block of pixels.

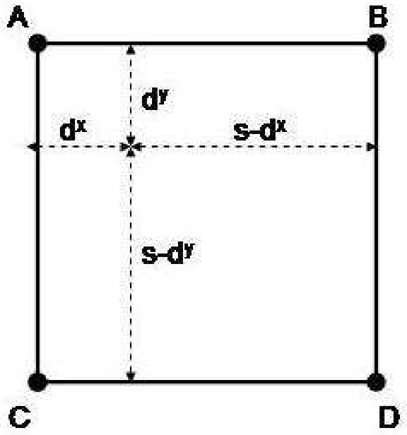

Figure 3.10: Chrominance Interpolation

If a block contains no non-zero coefficients, it can be represented by one EOB symbol.

Once the run–length symbols have been decoded, the bitstream is divided into a series of individual transform coefficient values. The series is then transposed into a 4x4 array using zig–zag scanning illustrated in Figure 3.11.

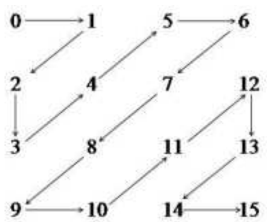

Figure 3.11: Zig–Zag Scan

The zig-zag method of placing the coefficients is used for all types of encoding with some small variations. If a block has been coded in*intra 16x16 mode*, the 16 DC coefficients from each of the 16 4x4 blocks are grouped together in their own 4x4 block and scanned separately from the AC coefficients. There are at most 15 AC coefficients left to scan in each of the 16 4x4 blocks.

Any of the 4x4 blocks which contain AC coefficients are then zig–zag scanned starting in the second position rather than the first because the first coefficient contains the DC value.

The 2x2 array of DC chrominance coefficients are scanned in raster order. The 4x4 chrominance blocks with at least one AC coefficient are then scanned in zig–zag order starting at the second position.

3.5.5 Inverse Quantization and Coefficient Transformation

The 4x4 blocks of coefficients are now dequantized, reconstructed transformed back into the pixel domain and prediction values are added on. These operations can happen in varying order.

The process of scaling the coefficient values is tied to the quantization step. Instead of explicitly applying a quantizer, the coefficient value is first scaled then a certain number of bits are shifted out to the left.

Left shifting the bits is equivalent to the lossy approximation step in quantization because precision is lost. If the binary representation of a coefficient is left–shifted by 2^n bits then in the inverse quantization step it will be right–shifted by 2^n. The result is that 2^n distinct values are all mapped to one value. The number of bits to shift is determined by the value of QP .

H.264 uses a 6 bit number QP, to signal which quantizer should be used. Although there is room for 64 distinct quantizers to be signaled, the value of this 6 bit number only ranges between -26 and 25. The signaled value corresponds to $QP_Y - 26$. QP_Y points to which quantizer should be used for luminance data and it ranges in value from 0 to 51. QP_C points to which quantizer should be used for the luminance components. To determine which quantizer to use for the chrominance components a simple table lookup is used based on the value of QP_Y. **From here forward**, QP shall refer to QP_Y when referring to luminance samples and QP_C when referring to chrominance samples.

There are three distinctions which are made by the decoder as to which kind of block is being decoded; luminance DC coefficients from an intra 16x16 mode macroblock, chrominance DC coefficients and residual 4x4 coefficients.

There are two seperate transforms which are used for returning to the pixel domain. Both the transforms are integer based to avoid any extra errors due to rounding. Note that this means that most of the signal noise is now concentrated in the quantization stage. Previously, in H.263, the DCT transform was used which incurred the rounding mismatch error between the encoder and decoder. The transforms can be calculated without any multiplication. Addition and bit shifting are the only operations which are needed.

3.5.5.1 *Intra 16x16 mode* DC luminance Transform

In the case of the luminance DC coefficients from an *intra* coded block the discrete 4x4 Walsh-Hadamard transform is used. Hence the mathematical equivalent of the operations performed by the decoder is the following:

$$\mathbf{X_{QD}} = \mathbf{H}^{\mathrm{T}} \begin{bmatrix} Y_{QD00} & Y_{QD01} & Y_{QD02} & Y_{QD03} \\ Y_{QD10} & Y_{QD11} & Y_{QD22} & Y_{QD13} \\ Y_{QD20} & Y_{QD21} & Y_{QD22} & Y_{QD23} \\ Y_{QD30} & Y_{QD31} & Y_{QD32} & Y_{QD33} \end{bmatrix} \mathbf{H} \tag{3.12}$$

where:

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \tag{3.13}$$

Once the data has been transformed back into the pixel domain, the values need to be rescaled and dequantized. This is done according to the following operation[13]:

$$X_D(i,j) = (X_{QD} \times R) << (\frac{QP}{6} - 2). \tag{3.14}$$

The value of R is obtained by performing a table lookup in row X of Figure 3.13 based on the value of (QP mod 6).

[13]Note: the calculations for scaling the value of X_{QD} require the use of multiplication and division operations which can not be performed by bit–shifting.

3.5.5.2 DC Chrominance Transform

Instead of a 4x4 block, there are two 2x2 blocks of DC coefficients, one for each colour component. The discrete 2x2 Walsh-Hadamard transform is used in this case. The scaling equations are:

$$X_D(i,j) = (X_{QD} \times R) << (\frac{QP}{6} - 1). \tag{3.15}$$

3.5.5.3 Residual 4x4 Transform

In the case of these blocks, rescaling and dequantization is performed before transformation. The block of residual values Y_Q is first partitioned into three sections X, Y, Z according to Figure 3.12. Let $G(i,j)$, $i,j = 0 \ldots 3$, be a function which returns the region (either X, Y or Z) that (i,j) belongs to. For example, $G(2,3) = Z$.

$$\begin{bmatrix} X & Z & X & Z \\ Z & Y & Z & Y \\ X & Z & X & Z \\ Z & Y & Z & Y \end{bmatrix}$$

Figure 3.12: Block division

The coefficients are then scaled according to which region (X, Y, Z) they belong to and the value of QP.

$$Y'(i,j) = (Y_Q \times R(QP, G(i,j)) << (\frac{QP}{6}). \tag{3.16}$$

R is a function, which given QP and the partition value will return the appropriate scaling value. There are six distinct triplets (see Figure 3.13) of scaling factors which are used. The three values in the triplet correspond to the X, Y, Z partition.

Once Y' has been calculated, its values are then transformed using a 1 dimensional transform $T4$. First each row, $\vec{y}$, of Y' is transformed, then each column of the resulting matrix. $T4$ has the form of equation (3.17):

$QP \bmod 6$	(X, Y, Z)
0	$(10, 16, 13)$
1	$(11, 18, 14)$
2	$(13, 20, 16)$
3	$(14, 23, 18)$
4	$(16, 25, 20)$
5	$(18, 29, 23)$

Figure 3.13: Scaling triplets

$$\mathbf{T4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix}. \tag{3.17}$$

The entries in the resulting matrix X are then scaled and shifted according to the following:

$$X'(i, j) = (X(i, j) + 2^5) >> 6 \tag{3.18}$$

The predicted values are added using either motion vectors, in the case of *inter* coding or prediction mode in the case of *intra* coding. Finally, the numbers are clipped to between 0 and 255.

3.5.6 Deblocking Filter

H.264 uses an in–loop deblocking filter on the edges between the 4x4 blocks. The rules for applying the filter are very intricate and provide little insight into how the filter works. Pages 68–73 in [6] detail the exact rules for applying the deblocking filter.

The filtering is done on a per macroblock basis. First, the 3 vertical edges between 4x4 blocks are filtered, then the 3 horizontal edges. Finally, the top and left side of the macroblock are filtered. The bottom and right side are not filtered because those blocks have not yet been decoded due to the scanning order.

A boundary strength is then calculated between each 4x4 block. This boundary strength is determined based on several properties of the block such as whether the block has been *intra* coded, if there are coefficients, if the block boundary is also a macroblock boundary, etc.... Depending on the boundary strength and other parameters obtained from an intricate table lookup scheme, one of two filtering schemes is then applied. In the first scheme up to two coefficients on either side of the border can be updated. In the second scheme, up to three coefficients on either side can be updated.

3.5.7 Entropy Encoding

3.5.7.1 Default Scheme

The default entropy encoding mode for H.264 uses a **universal variable length code** (UVLC) table. The table is represented in compressed form in Figure 3.14 where the value x_i can take on the value 1 or 0:

$$
\begin{array}{c}
1 \\
0\ 1\ x_0 \\
0\ 0\ 1\ x_1\ x_0 \\
0\ 0\ 0\ 1\ x_2\ x_1\ x_0 \\
0\ 0\ 0\ 0\ 1\ x_3\ x_2\ x_1\ x_0 \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots
\end{array}
$$

Figure 3.14: Default UVLC Table

Figure 3.14 specifies an instantaneously decodable code. Figure 3.14 could easily be rewritten as a binary tree with the codewords attached as leaves. Alternatively, Figure 3.14 could be represented in table format, Table 3.2. The codeword length l is $2n+1$ where the $n+1$ trailing bits are the INFO bits. Given l and INFO a simple formula can be used to find the **codeword number** and hence the associated value.

The UVLC can be used for encoding any parameter, macroblock type, 8x8 mode, motion vector data, coded bit pattern, any type of transform coefficients or *intra* prediction mode. Each set of parameters is associated with a set of codeword numbers { 0,1,2,...} through a 1 : 1 mapping. A

Codeword_Number	Codeword
0	1
1	010
2	011
3	00100
4	00101
5	00110
6	00111
7	0001000
⋮	⋮

Table 3.2: Explicit form of first codewords

simple formula (3.19) can then be used to decode the codewords by linking through their codeword numbers.

$$codeword\_number = 2^{\lfloor \frac{L}{2} \rfloor} + \text{INFO} - 1 \tag{3.19}$$

For example, in Table 3.2 codeword 6 is 00111. The length is 5 and the value of INFO is 11 in binary or 3 in decimal. Therefore $6 = 2^{\frac{5}{2}} + 3 - 1 = 4 + 2$.

It is instructive to observe the way in which different parameter values are assigned codeword numbers. The shortest codewords will be assigned to the most probable symbols.

Intra and *Inter* Macroblock Types

There are two *intra modes*, *4x4* and *16x16*. The information which has to be signaled for *4x4 mode* is its mode indicator, the prediction mode and residual coefficient values. Codeword number 0 (the shortest code) is assigned to the case of *4x4 mode*. Residual coefficient information and prediction mode are separate parameters and have their own tables. For *16x16 mode* there are three parameters which have to be signaled, see section 3.4.2.7. The rest of the code table is devoted to representing all possible combinations of the three parameters for *16x16 mode*. Codewords 1–4 are reserved for signaling the four possible prediction modes for *16x16 mode* block with no

DC coefficients and no AC coefficients. The next four codewords, 5–8, signal the four possible prediction modes where there are non–zero DC coefficients but all AC coefficients are 0. Continuing in this manner, there are 24 possible parameter combinations which can be distinguished for *intra 16x16 mode* macroblocks.

The first six codewords for *inter* coding are used for signaling the different block divisions, 16x16, 16x8, 8x16, etc.... If any of the 8x8 blocks are *intra* coded, *intra 4x4 mode* is used to code the four 4x4 blocks and the *intra 4x4 mode* signaling format is used.

Motion Vectors and QP Change

The components for motion vectors and delta-QP share the same code table. The codewords are assigned according to the ascending magnitude of the parameter. A positive value receives the first codeword and a negative value receives the next. It is therefore possible that two symbols which have equal magnitude will

be assigned codewords of a unequal length because the positive symbol was given the last possible codeword of a smaller length. The idea is that a large magnitude will rarely be sent. For both motion vectors and delta-QP this assumption would be correct because it is the residual motion vector which is signaled. For delta-QP the difference for a specific macroblock from the rest of the slice should never be very large.

Run and Level

The first codeword is assigned to the EOB symbol which has to be sent for every macroblock. The rest of the symbols are arranged in the following order 1) sign of level, 2) Run Value (ascending), 3) magnitude of Level (ascending).

There is another, slightly more involved way of coding the transform coefficients. Tables are still used to derive the codewords, but there is at least some context adaptation to reduce the bit–rate. The next more complicated/efficient option which is available for entropy coding is the Context Adaptive Binary Arithmetic Coding, CABAC, which will be presented in Chapter 4.

3.5.7.2 Context–Adaptive VLC for Transform Coefficients

This is an optional mode, used only for coding transform coefficients. It will increase the encoder complexity but should reduce the bit rate. The coefficient information is broken into three parameters

1. Num–Trail signals the number of trailing 1s (T1s)[14] and the number of non–zero coefficients in the 4x4 block. Only the sign needs to be decoded for the T1s. The number of T1s is clipped to 3.
2. Level information is decoded for all coefficients except the T1s.
3. Run information.

Coefficient data is decoded in reverse zig–zag order. First the signs of the T1s, then level information when it is first needed and run information. The run information is divided into two components, first the total number of zeros and second the sequence of the numbers of zeros between each coefficient.

There are three luminance VLC, one DC chrominance VLC and one FLC tables which can be used for coding num–trail. The blocks on the top and to the left of the block currently being decoded are used for determining which table to use. Table 3.3 details how the calculations are carried out.

Upper Block Present?	Left Block Present?	N
Y	Y	$\frac{N_u+N_l}{2}$
Y	N	N_u
N	Y	N_l
N	N	0

$$\begin{aligned} \text{First VLC table used if } 0 &<= N < 2 \\ \text{Second VLC table used if } 2 &<= N < 5 \\ \text{Third VLC table used if } 5 &<= N < 8 \\ \text{FLC table used if } N &>= 8 \end{aligned}$$

Table 3.3: VLC Table Choice

[14] When there are any non–zero coefficients in a 4x4 block there tends to be several high frequency coefficients that are ± 1, see [6]

The level information can be signaled using one of five possible tables which roughly approximate a context adaptive scheme. The first table, Table 0, is designed for the case when mainly low magnitude coefficients are present. As the table number gets higher, the corresponding binary tree becomes more balanced. Table 3.4 contains the first three entries for tables 0,2 and 5. Tables 1,2,3,4 were designed for the case where there is a high probability of large magnitude level values.

Lev-VLC0

Code no	Code	Level
0	1	1
1	01	-1
2	001	2
$\vdots$	$\vdots$	$\vdots$

Lev-VLC2

Code no	Code	Level
0–3	1XX	±1 to ±2
4–7	01XX	±3 to ±4
7–11	001XX	±5 to ±6
$\vdots$	$\vdots$	$\vdots$

Lev-VLC5

Code no	Code	Level
0–15	1XXXX	±1 to ±8
16–31	01XXXX	±9 to ±16
32–47	001XXXX	$\pm17$to ±24
$\vdots$	$\vdots$	$\vdots$

Table 3.4: VLC tables for coding Level information

The tables are selected using a simple algorithm which applies to both *intra* and *inter* coded macroblocks.

IF QP $\geq$ 21

Decode the first coefficient with VLC0

Decode second coefficient with VLC1
Increase VLC by one (up to 2) if $|\text{Level}| > 3$
ELSE
IF (number of coefficients > 10)
Decode the first coefficient with VLC1
Decode the second coefficient with VLC2
ELSE
Decode the first coefficient with VLC0
Decode the second coefficient with VLC1
IF (vlc == VLC1)
increase to VLC2 if $|\text{Level}| > 3$
IF (vlc $\geq$ VLC2)
Increase VLC by one (up to 4) if $|\text{Level}| > 5$

Run information is divided into *totalzeros* and run lengths. *Totalzeros* is the sum of all the zeros before the last coefficient in a forward scan through the 16 coefficients. For example, given 0040008050003000, totalzeroes = 2+3+1+3 = 9.

Since the number of non–zero coefficients is already known, this information can be used to limit the possible number of zero coefficients. For example, if there are 7 non–zero coefficients, there can not be more than 9 zeros before the last coefficient. There are 15 possible tables for coding the *totalzeros*. The number of non-zero coefficients[15] determines which table to use.

The run size before the next non–zero coefficient is signaled using one of seven VLC tables. The number of zeros remaining to distribute, zerosleft, starts at *totalzeros* and is decremented appropriately as each non-zero coefficient is decoded. The value of zerosleft is used to choose which VLC table should be used. For example, if there is only 1 zero left to distribute then the size of the run before the next coefficient is either 0 or 1 which requires only 1 bit to signal.

[15] Signaled in num–trail.

3.6 Hypothetical Reference Decoder

The hypothetical reference decoder (HRD) models the channel and decoder buffer. It acts as a throatle on the numbers of bits being released to the channel. There are three parameters which specify the HRD, the channel rate R, the buffer size B, and the initial buffer fullness F in bits. The HRD parameters represent the available resources of the decoder, bandwidth, buffer space and delay respectively.

The HRD starts with an initially empty buffer. At time t_{start}, the buffer begins to fill with bits at an instantaneous bit rate $R(t)$. When $\int_{t_{start}}^{t_0} R(t)dt = F$, t_0 is identified as the decoding time of the first picture. It can be seen by equation (3.20) how F measures decoder delay.

$$\text{Time until the first picture can be decoded} = \frac{F}{\frac{1}{t}\int_{t_{start}}^{t_0} R(t)dt}. \tag{3.20}$$

The HRD can be easily described using a leaky bucket (LB). A leaky bucket leaks at a rate R_1 bits/second, the bucket has a size B_1 bits and it has an initial fullness of $B_1 - F_1$ bits.

The LB parameters are signaled in the header of the bitstream. They specify the decoder resources which are necessary to completely decode the bitstream in real time. A bitstream may be contained in any number of leaky buckets. The encoder can specify any number of distinct LBs in the bitstream, $(R_1, B_1, F_1), \ldots, (R_N, B_N, F_N)$ where the R_i's are in strictly increasing order and the B_N's and F_N's are in strictly decreasing order.

In the context of the HRD, a bitstream is completely characterized by the 2–D integer sequence, $(t_0, d_0), (t_1, d_1), \ldots$, where t_i is the decoding time of picture i and d_i is the numbers of bits making up picture i.

A bitstream is contained in a given LB if the bucket doesn't overflow under the following conditions:

1. The bucket starts out with $B_1 - F_1$ bits.
2. At time t_0, d_0 bits are inserted into the bucket in addition to the $B_1 - F_1$ bits.
3. The bucket begins to drain at a rate of R_1 bits/second.

4. If the bucket empties, then it will stop draining and remain empty until the next insertion

5. At time t_i $i \geq 2$, d_i bits are inserted into the bucket which will drain at rate R_1

In other words, the initial bucket fullness is defined by equation (3.21) and the number of bits in the buffer at time t_{i+1} is defined by equation (3.22).

$$b_0 = B_1 - F_1 \tag{3.21}$$

$$b_{i+1} = max\{0, b_i + d_i - R(t_{i+1} - t_i)\}. \tag{3.22}$$

The bucket will not overflow if $b_i + d_i \leq B_1 \quad \forall i \geq 0$.

As mentioned above, the encoder can specify several different LBs to contain the bitstream. When the decoder receives the bitstream over a channel with rate R, it must check if its HRD parameters, R,B and F are sufficient to decode the bitstream. This is done through linear interpolation and extrapolation. Because of the enforced ordering in the bitstream, the sequence of LBs form a convex curve as seen in Figure 3.15. The decoder will test to see if the point (R, B) resides above the curve.

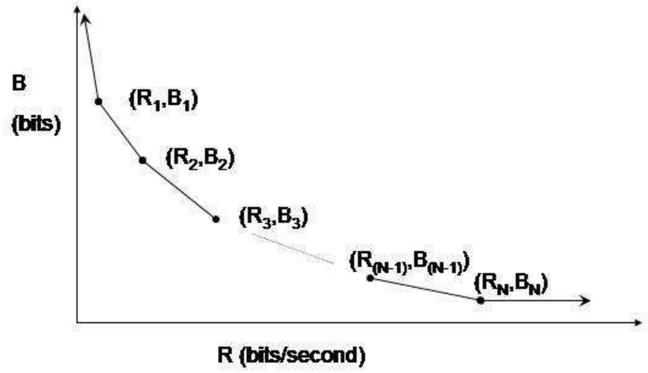

Figure 3.15: Leaky Bucket

If $R_i \leq R \leq R_{i+1}$ then the HRD will not overflow or underflow if $B \geq B_{min}(R)$ and $F \geq F_{min}(R)$.

$$B_{min}(R) = \alpha B_n + (1 - \alpha) B_{n-1} \tag{3.23}$$

$$F_{min}(R) = \alpha F_n + (1 - \alpha) F_{n-1} \tag{3.24}$$

$$\alpha = \frac{R_{n+1} - R}{R_{n+1} - R_n}. \tag{3.25}$$

If $R < R_1$

$$\begin{aligned} B_{min}(R) &= B_1 + (R_1 - R)T && (3.26) \\ F_{min}(R) &= F_1. && (3.27) \end{aligned}$$

Where T is the time from the decoding of the first picture to the last picture.

If $R > R_n$

$$\begin{aligned} B_{min}(R) &= B_N && (3.28) \\ F_{min}(R) &= F_N. && (3.29) \end{aligned}$$

Alternatively, given a buffer size, B, the bitstream is decodable provided $R \geq R_{min}(B)$ and $F \geq F_{min}(B)$ where $B_n \geq B \geq B_{n+1}$.

$$\begin{aligned} R_{min}(B) &= \alpha R_n + (\alpha - 1)R_{n-1} && (3.30) \\ F_{min}(B) &= \alpha F_n + (\alpha - 1)F_{n-1} && (3.31) \\ \alpha &= \frac{B_{n+1} - B}{B_{n+1} - B_n}. && (3.32) \end{aligned}$$

If $B > B_1$

$$\begin{aligned} R_{min}(B) &= R_1 + \frac{(B - B_1)}{T} && (3.33) \\ F_{min}(B) &= F_1. && (3.34) \end{aligned}$$

If $B < B_N$ then the stream may not be decodable.

3.7 Analysis

3.7.1 Block Sizes

By grouping pixels in smaller block sizes, 4x8, 8x4 and 4x4, there is a higher probability that the best match in the previous frame will be found. A better match results in a residual block with low magnitude coefficient values. The extra signaling required to indicate the sub–division mode and motion vectors must be offset by the savings in transmitting the residual coefficient values.

The smaller block sizes also mean more searches must be made to determine the motion vectors. For example, with a QCIF image, there are

99 macroblocks or 396 8x8 blocks. If only 8x8 blocks are used and only one other previous picture is searched, 396 searches must be made. If 4x4 blocks are used then 1548 searches must be made. If 4x4 blocks are used and the four most recent pictures are searched then 6336 searches must be made. If the largest motion vector component allowed is 32 pixels, then 16382 subtractions have to be performed which results in approximately 1.0×10^8 operations which have to be performed for each picture. At 29.9 pictures per second this amount to 3.0×10^9 operations per second.

Smaller block sizes also lead to fewer blocking artifacts. Smaller blocks will result in better representation of edges in the picture.

3.7.2 Prediction Equations

The impact of the expensive searches is balanced by simple prediction operations for *inter* mode. The vector components are basically predicted in only one way. This is a marked difference from *intra* mode where, for *4x4 mode*, there are 8 available predictions modes, some of them very intricate. For example, prediction mode 7 requires 10 separate calculations for each 4x4 block. These expensive calculations are balanced by the lack of block searches which are not required in *intra* mode.

Inter mode seems ideal for the situation where computing power is readily available but bandwidth is limited. Conversely *intra* mode is best suited to situations where little computing power is available but bandwidth is plentiful. An intelligent encoder could use channel–traffic feedback to switch between *intra* and *inter* modes as necessary.

The coded bit pattern (*cbp*), which is sent for every *inter* block and *intra 16x16 mode* blocks can save a considerable number of bits. In the case of an *inter* macroblock, the four least significant bits are used to indicate which of the 8x8 blocks do not contain any non-zero coefficients. If all four bits are set to 0 then none of the 8x8 blocks contain transform coefficients. If these four bits were not used, 16 bits of EOB symbols would have to sent. This results in a savings of 12 bits for every empty macroblock. The *nc* flag, which occupies the 2 most significant bits of the *cbp*, can also save unnecessary EOB symbols for chrominance information. The case for nc=0 results in a savings of 9 bits because 10 bits would have to have been sent, 2 for the 2x2 DC blocks and 8 for the 4x4 AC blocks. The benefit of using the *cbp* is of course minimized when a macroblock does contain transform coefficients. The flags do not impose a heavy burden on the bitstream. No more than one byte is taken up for each macroblock.

3.7.3 Picture Referencing

The ability to reference more than the immediately surrounding pictures is a new feature in H.264. Each extra picture searched results in millions of extra operations. It is therefore not a feature to be used lightly. Multiple picture referencing (MPR) also results in an increase in signaling data. The reference parameter must be sent indicating which previous picture each motion vector is pointing to. MPR could also involve the use of the long–term buffer detailed in section 3.5.2.

The standard does not state the specific reason for using a long–term picture buffer. The reason can, however, be inferred. A long–term picture buffer would be very useful in the case when several pictures were repeated many times over a long period. For example, a news broadcast often consists of two distinct types of video, shots of the news reader sitting still reading the news and short clips of video for each news story. If one picture for each camera angle was coded in *intra* mode and saved in the long–term buffer, then every subsequent shot of the news reader would have an excellent reference frame, no new *intra* frames would be needed. The complexity involved in implementing such a system would be high by today's standards, because this would require and intelligent encoder to examine the bitstream and identify commonly occuring scenes.

3.7.4 Quantization Operation

The standard specifies that the values which result from dequantization[16] should be constrainted to $(-2^{15}, 2^{15}-1)$. It can be inferred that each transform coefficient is assigned at least 2 bytes.

[16]These values are usually transform coefficients.

There are 51 distinct values QP can take on, however this does not mean that there are 51 different quantizers to choose. Rather, the way the equations are defined, there are only 9 quantizers which operate by shifting out a differing number of bits. For each quantizer there are 6 possible values which can be used to scale the coefficient values.

Given that equations (3.14), (3.15) and (3.16) are used for performing the inverse quantization operation, the corresponding forward quantization must be of the form of equation (3.35).

$$X_{QD} = (\frac{X_D}{R(QP, G(i,j)}) >> (QP/6). \tag{3.35}$$

Note that for each increment of 6 for QP the scaling parameter halves. Since many of the scaling parameters are not powers of two, integer round off should be used to remove any mismatch between the encoder and decoder.

In order to help visualize how the quantizer works, Figures 3.16 and 3.17 show the result of applying the quantizer to luminance components without the additional scaling operations[17]. As the value of QP increases, many of the transform coefficient (TC) values are decimated to 0.

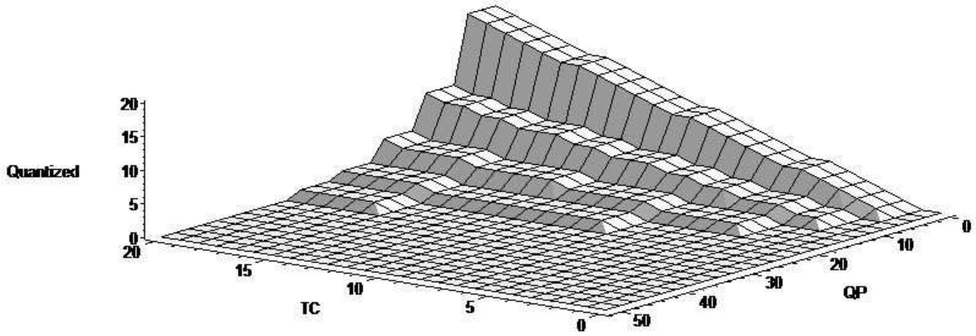

Figure 3.16: Zoomed view of H.264 Quantizer

[17]Since the scale in Figure 3.17 is too large to show fine detail, Figure 3.16 was included to show a zoomed view.

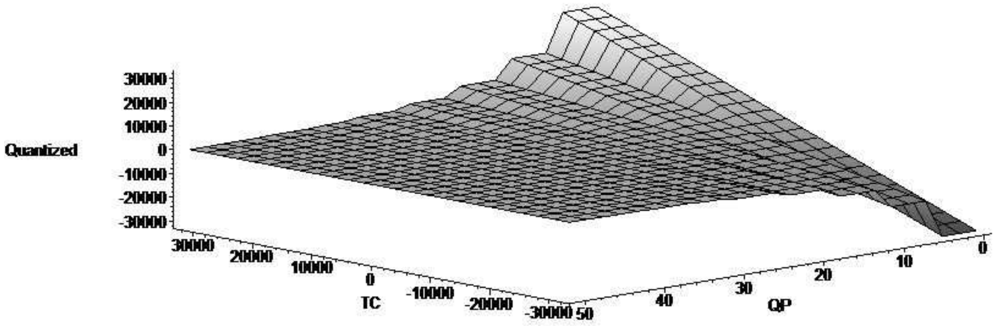

Figure 3.17: Full View of H.264 Quantizer

3.7.5 Transform Coding

Both the transforms used for H.264 are very simple and provide correspondingly less signal compaction.

In the case of residual AC blocks, three distinct scaling values have to be used. Recall, these residual blocks are transformed according to equation (3.36) and the resulting block is partitioned for scaling according to Figure 3.12. In order to understand why all this is done, it should be noted that the basis function of $T4$ have two distinct norms which results in three distinct norms for the two–dimensional transform of equation (3.36). These three distinct norms are distributed in $\mathbf{X}$ in the pattern of Figure 3.12.

It should also be noted that the scaling factors associated with one of norms of $T4$ are the same scaling factors which are used for the 4x4 Hadamard transform which shares the same norm.

$$\mathbf{X} = \mathbf{T4} * \mathbf{Y} * \mathbf{T4}^T. \tag{3.36}$$

3.7.6 Entropy Encoding

The default UVLC tables which are provided are not especially noteworthy. They can only have been designed with the most general image characteristics. For example, EOB is given the shortest code word for transmitting transform coefficient information. This must mean that EOB is the most frequently transmitted symbol in general. There is no ability to adapt to and take advantage of image characteristics.

The context adaptive VLC for transform coefficients allows a limited degree of adaptability however results in chapter 5 will show that the CABAC system provides a much better compression rate.

3.8 Comments

The primary characteristic of most of the H.264 tools is to sacrifice run–time for better rate–distortion performance. This is a consequence of the recent growth in computer power which has been steadily following Moore's Law [18] since the late sixties. Computers are now roughly 16 times more powerful then they were in 1996 when H.263 was first produced. Even so, H.264 is not being designed with current computing limits in mind. In order for the standard to stand the test of time it must challenge today's technology while anticipating and driving the creation of new technology.

Many of the new features in H.264 require not only additional computing resources but also increased bandwidth. Since it is not always possible to increase the available bandwidth the only variable which can be adjusted is the rate-distortion performance. If the bandwidth stays constant and the amount of control signalling increases then the picture quality will have to suffer.

As the results in Chapter 5 will bear out, using all the features of H.264 in real time is well outside the capability of a powerful personal computer. However, H.264 has been designed in a modular fashion; as technology evolves we will be able to combine the tools in increasingly powerful combinations.

[18] Computing power doubles every 18 months.

Chapter 4

H.264 – Main Profile

The main profile contains all the features in the baseline profile with the addition of three features.

- B pictures
- Context Adaptive Binary Arithmetic Coding (CABAC)
- Adaptive block-size transform

4.1 B Pictures

B–pictures, or bidirectionally coded pictures, are predictive coded pictures with up to two reference pictures. There are several fine distinctions to be made in understanding B–pictures which are not made any more clear by the use of confusing terminology.

A B–picture can be predicted by motion vectors which point into two different buffers, a forward reference frame buffer and a backward reference frame buffer. The terms forward and backward do not mean that, relative to the B–picture, the buffers contain pictures in preceding or subsequent order respectively. In other words, a forward reference frame buffer does not *necessarily* have to contain only pictures which precede the B-picture in the video stream; as with the backward reference frame buffer, it does not *necessarily* have to contain picture which are subsequent to the B-picture.

If the B–picture has two motion vectors for a block of pixels, the prediction block will be a weighted combination of the two referenced blocks.

The division of the macroblock for a B–frame is the same as for P–frames: 16x16, 16x8, 8x16 and 8x8 divisions with 8x8,4x8,8x4 and 4x4 divisions available for 8x8 coded macroblocks.

When B–pictures are used, the location of picture data is no longer in temporal or display order. For example, if the following dependency structure exists in Figure 4.1 then the resulting bitstream will be $P0, P1, B1, B3, B2, \ldots$ instead of $P0, B1, B2, B3, P1, \ldots$:

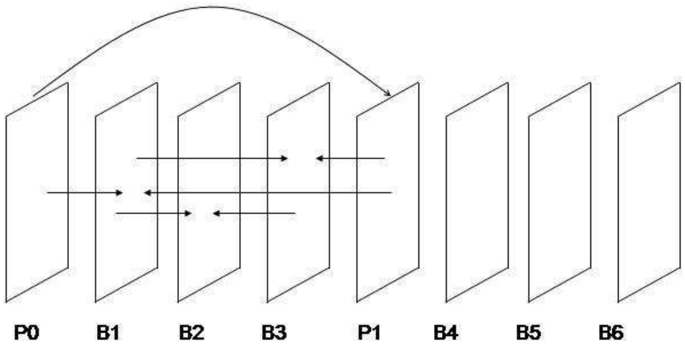

Figure 4.1: Example of a picture sequence containing B-pictures

4.1.1 Macroblock Modes

There are five available macroblock modes for B–pictures which can be broadly classified into three functional groups.

4.1.1.1 Forward and Backward

Forward mode and *backward mode* are basically equivalent to the *inter* coding mode. The prediction block is taken from the forward reference frame buffer if *forward mode* is used and from the backward reference frame buffer if *backward mode* is used.

4.1.1.2 Bi–predictive and Direct

Bi–predictive mode and *direct mode* reference two distinct blocks for each block being coded. These blocks could be in either of the reference frame buffers. It should again be emphasized how loose the constraints are for these buffers. For example, the same picture could be assigned to both buffers. It is up to the encoder to control how pictures are added to the buffers.

The only difference between *bi–predictive mode* and *direct mode* is in how certain parameters are coded. In *bi–predictive mode*, the reference frame indices and the forward and backward motion vectors are coded separately. In *direct mode*, the reference frame indices and forward and backward motion vectors are derived from those in the corresponding macroblock in the picture with reference index 0 in the backward reference frame buffer. Essentially this means that the structure of the picture with reference index 0 dictates the structure of the current B-picture. It is also clear that in order to begin using *direct mode*, *bi–predictive* mode must be used first to create the macroblock divisions which are found in the picture with reference index 0.

The two reference macroblocks will be combined together using one of several weighting schemes.

4.1.1.3 *Intra* Modes

B–picture *intra* mode is no different to P–picture *intra* mode. The picture is coded using spatial prediction using one of the nine available modes.

4.1.2 Coding the Modes

If a block is coded using *direct mode*, no information needs to be sent for the sub–blocks because this information can be found in the picture with reference index 0 in the backward buffer. There is therefore only one codeword needed to signal this case.

If 16x16 division size is used, the block could be coded using *forward mode*, *backward mode* or *bi–directional mode*. Three codewords are required to distuinguish these modes.

If 8x16 or 16x8 blocks are used for dividing the macroblock, each of the two blocks may use a different mode, either *forward mode*, *backward mode* or *bi–directional*. That makes 9 possible prediction choices for two ways of dividing the macroblock. This results in 18 distinct codewords which are needed to describe these situations.

If 8x8 divisions are used a further 12 codewords are needed to describe the combinations of blocks and prediction choices.

4.1.3 Prediction Generation

Unless explicitly specified, when *bi-directional* mode or *direct mode* are used, the sum of the two referenced macroblocks is divided by two to form the prediction for the currently regarded block. There is however, the capability to signal much more explicit parameters for combining the two macroblocks.

If so indicated in the bitstream, the parameters FWF,SWF, LWD and D are sent to be used in equation 4.1.

$$P = clip\ \left(\frac{P_1 FWF + P_2 SWF}{2^{LWD}} + D\right) \tag{4.1}$$

Where P_1 and P_2 are the two referenced macroblocks. P is clipped between 0 and 255 which is to be expected because the values in the macroblock being predicted are also confined to [0,255].

4.1.4 Decoder Process

The motion vectors for B–pictures are differentially encoded as with I–pictures. That is to say, a prediction is formed for the motion vector and then the prediction error is sent over the channel. The predictions are formed in the same manner as *inter* pictures, using median prediction except in one special case.

If *bi-directional mode* or *direct mode* are used and the two picture are in the same direction[1] then the motion vector for the farthest picture is formed using a differential motion vector from a scaled motion vector pointing to the nearest picture. The concept is illustrated in Figure 4.2.

The motion vector, MV1, is scaled according to two parameters, distance_1 and distance_2. Then the differential motion vector, DMV, is added to reach the appropriate block in the second reference picture. Equation 4.2 details the calculations necessary to obtain MV2.

$$MV2 = \frac{distance\_1}{distance\_2} MV1 + DMV \tag{4.2}$$

[1]That is to say in the same buffer.

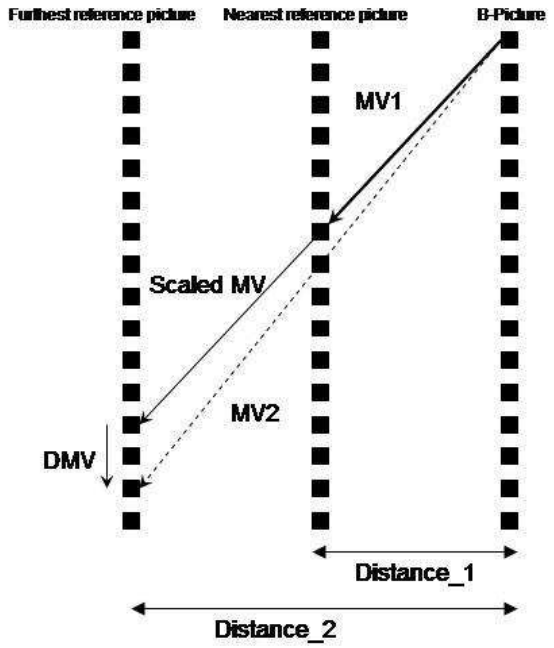

Figure 4.2: Calculation of Second Motion Vector

4.2 CABAC

The CABAC system allows for more efficient compression at the expense of much higher complexity. Each bit encoded, entails a significant amount of processing. Every encoded parameter has its own context modeling system, some more complicated then others. Table 4.1 lists most of the information which has to be encoded for H.264.

Macroblock Type (I,P,B Slices)
Motion Vector Data (P,B vectors)
Reference Frames
Coded Block Patterns (*intra 16x16 mode, inter mode*)
Intra Predictions Modes
Coefficient Information

Table 4.1: H.264 Parameters

Non–binary parameters are mapped to binary code words. Each of the bits in the binary codewords will have an associated probability distribution which is determined by context modeling. Context is derived from at most the two surrounding samples on top and to the left of the currently regarded sample, as shown in Figure 3.5–(a). As each bit is encoded, these probability distributions are updated accordingly.

Many of the computations which are normally associated with arithmetic encoding are side–stepped in CABAC by the use of table look up schemes. Tables are used to determine the new size of the encoding range and for updating the context models. The disadvantage of using tables is that approximations have to made.

It will be helpful to make some terms clear for the context of CABAC. Each bit which is encoded has a corresponding **context variable**. These context variables can take on one or several possible values. For example: if pixel bit i of pixel c was being encoded then the context variable,ctx_c_i, *could* look something like equation 4.3. Each of the possible values of ctx_c_i is a context.

$$ctx\_c_i = (ctx\_a_i + ctx\_b_i)/2 \tag{4.3}$$

The basic steps for CABAC are summarized below.

1. Determine the context
2. Initialize the context model
3. Perform arithmetic encoding on each individual bit
4. Update the context, rescale if necessary, and return to step 3.

The decoder will know in which order to expect data from the bitstream. Given this knowledge, the context model has to be initialized. If the information being encoded/decoded is non–binary valued then it has to be binarized[2]. This process involves traversing a binary tree where each branch in the tree is a **decision** which has its own context model. Since these contexts only model the possible values of one bit, the random variable is defined by one value, p which represents the probability of the least likely value of the bit. After encoding each bit, the context model is updated accordingly.

[2]This awkward term was taken from [6].

4.2.1 Context Modeling

The context modeling for every type of parameter is presented in the standard. For the sake of brevity, only selected parameters will be discussed here, encoding *inter* macroblock mode information, motion vector data and transform coefficients.

4.2.1.1 *Inter* Macroblock Modes

Recall, for *inter* macroblocks there are six ways of dividing the macroblock and for 8x8 mode there are a further five ways of dividing each of those blocks. All these modes must be sent to the decoder. Table 4.2 details these modes with their associated codewords.

Macroblock Mode	Binarization
SKIP	0
16x16	1 0 0 0
16x8	1 0 1 1
8x16	1 0 1 0
8x8	1 0 0 1
Intra 4x4	1 1 0
Intra 16x16	1 1 1
Decisions	1 2 3 4

8x8 Partition Mode	Binarization
8x8	1
8x4	0 0 0
4x8	0 0 1 1
4x4	0 0 1 0
*Intra*4x4	0 1
Decisions	1 2 3 4

Table 4.2: Binarization for *intra* coding modes

For encoding the macroblock mode, the first decision has three possible context models based on the value of equation 4.4. See table 4.3 for the associated count values, Section 4.2.2 details how the values c_i and r_i are used. Therefore, the value of ctx_mb_type dictates which probability distribution p will be used for encoding the first bit for the codeword denoting the macroblock mode.

$$ctx\_mb\_type\_P(C) = \begin{cases} 0 & \text{if A and B were both skipped} \\ 1 & \text{if A XOR B was skipped} \\ 2 & \text{if A and B were not skipped} \end{cases} \tag{4.4}$$

The rest of the decisions have only one context model each to choose from, i.e. only one possible distribution for p. The second decision is coded using $ctx\_mb\_type\_P(C) = 3$ (context 3). The third decision is based on the value of the second. If a value of 0 is encoded then context 4 is used, context 6 otherwise. For the fourth decision, if a 0 is encoded then context 5 is used, context 6 otherwise. The 8x8 divisions are encoded in a similar manner using a different table.

$ctx\_mb\_type\_P =$	c_0	c_1	r_0	r_1
0	7	2	5	0
1	1	2	0	0
2	1	2	0	0
3	30	1	0	0
4	2	1	0	0
5	5	9	9	-9
6	4	3	0	0

Table 4.3: Contexts for encoding *intra* macroblock modes

4.2.1.2 Motion Vector Data

As with *intra* frames, only the first bit of motion vector data is encoded using a context variable with more than one possible value. Equation 4.5 is used to derive the contexts shown in equation 4.6.

$$e_k(A, B) = |MD_k(A)| + |MD_k(B)| \tag{4.5}$$

$$ctx\_MD(C, k) = \begin{cases} 0 & \text{if } e_k(A, B) < 3 \\ 2 & \text{if } e_k(A, B) > 32 \\ 1 & \text{otherwise} \end{cases} \tag{4.6}$$

These three contexts represent three different probability distributions. Context–0 corresponds to a much higher weighting given to zero as the most probable symbol. This is due to how the motion vector components are binarized. The codeword for a zero magnitude component is 0, therefore every other codeword must start with a 1[3]. The extra weight given to the zero in context–0

[3]For unique decodability.

now makes sense because it corresponds to a context where the surrounding residual motion vectors have a small magnitude.

Symbols corresponding to component magnitudes from 1 to 63 are encoded using a unary code of the following form: magnitude n is encoded as $1_0 1_1 \dots 1_n 0$. Codewords 64 and upwards are encoded with an initial run of 63 1's followed by an additional $\lfloor log_2(n-62) \rfloor$ 1's, a 0 then finally some suffix bits. Table 4.4 shows codewords 64 to 72.

Context–1, which assigns an equi-probable distribution to the counts, is used for the case when the surrounding motion vector residuals are not very large. Context–2, which corresponds to high magnitude surrounding motion vector residuals, assigns a slightly higher probability to 1. The rest of the bits are encoded using a fixed context model.

Index	Prefix	Suffix
63	$1 \dots 10$	-
64	$1 \dots 110$	0
65	$1 \dots 110$	1
66	$1 \dots 1110$	00
67	$1 \dots 1110$	01
68	$1 \dots 1110$	10
69	$1 \dots 1110$	11
70	$1 \dots 11110$	000
71	$1 \dots 11110$	001
72	$1 \dots 11110$	010
Decision	123...	...

Table 4.4: Binarization of motion vector magnitudes

4.2.1.3 Transform Coefficients

For *each* block of transform coefficients, a one bit variable, *cbp*4, is signaled which indicates if there are any non–zero coefficients. *cbp4* is not sent if the macroblock header indicates there are no non-zero coefficients in that region of the macroblock. If *cbp4* equals zero then no further information is sent for the block.

The block types[4] are divided into 5 categories. Equation 4.7 is used for all five categories of blocks. There are four possible values for equation 4.7 which results in a total of 20 different possible contexts for encoding the *cbp4* bit.

$$ctx\_cbp4(C) = cbp4(A) + 2 \text{ x } cbp4(B) \tag{4.7}$$

If *cbp4* indicates there are non–zero coefficients, a **significance map** of the block must be sent indicating the positions of the non–zero coefficients. For every coefficient position, a 1–bit variable, SIG, is transmitted. If SIG equals 1, the coefficient is non–zero. Furthermore, if SIG 'equals 1 then a 1–bit variable, LAST, is signaled to indicate if this is the last non–zero coefficient for the block. Table 4.5 gives an example of this coding style.

Coefficients	10	2	0	0	0	5	1	1	0	0	0	0	1	0	0	1
SIG	1	1	0	0	0	1	1	1	0	0	0	0	1	0	0	(1)
LAST	0	0	-	-	-	0	0	0	-	-	-	-	0	-	-	(1)

Table 4.5: Example of SIG,LAST coding

There is a different context variable for each coefficient position. The same context is used for both SIG and LAST.

Now that a map has been formed indicating where non–zero coefficients are located, the magnitudes and signs of those coefficients must be encoded. This is accomplished in a manner very similar to that of motion vector information.

4.2.2 Initialization of Context Models

Having chosen a context, the variables c_1 and c_0 are assigned values for each needed bit. These variables now have to be initialized to make them ready to be sent to the encoder/decoder.

There are three types of context models, those with QP–independent initial counts, those with QP–dependent initial counts and uniform initial counts. These models are initialized in the following manner:

[4]See table 3.1.

Note: $c_{total} = c_0 + c_1$

QP–INDEPENDENT

while($c_{total} > 16$)

$c_{temp} = c_1$

$c_1 = (c_1 + 1) >> 1$

$c_{total} = c_1 + ((c_{total} - c_{temp} + 1) >> 1)$

QP–DEPENDENT–case1

qp_factor = min(33, max(0,$QP - 22$)) - 12

$cs_0 = c_0 + (r_0$ x qp_factor$)>> 3$

$cs_1 = c_1 + (r_1$ x qp_factor$)>> 3$

QP–DEPENDENT–case2

qp_factor = min(36, max(0,$40 - QP$)) - 12

$cs_0 = c_0 + (r_0$ x qp_factor$)>> 4$

$cs_1 = c_1 + (r_1$ x qp_factor$)>> 4$

The QP–Independent algorithm is actually used whenever the count values get too high and have to be rescaled. The rescaling operation ensures that c_{total} is never higher than 16. This places an upper bound of 64 on the possible number of probability distributions and correspondingly, a bound of 64 on the number of encoder/decoder states.

The most probable symbol (MPS), the least probable symbol (LPS) and state index STATE have to be determined from the initialized distribution. The STATE index is used for making transitions in

the encoding tables.

GET–MPS–AND–STATE

MPS = 0

IF $((c_{total} - c_1) < c_1)$

MPS = 1

STATE = StateTab$[c_{total} - c_1 - 1][c_{total} - 2]$

ELSE

STATE = StateTab$[c_1 - 1][c_{total} - 2]$

StateTab is a table containing encoder states which are referenced by the value of c_{LPS} and c_{total}.

For the uniform distribution, $c_1 = c_0 = 1$, MPS=0 and STATE=0.

4.2.3 Implementation Details

CABAC is implemented using mainly table lookups. Instead of making costly calculations, the encoder/decoder moves from state to state based on the values of c_0 and c_1 as they are updated when each bit is encoded/decoded. Each state contains an approximation of two variables R and V.

The range R, is a 16–bit number. The pointer V, points into the current sub–range of R which contains the most recently encoded symbol. R is initialized as 0x8000 (32768) and is never allowed to shrink smaller than 0x4000 (16384). In other words, because of the nature of of arithmetic coding, R will continue to shrink as more numbers are encoded. Eventually, R will reach a value which triggers a re–scale operation.

First, R is initialized to 0x8000, then the first bit to be encoded is examined. If this bit is the LPS, then R is shrunk to $R * p$[5] and V is set to point into $[0, R * p)$. If the bit is the MPS, then R is shrunk to $R * (1 - p)$ and V is set to point to $[R * p, R)$. When the next symbol is encoded, R will be shrunk even further. In order to avoid these costly multiplication operations, R is quantized to four values; that is to say, the range (0x4000,0x8000] is quantized to four values using equation 4.8.

There are 64 different values of p. Many of these distributions are equal but they must remain distinct in order to preserve the count values. For example, if $c_0 = 1$ and $c_1 = 2$ then $p = \frac{1}{3}$; if $c_0 = 2$ and $c_1 = 4$ p is again equal to $\frac{1}{3}$. Table 4.2.3 illustrates all these different distributions.

Since there are only 4 values for R and 64 values for p, every possible value of $R * p$ can be stored in a 64x4 table.

$$Q = (R - 0x4001) >> 12 \tag{4.8}$$

[5] Remember, p is the probability of the least probable symbol.

c_{total}	$p_{LPS} = p$
2	$\frac{1}{2}$
3	$\frac{1}{3}$
4	$\frac{1}{4}, \frac{2}{4}$
5	$\frac{1}{5}, \frac{2}{5}$
6	$\frac{1}{6}, \frac{2}{6}, \frac{3}{6}$
7	$\frac{1}{7}, \frac{2}{7}, \frac{3}{7}$
8	$\frac{1}{8}, \frac{2}{8}, \frac{3}{8}, \frac{4}{8}$
9	$\frac{1}{9}, \frac{2}{9}, \frac{3}{9}, \frac{4}{9}$
10	$\frac{1}{10}, \frac{2}{10}, \frac{3}{10}, \frac{4}{10}, \frac{5}{10}$
11	$\frac{1}{11}, \frac{2}{11}, \frac{3}{11}, \frac{4}{11}, \frac{5}{11}$
12	$\frac{1}{12}, \frac{2}{12}, \frac{3}{12}, \frac{4}{12}, \frac{5}{12}, \frac{6}{12}$
13	$\frac{1}{13}, \frac{2}{13}, \frac{3}{13}, \frac{4}{13}, \frac{5}{13}, \frac{6}{13}$
14	$\frac{1}{14}, \frac{2}{14}, \frac{3}{14}, \frac{4}{14}, \frac{5}{14}, \frac{6}{14}, \frac{7}{14}$
15	$\frac{1}{15}, \frac{2}{15}, \frac{3}{15}, \frac{4}{15}, \frac{5}{15}, \frac{6}{15}, \frac{7}{15}$
16	$\frac{1}{16}, \frac{2}{16}, \frac{3}{16}, \frac{4}{16}, \frac{5}{16}, \frac{6}{16}, \frac{7}{16}, \frac{8}{16}$

Table 4.6: All Possible Probability Distributions for p

The value of p is part of the family of variables which make up the current state of the encoder/decoder. The state contains, p, c_{total}, V, R and the MPS.

Transitions between states are handled using a 64x2 table. The two columns in this table, Next_State_MPS and Next_State_LPS, contain the index of the next state which the encoder/decoder will enter if the current symbol being encoded/decoded is the MPS or LPS respectively.

4.3 Adaptive block-size transform (ABT)

No matter which block division method is chosen for a macroblock, in the baseline profile, the pixels are always transformed in 4x4 blocks. The advantage to this situation is simplicity. However, using a transform which can take advantage of the entire signal length of larger blocks such as 8x8, 4x8 and 8x4 then the transform can achieve a greater signal compaction.

Prediction

As with regular *intra* luminance coding, there are nine available prediction modes for luminance *intra* ABT coding. The prediction modes for regular *intra* coding were defined for 4x4 blocks while the *intra* ABT prediction modes are defined for NxM blocks where N=4 or 8 and M=8 or 4. The prediction modes for two blocks are coded together unless there is an odd number of blocks, in which case the last block is grouped with a 0 which the decoder will subsequently discard.

The Transforms

Two transforms are used for the two signal lengths. For 4x4 and 8x4 blocks the transform from equation (3.17) is used. For 4x8 and 8x8 blocks, the 8x8 transform in equation (4.9) is used. Note that the bases in the transform $T8$ has a common norm which will simplify the scaling process.

$$\mathbf{T8} = \begin{bmatrix} 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 \\ 19 & 15 & 9 & 3 & -3 & -9 & -15 & -19 \\ 17 & 7 & -7 & -17 & -17 & -7 & 7 & 17 \\ 9 & 3 & -19 & -15 & 15 & 19 & -3 & -9 \\ 13 & -13 & -13 & 13 & 13 & -13 & -13 & 13 \\ 15 & -19 & -3 & 9 & -9 & 3 & 19 & -15 \\ 7 & -17 & 17 & -7 & -7 & 17 & -17 & 7 \\ 3 & -9 & 15 & -19 & 19 & -15 & 9 & -3 \end{bmatrix}. \tag{4.9}$$

Block Scanning

After the blocks have been transformed, they are scanned using a zig-zag pattern. For the 8x4 and 4x8 blocks, a slightly modified version of zig-zag is used to accommodate the non-square shape of the blocks. The modified 4x8 scan is shown in Figure 4.3.

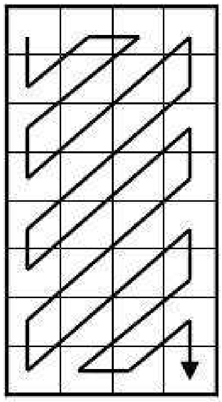

Figure 4.3: Example of the 4x8 Scan

Inverse Quantization and Reconstruction

DC blocks are still transformed dequantized in the manner defined in Sections 3.5.5.1 and 3.5.5.2. The residual 8x8, 8x4, 4x8 and 4x4 blocks are transformed and dequantized in a manner similar to that defined in Section 3.5.5.3. Table 4.7 shows all the scaling values which are needed for the different combinations of transforms.

For an 8x8 block, the rows and columns are both transformed by $T8$. Since every basis function in $T8$ has the same norm, only one set of scaling values is needed. These values are presented in column $S_{8\times 8}$ of Table 4.7.

For a 4x8 or 8x4 block, both $T8$ and $T4$ are needed. In the case of an 8x4 block, first the rows are transformed using $T4$ then the columns of the resulting matrix are transformed using $T8$; for a 4x8 block, the rows are transformed using $T8$ and the columns of the resulting matrix are transformed using $T4$. Since there are two distinct norm values for $T4$, but only one norm for $T8$, only two sets of scaling values are needed. These values are presented in column $S_{8\times 4,4\times 8}$ of Table 4.7.

For a 4x4 block, $T4$ is used for both the rows and columns. It should be noted that the scaling values presented in column $S_{4\times 4}$ of Table 4.7 are roughly four times larger than the values defined in Table 3.13. This essentially amounts to a left bit–shift by two. It is rather curious that the scaling values were changed, otherwise there is no difference between the regular transformation process for 4x4 blocks and ABT.

QP mod 6	$S_{8\times8}$	$S_{8\times4,4\times8}$		S_{4x4}		
0	15	9	11	40	64	51
1	17	10	12	45	72	57
2	19	11	14	50	81	64
3	22	12	16	57	91	72
4	24	14	17	63	102	80
5	27	15	20	71	114	90

Table 4.7: Scaling values for the different transform sizes

4.4 Summary

The additional options which form the main profile try to increase the compression rate by taking advantage of further redundancy in the video data.

B–pictures use as much temporal information as possible by examining not just preceding pictures but also subsequent ones. All the prediction modes for B–pictures provide great flexibility for finding a prediction block but a heavy

computational burden is imposed. In order for B–pictures to be useful, the extra signaling which is required must not encroach significantly on the gains made by the extra compression. If the extra signaling results in the same bit–rate as without B–pictures then computing time will have been wasted.

CABAC achieves additional compression by exploiting statistical redundancy between subsequent bits are they are encoded. This essentially entails paying careful attention to every data bit sent over the channel. The obvious result is a much higher complexity. This is borne out in the test results in Chapter 5. CABAC depends much less on spatial continuity between subsequent pictures than other tools such as B–pictures which will not work well if each picture shows a new scene. By continually updating the context model, CABAC achieves a consistant performance gain over UVLC.

Adaptive Block Size transforms add the least amount of additional computation of the three extra features in the main profile. Matching the transform to the block size will result in greater signal compaction and potentially better run–length coding performance. If an 8x8 block does not contain any non–zero coefficients then only one EOB symbol must be sent, instead of 4 if the block had been divided into 4 4x4 blocks.

Chapter 5

Performance of H.264 encoder

5.1 Testing Conditions

Table 5.1 details the test sequences which were compressed to show the performance of the latest H.264 encoder. The sequences were stored in YUV-concatenated format, with 8 bits used for each video component with 30 frames per second. The sequences were obtained from [9].

Sequence Name	Sequence Number	Resolution	Number Of Frames	Size (MB)	Rate (kb/s)
Container	1	QCIF	300	10.8	8640
Foreman	2	QCIF	400	14.5	8640
News	3	QCIF	300	10.8	8640
Silent Voice	4	QCIF	300	10.8	8640
Mobile and Calendar	5	CIF	300	43.5	34800
Paris	6	CIF	1000	145	34800
Tempete	7	CIF	260	37.7	34800

Table 5.1: List of test video sequences

The first frame of each sequence is shown in Figures 5.1 and 5.2; the sequences have the following visual and temporal characteristics:

- *Container*: a large container ship moving slowly in the distance.

Figure 5.1: QCIF video sequences; starting from top left, container, foreman, news and silent

- *Foreman*: a close–up face shot of a person talking, the camera then pans to show a completely new scene of a construction site.
- *News*: two newsreaders, who do not move very much, in the center background a ballet is performed, while the left and right background do not change.
- *Silent Voice*: a person is speaking in sign-language; the background does not change.
- *Mobile*: a toy train moves along tracks in front of a calendar. This scene has perhaps the most movement.
- *Paris*: two people are seated, talking. There arHadamarde many hand gestures and changes of facial expression.
- *Tempete*: a camera slowly zooms out from a nature scene. Leaves are continually blowing between the camera and the background.

The tests were run using the JM H.264 software encoder, version 3.40a. The encoder ran in Mandrake Linux 8.2 on a Pentium-4M 1.4 GHz with 256 MB of 266MHz RAM and 512 KB CPU cache.

Figure 5.2: CIF video sequences; starting from top left,mobile, paris and tempete

For a thorough comparison of a slightly older software implementation of H.264, H.263 and MPEG4, see [7] and [2].

The following features were tested to demonstrate the effect on run–time and the bit–rate; PSNR values[1] are given throughout, however, they are kept relatively constant.

1. I–frames vs. P–frames

2. Effect of decreasing the block size

3. 1 reference frame vs. 5 reference frames

4. UVLC vs. CABAC

5. P–frames vs. B–frames

[1] The PSNR values presented are the mean of the three Y, U and V PSNR values.

Adaptive block size transforms were not tested because that option was unavailable on the software.

Emphasis will be placed on the tradeoff between the additional run–time incurred by using each additional option and the savings on the bit–rate. Two situations will be assumed to provide a context in which to analyse the results, an office setting, with powerful desktop computers and a high bandwidth

LAN or a small handheld device using wireless communications. In both situations, computing power and network bandwidth can be assumed to be equally valuable. In an office setting it would be very costly to upgrade all the desktop computers; it would also be very costly to upgrade all the internal LAN equipment such as routers, switches and cabling. For the handheld device, computing power is at a premium due to size and heat constraints; wireless bandwith is also not cheap. These two situations are fairly typical instances where H.264 will be used in real time.

However, it should be remembered that non–real time video has fewer restrictions. In this context, run–time is almost irrelevant; the media only has to be encoded once and can be decoded any number of times. For example, if a particular option quadruples the run–time but results in the material fitting on one CD–ROM instead of two, the tradeoff is probably worthwhile.

All tests were run with the following options, unless otherwise specified:

1. One reference frame.

2. UVLC Entropy Encoding.

3. P–frames on.

4. No forced I–frames (i.e. only the first frame is an I–frame) item.

5. Motion vector search range clipped to 16.

6. Slices turned off.

7. All seven block sizes used.

5.2 Using P–frames

The first test was performed to determine the effect of using P–frames. Table 5.2 illustrates that P–frames have a significant impact on both the run–time and the bit–rate. On average, the cost of using P–frames seem worthwhile; for the price of more than tripling the run–time, the bit–rate can be decreased by more than six times.

Sequence	Only I			Only P		
Number	Time (s)	Rate (kb/s)	PSNR (dB)	Time (s)	Rate (kb/s)	PSNR (dB)
1	66.375	619.32	37.9	254.779	26.43	37.5
2	90.467	637.30	37.4	343.931	102.23	37.1
3	80.64	687.36	37.7	255.99	56.52	37.4
4	273.65	919.82	34.8	969.897	213.91	33.9
5	335.507	6336.76	34.3	1137.124	1342.84	33.3
6	995.46	3626.25	36.5	3453.104	306.61	36.0
7	257.339	3818.18	36.0	925.903	953.74	35.5

Average time cost of using P–frames:	345%
Average rate of the P–frame videostream:	14.7%
Average PSNR decrease from I–frames to P–frames:	0.55 dB

Table 5.2: I–frames vs. P–frames

5.3 Using Different Block Sizes

The second test was performed to determine the impact of varying the range of block sizes available to the encoder. Previously, the smallest block size available, was 8 pixels by 8 lines. H.264 provides seven different block sizes which can be grouped together as shown in Table 5.3.

Table 5.4 shows the results from testing each of sequences, starting with only using type–1 blocks and gradually increasing to include all five types of blocks. Table 5.5 summarizes the impact each block type has on the run–time and bit–rate.

The most significant reduction in bit–rate can be seen from using block types 1–4 versus just using block type 1 blocks. This should be expected

Type 1	16x16
Types 2 and 3	16x8 and 8x16
Type 4	8x8
Types 5 and 6	8x4 and 4x8
Type 7	4x4

Table 5.3: Block Types

however, previous standard such as H.263 and MPEG4 used block types 1–4. It is more important to observe the effect of including block types 5–7 which have not appeared in any other international standards. The results show that these block sizes provide a minimal decrease in block size while adding a significant portion to the run–time. At high bit–rates there are marginally more savings in the bit–rate.

5.4 Using 5 Reference Frames

Table 5.6 demonstrates the effect of using 5 reference frames. This data is compared with the P-frame results from Table 5.2. Both tests were run using the same parameters, differing only in the number of number of reference frames; the results from Table 5.2 were obtained using only one reference frame. The reference frames were implemented using a sliding window policy. There were no long–term pictures used.

There are two important observations to be made about increasing the use of reference frames. Firstly, the run–time penalty for using more search frames decreases at higher resolutions. Secondly, the benefits of increasing the number of search frames seems to be highly subjective. For five of the seven sequences there was only a small reduction in the bit–rate. However, for two of the sequences *tempete* and *mobile*, the bit–rate was reduced by approximately 18%. These findings are borne out by [2].

Sequence	1			1-3		
Number	Time (s)	Rate (kb/s)	PSNR (dB)	Time (s)	Rate (kb/s)	PSNR (dB)
1	152.092	30.44	37.38	161.566	27.02	37.44
2	199.966	120.17	36.93	226.944	106.35	37.04
3	149.706	69.61	37.20	183.404	61.47	37.29
4	569.295	232.26	51.1	623.392	220.41	51.14
5	643.951	1470.62	33.22	721.586	1389.61	33.30
6	2066.763	371.40	35.81	2204.749	330.78	35.97
7	510.030	1029.69	35.39	589.005	976.21	35.44
	1-4			1-6		
1	218.636	26.75	37.49	232.154	26.37	37.48
2	287.92	105.15	37.06	315.674	102.53	37.09
3	213.430	59.72	37.35	233.545	56.86	37.41
4	775.857	216.17	51.14	873.902	214.05	51.15
5	993.688	1375.85	33.33	1045.930	1346.46	33.38
6	3132.981	323.09	35.96	3219.810	307.98	36.05
7	772.311	965.54	35.49	847.227	954.88	35.54

Sequence	1-7		
Number	Time (s)	Rate (kb/s)	PSNR (dB)
1	254.779	26.43	37.5
2	343.931	102.23	37.1
3	255.99	56.52	37.4
4	969.897	213.91	51.1
5	1137.124	1342.84	33.3
6	3453.104	306.61	36.0
7	925.903	953.74	35.5

Table 5.4: Results from decreasing the block sizes

5.5 Using CABAC

The CABAC encoder is supposed to provide near optimal encoding, see [3]. The results of using CABAC are presented in Table 5.7 and they are compared with those of the P-frame results from Table 5.2. From the point of view of run–time versus bit–rate, CABAC proves disappointing. For an average increase in run–time of 66% the bit–rate is reduced, on average, 7%.

–	Average run–time increase	Average bit–rate decrease
1 to 1-3	111%	90%
1-3 to 1-4	130%	98%
1-4 to 1-6	107%	97%
1-6 to 1-7	109%	99%
overall increase	172%	87%

Table 5.5: Impact of decreasing the block sizes

Sequence	5 Search Frames		
Number	Time (s)	Rate (kb/s)	PSNR (dB)
1	712.168	24.53	37.60
2	1056.149	98.28	37.22
3	777.905	56.37	37.48
4	2824.667	203.60	51.16
5	3259.739	1092.06	33.59
6	9663.227	297.18	36.09
7	2553.817	812.61	35.77

Average cost of using 5 reference frames:	290%
Average bitrate reduction when using 5 reference frames:	92%
Average PSNR change when using 5 reference frames:	0.13

Table 5.6: Impact of using 5 reference frames

This does not mean CABAC should not be used; rather, it should not be used indiscriminately. Remember that the reduction of 7% is achieved without losing any information. CABAC would therefore be an ideal option in the case of non–real time video.

5.6 Using One B–frame

The results of compressing each of the sequences using one B-frame are presented in Table 5.8. The resulting bitstream started with one I-frame, and then alternated between I-frames and B–frames. The results were, again, highly subjective. The bit-rate was only reduced for the mobile and tempete

Sequence	CABAC		
Number	Time (s)	Rate (kb/s)	PSNR (dB)
1	432.301	24.57	37.5
2	572.870	96.87	37.1
3	427.273	53.36	37.4
4	1642.476	192.65	33.9
5	1856.461	1234.15	33.3
6	5792.271	283.93	36.0
7	1535.987	886.93	35.5

Average cost of using CABAC:	166%
Average bitrate reduction when using CABAC:	92.7%
Average PSNR change when using CABAC:	0

Table 5.7: Impact of using CABAC

sequences; for the other five, the bit–rate was actually increased along with the run–time.

Sequence	1 B–Frame		
Number	Time (s)	Rate (kb/s)	PSNR (dB)
1	321.149	38.78	37.59
2	436.778	104.63	37.23
3	319.583	64.44	37.50
4	1236.086	222.71	51.15
5	1372.299	1233.20	33.56
6	4224.252	364.70	36.20
7	1152.049	908.47	35.68

Average cost of using 1 B-frame:	124%
Average bitrate reduction when using 1 B-frame:	110%
Average PSNR change when using 1 B-frame:	0.13

Table 5.8: Impact of using 1 B-frame

Chapter 6

Conclusions

The baseline and main profiles for the emerging H.264 video encoding standard have been presented and analyzed. Test results have shown the effectiveness of some of the available compression tools. It was found that most of the tools impose a heavy computational burden and should not be used indiscriminately.

It can be argued that an intelligent encoder is required to independently apply the capabilities of H.264. For example, in some cases B-frames reduce the bit–rate while in other cases the bit–rate is actually increased. It would be helpful if the encoder could make this determination either beforehand or during encoding. There are many possible methods of implementing an intelligent encoder; many of which would be a mix of brute–force searching and subjective analysis.

For brute–force, multiple encoding processes with different combinations of options could be run; whichever set of options yielded the best rate–distortion/run–time performance would be picked as the best possible choice. A subjective analysis algorithm would follow the change in the video content and determine through empirical results which options to use. Approximations could be made to save resources; not every possible combination of options would need to be tested in the brute–force method and any number of parameters could be used to follow the video content changes for the subjective analysis.

The intelligence part of the encoder would have to be light–weight and scalable to meet available resources. Given some reasonable approximations, an intelligent encoder should be able to be implemented which could effectively manage the H.264 algorithm.

Bibliography

[1] T. Cover, J. Thomas, *Elements of Information Theory.* Toronto, John Wiley & Sons Inc., 1991.

[2] A. Joch, F. Kossentini, P. Nasiopoulos, "A Performance Analysis of the ITU-T Draft H.26L Video Coding Standard." *http://pv2002.ece.cmu.edu/papers/* (Current August 2002).

[3] D. Marpe. G. Blattermann, G. Heising, T. Weigand, "Further Results from CABAC Entropy Encoding Scheme," Document VCEG-M59, VCEG 13th annual meeting, Austin Texas, USA, 2–4 April 2001

[4] K. Sayood, *Intoduction to Data Compression.* Toronto, Morgan Kaufmann Publishers, 2000

[5] P. Topiwala, G. Sullivan, F. Kossentini, "Overview and Performance Evaluation of the ITU-T Draft H.26L Video Coding Standard." Proc. SPIE, Vol. 4472, 2001.

[6] ITU-T/SG16/VCEG, ISO/JTC1/SC29/WG11, H.264 Joint Committee Draft, Document JVT-C167, 3rd Meeting: Fairfax, Virginia USA, 6-10 May, 2002

[7] ITU-T/SG16/VCEG (Q.6/16), "Summary Information for ITU-T VCEG Draft H.26L Algorithm In Response to Video and DCinema CfPs." Document M7511, July 2001, *ftp://standard.pictel.com/video-site/h26L/m7511_H26LsummaryInfo.doc.*

[8] "Emerging H.26L Standard." *http://www.ubvideo.com/public/h.26l_white-paper.pdf* (Current February 2002).

[9] *http://kbs.cs.tu-berlin.de/ stewe/vceg/sequences.htm.*

[10] *http://bs.hhi.de/ suehring/tml/download/.*

[11] *http://www.intel.com/pressroom/archive/releases/mmxbkgrn.htm.*

[12] http://mpeg.telecomitalialab.com