Recasting a Scene Adaptive Video Coder for Real Time Implementation

By

Jonathan David Rosenberg

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Author.. [illegible]
Department of Electrical Engineering and Computer Science
January 23, 1995

Certified by.. [illegible]
Dr. Jae Lim
Professor, MIT
Thesis Supervisor

Certified by..
Dr. John Hartung
MTS, AT&T Bell Laboratories
Thesis Supervisor
[illegible]

Accepted by.. [illegible]
Prof. Frederick R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Recasting a Scene Adaptive Video Coder for Real Time Implementation

By

Jonathan David Rosenberg

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of
Bachelor of Science
and
Master of Science

Abstract

The Scene Adaptive Video Coder is a new algorithm used to code video sequences at very low bitrates (8 kb/s to 40 kb/s). It is a constant bitrate, constant frame rate encoder. Low rates are obtained through the use of sub-pel motion estimation, aggressive adaptive vector quantization, and dynamic bit allocation. Unfortunately, the algorithm is too complex to be implemented in real time, which is important for practical applications of low bitrate video. This thesis develops a recast of the algorithm to reduce its computational complexity while maintaining its important features, and minimizing the performance degradation as compared to the original algorithm. The computational trouble spots of the algorithm are determined, models are developed to assess the computational complexity and performance trade-offs, and the newly designed algorithm is implemented.

Thesis Supervisor: Dr. John Hartung
Title: Member of Technical Staff, AT&T Bell Laboratories

Thesis Supervisor: Dr. Jae Lim
Title: Professor, MIT

Acknowledgments

I would like to thank my thesis advisors, Dr. John Hartung and Prof. Jae Lim, for their guidance and support throughout this project.

I would also like to thank Dr. Nikil Jayant, and the rest of the staff in the Signal Processing Research Department, for their assistance. A special thanks go to Dr. Rich Pauls, for many technical and philosophical discussions.

Lastly, I would like to thank my fiancé, Michelle Greene, for her continual support of my work and her patience with me. My parents also deserve a special thanks for their constant support throughout my education.

Contents

Chapter 1

Introduction

With the dawning of the information age, video communications systems have become a source of great interest. The main driving force behind many of these technologies are video compression algorithms, which allow a great deal of video information to be transmitted over channels with low bandwidth. Several standards for video coding have been, or are in the process of being developed for a variety of applications. MPEG 1, which is geared towards video on CD-ROM, operates around 1.5 Mb/s [1]. MPEG 2 [2] is meant for more general video compression applications, including digital broadcast TV. It operates at rates anywhere between 3 to 20 Mb/s. HDTV [3], which is still in the process of specification, is meant for high quality broadcast TV, and operates around 20 Mb/s. Px64, standardized in CCITT recommendation H.261 [4] is geared for video conferencing over ISDN phone lines. As a result, it operates at rates ranging from 64 kb/s to 384 kb/s, in multiples of 64 kb/s (thus the name, Px64). However, there is no established standard for video at rates below 64 kb/s.

There is an enormous variety of applications for video at rates below 64 kb/s. This is mostly due to the existence of the PSTN (Public Switched Telephone Network). The PSTN, which connects nearly every person on the planet to just about everyone else, is the most widespread of all networks. Since each subscriber is bandlimited to around 3 1/2 kHz, bitrates on the PSTN are fundamentally limited to around 30 kb/s.

The possibilities for video applications over the PSTN are endless. They include videophones, multimedia email, remote sensing, electronic newspapers, interactive multimedia databases, multimedia videotext, video games, interactive computer imagery, telemedicine, and communication aids for the deaf, just to name a few [5].

Despite the tremendous number of applications, very low bitrate video has only recently received attention because of its daunting requirements. A simple calculation is very illustrative. Assuming video at CIF resolutions (352x288, 12 bits/pel), at 30 frames/second, the bit rate required is:

$$352*288\frac{\text{pels}}{\text{frame}}*12\frac{\text{bits}}{\text{pel}}*30\frac{\text{frames}}{\text{sec ond}} \approx 36\text{Mb} / \text{s}$$

To reduce this to 28.8 kb/s for transmission over the PSTN (using the V.34 modem standard) requires a compression ratio of over 1200!

1.1 Algorithms for Very Low Bitrate Video

In order to achieve these enormous compression ratios, low bitrate video coders must restrict the range of video sequences they can code. The type of sequence most studied is the head-and-shoulder sequence, which is common in videophone and video-teleconferencing applications. There are several features of head-and-shoulders sequences which very low bitrate coders take advantage of [5]:

1. *Fixed Scene Content.* The image to be transmitted is a fixed head and shoulders, with some sort of background that does not change.
2. *Limited Motion.* The only motion is that of the head and facial features. There are no background or camera movements.
3. *Lower Resolution.* In general, such sequences require less resolution to sufficiently convey the facial expressions and emotional content of the sequence.

Additionally, most VLBV (Very Low Bitrate Video) coders aim to achieve the following goals:

1. *Very Low Bitrates.* This is the most important goal. The range is usually between 8 and 40 kb/s.
2. *Low Delay.* Since the intended use is for real-time person to person communication, delay must be kept as low as possible.
3. *Achievable Computation.* Another necessity for practical systems is that the computational requirements are sufficiently low to be accomplished using currently available hardware.

Currently, the ITU (International Telecommunications Union, formerly the CCITT) is preparing to develop a new standard, called MPEG4, which will deliver, among other sequences, head-and-shoulders video at very low bitrates [6]. Algorithms for achieving these rates fall under three categories: waveform based coders, model-based coders, and hybrid coders (which are a mix of the first two).

1.1.1 Waveform Coders

Very low bitrate waveform coders treat the image sequence as a two-dimensional signal, and attempt to process it using common signal and image processing techniques. This method has been very successfully employed at higher bitrates (MPEG 1, MPEG 2, HDTV and Px64 are waveform coders). Typically, waveform coders use techniques such as motion compensation, transform coding (such as the DCT, WHT, KLT, and DFT), Huffman coding, and scalar and vector quantization. Important approaches less commonly used by other coders include subband coding, wavelet coding, and fractal coding. Each of these techniques have been studied quite thoroughly. An excellent overview of motion compensation can be found in [7], transform coding in [8], quantization in [9], subband coding in [10], wavelet coding in [11], and fractal coding in [12].

Of all these waveform coding techniques, the most successful has been the MC-DCT (Motion Compensated Discrete Cosine Transform), also known as Hybrid III. Figure 1.1 is a block diagram of a typical MC-DCT based coder. Temporal redundancy is reduced by

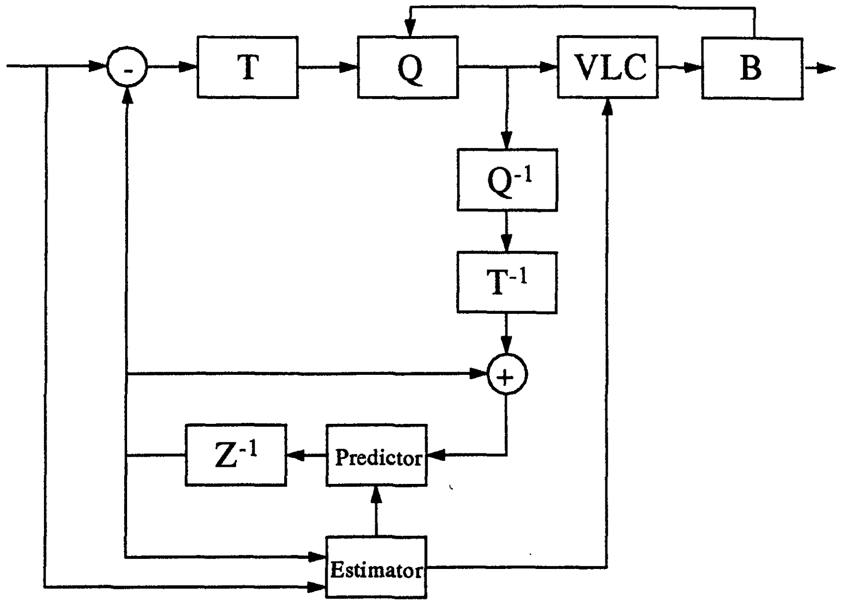


Figure 1.1: MC-DCT Based Coder. T is the transform (DCT), T^{-1} is the inverse transform, Q is the Quantizer, and Q^{-1} the Reconstructor, VLC is a Variable Length Code generator, and B is the Output Buffer.

the motion prediction, and spatial redundancy is removed by the DCT. The quantizer and Huffman coder then reduce the bitrate of the coded residual. The MC-DCT is an efficient and mature coding technique [5]. However, three properties of the MC-DCT make its direct application at very low bitrates problematic:

1. Bit allocation is a function of output buffer occupancy only. Since few bits are available to code the residual, heavy quantization must be applied. This leads to uniformly poor image quality.
2. Since the residuals are so coarsely quantized, the reconstructed frame will accumulate errors over time, reducing image quality.
3. To achieve constant bitrates, the quantization is adjusted based on the fill rate of the output buffer. Known as *rate buffering*, this technique increases delay at low frame rates (several frames must be stored in the buffer), making real time communication difficult. Rate buffering also introduces frame dropping, causing very jerky motion at low bitrates.

As a result, very low bitrate MC-DCT based coders usually employ some additional features, such as improved motion estimation, vector quantization, and intelligent bit-allocation to address these problems [5].

1.1.2 Model Based Coders

Unlike Waveform based coders, Model based coders do not transmit information about the pels in the image. Instead, they seek to transmit information about the object content. This is achieved by modeling the objects in the scene, and transmitting the model information to the decoder.

There are two types of model-based coding schemes. The first is *object-oriented image coding*, first proposed by Musmann in [13]. In this method, a general model is extracted from the scene. The model consists of generalized objects, each of which contain three parameters: motion (A_i), shape (M_i), and color (S_i). Several models are available for analyzing the image, and include rigid 2D objects with 3D motion, flexible 2D objects with 2D motion, rigid 3D objects with 3D motion, and flexible 3D objects with 3D motion. Each has its own advantages and disadvantages in terms of bitrate, computational complexity, and accuracy.

The other type of model-based coding is *semantic-based image coding*, first proposed independently in [14] and [15]. It uses explicit object models, instead of general ones as in object-oriented image coding. A common object model would be a wireframe mesh of a head. The encoder uses the model to extract parameters from the image, and then synthesizes the image from the parameters, computes some sort of error criteria, and uses the error to re-extract the parameters. The process iterates until the error is acceptable. Texture mapping was later added to this technique to improve the naturalness of the reconstructed images [16].

Although model-based coders can achieve very low bitrates, they suffer from two difficulties. First, the computational complexity involved in determining model parameters is far too great to permit a real-time system to be developed using current technology. Furthermore, the resulting image sequences often have a cartoon-like quality, which can be a very disturbing artifact.

1.1.3 Hybrid Coders

Hybrid Coders fall into two categories. The first is known as *model-assisted waveform coders*, and the second as *waveform-assisted model coders*. They differ in the amount upon which waveform and model-based coding is used [5].

An example of a model-assisted waveform coder is Px64 with facial tracking, as outlined in [17]. In this coder, a simple elliptical model of the face is used. By applying various symmetry rules, an ellipse is fitted to the face. Px64 is then used to code the image. However, two quantization values are defined - one for coding blocks inside of the face, and one for coding blocks outside the face. As a result, more bits can be spent in the facial area, and less outside, yielding improved perceptual performance.

An example of waveform-assisted model coders can be found in [18]. In this coder, the "clip and paste" method is applied to regions where semantic-based image coding failed. In these regions, waveform coding approaches are used instead, and the results are "pasted" on top of the semantic-based coded frame. This helps to reduce some of the cartoon-like artifacts.

1.2 Scene Adaptive Video Coder

There are problems associated with many of the low bitrate coding schemes. Model based coders often produce results which appear cartoon-like and artificial. Additionally, they are extremely complex, and are often difficult to implement in real time. Hybrid coders are also often too complex for real time implementation. While many waveform coders (such as Px64), can run in real time, they do not take advantage of the properties of head-and-shoulders sequences to reduce the bitrate enough. Additionally, rate-buffering methods, which are commonly employed to achieve a constant bitrate, lead to excessive delay and choppy motion.

The *Scene Adaptive Coder*, developed at AT&T Bell Labs [19], addresses several of these problems. It is a waveform coder, similar to Px64 in its use of the motion compensated DCT. However, it features a constant bitrate and constant frame rate output. It also uses fractional pel motion compensation, along with very high dimensionality adaptive vector quantization, to address the problems with the MC-DCT at low bitrates.

Although it meets the criteria of low bitrates and low delay, the last criteria, that of achievable computation, has not been met. The algorithm is less complex than model-based coders, but does not yet work in real time.

Addressing this difficulty is the subject of this thesis. Several experiments and analyses are carried out to determine the optimal recast (optimal in the sense of the least amount of degradation in image quality) of the algorithm in order to be implemented in real time.

1.3 Outline of the Thesis

The second chapter will outline the algorithm which was originally implemented for simulation in a non-real time environment. The third chapter will then discuss the hardware platform that will be used to implement the algorithm. The fourth chapter discusses the direct port of the algorithm to the hardware. Then, the fifth chapter analyzes the computation time required for the algorithm after being ported to the hardware, and determines the tasks that need to be modified. Chapter 6 addresses motion vector rate computation. Chapter 7 addresses vector quantization rate estimation. Chapter 8 addresses motion estimation, and Chapter 9 presents the conclusions of the research.

Chapter 2

The Scene Adaptive Video Coder

The Scene Adaptive Video Coder (SAVC) has several features in common with other transform based coders, such as Px64. It is based on the MC-DCT, and operates on 8x8 blocks. It codes sequences at SIF (352x240, easily derived from CCIR601, which is 704x480, 16 bits/pel), but operates internally at 112x80 (coined *NCIF*), which is the closest resolution to one ninth of SIF that is also divisible by 8. The SAVC has three features which make it very different from other coders.

The first feature is its sub-pel motion estimator. At resolutions as small as NCIF, full pel, and even half-pel motion estimation cannot produce adequate prediction gains. Therefore, a one-third pel motion estimator is used.

The second feature is its hybrid vector/scalar quantizer. The coder uses a fixed number of bits each frame (the bitrate divided by the frame rate). These bits are optimally allocated between scalar quantization and vector quantization. The vector quantization is 64 dimensional (i.e., over an entire 8x8 block). Both encoder and decoder maintain two, 256 entry codebooks of codevectors. The advantage of VQ at such high dimensions is the enormous savings in bits. The common disadvantage of high dimensionality VQ, poor codebook matches, is offset by the use of scalar quantization. Scalar quantization is obtained by using the DCT, quantizing the coefficients using a quantization matrix tuned to head-and-shoulder sequences, run-length-amplitude (RLA) encoding those coefficients, and then Huffman coding the RLA tokens. The allocation of bits between the vector and scalar quantization is dynamic, and is redone each frame in order to minimize the total frame MSE.

The third feature of the SAVC is its adaptive VQ codebook. Each frame, the codebook is adapted to best match itself to the sequence being coded. A competitive learning algorithm is used to implement this. Codewords are replaced or altered depending on how often they are used, and how much gain they have been able to provide.

The next section will outline the details of the algorithm.

2.1 Algorithm Detail

Since the SAVC is a constant frame rate and bitrate encoder, each frame has a fixed number of bits, R_{FRAME}, allocated to it. These bits are divided into three sections: the motion information, the vector quantization, and the scalar quantization. The number of bits used for each one is R_{MV}, R_{VQ}, and R_{SQ}, respectively.

The first step in the algorithm is to generate the motion information. Each frame is divided into 140 8x8 blocks, such that there are fourteen blocks horizontally, and ten blocks vertically. Using forward motion estimation, each coding block is searched for in the previous input frame (i.e., the frame obtained from the camera), not the previous reconstructed frame (the frame the decoder generated). Searching over the previous input frame will generate motion information which better represents the true motion of the sequence. Searching over the previous reconstructed frame will give motion information which better reduces the MSE of the predicted frame. However, smoother motion is visually more important [5].

In order to do the motion estimation, full-search block matching is done, using motion vectors in the range of -5 to 5 NCIF pels, in increments of 1/3 pel, for a total of 31 possible motion vectors in both the X and Y directions. Sub-pel motion estimation is accomplished by interpolating up to SIF resolutions, and then searching in a range of -15 to 15 pels, in 1 pel increments.

The motion estimator is biased towards a zero motion vector. If the prediction error for the zero motion vector is below a certain threshold, then zero motion is assumed even if it is not the best prediction. This is done in order to insure that no motion is detected in regions of constant texture, and also to help reduce bitrate, as the zero-motion vector is coded with run-length codes.

After obtaining a motion vector for each block, the motion vectors must be coded. There are three methods which can be used:

1. *PCM Codes.* 10 bits can be allocated to encode each non-zero motion vector.
2. *Global Huffman Codes.* Both decoder and encoder have a fixed Huffman table which is optimized for motion in head-and-shoulders sequences. This table can be used to entropy code the motion vectors.
3. *Local Huffman Codes.* The encoder can generate a local Huffman table based on the statistics of the vectors used in the current frame. The motion vectors can then be entropy coded using the local table.

To decode the local Huffman codes, the decoder must be able to reproduce the table used to generate them. Therefore, the encoder must transmit information about the local table to the decoder. The most compact method for sending this information is to transmit the statistics used to create the code, given that the encoder and decoder use the same algorithm for generating Huffman codes. The statistics are represented in the *motion vector list*, which is a catalogue of each unique motion vector and its rate of occurrence. There are two options for encoding the motion vector list when it is required:

1. *PCM Codes.* 10 bits are allocated for each unique motion vector, and 6 bits for its rate of occurrence.
2. *Global Huffman Codes.* Each unique motion vector can be looked up in the global Huffman table and thus represented by a variable length code. The rate of occurrence can be PCM coded with 6 bits.

The method that is used is the one which results in the least number of bits to represent the motion vectors. If the resulting motion vector coding rate is above the maximum, R_{MVMAX} (a fixed coder parameter, usually 15% of the total bits per frame) the coding rate is reduced below R_{MVMAX} by discarding motion vectors.. The motion vectors to be discarded are the ones with the least *prediction gain.* Prediction gain, G_{PR}, can be defined as follows:

$$G_{PR} = \frac{1}{64}\sum_{j=0}^{63}\left(B_{K,T}[j]-\overline{B}_{K,T}[j]\right)^2-\left(B_{K,T}[j]-\hat{B}_{K,T}[j]\right)^2$$

Where $B_{K,T}$ is the coding block in the current frame, $\overline{B}_{K,T}$ is the prediction of B_{KT} using a zero motion vector, and $\hat{B}_{K,T}$ is the coding block predicted using the actual motion vector. In other words, the prediction gain is the improvement obtained from using a motion vector as opposed to assuming a zero motion vector. The coding block with the least prediction gain is that block for which a motion vector provides the least improvement above doing nothing at all (i.e., using a zero motion vector). To reduce the coding rate, R_{MV} , below R_{MVMAX}, the motion vector which yields the smallest G_{PR} is discarded. The motion vectors are then re-encoded, and the new rate is calculated. If this rate is still above R_{MVMAX}, the process is repeated until R_{MV} is small enough.

The next step in the algorithm is to vector quantize the coding blocks. The encoder will sort the coding blocks in order of decreasing *prediction error*, E_{PR}, which is defined as:

$$E_{PR} = \frac{1}{64}\sum_{j=0}^{63}\left(B_{K,T}[j]-\hat{B}_{K,T}[j]\right)^2$$

The block with the worst prediction error will then be vector quantized. There are two codebooks available, an intra-frame codebook, and an inter-frame codebook. The intra-frame codebook contains actual coding blocks, while the inter-frame codebook contains residuals of coding blocks after motion compensation. Each codebook contains up to 256 codewords. The encoder will do a full search over both codebooks, and come up with a best match in each one. The results are two 64-dimensional vectors, V_{INTRA} and V_{INTER}. Then, the quantization error is computed for each one. The errors are defined as:

$$E_{INTRA} = \frac{1}{64}\sum_{i=0}^{63}\left(V_{INTRA}[i] - B_{K,T}[i]\right)^2$$

$$E_{INTER} = \frac{1}{64}\sum_{i=0}^{63}\left(V_{INTER}[i] - \left(B_{K,T}[i] - \hat{B}_{K,T}[i]\right)\right)^2$$

If E_{INTRA} is smaller than E_{INTER}, the intra-frame codebook is used. Otherwise, the inter-frame codebook is used. If E_{INTRA} and E_{INTER} are both larger than E_{PR}, the vector quantization does not reduce the total frame MSE. In that case, there is no match for the block in the codebook, and its VQ is discarded. Coding blocks, in order of decreasing prediction error, are vector quantized and coded in this fashion until the total rate left, R_{FRAME} - R_{MV}, has been exhausted.

The coding rate required for the vector quantization is composed of several parts. First, each codeword is coded as a fixed 8 bit quantity, which refers to the address into the appropriate codebook. Secondly, a mask must be transmitted to the decoder which indicates the coded block positions for those blocks which have a VQ. In fact, the coded block positions for those blocks with motion information must be transmitted as well. The two coded block positions are lumped together, and coded as one unit, since there is a great deal of correlation between blocks with motion and blocks with a VQ. A block which contains either a motion vector or a VQ is called an *MV/VQ instance*. Blocks without a motion vector or a VQ are left uncoded, and the decoder will assume a zero-motion vector for those blocks, essentially repeating those blocks from the previous frame. The coded block positions can be encoded in several ways:

1. *Fixed Mask.* Transmit a 140 bit mask.
2. *Addressing.* Transmit an 8 bit address with each MV/VQ instance which indicates the coded block position.
3. *Huffman Run Length Codes.* Run length code the mask, and then Huffman code the Run Length tokens.

Two bits are required to describe which coding type is used. Additionally, if the Addressing method is used, a count of the number of MV/VQ instances must be transmitted as well. In the case of either the fixed mask, or the Huffman Run Length Codes, a count is not required, since the mask itself contains this information.

The next item that needs to be coded is the block type, which indicates whether an MV/VQ instance has a motion vector only, a motion vector and an inter-frame VQ, an inter-frame VQ only (i.e., a zero motion vector), or an intra-frame VQ (no motion information needed). Two bits are required to encode this information for each MV/VQ instance.

Now that the motion and vector quantization information have been coded, the next step is to begin the dynamic bit allocation, which involves trading off vector quantization for scalar quantization. To choose which blocks will be scalar coded, the coding blocks are sorted in order of decreasing *approximation error*, E_{AP}, which is defined as E_{PR} for blocks with motion vectors only (including the zero motion vector), E_{INTRA} for blocks coded using the intra-frame codebook, or E_{INTER} for blocks coded using the inter-frame codebook. The block with the largest E_{AP} is then scalar quantized.

Scalar quantization is achieved in two ways, intra-frame and inter-frame. In intra-frame quantization, the actual coding block is transformed using the 8x8 DCT, and the result is quantized using a fixed quantization matrix. In inter-frame quantization, it is the residual after motion compensation which is transformed and quantized. The encoder will do both, and choose the method which results in the least number of bits.

Given a well-designed matrix, the quantization will result in a block with many zero coefficients. The most efficient way to transmit the quantized block to the decoder is to send only the nonzero coefficients, along with information indicating the coded coefficient positions. The encoder can use two methods for encoding the actual coefficients:

1. *PCM Codes*. Each coefficient is coded using Pulse Code Modulation with a fixed number of bits.

2. *Huffman Codes.* Each coefficient is looked up in a Huffman Table, and the Huffman code for each is transmitted.

The coded coefficient positions can be represented three different ways:

1. *Addressing.* For a small number of nonzero coefficients (typically one or two), it is easiest to just transmit the address of each nonzero coefficient using a fixed 6 bit quantity (there are 64 possible addresses). In addition, the encoder must also transmit the number of nonzero coefficients, which is encoded using 4 bit count (there can never be more than 10 non-zero coefficients, since it is always more efficient to use the Mask coding method in this case)
2. *Fixed Mask.* Send a 64 bit mask which indicates the positions of the nonzero coefficients.
3. *Huffman Run Length Codes.* Scan the Mask in zigzag order, and generate run-length tokens. Look up each of the run-length tokens in a Huffman table, and transmit the Huffman codes to the decoder.

The encoder will choose the most efficient method for the coefficient coding and the position coding.

Lets say the resulting encoding requires M bits. In order to obtain these bits (no bits are available since the VQ exhausted the total rate), VQ information must be discarded. To do this, the coding blocks are also sorted in order of increasing Δ_{MSE}, which is defined as $E_{PR} - E_{INTRA}$ or $E_{PR} - E_{INTER}$ depending on the codebook used. In other words, Δ_{MSE} is a measure of how much improvement is obtained by using the codebook as opposed to just using a motion vector. For the coding block with the least Δ_{MSE}, the improvement is small, so discarding its VQ information does not increase the frame MSE by too much.

The procedure for the dynamic bit allocation is as follows. The coding block with the least Δ_{MSE} has its VQ discarded, giving an increase in the total rate available, but increasing the frame MSE by Δ_{MSE} as well. VQ bits will be discarded, in order of increasing Δ_{MSE}, until enough bits have been obtained to transmit the scalar quantized block. Next, the encoder checks to see if scalar quantizing the block improved the frame MSE. This will be the case if:

$$E_{AP} \geq \sum_{VQ} \Delta_{MSE}$$

In other words, the scalar quantization was worthwhile if the improvement in frame MSE obtained by using it (E_{AP}) exceeds the sum of the losses incurred by throwing away VQ instances.

The dynamic bit allocation continues until either the scalar quantization no longer improves the frame MSE, all the VQ instances have been discarded, or there are no more blocks to scalar quantize.

After the actual encoding process is complete, the encoder adapts its VQ codebooks. This feature of the algorithm is what really makes the scene adaptive coder scene adaptive. Since this portion of the algorithm is currently under study, a preliminary procedure is used.

The codebook adaptation algorithm uses a split and merge technique. For each scalar quantized block, the distance to the closest codeword is computed. If this distance is above a certain threshold, then the Voronoi region corresponding to the closest codeword is split by simply adding the scalar quantized block to the codebook. However, since only a fixed number of codewords can exist in the codebook at a time, the addition of vector must be accompanied by the removal of another.

To choose which vector to remove from the codebook, the encoder maintains a usage count and improvement measure for each codeword. Every time a codeword is used, its usage count is incremented. Every frame, the usage count for every codeword is decreased by one. Thus, the usage count remains fixed if the codeword is used once a frame, on average. The improvement measure is the average improvement over just using motion compensation that the codeword has provided. A figure of merit, which is the product of the usage count and improvement for each codeword, is then computed. The codeword with the smallest figure of merit is removed from the codebook, therefore merging its Voronoi region with its closest neighbor's.

This process is repeated for every scalar quantized block in the frame. Since the decoder must also adapt its codebook in a similar fashion, each scalar quantized block is accompanied by an address, which indicates where in the codebook it should be placed. Additionally, a single bit indicates whether the scalar quantized block should be placed in the codebook at all.

The last step in the encode loop is then to generate and transmit the bitstream, and to build the reconstructed frame.

Chapter 3

The IIT VCP Platform

The platform chosen to implement the algorithm is the VCP (Video Compression Processor). This chip, made by IIT (Integrated Information Technology) is unique among compression chips in that it is both fast and fully programmable. It is capable of performing MPEG 1 Decode, MPEG 1 I-Frame Encode, Px64 Encode and Decode, and MPEG 2 Decode, all in real time, with only a change in software. It is this flexibility that makes it ideal for developing non-standard algorithms [20].

The VCP has several external connections [24]. It has ports for interfacing to DRAM, which is used as a frame buffer, and to SRAM, which is used for text and data storage. It has a pair of video input and output ports which operate at CCIR 601 resolution at 30 fps.

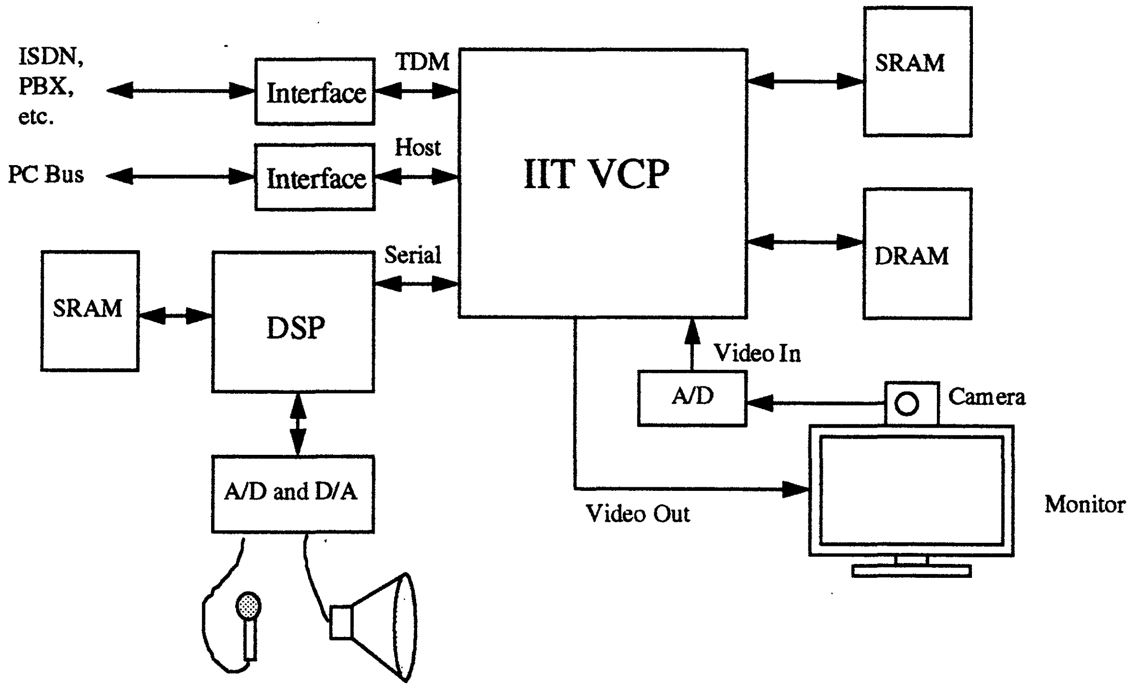


Figure 3.1: Typical VCP System

It also has ports for interfacing to serial and TDM (Time Division Multiplexed) devices. It can be controlled through its host interface port, which is usually connected to a PC bus. Figure 3.1 is a block diagram of a typical VCP system.

IIT provides this complete system on a PC development card, known as the DVC III Board (Desktop Video Compression III) [25]. The board plugs into a standard ISA slot on a PC, and has ports on the back which connect to an NTSC monitor and camera. The board has 2M of DRAM, 512k of SRAM, and a VCP running at 28 MHz. A simple program that runs on the PC allows a user to download code to the VCP and execute it, in addition to doing some basic run-time debugging.

3.1 Overview of the VCP

There are two major processing elements inside the VCP [24], the *MIPS-X* (a simple RISC, also known as the R-2000), and the *Vision Processor*. The MIPS-X controls the operation of all of the other sections of the chip, including the Vision Processor, the Huffman Encoder and Decoder, the DRAM and SRAM DMA controllers, the H.221 units, the video ports, and the Portal. Figure 3.2 is a block diagram of the VCP chip.

The Vision Processor (VP) is the real processing engine. It is a special-purpose video DSP, with an instruction set that is tailored for MC-DCT based compression routines. It contains its own memories, high-precision multiply-accumulate units, register file, matrix transpose unit, ALU, and motion estimation unit. The MIPS-X can call routines in the VP and transfer data in and out of it.

The Video Input and Output sections perform basic pre- and post- processing tasks. This includes arbitrary horizontal scaling and 7-tap filtering, temporal filtering, chroma sub-sampling, signed to unsigned data conversion, and clamping.

The Huffman Decoder unit is completely programmable, and can be made to decode almost any Huffman code. The Huffman encoder can generate Huffman codes for H.261 and MPEG 1 only.

The two DMA controllers move data around the chip. The RISC can program these DMA controllers to transfer two dimensional data from any one of the sections to any of the other sections on the same bus. A variety of priorities and options are available. Once

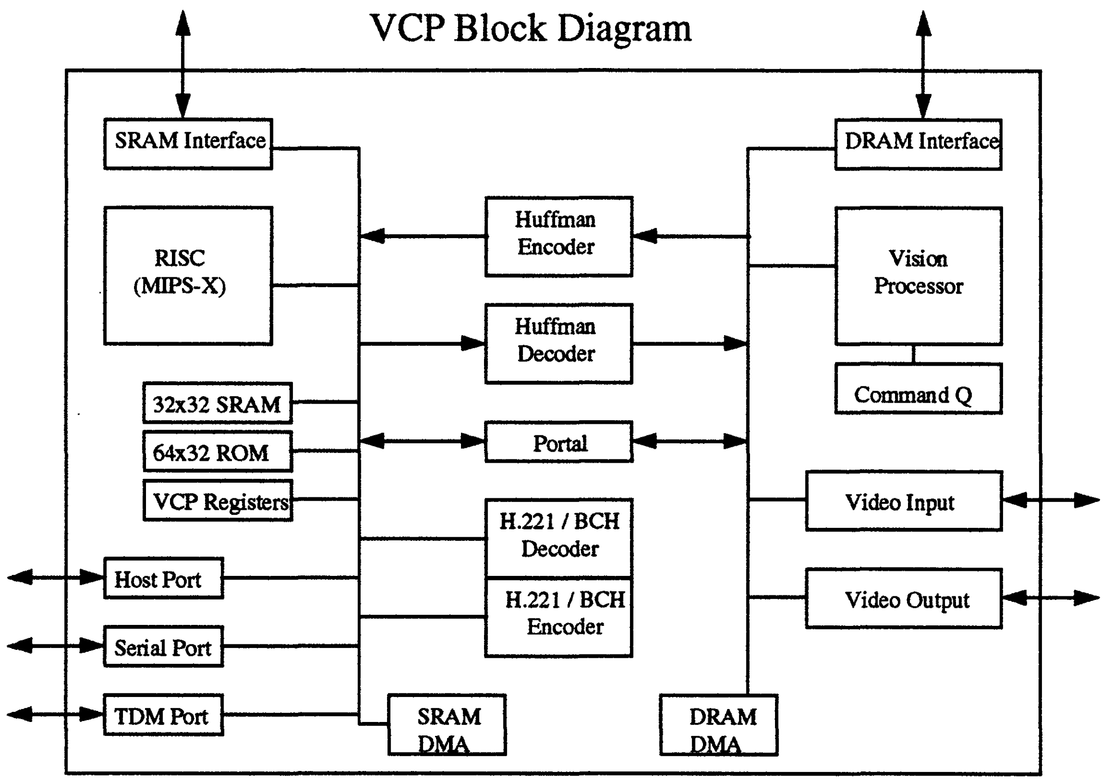


Figure 3.2: VCP Block Diagram

initiated, the DMA will take place without MIPS-X supervision. The DMA controllers can be programmed to interrupt the MIPS-X when the transfers complete, or just change a flag. Movement of data from the SRAM bus to the DRAM bus occurs through the Portal, which consists of a pair of holding registers.

The H.221/BCH Sections assist in generating and decoding framing information and error detecting codes.

3.2 Flow of Data

The best way to understand the operation of the device is to trace the flow of data for a typical frame of encoding and decoding.

For encoding, data from the camera comes into the Video Input section. The MIPS-X can program the video input section to perform any of the available pre-processing options. It then sets up a DMA channel from the video section to DRAM. When the transfer of the frame to DRAM has completed, the MIPS-X may then begin to code it. In a JPEG application, for example, the next step will be to perform an 8x8 DCT on each of the coding blocks. To do this, the MIPS-X initiates a DMA transfer of a single 8x8 block in the image from DRAM into the VP. Once this DMA has completed, the MIPS-X calls the appropriate DCT-Quantizer routine in the VP ROM. After this routine has finished, the MIPS-X transfers the 8x8 block of DCT coefficients out of the VP into DRAM. These three steps (DMA in, execute DCT, DMA out) are repeated for all the coding blocks in the frame. Next, the MIPS-X transfers the quantized frame, one block at a time, from DRAM into the Huffman Encoder (if a standard Huffman code is to be used), or from DRAM to the Portal (if a non-standard Huffman code is to be used). From either the portal, or the output of the Huffman encoder, data can be transferred into SRAM. There, the MIPS-X can generate its own Huffman codes (if the Huffman Encoder Unit was not used), and then assemble a bitstream. Data is then transferred from SRAM to the Host port, the serial port, or the TDM port, depending on the desired channel.

The decoding process follows almost the same path, but backwards. Data comes in from either the TDM, serial, or host ports. It is then transferred to either the Huffman decoder, or to SRAM, depending on the contents of the bitstream. Huffman decoded blocks can then be transferred from the output of the Huffman decoder to DRAM. From there, each block is transferred to the VP. The MIPS-X then calls the IDCT routine. When this has completed, the block is transferred out to DRAM. This process is repeated for each coding block in the frame. Once the whole frame has been reconstructed, a DMA is initiated to transfer the frame from DRAM to the Video output section.

3.3 The Vision Processor

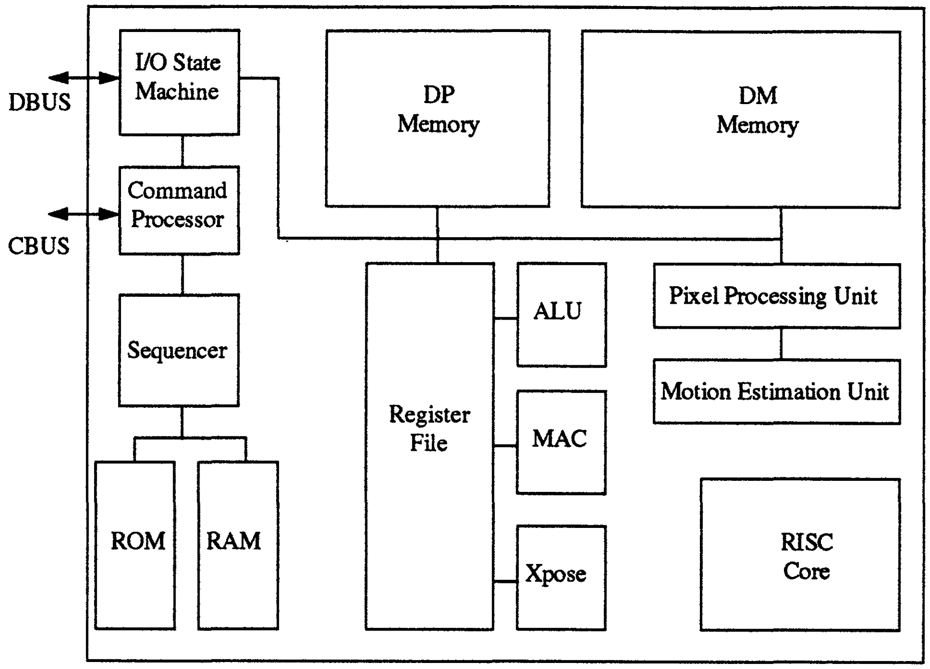


Figure 3.3: Block Diagram of VP

3.3.1 VP Architecture

The VP is a very high performance Video DSP engine [26]. It runs at twice the system clock speed (56 MHz for the DVC III Board). Figure 3.3 is a block diagram of the VP.

Commands from the MIPS-X are issued to the VP over the CBUS (Command Bus), and data is transferred between DRAM and the VP over the DBUS (Data Bus). The two most common commands are a subroutine call and a DMA transfer.

The main functional units in the VP are:

1. *The Sequencer*, which is a simple microsequencing unit that can run routines in either the RAM or ROM. The sequencer allows for conditional branching, up to 2 levels of subroutine call, programmable loop counting, and unconditional jumps.
2. *The I/O State Machine*, which handles DMA transfers into and out of the VP. It can place data in either the DP or DM memories, and runs independently of the sequencer. As a result, DMA and code execution can occur simultaneously. The I/O state machine can be programmed to automatically run-length encode, in a zigzag fashion, an 8x8 block stored in the DP memory. This is useful for coding DCT blocks. Additionally, the I/O state machine can be used to transfer a single 16 bit quantity from the VP to the MIPS-X directly, without using the DRAM or DMA. This feature, called *direct parameter transfer*, is useful for returning motion vectors, VQ codebook addresses, and other scalar results.
3. *The ROM and RAM*, which provide storage for programs. The ROM contains code for DCT, IDCT, motion estimation, etc.
4. *The DP*, which is a general purpose memory. It contains 1024 bytes, and is multi-ported for simultaneous reading and writing. Typically, it is used for storing 8x8 coding blocks.
5. *The DM*, which is also a general purpose memory. It contains 1440 bytes, and is multi-ported for simultaneous reading and writing. Since it is larger than the DP, the DM is ideal for storing reference frames for motion estimation. The DM has a special feature that allows all addresses to be offset from a base. This results in a wrap-around effect, so that addresses which fall off the edge of the memory wrap back around to the beginning.
6. *The Register File*, which has 64 registers, each of which holds 8 bytes. Data can be exchanged between the register file and either the DP or DM memories. Most instructions operate on the register file.
7. *The Multiply-Accumulators*, which can perform four 16x16 multiplies in parallel. Of the 32 bits which result, any 24 can be accumulated into one of two accumulators. A variety of post and pre shifting options are available.

8. *The ALU*, which can add and subtract sixteen 8 bit quantities or eight 16 bit quantities. In fact, it can do both the addition and subtraction simultaneously, an operation required for the DCT.
9. *The Transpose Unit*, which can swap 16 bit quantities within a 64 bit word. It is used for transposing matrices. It is most efficient at transposing 4x4 matrices, which can be accomplished in 6 cycles.
10. *The Pixel Processing Unit (PPU)*, which is ideal for vertical filtering of data. It operates directly from the DM or DP, and operates on many pels at a time. However, its precision is very limited.
11. *The Motion Compensation Unit* , which is one of the VP's strong points. It can compute the SAD (Sum Absolute Difference) between two 8x8 blocks (one in the DM, one in the DP) in only 4 cycles. It operates directly on the DM and DP memories.
12. *The Risc Core*, which is a simple 16 bit computer. It has 16 general purpose 16 bit registers. It can perform basic scalar operations.

Put together, the Motion Compensation Unit, the PPU, the ALU, the MACs and the Transpose unit form the *datapath*, which is the main computational unit in the VP. Every cycle, the VP can execute one datapath instruction (MAC, transpose, motion compensate, PPU, etc.), one instruction from the risc core, one sequencer instruction, and one data movement instruction, depending on the particular combination.

3.3.2 Assessment of the VP

This very unusual architecture leads to a machine that has a set of very particular strengths and weaknesses. Assessing these characteristics is important for deciding on how to recast the SAVC.

1. The motion compensation unit makes *any* comparisons involving the SAD very fast and efficient. Any other method of comparison, such as the MSE, is slower by at least an order of magnitude.

2. Due to the way the four multiply-accumulators obtain their data, vertical filtering is very fast, but horizontal filtering is slower by nearly an order of magnitude. The result of this unusual asymmetry is that separable 2D filters are best implemented by filtering vertically, transposing, filtering vertically, and then transposing back. Since the transpose units are very flexible and fast, this method is optimal.

3. The PPU is not very effective. Most vertical filtering tasks require filters with more than three taps, which usually means that the PPU's limited precision accumulator is not sufficient, and the multiply accumulators must be used.

4. The ALU's addition and subtraction capabilities, when coupled with the multiply accumulators, make an efficient mechanism for computing DCTs and IDCT's. This is certainly no surprise, considering that the VCP was designed to handle MC-DCT based waveform coders.

3.4 Video Input Section

Figure 3.4 is a block diagram of the video input section. Video comes in at CCIR601 resolution at 30 fps. A complementer can convert the data from signed to unsigned. Then, an optional scaler with a 7 tap filter is available. This scaler can decimate the input horizontally by almost any scaling factor. The filter is programmable on the fly. A UV Decimator can further reduce the chroma from 4:2:2 to 4:2:1. A temporal filter can apply a non-linear filtering algorithm to the data. The pels build up in the output FIFO's. The MIPS-X can establish a DMA channel to move the data into reference memory (DRAM).

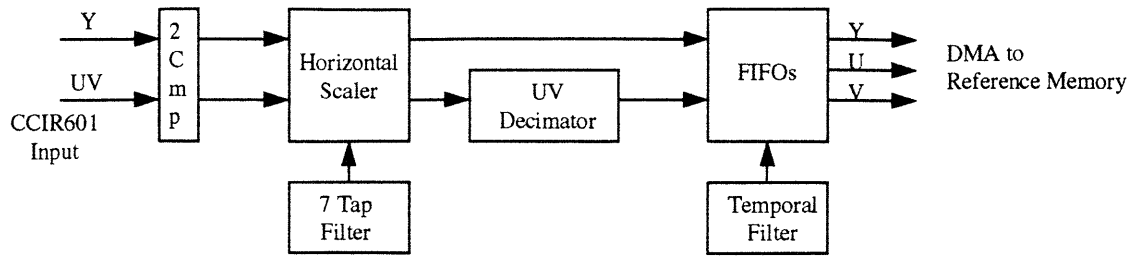


Figure 3.4: Video input Section

All the components of the video input section are programmable and optional.

3.5 Video Output Section

Figure 3.5 is a block diagram of the video output section. It is similar to the input section, but in reverse. Data from reference memory accumulates in the FIFO's until it is needed. A non-linear temporal filter is available, with programmable coefficients. The chrominance can be upsampled to 4:2:2 or 4:4:4. A 7 tap horizontal filter and scaler is available in the horizontal direction. A CCIR clamp automatically limits out-of-range pixel values. An optional two's complementer can convert from signed to unsigned arithmetic. There is also an optional YUV to RGB converter.

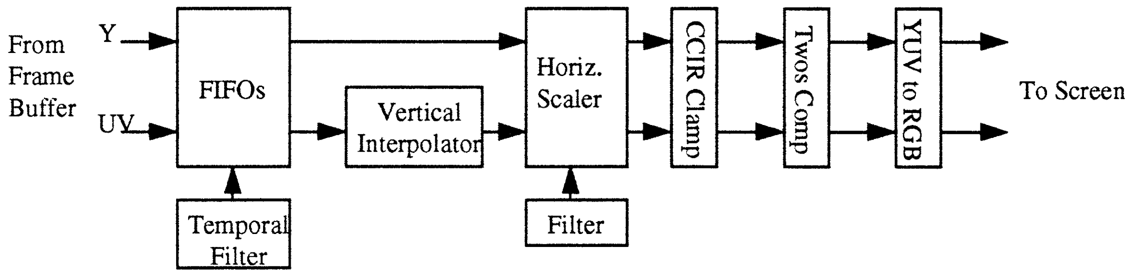


Figure 3.5: Video Output section

3.6 Reference Memory and DMA

The reference memory (DRAM) is used to store frames of video along with any other data which the video sections and VP may need to access. On the DVC III board, 2 Megabytes of DRAM are provided. Logically, this 2M is implemented as a two dimensional memory. It is 128 "longwords" wide, by 8192 rows deep. A longword is the smallest unit of data that can be transferred into and out of memory. It is 32 bits long, which implies that there are 4 pels per longword (a pel of luma occupies a single byte). Luminance and chrominance are stored separately.

For example, a frame of SIF luminance would require a section of memory 88 longwords wide by 240 rows deep.

The DMA controller is responsible for moving data into and out of the reference memory. DMA occurs on two-dimensional blocks. To initiate a DMA, the MIPS-X must provide several pieces of information: the x and y address and size of the block to be transferred, its destination, and a flag indicating if the data contains RLA tokens. When the data contains RLA tokens, the DMA controller will ignore the block size, and continue to read data until it reaches an end-of-tokens marker. This is useful since the size of a block of RLA data is data-dependent.

In fact, there are two separate DMA controllers on the DRAM bus. One is for DMA into and out of the VP, and one is for all other DMA. VP DMA is different since it also requires an address into the DM or DP memories, in addition to the above information. The VP DMA controller is structured as a queue. DMA requests from the MIPS-X queue up in the "Bus Controller Command Queue", which is 32 instructions deep. Besides DMA requests, VP subroutine calls and "wait" instructions also queue up in the command queue. Wait instructions tell the DMA controller not to execute the next instruction in the queue until a certain condition is satisfied. Two conditions are available: VP has completed executing a subroutine, and VP DMA has completed. These wait instructions are necessary to insure that the VP has completed a routine before the results are DMA'ed out, and to insure that a routine does not begin to execute until the input data has completed DMA'ing into the VP. The VP Bus Controller is smart enough to automatically wait for a DMA to complete before initiating the next DMA, and to wait for a subroutine to complete before calling the next subroutine.

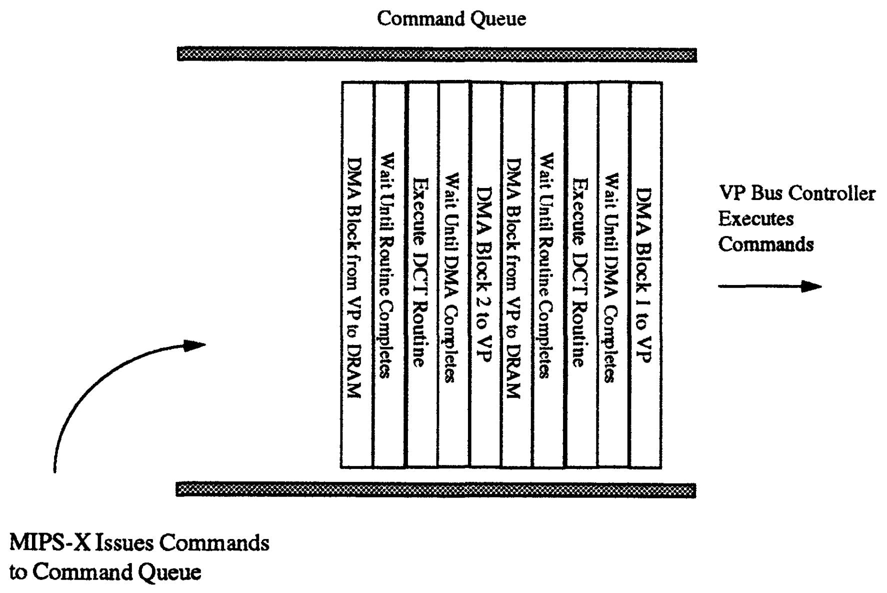

Command Queue
DMA Block from VP to DRAM
Wait Until Routine Completes
Execute DCT Routine
Wait Until DMA Completes
DMA Block 2 to VP
DMA Block from VP to DRAM
Wait Until Routine Completes
Execute DCT Routine
Wait Until DMA Completes
DMA Block 1 to VP
VP Bus Controller Executes Commands
MIPS-X Issues Commands to Command Queue

Figure 3.6: State of the Command Queue while performing the DCT on the blocks in a frame.

Figure 3.6 is a diagram of the state of the command queue for a situation in which all the blocks in the current frame are being transformed via the DCT.

Chapter 4

Porting the Algorithm to the VCP

The first step in recasting the algorithm is to port it directly to the VCP without any changes. After this has been accomplished, the timing requirements of the ported algorithm can be estimated. From there, the various tasks of the algorithm can be redesigned in order to reduce computation time.

There are three steps in porting the algorithm: extraction, partition, and modification. The first step, extraction, is to determine what the main tasks of the algorithm are, and to develop a flow chart for their interaction. Once the algorithm is understood, this step is quite simple. However, it is necessary to explicitly determine the various tasks in order to accomplish the second step. In partitioning, the various tasks of the algorithm are distributed among the resources available. In the case of the VCP, the available resources include the MIPS, VP, Video Filters, Huffman State Machines, BCH encoders, etc. Additionally, the data must be distributed among the two available memories: MIPS SRAM and reference memory (DRAM). After partitioning, the last step is to modify the way the section of the algorithm is being implemented in order to best suit the available resource.

4.1 Extraction

The algorithm is quite amenable to the extraction process. Figure 4.1 depicts a flow chart which represents the various tasks involved, and their interaction. It is important to note that several tasks occur in loops. In particular, *Find VQ rate* appears in a double loop, and *Encode MV* appears in a single loop.

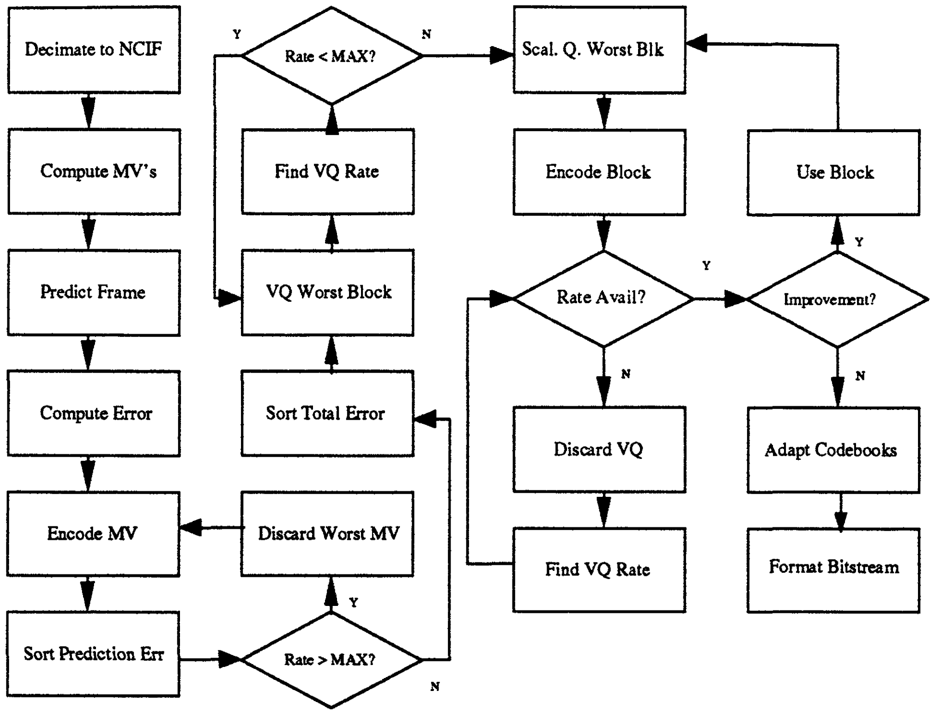


Figure 4.1: SAVC Flowchart

4.2 Partition and Modification

Partition and modification are much more difficult than extraction. The difficulty is exacerbated by the fact that one directly affects the other. To simplify the discussion, both will be considered at the same time.

4.2.1 Data Partition

The main pieces of data used in the encoding process are:

- SIF Video frames
- NCIF Video frames
- Intermediate NCIF frames (motion compensated frame, motion residual, etc.)
- Intra and Inter frame codebooks
- Motion Vectors
- VQ Addresses
- RLA Encoded, Quantized, DCT blocks

- Block MSE's
- Huffman tables

There are two criteria used to determine which memory is to be used for each piece of data. The first criteria is *size*. Since there is much more DRAM than SRAM, pieces of data which are very large should probably be stored in DRAM. The other factor, and by far the most important one, is *accessibility*. Data which needs to be manipulated often by the MIPS-X should probably be stored in SRAM. Data which is used mainly by the VP or Video Units should probably be stored in DRAM.

For the SIF and NCIF Video frames, and for the intermediate NCIF frames, the reference memory (DRAM) is the best place to store the data. The only units which ever need to access the data are the VP and video units, both of which are attached to the DRAM bus. Additionally, video data often consumes a great deal of space, making the reference memory an even better choice.

For the codebooks, the DRAM is also the optimal place for storage. The MIPS-X never actually needs to manipulate the contents of the codebooks. The codebook search is performed by the VP, with the VQ address returned via direct parameter transfer. Although the MIPS-X will manage the codebook adaptation algorithm, the algorithm itself is independent of the codebook content, and only depends on the usage count and improvement measures, which are stored in the SRAM. Thus, only the VP ever uses the actual codebooks, making DRAM ideal for codebook storage.

The motion vectors, vector quantization addresses, and block MSE's are generated by the VP and can be transferred to the MIPS-X via direct parameter transfer. The MIPS-X uses the data to make encoding decisions and to generate the bitstream. Therefore, it is more efficient to store them in SRAM.

The RLA coded, quantized DCT blocks are generated by the VP. The MIPS-X uses them to determine the encoding rate, and also includes them in the bitstream. Therefore, they are more efficiently stored in the SRAM. However, since the coded blocks are large, they cannot easily be transferred to the MIPS-X by direct parameter transfer. As a result, they must be DMA'ed to DRAM, and then transferred to the SRAM through the portal. This requires a small buffer to be maintained in DRAM.

The Huffman codes are better suited to storage in SRAM. Although they can be large in size, the MIPS-X performs all the Huffman encoding, not the VP (the VP does not have a large enough memory).

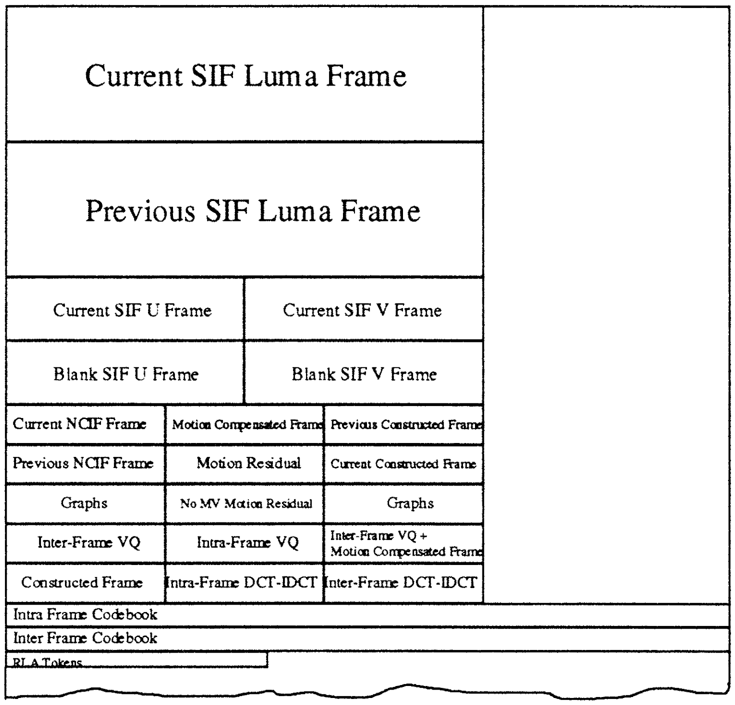


Figure 4.2: DRAM Memory Map

Figure 4.2 is a diagram of the memory map of the DRAM.

4.2.2 Decimation

The first step in the algorithm is to decimate the input frame from SIF (352x240) to NCIF (112x80). The conversion first requires that the SIF be converted from 352x240 to

336x240. This is accomplished by clipping 8 pels from each side. Then, the image is decimated by three both vertically and horizontally.

There are two options for performing the decimation. The first is to use the horizontal filters in the video input section. These provide 7 taps. The horizontally filtered and subsampled frame would then be DMA'd to the reference memory. Block at a time, pieces of the subsampled frame would then be transferred to the VP for vertical filtering. The fully decimated blocks would then be transferred to the current NCIF frame in reference memory. The other option is to use the VP for both vertical and horizontal filtering. SIF frames would be transferred directly from the video input section to the current SIF frame in DRAM. Then, block at a time, pieces of the frame would be transferred to the VP for vertical and horizontal filtering and subsampling. The fully decimated blocks would then be transferred to the current NCIF frame in DRAM.

The second option is more efficient. Since motion estimation is done at 1/3 pel accuracy, an NCIF frame must be interpolated up to SIF later on for the motion search. By doing the decimation entirely in the VP, the unfiltered input SIF frame will be available. If the horizontal filters in the video input section are used, the unfiltered SIF frame will never be stored.

The decimation is accomplished by first filtering the SIF image with the FIR filter pictured in Figure 4.3. This filter was obtained by taking an ideal lowpass filter with a cutoff frequency of $\frac{\pi}{3}$, limiting it to seven taps using a rectangular window, and rounding each coefficient to an integral multiple of $\frac{1}{65536}$. The rounding is needed since the VP is a fixed point processor.

Figure 4.4 is a plot of the frequency response of this filter.

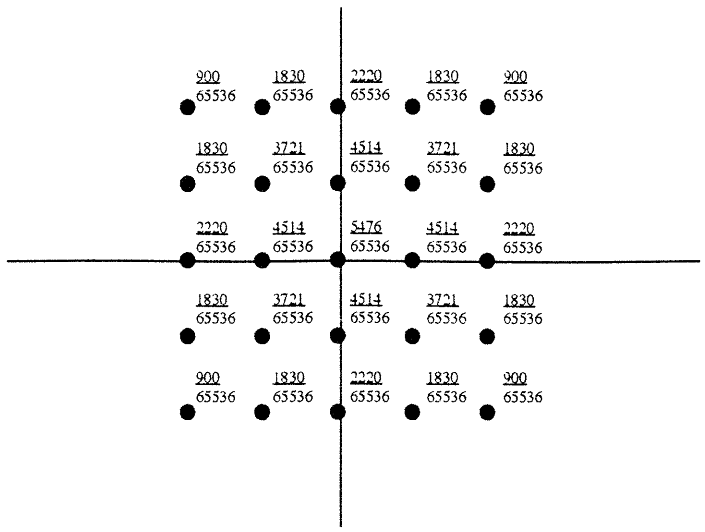


Figure 4.3: Filter used for decimation

In the non-real time implementation of the algorithm, the decimation is accomplished by taking the entire image, reflecting the sides, filtering vertically, and then filtering horizontally. Then, the image is subsampled by three in each direction. This implementation is not appropriate for the VP, however, for several reasons:

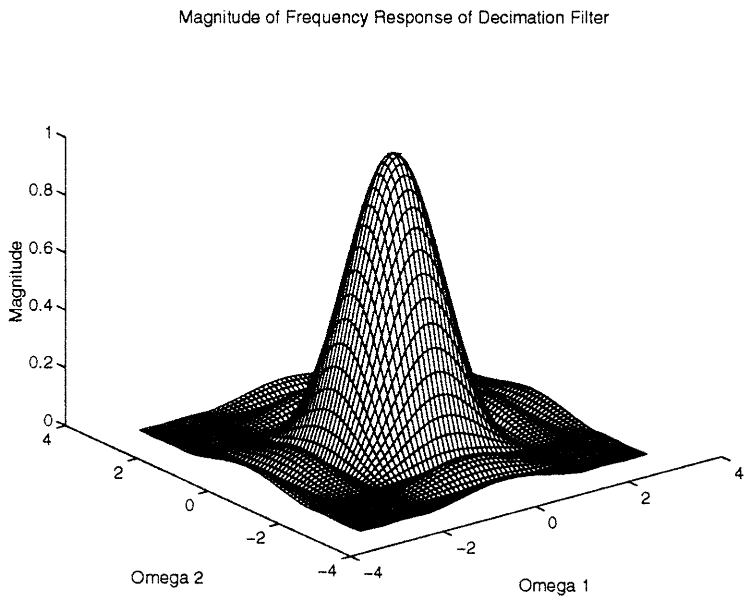


Figure 4.4: Frequency Response of Decimation Filter

1. The VP does not have enough memory to hold an entire SIF frame at a time.
2. Filtering the entire frame is too computationally intensive.
3. The VP cannot filter horizontally very well.

To tackle the first problem, the implementation on the VP operates on a single block at a time. The size of the block can be determined by working backwards. The final result needs to be an 8x8 NCIF block. To generate such a block, a 24x24 SIF block is decimated. However, since the filter has seven taps, three additional pixels on the left (top), and one on the right (bottom) are needed. This brings the size up to 28x28. Since the reference memory operates on data in four pel units, an additional pel on the left, and three pels on the right must also be transferred to the VP. Thus, the blocks to be operated on are 32x28 pels.

Instead of filtering every pel in the 32x28 block, a multirate filtering technique is used whereby only those pels which will be sampled are ever filtered [21]. This reduces the computation significantly (by roughly a factor of nine), solving the second problem. Since the VP is efficient at vertical filtering, the block is filtered vertically first. Only the pels in every third row are actually filtered. Figure 4.5 illustrates this concept. The 'x' represents a pel which will eventually be sampled to generate the NCIF block. The 'o' represents pels which will not be sampled. The lightly dotted box represents the window of the vertical filter, and the heavy dotted box represents the pel which is generated by the filter. Notice that *all* the pels in a row must be filtered, since they will all be needed for horizontal filtering.

After the vertical filtering is complete, the 28x24 block is subsampled vertically, resulting in a 28x8 block. This block is then transposed, and filtered in a similar manner, with only the pels in every third row being filtered. The transposed 24x8 block is then subsampled horizontally, resulting in an 8x8 block, which is transposed once again to yield the final 8x8 result.

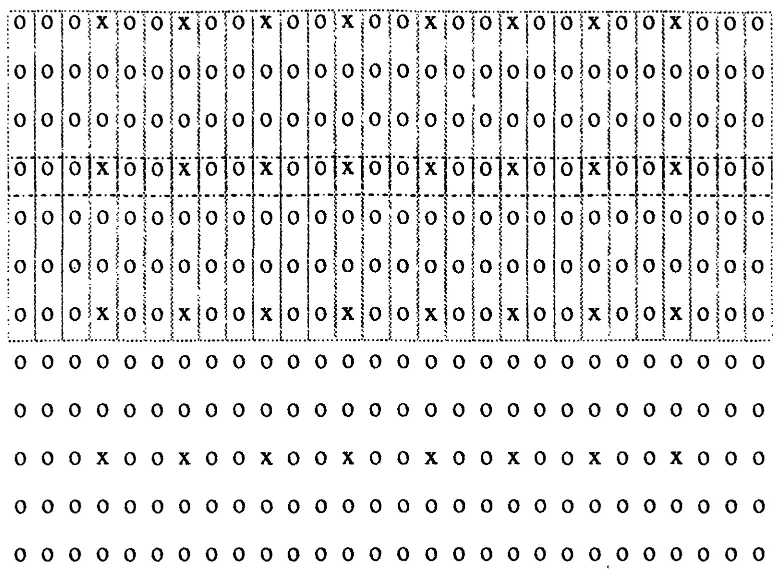


Figure 4.5: Multirate filtering

So, in order to decimate the entire frame, the MIPS-X transfers a 32x28 block from the current SIF input frame into the VP, calls the filter routine, and then transfers the 8x8 block from the VP to the current NCIF frame. This procedure is repeated for all 140 blocks in the current frame.

4.2.3 Motion Estimation

For each block in the current frame, a motion vector in the range of -15 to 15 SIF pels in both the x and y directions must be determined. A full search block-based motion estimation technique is used. Full search at SIF resolutions requires a reference block of size 24x24, plus 15 pels on each side, bringing the size up to 54x54. The MSE between the current block and all 961 possible blocks in the previous frame are computed. The motion vector which yields the smallest error is then chosen.

The VP is ideally suited for motion estimation. However, there are three main difficulties in implementing this specific task on the VP:

1. The 54x54 reference block does not fit in the DM.
2. Searching all 961 blocks in the previous frame requires too much computation.

3. Computing the MSE requires too much computation.

The third problem is easily overcome by using the SAD as an error criterion, since the motion compensation unit in the VP computes this figure. Overcoming the first difficulty can be accomplished by dividing up the search region. Assume that the -15 to 15 pel range is being split up into sub-ranges, each of size Δx in the x direction and Δy in the y direction. To minimize the amount of time spent DMA'ing sections into the DM, both Δx and Δy should be made as large as possible. The maximum size is limited by the space in the DM memory. The size of a block of reference is $24+\Delta x$ by $24+\Delta y$. The size of the DM is 1440 bytes. This limits Δx to 16 and Δy to 12, given the size limitations of the DM. The final constraint is that Δx and Δy must evenly divide the search range, which is 31 values in the x direction and 31 in the y. A reasonable choice for Δx is therefore 16, and for Δy is 11. These values evenly divide a range which is slightly greater than the actual search range.

These values of Δx and Δy imply that there are 6 sub-ranges. For each range, the appropriate reference block must be DMA'ed to the DM, searched, and the results stored for comparison with the results of the next search range.

Unfortunately, the second problem, that of computational complexity, is only exacerbated by the division of the search range into sub-ranges (more DMA is now required). As Chapter 5 will illustrate, the computation time is still too large, and a new motion estimation technique is needed. A coarse-fine approach requires much less computation, and is also well suited to implementation on the VP. Chapter 8 details the analysis of coarse-fine motion estimation.

4.2.4 Motion Compensation

Once the motion vectors have been obtained, they must be applied to the previously constructed frame to generate the predicted frame. This procedure, called *motion compensation*, is somewhat complicated since the motion vectors are in 1/3 pel increments. Thus, in order to use them, the previously constructed frame (which is at

NCIF) must be interpolated up to SIF. At SIF resolutions, the motion vectors are now in a range of -15 to 15 pels, in integral increments. The appropriate 24x24 block can be located, and then decimated back to an 8x8 NCIF block. In the non-real time version of the coder, the entire previously constructed frame is interpolated to SIF using a separable 23-tap filter. The motion estimates are then used to select a 24x24 block. This block is then subsampled to 8x8.

Previous results show that a long filter is needed for the interpolation [19]. Any aliasing noise introduced by filter errors is directly translated into prediction errors. At such low bitrates and resolutions, the motion compensation needs to be as accurate as possible. With 23 taps, the aliasing noise is reduced to less than -40 dB [19].

As with decimation, there are several aspects of the implementation in non-real time that do not fit well on the VCP:

1. The entire NCIF frame does not fit into the DM memory.
2. 23 point convolution requires too much computation.
3. Interpolation of an entire frame to SIF requires too much computation.

Combating the first difficulty is achieved by performing the interpolation on single blocks at a time. Battling the other two problems requires more effort.

As soon as interpolation is moved to block-based (as opposed to frame-based), it becomes clear that interpolation is not always necessary. When a motion vector is zero, no interpolation is needed. More generally, however, when a motion vector is a multiple of three in both x and y directions, no interpolation is needed either. If the interpolating filter preserves the original pels in the NCIF image (as an ideal lowpass filter would), any predicted block whose pels come entirely from the NCIF frame does not need to be generated by interpolation. Further gains can be made by realizing that no interpolation is needed horizontally when the x motion vector is a multiple of three, and no interpolation is needed vertically when the y motion vector is a multiple of three.

When interpolation is needed, multirate filter theory can once again reduce the computation enormously. Since the final result of the motion compensator is an 8x8 block, there is no need to generate all 24x24 pels in the interpolated block - only those which are in the output need to be computed. In addition, while generating those pels, it is not necessary to do all 23 multiplication's. Only 8 of the points will be non-zero. Figure 4.6 illustrates this concept. The x's represent pels in the original 8x8 block. The o's represent the zeroes in the 24x24 interpolated block which need to be filtered. The boxes are around those pels which are needed for a motion vector of (1,1) at SIF (one to the right, and one down). As with decimation, the first step is to filter vertically. All the pels in row A, row B, etc. need to be filtered. However, only those pels which are in the same column as an 'x' will have any non-zero inputs. Therefore, the vertical filtering only needs to be done in 8 of the columns. For each of those eight columns, only every third pel is nonzero. Therefore, only 8 of the 23 taps in the filter will be multiplied by nonzero inputs. Which of the 8 depends on the distance away from an 'x' - 1 or 2. Thus, the 23 tap filter can actually be thought of as two 8 tap filters, both of which represent one of the three polyphases of the 23 tap filter. Which filter to use depends on the motion vector. If the motion

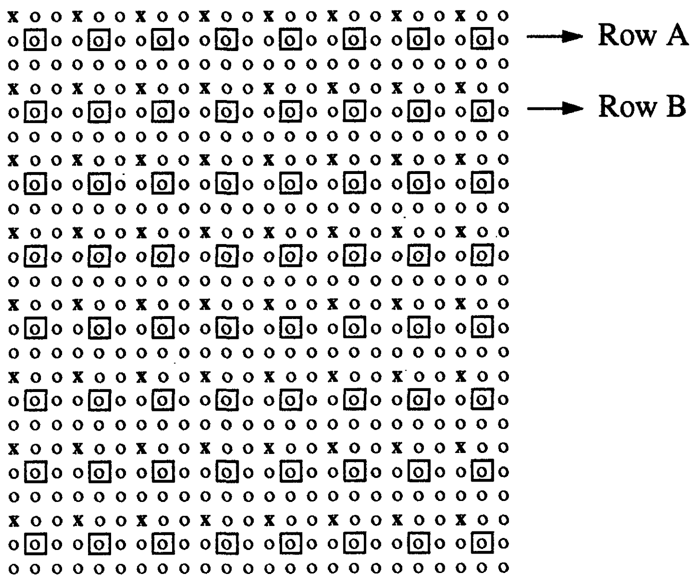


Figure 4.6: Multi Rate filtering for motion compensation

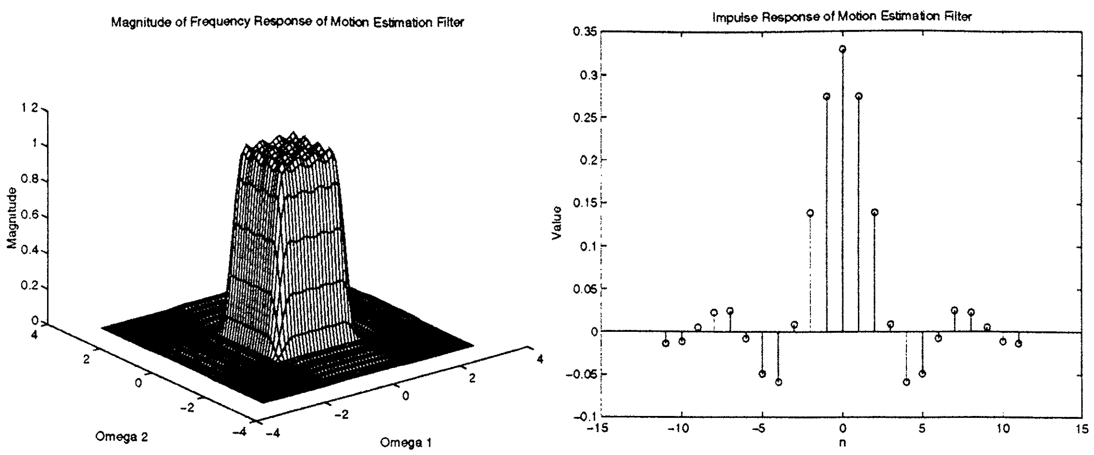


Figure 4.7: Frequency Response and Impulse Response of Compensation Filter

vector, modulo 3, yields 1, then phase A is used. If it yields 2, then phase B is used (if it yields 0, no filtering is needed, as described above).

After the vertical filtering of the 8x8 input block by one of the 8 tap filters, the block is transposed, and the horizontal filtering is done in a similar manner. Thus, the motion compensation problem has been changed from one of a 23 tap filter interpolating an NCIF frame to SIF, to one of filtering an 8x8 block with an 8 tap separable filter.

The implementation follows directly. An appropriate section of the previously constructed frame is DMA'ed into the DM memory. The location of the block needs to be offset by the integral portion of the motion vector. Again, due to the 4-pel alignment rule, some extra pels must be transferred, along with the extra 3 needed on each side for the filtering near the edges. Then, the MIPS-X can transfer a parameter to the VP (via the direct parameter transfer mechanism) indicating which of the filters to use. Then, one of three routines is called. One routine is for both horizontal and vertical filtering, one is for just horizontal, and one is for just vertical. After the VP has completed, the MIPS-X can DMA the 8x8 compensated block out of the VP into the motion-predicted frame in DRAM.

As with decimation, the filter needed to be redesigned in order to meet the fixed point architecture. While redesigning, it was essential to make sure that each of the polyphase components of the filter had a DC coefficient of one. Otherwise, a constant background input block would not be interpolated to a constant background output block - resulting in a very patchy artifact. Figure 4.7 is a plot of the impulse response and magnitude of the frequency response of the original filter. Figure 4.8 is a plot of the impulse response and frequency response for the $+\frac{\pi}{3}$ phase, and figure 4.9 is a plot of the impulse response and frequency response for the $-\frac{\pi}{3}$ phase.

4.2.5 Error Computation

After the motion compensation has occurred, the residual must be computed. The residual is the difference between the motion compensated (predicted) frame, and the original

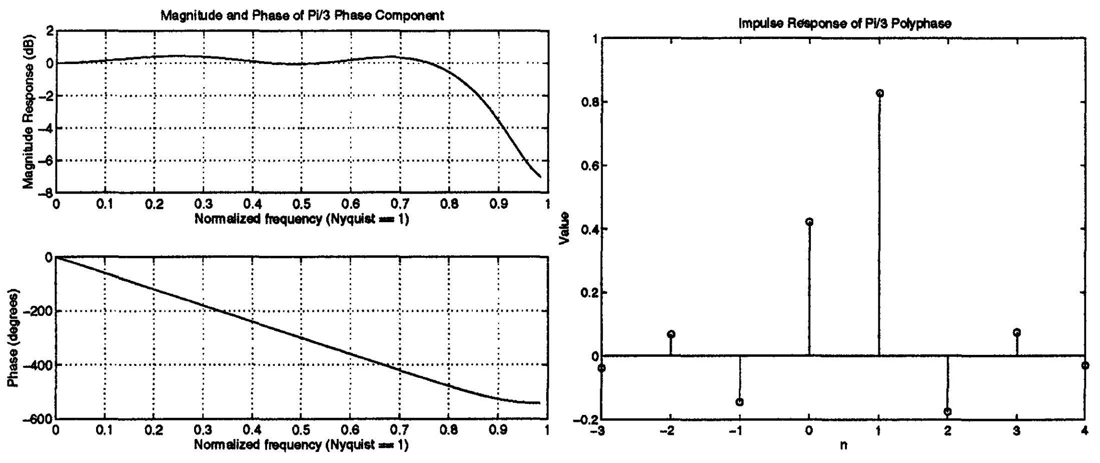


Figure 4.8: Frequency Response and Impulse Response of $+\frac{\pi}{3}$ Polyphase

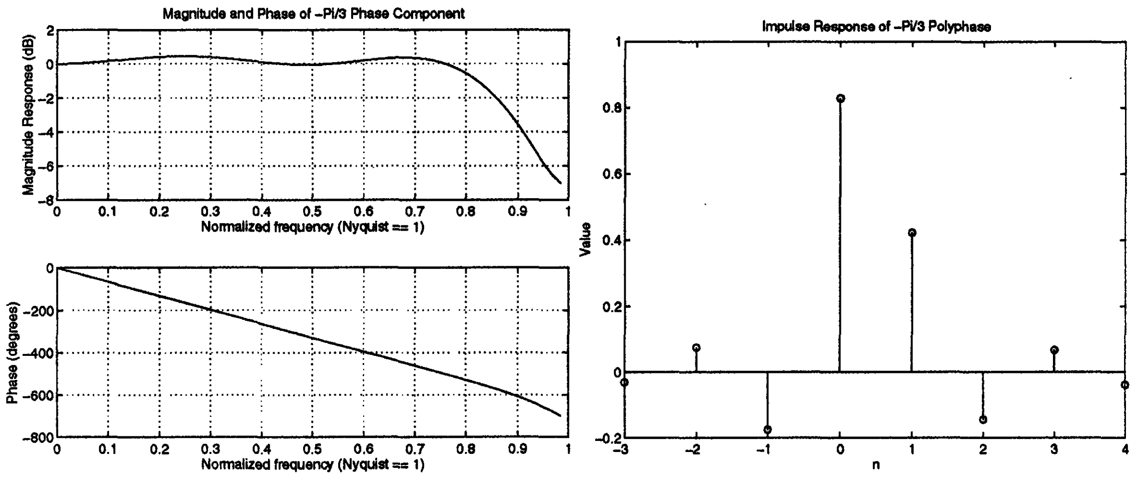


Figure 4.9: Frequency Response and Impulse Response of $-\frac{\pi}{3}$ Polyphase

frame. The residual after motion compensation with a zero motion vector (i.e., the difference between the current and previous frames) needs to be computed also. For both the zero-motion residual, and the compensated residual, the SSE (Sum Squared Error) for each block needs to be computed.

This portion of the algorithm maps very well to the VP. The datapath ALU can be used to subtract the two frames. Since eight 8 bit subtractions can be done at once (the 8 additions that are also done are unneeded), eight uses of the ALU is all that is required to compute the residual. The VP can then move the residual right back to the DM, where the MIPS-X can read it back via DMA. Computing the MSE is then a matter of squaring each term in the residual an adding them up. The multiply-accumulators can do this very efficiently. The final result can then be moved to the MIPS-X via direct parameter transfer.

The procedure, then, is as follows. The MIPS-X initiates a DMA to transfer an 8x8 block from the current NCIF frame in DRAM to the DP. It also initiates a transfer to move the corresponding block in the predicted frame into the DM, and the corresponding block in the previous input frame to a different address in the DM. The MIPS-X then calls the error computation routine in the VP. This routine will generate the difference between the current input frame and previous input frame, place it in the DM, and pass the SSE back to the MIPS-X via direct parameter transfer. By calling another routine, the difference between the current input frame and the motion compensated frame will be computed, the residual placed in the DP, and the SSE passed to the MIPS-X. The MIPS-X can then transfer the residuals to reference memory for later use. The procedure is repeated for all 140 blocks in the frame.

4.2.6 Sorting

At this point, the coder can compute the prediction gain (see Chapter 2) for each block. These gains must then be sorted to determine which motion vectors to discard if the rate required for coding them is too high. Sorting is needed in several other places in the code as well, so a generic sorting procedure was written. The non-real time coder uses simple insertion sorts, as these are the most intuitive sorts, and among the easiest to write code for. However, a faster sort is needed for the real-time system.

The VP is not well suited for sorts. It is very fast at finding the largest or smallest element in a set, but it is incapable of giving the index of that element in the set. Since it is the index that the MIPS-X needs (the index is the coded block position), the sorts are more efficiently done in the MIPS-X. A generic heapsort routine [22] was used. It takes two arrays, and performs the same element manipulations on both, but makes decisions based on only one. Therefore, one array can contain the SSE's, and the other the coded block positions. Upon return, the coded block positions will be sorted in order of increasing SSE.

4.2.7 Motion Vector Rate Computation

After all the motion vectors have been computed, their rate is calculated. If this rate exceeds a threshold, then a motion vector is discarded, and the rate is recomputed. This procedure is repeated until the rate falls below the threshold.

In order to compute the motion vector rate, a Huffman table must be generated based on the statistics of the motion vectors. Since these statistics change each time a vector is discarded, the table must be continually regenerated. Each motion vector must also be looked up in a global Huffman table which is read in upon initialization of the coder. Since both of these operations require pointer manipulations and array indexing, they are not well suited for the VP or any other unit on the chip. The Huffman Encoding unit has tables for MPEG I and Px64 DCT coefficients, but nothing for motion vectors. Therefore, the motion rate computation must be implemented on the MIPS-X. There are no obvious ways to improve the implementation of the Huffman table generation, which is a very time intensive operation. Chapter 6 considers a modification of this portion of the algorithm which resolves this difficulty.

4.2.8 Vector Quantization

Vector quantization was one of the most difficult routines to port to the VCP. In the non-real time encoder, two 256 entry codebooks of 8x8 blocks are maintained. Each of the blocks to be quantized is compared to every entry in each codebook, and the MSE is computed. The entry with the smallest MSE is deemed the best match. Over half of the time required for encoding in the non-real time coder was spent on this operation. Several aspects of the non-real time implementation make its application to the VCP difficult:

1. The VP does not have enough memory to hold either codebook.
2. The VP does not have enough memory to hold all the coding blocks.
3. The MSE takes too long to compute.

The first step towards battling the third difficulty is to use the SAD, which does not require multiplication's. With this error criteria, an enormous improvement in performance can be made by realizing that vector quantization is very much like motion estimation. The similarities are numerous:

- Both are searches.
- Both seek to find the best match by checking all possibilities and choosing the match with the smallest error.
- Both compare a single 8x8 block to many 8x8 blocks.

Therefore, by utilizing the structures in the VP which are meant for motion compensation, vector quantization can be performed efficiently as well, even though the VP was not specifically designed for it.

Unfortunately, the one difference between motion estimation and vector quantization is a big one: search space. In motion estimation, the search range for each coding block is different, and can easily be placed in the DM memory. In VQ, however, each coding block has the same search space (both codebooks), which does not fit into the DM memory at all. In fact, only 16 codewords can fit in the DM memory at a time.

The difficulty does not imply that vector quantization is impossible - it just implies that it may not be fast. A first stab at the implementation, called *Approach I*, in pseudo-C, is as follows:

```
for(CodingBlock = 0; CodingBlock < NumberBlocks; CodingBlock++)
      {
          DMA_From_DRAM_To_DP(CodingBlock);
          WaitUntilDMAFinishes();
          for(CodebookBlock = 0; CodebookBlock < 16; CodeBookBlock++);
              {
                  DMA_From_DRAM_To_DM(CodebookBlock);
                  WaitUntilDMAFinishes();
                  CallVPRoutine(VectorQuantize);
                  WaitUntilRoutineFinishes();
              }
          DirectParamTransfer(VQMatch[CodingBlock]);
      }
```

A simple model can be developed to study the amount of time required for vector quantization. We can define the following constants:

α = Time for the VP to search one coding block into 16 codewords (one section)

β = Time required to DMA 16 codewords (one section) into the DM

δ = Time required to DMA a single coding block to the DP

N = Number of coding blocks (140)

A = Number of codebook sections in a codebook (16)

Using these constants, the time required for Approach I can be written as:

$$T = N[\delta + A[\beta + \alpha]]$$
$$T = N\delta + NA\beta + NA\alpha$$

As a standard for comparison, we can expect that the VQ can occur no faster than the time required to search every block into the codebook. This time, T_{OPT}, can be written as:

$$T_{OPT} = NA\alpha$$

Even though Approach I will work, it is wasteful. Most of the time is spent DMA'ing sections of the codebook into the DM (the $NA\beta$ term above). In fact, each section of the codebook will be DMA'd into the DM 140 times (given that there are 140 coding blocks). Instead, it is faster to keep several coding blocks in the DP at one time. When a section of the codebook is placed in the DM, all the coding blocks in the DP are searched into it. Although this will not reduce the time the VP must spend computing, it reduces the DMA time significantly. Let Δ (DELTA) represent the number of coding blocks placed in the DP at a time. This approach, called Approach II, can be coded as follows:

```
for(CodingBlock = 0; CodingBlock < NumberBlocks; CodingBlock+= DELTA)
      {
          for(i = CodingBlock; i < CodingBlock + DELTA; i++)
              DMA_From_DRAM_To_DP(i);
          WaitUntilDMAFinishes();
          for(CodebookBlock = 0; CodebookBlock < 16; CodeBookBlock++);
              {
                   DMA_From_DRAM_To_DM(CodebookBlock);
                   WaitUntilDMAFinishes();
                   for(i = 0; i < DELTA; i++)
                         CallVPRoutine(VectorQuantize);
                   WaitUntilRoutineFinishes();
              }
          for(i = CodingBlock; i < CodingBlock + DELTA; i++)
              DirectParamTransfer(VQMatch[i]);
      }
```

Using the constants defined above, the time required for Approach II can be modeled as follows:

$$T = \frac{N}{\Delta}[\Delta\delta + A[\beta + \Delta\alpha]]$$

$$T = N\delta + \frac{NA\beta}{\Delta} + NA\alpha$$

The second term is thus reduced by a factor of Δ compared to Approach I. Approach II, however, is still wasteful, and not yet close enough to T_{OPT}. While the DMA of codebook sections is occurring, the VP is idle. Instead, it would be more efficient if the VP could be working on a section of the codebook while the next section is being DMA'd to the DM. Since the DM memory is multi-ported, the I/O State Machine can write to the DM while code is executing - as long as the code does not try and write to the DM either. However, the VQ routine has no need to ever write to the DM, so this restriction is no problem.

Instead of using a 16 codeword section of the codebook, an 8 codeword section of the codebook will be transferred to the DM. The DM memory can then be split into two halves, each half containing a section. While one half is being searched, the other half can be filled with data from the I/O state machine. Once both the DMA and the computation have completed, the DM memory can be rotated, so that the VP can now search the other side of the DM with the new data, and the next section of data can be DMA'd in on top of the old data.

This approach, called Approach III, can be coded as follows:

```
for(CodingBlock = 0; CodingBlock < NumberBlocks; CodingBlock+= DELTA)
      {
          for(i = CodingBlock; i < CodingBlock + DELTA; i++)
```

```
            DMA_From_DRAM_To_DP(i);
        WaitUntilDMAFinishes();
        for(CodebookBlock = 0; CodebookBlock < 32; CodeBookBlock++);
            {
                DMA_From_DRAM_To_DM(CodebookBlock);
                for(i = 0; i < DELTA; i++)
                    CallVPRoutine(VectorQuantize);
                WaitUntilRoutineFinishes();
                WaitUntilDMAFinishes();
                RotateMemory();
            }
        for(i = CodingBlock; i < CodingBlock + DELTA; i++)
            DirectParamTransfer(VQMatch[i]);
    }
```

Modeling the time required is a little more complicated now. Let T(Δ) represent the amount of time required for both the DMA of a codebook section, and the VQ of all the blocks, to complete. This time is a function of Δ since the I/O state machine will perform its DMA by stealing memory cycles from the sequencer. The larger Δ is, the more unused memory cycles available. Assume that some fraction, λ, of the searching time $\frac{\Delta\alpha}{2}$ is available time for the I/O state machine to work. Thus, if:

$$\lambda\left(\frac{\Delta\alpha}{2}\right) > \frac{\beta}{2}$$

the I/O state machine will have enough time to complete its transfer before the search is completed. In that case,

$$T(\Delta) = \frac{\Delta\alpha}{2}$$

However, if the inequality is reversed, the sequencer will complete before the I/O is finished, and:

$$T(\Delta) = \frac{\Delta\alpha}{2} + \left(\frac{\beta}{2} - \lambda\frac{\Delta\alpha}{2}\right)$$

$$T(\Delta) = \frac{1}{2}[\Delta\alpha(1-\lambda) + \beta]$$

Using this model, we can estimate the time required for Approach III as:

$$T = \frac{N}{\Delta}[\Delta\delta + 2A[T(\Delta)]]$$

Substituting T(Δ):

$$T = \frac{N}{\Delta}\left[\Delta\delta + 2A\left[\frac{\Delta\alpha}{2}\right]\right] \quad \Delta > \frac{\beta}{\alpha\lambda}$$
$$T = \frac{N}{\Delta}\left[\Delta\delta + \frac{2A}{2}[\Delta\alpha(1-\lambda)+\beta]\right] \quad \Delta < \frac{\beta}{\alpha\lambda}$$

Reducing further:

$$T = N\delta + NA\alpha \qquad \Delta > \frac{\beta}{\alpha\lambda}$$
$$T = N\delta + NA\left[\alpha(1-\lambda)+\frac{\beta}{\Delta}\right] \qquad \Delta < \frac{\beta}{\alpha\lambda}$$

Figure 4.10 is a plot of T vs. Δ.

Thus, if Δ is made larger than $\frac{\beta}{\lambda\alpha}$, the time required can be brought close to T_{OPT}. The time is especially close to T_{OPT} if $A\alpha >> \delta$. This relationship is clearly true, since the time to DMA a single 8x8 block is much less than the compute time for the search of an entire codebook. Thus, Approach III is optimal.

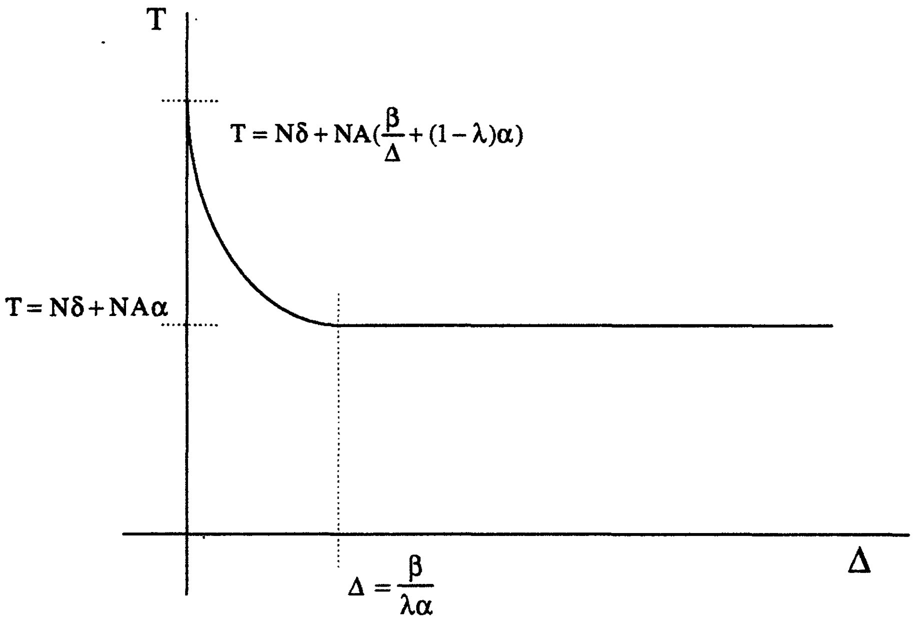


Figure 4.10: T vs. Δ

To determine the value of Δ, the actual running time of the implemented routine was computed as a function of Δ, for values of 2,4,5,7,10,14, and 20 (since these values also divide 140). Figure 4.11 plots this graph. Notice that the curve flattens out around Δ=14. Implementing Approach III's routine on the VP was somewhat tricky, for two reasons:

1. Δ reaches its optimal value at 14. Since the SAD and location of the best match need to be stored for all 14 coding blocks, there is not enough room in the general purpose risc registers for this information.
2. Overhead can be significant (as it is in coarse motion estimation)

To deal with the first issue, the risc registers were made to act as a cache for the SAD and location of the best match for the current coding block. The actual data is stored in the datapath register file, and is swapped in when needed. Although this wastes a few cycles, there is no other reasonable place to store the data. Great improvements were made in the combating the second problem using a concept known as *loop unrolling*. Long known to computer scientists as a way to speed up code, it can reduce overhead.

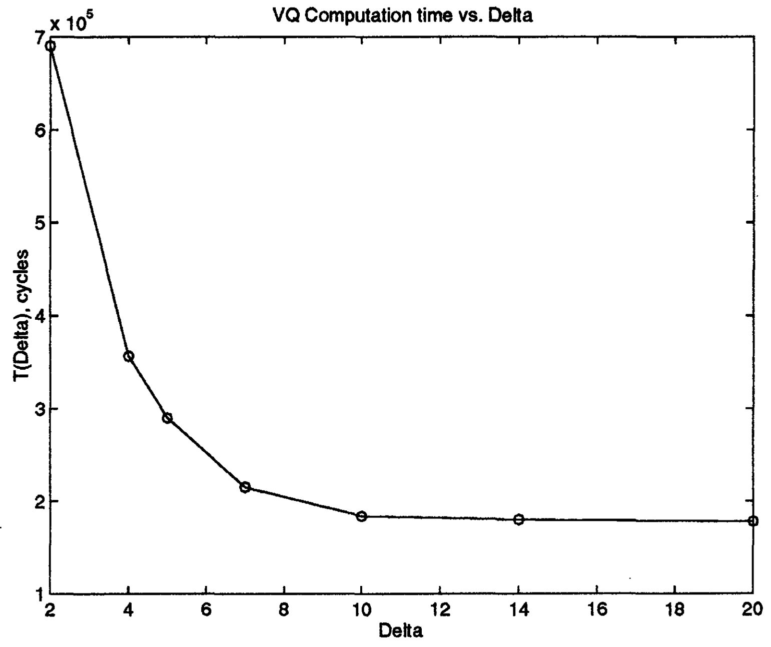


Figure 4.11: VQ Compute Time vs. Delta

For example, the following piece of code:

```
          for(i = 0; i < 8; i++)
              {
                   x[i] = a[i]-b[i];
              }
```

Can be unrolled into:

```
      x[0] = a[0]-b[0];
      x[1] = a[1]-b[1];
      x[2] = a[2]-b[2];
      x[3] = a[3]-b[3];
      x[4] = a[4]-b[4];
      x[5] = a[5]-b[5];
      x[6] = a[6]-b[6];
      x[7] = a[7]-b[7];
```

Although it requires more memory, the unrolled loop does not have to deal with the overhead of incrementing and checking i against 8. By using this method on the main loop of the quantization routine, overhead was reduced dramatically.

4.2.9 Vector Quantization Rate Computation

After the vector quantization has occurred, the rate required must be computed. This rate consists of two parts. The first is the codebook address, and the second is the coded block position. This position can be transmitted to the decoder as either a mask, by Huffman coding the zero-run length coded mask, or by direct addressing. Every time a VQ is added or discarded, the rate must be recomputed.

Since the VP is not well suited to Huffman coding, and the Huffman Encoder Unit on the VCP is not programmable, the rate computation must occur in the MIPS-X. Although the routine is fairly simple, it can be iterated many times during a single frame. Both a modification to the algorithm, and an improvement in the implementation of this routine can reduce computation time. Chapter 7 deals with this problem.

4.2.10 Scalar Quantization

Scalar quantization involves taking the DCT of a block, quantizing each coefficient using a quantization matrix, RLA coding the nonzero coefficients, and then inverse quantizing and

IDCT'ing the blocks. More than any other section of the algorithm, this section is exceptionally well suited to implementation in the VP.

The 2D DCT is performed using the separable DCT-II structure [23]. Since the multiply accumulators in the VP can operate independently, four vertical DCTs can be performed at once. After all 8 vertical DCTs have been computed (two iterations of the main loop), the block is transposed and 8 more vertical DCTs are computed. The quantization is achieved by multiplying each coefficient by a stepsize (which must be an integer over 65536). The quantized DCT can then be RLA coded by special purpose hardware in the VP. The block does not have to be transposed back to its normal orientation for the RLA coding to work. The VP RLA coder compensates for the change in orientation. The inverse quantization routine is achieved by multiplication of each term by an integral factor. The IDCT, which takes its data in transposed form, can then be performed. It operates almost identically to the DCT routine.

The procedure for scalar quantizing a block can be summarized as follows. The MIPS-X DMAs an 8x8 block from the current NCIF frame in DRAM to the DP memory in the VP. The DCT routine is then called, followed by the quantization routine. Then, the MIPS-X initiates an RLA DMA transfer of data from the DP memory to the RLA buffer in DRAM. The MIPS-X then calls the inverse quantization routine, followed by the IDCT routine. The quantized block is then DMA'd from the DP memory to the current reconstructed frame in DRAM.

4.2.11 Scalar Quantization Rate Computation

Coding the scalar quantized blocks can occur in several different ways. There are three methods for transmitting the coded coefficient positions: run length coding, a 64 bit mask, or direct addressing. There are two methods for encoding the values of the nonzero coefficients: Huffman codes or PCM codes. Given the RLA tokens produced by the VP, calculating each of these quantities is fairly simple.

As with the other rate calculation routines, the MIPS-X is the appropriate resource to use. The VP does not have a large enough memory or control structure to handle memory lookups and comparisons. The routine is fairly simple to implement on the MIPS-X, and requires no major modifications from the non-real time coder.

4.2.12 Codebook Adaptation

The codebook adaptation algorithm adds some of the scalar quantized blocks to the codebook, replacing other blocks which have not been useful, according to a specific figure of merit.

The adaptation algorithm is also better suited to implementation on the MIPS-X. The algorithm requires that an array of usage counts, and an array of improvement measures, be maintained for each codebook entry. A search must then be made to find the block with the smallest product of the two values. The searching and array indexing are not efficiently handled in the VP.

Implementation of the routine on the MIPS-X is straightforward, and requires no major modifications to the non-real time implementation.

4.2.13 Bitstream Formatting

The last step in the algorithm is to generate the actual bitstream. This portion is most efficiently handled by the MIPS-X, since it requires a great deal of array indexing and table lookups. As with codebook adaptation and scalar quantization rate computation, no major modifications to the non-real time implementation are required.

Chapter 5

Timing Analysis

The results of Chapter 4 indicate how to optimally port the algorithm to the VCP without modification. The next step is to perform a timing analysis to determine the computation time required for each of the tasks in the algorithm. The timing analysis will answer two questions. First, can the algorithm be reasonably ported to the VCP, and second, what aspects of the algorithm are most time consuming? It is the aggregate computation time which provides the answer to the first question, and the individual times which answer the second.

In cases where the aggregate time is too large, two possibilities exist. The first is to redesign the portions of the algorithm which are too computationally complex, and hope that in the worst case situation, the algorithm can still be made to work in real time. The second possibility is to re-design the algorithm so that in the *average* case it can work in real time. To handle extreme cases, a resource manager can be constructed to monitor the progress of the algorithm. The resource manager can instruct various tasks in the algorithm to reduce computation, during run time, if necessary. Such reductions in computation are often achieved by reducing search ranges, assuming motion vectors and vector quantization addresses based on previous frames, etc.

The first possibility is much more attractive, since it does not require the additional complexity of a resource manager, nor does it require the algorithm to be modified in unpredictable ways. As a result, the timing analysis which follows is done based on worst case situations, in the hopes that the algorithm can be recast so that even the worst case scenario can be computed in real time.

5.1 Assumptions

Some assumptions need to be made. First, the MIPS-X operates at 28 MHz. Given that a 10fps coder is being constructed, that leaves 100ms per frame, which is 2.8 million cycles of MIPS computation time. All figures will be measured in units of MIPS computation time. Additionally, all times will be cited as a percentage of total time available, which is equivalent to the number of cycles required, divided by 2.8×10^6, times 100.

It is also assumed that the VP operates at 56 MHz. Therefore, one instruction on the VP requires .5 MIPS cycles. The DMA requires 1.5 MIPS cycles per longword (32 bit) transfer.

The analysis will try to measure the main tasks involved in each computation. Typically, this will include number of multiplication's, number of additions, number of swaps, etc. Overhead, such as address computations, accessing variables, pointer de-references, etc., is assumed to be an additional 15% of the estimated time.

5.2 Analysis

5.2.1 Decimation

The decimation requires three steps: DMA the 32x28 block from the current SIF frame to the VP. Then, execute the decimation routine. Finally, DMA the 8x8 result from the VP to the current NCIF frame in DRAM. The total time is thus:

$$T_{DEC} = T_{DMA\ IN} + T_{COMP} + T_{DMA\ OUT}$$

A 32x28 block is composed of 8x28 longwords, which is 224 longwords. At 1.5 cycles per longword, $T_{DMA\ IN} = 336$ cycles. An 8x8 block is 2x8 longwords, which is 16. Multiplied by 1.5 cycles per longword, $T_{DMA\ OUT} = 24$ cycles.

The computation time is broken down into vertical filtering, transposing, filtering again, and transposing back:

$$T_{COMP} = T_{VFILT} + T_{XPOSE1} + T_{HFILT} + T_{XPOSE2}$$

The vertical filtering requires that every third row in the 32x28 block be filtered (8 rows). Each row consists of 32 pels. However, since the four MAC units operate in parallel, 4 pels can be done at a time, which implies that each row requires eight iterations. Each iteration requires 7 multiplication's, as the filter has seven taps. Multiplying it all out, T_{VFILT} = 8 rows x 8 iterations/row x 7 multiplies/iteration x .5 cycles/instruction= 224 cycles.

The first transpose requires a 32x8 matrix to be transposed into an 8x32 matrix. All transposes are done in 4x4 blocks, with each block of 4x4 requiring 6 VP instructions (3 MIPS-X) cycles to complete. The transpose will require 16 4x4 transposes. Therefore, T_{XPOSE1} = 16 transposes x 3 cycles/transpose = 48 cycles. The second transpose requires an 8x8 matrix to be transposed. This requires 4 4x4 transposes, which implies that T_{XPOSE2} = 4 transposes x 3 cycles/transpose = 12 cycles.

The horizontal filtering requires every third row in the 8x32 block to be computed (that's actually only 8 rows, since extra garbage on the side has been included so far). In each row, there are 8 pels to be filtered. Since there are four MAC units in parallel, this implies two iterations. Each iteration requires 7 multiplication's. Multiplying this out: T_{HFLIT} = 8 rows x 2 iterations/row x 7 multiplication's/iteration x .5 cycles/instruction = 56 cycles.

Adding them up, $T_{COMP} = T_{VFILT}$ (224) + T_{XPOSE1} (48) + T_{HFILT} (56) + T_{XPOSE2} (12) = 340 cycles. Plugging into the formula for T_{DEC}, $T_{DEC} = T_{DMA\ IN}$ (336) + T_{COMP} (340) + $T_{DMA\ OUT}$ (24) = 700 cycles. Since this is iterated for all 140 blocks in the frame, the total time is 140 x 700 cycles, which is $98\text{x}10^3$ cycles. Adding fifteen percent more for overhead, the total becomes $112.7\text{x}10^3$ cycles.

Decimation	$112.7\text{x}10^3$ cycles	4% Total Time

5.2.2 Motion Estimation

Motion estimation also consists of three parts. For each block, a 24x24 section of the current frame is DMA'ed to the DP memory in the VP. Then, 6 40x35 sections of the previous frame must be DMA'd to the DM, and 6 iterations of the motion search routine must occur. Finally, the resulting motion parameter must be transferred back to the MIPS-X. Thus,

$$T_{MVEST} = T_{DMA1} + 6\ [\ T_{DMA2} + T_{COMP}]$$

The 24x24 block is 6x24 longwords, which is 144. At 1.5 cycles/longword, $T_{DMA1} = 216$ cycles. A 40x35 block is 10x35, or 350 longwords. Also at 1.5 cycles/longword, $T_{DMA2} = 525$ cycles.

Most of the time computing is spent in the motion compensation unit. Overhead for comparisons will be ignored, and included in the 15% figure later on. A 24x24 comparison requires nine 8x8 comparisons. Since each 8x8 compare takes 4 cycles on the VP, this corresponds to 2 cycles on the MIPS-X. In each 40x35 reference frame, there are $(\Delta x)(\Delta y)$ 24x24 blocks to be compared, which is (16)(11) 24x24 blocks, or 176. Multiplying out, T_{COMP} = 176 blocks x 9 compares/block x 2 cycles/compare = 3168 cycles.

Plugging in for T_{MVEST}:

$$T_{MVEST} = T_{DMA1}\ (216) + 6\ [\ T_{DMA2}\ (525) + T_{COMP}\ (3168)]$$

$$T_{MVEST} = 22.37\text{x}10^3 \text{ cycles}$$

Where we have chosen to ignore the time spent in direct parameter transfer. Since this time, in the worst case, must be applied to each of the 140 blocks in a frame, the total time is 140 x T_{MVEST}, which is $3.13\text{x}10^6$ cycles. Adding 15% for overhead, the total becomes $3.60\text{x}10^6$ cycles.

Motion Estimation	$3.60\text{x}10^6$ cycles	129% Total Time

5.2.3 Motion Compensation

The time required for motion compensation is similar to the time for decimation. A 16x15 block is DMA'd from the previous reconstructed frame into the VP. Then, after the filter type is passed via direct parameter transfer, the appropriate compensation routine is called. The 8x8 result is then DMA'd from the VP to the predicted frame in DRAM. We will assume that the time for direct parameter transfer is negligible. In addition, since the computation is for a worst case scenario, we will assume that both vertical and horizontal filtering must be done for each block. With this in mind,

$$T_{MVCOMP} = T_{DMA\ IN} + T_{COMP} + T_{DMA\ OUT}$$

The block that is DMA'd in is 16x16 (8x8 plus 8 extra pels for the edges). This is 4x16, or 64 longwords. At 1.5 cycles/longword, $T_{DMA\ IN}$ = 96 cycles. An 8x8 block takes 24 cycles to DMA, so $T_{DMA\ OUT}$ = 24 cycles.

As with decimation, T_{COMP} involves vertical filtering, transposing, filtering again, and transposing once more:

$$T_{COMP} = T_{VFILT} + T_{XPOSE1} + T_{HFILT} + T_{XPOSE2}$$

For vertical filtering, 8 rows must be computed. In each row, there are 16 pels. However, four can be done in parallel, yielding 4 iterations. In each iteration, there are 8 filter taps, each requiring a single multiply. Multiplying out, T_{VFILT} = 8 rows x 4 iterations/row x 8 multiplication's/iteration x .5 cycles/instruction= 128 cycles. T_{HFILT} can be computed in a similar manner. An 8x16 block must have 8 rows computed, with 8 pels (2 iterations) in each row, each requiring 8 multiplication's. T_{HFILT} = 8 rows x 2 iterations/row x 8 multiplication's/iteration x .5 cycles/instruction = 64 cycles.

The first transpose requires a 16x8 block to be transposed to an 8x16 block. This requires 8 4x4 transposes, each of which requires 3 cycles. Thus, T_{XPOSE1} = 8 transposes x 3 cycles/transpose = 24 cycles. The second transpose is from an 8x8 block to an 8x8 block, which requires 4 4x4 transposes, which is 12 cycles, so T_{XPOSE2} = 12 cycles.

Plugging in for T_{COMP}:

$$T_{COMP} = T_{VFILT}\ (128) + T_{XPOSE1}\ (24) + T_{HFILT}\ (64) + T_{XPOSE2}\ (12)$$

$$T_{COMP} = 228 \text{ cycles}$$

Plugging into the formula for T_{MVCOMP}:

$$T_{MVCOMP} = T_{DMA\ IN}\ (96) + T_{COMP}\ (228) + T_{DMA\ OUT}\ (24)$$

$$T_{MVCOMP} = 348 \text{ cycles}$$

This must be repeated for all 140 blocks, yielding a time of 48.72×10^3 cycles. Adding 15% for overhead, the total becomes 56.02×10^3 cycles.

Motion Compensation	56.02×10^3 cycles	2.0% Total Time

5.2.4 Error Computation

Error computation must actually be done twice. Once to compute the error after motion compensation, and once again to compute the error with a zero motion vector. For each one, an 8x8 block from the current frame must be DMA'd to the VP, and an 8x8 block to compare it with must also be DMA'd to the VP. Then, the error is computed, and the difference block can be DMA'd out. The SSE is read out via direct parameter transfer. Ignoring the time for the parameter transfer:

$$T_{ERR} = T_{DMA\ IN} + T_{DMA\ IN} + T_{COMP} + T_{DMA\ OUT}$$

All the DMA's are for 8x8 blocks, so $T_{DMA\ IN} = 24$ cycles, and $T_{DMA\ OUT} = 24$ cycles.

The compute time is composed of three factors. The first is to perform the subtraction, the next is to perform all the multiplication's and additions, and the last is to add the accumulators of the four MAC units. Thus:

$$T_{COMP} = T_{SUB} + T_{MUL} + T_{ADD}$$

As Chapter 4 indicated, the datapath ALU can be used to compute the difference. Since 8 subtractions can occur in one VP cycle, and 64 subtractions are required, $T_{SUB} = 8$ instructions x .5 cycles/instruction = 4 cycles. Each of the 64 pels in the block must then be multiplied by themselves and added up. This requires 64 multiply-accumulates. However, 4 can be done in parallel, so 16 iterations are required. Thus, $T_{MUL} = 16$ iterations x .5 cycles/instruction = 8 cycles. To sum the accumulators in all four multiply-accumulators requires 6 instructions, so T_{ADD} = 6 instructions x .5 cycles/instruction = 3 cycles. Plugging in for T_{COMP}:

$$T_{COMP} = T_{SUB}\ (4) + T_{MUL}\ (8) + T_{ADD}\ (3)$$

$$T_{COMP} = 15 \text{ cycles}$$

Plugging this into T_{ERR}:

$$T_{ERR} = T_{DMA\ IN}\ (24) + T_{DMA\ IN}\ (24) + T_{COMP}\ (15) + T_{DMA\ OUT}\ (24)$$

$$T_{ERR} = 87 \text{ cycles}$$

This must be repeated twice for all 140 blocks, bringing the total to $24.36\text{x}10^3$. Adding 15% for overhead, the result is $28.01\text{x}10^3$ cycles.

Error Computation	$28.01\text{x}10^3$ cycles	1.0% Total Time

5.2.5 Sorting

Sorting is completely done in the MIPS-X, using the standard HeapSort routine from *Numerical Recipes in C* [22]. Sorting is required in several places:

1. Sorting the Prediction Gains for discarding motion vectors
2. Sorting the rates of occurrence of the motion vectors for the initial Huffman rate computation
3. Sorting the rates of occurrence for the motion vectors for the final Huffman code generation
4. Sorting the blocks in order of prediction error to determine which blocks to scalar quantize
5. Sorting the blocks in order of Delta MSE to determine which VQ to discard

In the absolute worst case, it may be possible that each of these sorts has 140 elements. The HeapSort is an O(NlogN) operation, but it is difficult to estimate the amount of time required for an actual sort. Measuring the time, however, is fairly simple. Using the worst case ordering of data, the time measured for a heapsort was $36.18\text{x}10^3$ cycles. Multiplying by 5 sorts, the total sort time is $180.9\text{x}10^3$ cycles.

Sorting	180.9×10^3 cycles	6.5% Total Time

5.2.6 Motion Vector Rate Computation

It is very difficult to approximate the time required for this operation. The main computation is in the generation of the Huffman table based upon the motion statistics in the current frame. Determining the rate based on the local tables and PCM codes involves only table lookups and fixed point multiplies.

The time required to generate the Huffman code is proportional to the number of unique motion vectors, N. N can theoretically be as high as 140 (i.e., every motion vector is different). This is, however, almost impossible. A more reasonable worst case assessment is around 70 distinct motion vectors.

A very simple computational model can be developed to estimate the amount of time required to compute the Huffman code for N distinct symbols. Huffman code generation works in *stages*. At the first stage, there are N rates of occurrence. The two smallest rates are combined, and then the new rate is inserted into the list of rates. The amount of time for this insertion is proportional to the number of elements in the list, n. In the first stage n = N, and in the i^{th} stage, n = N-i. Let α represent the amount of time required to insert into a list with one element, so that insertion into a list of n elements requires $n\alpha$ cycles. After the tree is generated, it must be traversed in order to generate the actual code. This procedure requires an amount of time proportional to the total number of bits in the code. In the worst case, the total number of bits is $N\log_2 N$. Let β be this constant of proportionality. This is the case for input data whose rates of occurrence are all the same. It is this case which also presents the worst case for inserting the new rate. Therefore, the total time required, as a function of N, can be modeled as:

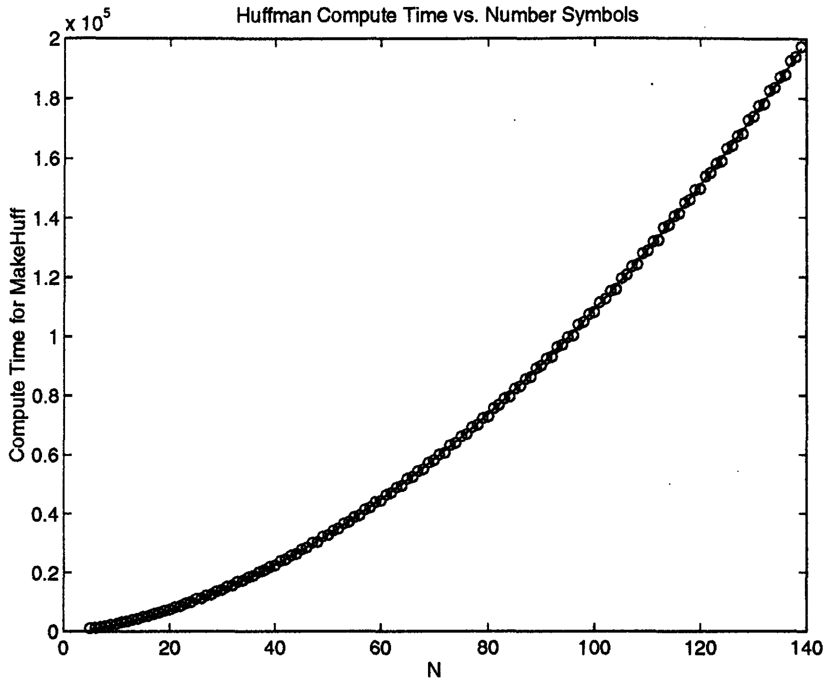


Figure 5.1: Huffman Code Generation time vs. N

$$T(N) = \left[\sum_{n=N}^{1} \alpha n\right] + \beta N \log_2 N$$

$$T(N) = \frac{\alpha}{2}\left(N^2 + N\right) + \beta N \log_2 N$$

T(N) was then measured using the Huffman Code Generation routine described in Chapter 7. Figure 5.1 is a plot of the actual time required vs. N. The least squares fit of the curve described above is also plotted as the solid line. For the best fit, $\alpha = 15.45$ and β=46.73. The amount of time required when N is 70 is roughly $65\text{x}10^3$ cycles. However, every time a motion vector is dropped, the code must be recomputed. In cases where there is a lot of uncorrelated motion (i.e., someone gets up, rotates the camera, etc.), as many as 100 motion vectors can be dropped. If the average time for the computation is $17.3\text{x}10^3$ cycles (that's the time for computing when there are 30 distinct vectors), the total time can be 100 times that, which is $1.73\text{x}10^6$ cycles.

Motion Vector Rate Comp.	1.73×10^6 cycles	61.7% Total Time

5.2.7 Vector Quantization

Chapter 4 developed the models necessary to determine the compute time required for the VQ routine. As figure 4.9 indicated, the time required for one of the two VQ (intra and inter-frame) was 180×10^3 cycles. Multiplying by two, the total time becomes 360×10^3.

Vector Quantization	360.0×10^3 cycles	12.8% Total Time

5.2.8 Vector Quantization Rate Computation

Most of the computation of the VQ rate is spent on determining the number of bits required to transmit the coded block positions of the MV/VQ instances. Every time an instance is added or discarded, the positions must be recoded. The most commonly used coding method, Huffman run-length codes, requires some amount of time to compute.

Since this routine is also implemented completely in the MIPS-X, the computation time is hard to estimate, as it is composed of table lookups and overhead. Instead, it is simpler and more efficient to simply measure the amount of time required, using a coded version of the routine. The amount of time is proportional to the number of instances. Figure 5.2 is a plot of the time required vs. N, the number of MV/VQ instances.

In and of itself, the routine does not take long to execute. However, it can be iterated many times. When every motion vector is zero, an MV/VQ instance is the same as a VQ instance. As VQ's are being added after motion estimation, the VQ rate must be computed after every block is VQ'd. Since there can be as many as 140 VQ's, the rate can be computed up to 140 times. As VQ's are discarded during the dynamic bit allocation, the rate must also be recomputed after every discard. Since all 140 VQ's can be discarded,

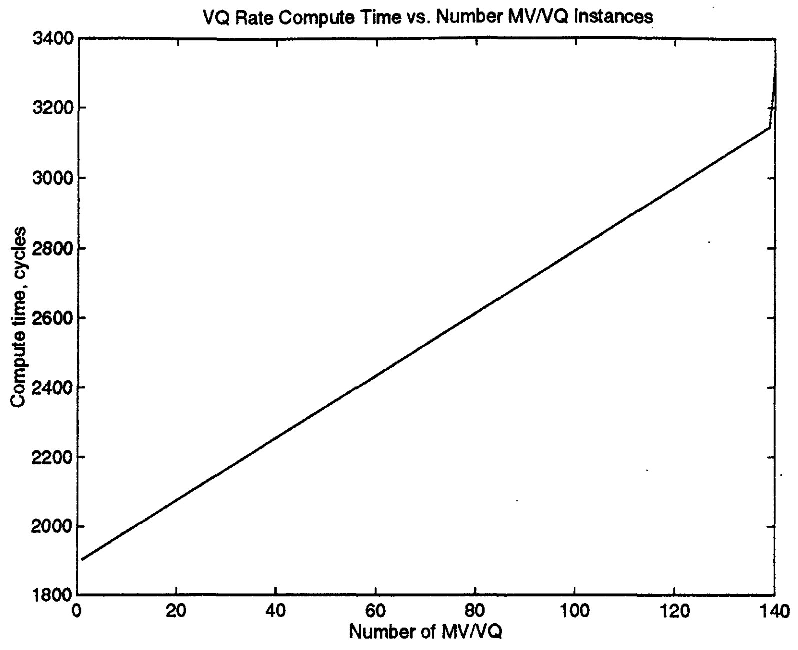


Figure 5.2: VQ Rate Compute Time

this adds another 140 iterations. Therefore, the VQ rate function can be iterated as many as 280 times in a single frame.

The total amount of time required for this computation will be:

$$2\sum_{i=0}^{140} T(i)$$

Where T(i) is the amount of time required for the routine to execute when the number of MV/VQ instances is i. Plugging in the data from Figure 5.2 into the equation, the total amount of time is 708.30×10^3 cycles.

VQ Rate Computation	708.3×10^3 cycles	25.2% Total Time

5.2.9 Scalar Quantization

Scalar quantization involves several steps. First, the 8x8 block must be DMA'd to the VP. Then, the forward DCT routine must be called, followed by the quantization routine, the inverse quantization routine, and the inverse DCT. Then, the RLA tokens must be DMA'd out, and then the quantized block must be DMA'd out. Mathematically:

$$T_{SQ} = T_{DMA\ IN} + T_{FDCT} + T_Q + T_Q^{-1} + T_{IDCT} + T_{RLA\ DMA} + T_{DMA\ OUT}$$

The DMA in and DMA out times are both 24 cycles. The RLA time is different, however. The RLA data consists of one longword per RLA token. Since there can be 65 RLA tokens, the time required is the DMA time on a 65 longword block, which is 97.5 cycles. Thus, $T_{DMA\ IN} = 24$, $T_{DMA\ OUT} = 24$, and $T_{RLA\ DMA} = 97.5$.

The forward DCT consists of a 1D vertical DCT, a transposition, and another 1D DCT. There is no second transposition since the IDCT routine expects its input to be transposed.

$$T_{FDCT} = T_{1D\text{-}DCT} + T_{XPOSE} + T_{1D\text{-}DCT}$$

The transpose of an 8x8 block requires 4 4x4 transposes, each of which requires 3 cycles. Therefore, $T_{XPOSE} = 4$ transposes x 3 cycles/transpose = 12 cycles. The 1D DCT must be done on all 8 columns of the block. However, since there are four multipliers in parallel, this requires only 2 iterations. Code for the DCT is available in the microcode ROM in the VP; it takes 17 cycles to run. Therefore, $T_{1D\text{-}DCT} = 2$ iterations/block x 17 cycles/iteration = 34 cycles. Plugging in:

$$T_{FDCT} = T_{1D\text{-}DCT}\ (34) + T_{XPOSE}\ (12) + T_{1D\text{-}DCT}\ (34)$$

$$T_{FDCT} = 80 \text{ cycles}$$

The IDCT requires exactly the same amount of time, so $T_{IDCT} = 80$ cycles.

Both the forward and inverse quantization routines are accomplished with multiplication's. Since there are 64 coefficients, 64 multiplication's are required. However, as there are four multipliers in parallel, only 16 iterations are necessary. Each iteration requires a single cycle, so $T_Q = T_Q^{-1} = 16$ iterations x .5 cycles/instruction = 8 cycles.

Plugging in:

$$T_{SQ} =$$

$$T_{DMA\ IN}\ (24) + T_{FDCT}\ (80) + T_Q\ (8) + T_Q^{-1}\ (8) + T_{IDCT}\ (80) + T_{RLA\ DMA}\ (98) + T_{DMA\ OUT}\ (24)$$

$$T_{SQ} = 314 \text{ cycles}$$

In the worst case, every block must be inter and intra-frame quantized, requiring 280 iterations of the DCT routine. Multiplying out, the total comes to 87.92×10^3 cycles. Adding 15% for overhead results in 101.11×10^3 cycles.

Scalar Quantization	101.11×10^3 cycles	3.6% Total Time

5.2.10 Scalar Quantization Rate Computation

Like motion vector rate computation and vector quantization rate computation, scalar quantization rate computation is difficult to estimate. For each set of RLA tokens returned from the VP, the coding rate using the three different position encoding methods and the two different amplitude coding methods must be computed. This involves table lookups, additions, and counting. The time required is proportional to the number of nonzero coefficients (i.e., the number of RLA tokens).

The actual amount of time required, as a function of the number of RLA tokens, is plotted in Figure 5.3.

The slope of the line is 120, meaning that every coefficient requires roughly 120 cycles to compute. The maximum number of times that the DCT Rate routine can be called is a function of the bitrate. The more bits available, the more DCT blocks that can be coded. At the very least, each coefficient of a DCT block requires 1 bit to code. Since it requires 120 cycles to compute the rate for a single coefficient, at 1600 bits/frame, the computation will require 120 cycles/coefficient x 1600 coefficients/frame = 192.0×10^3 cycles in an absolute worst case scenario.

DCT Rate Computation	192.0×10^3 cycles	6.8% Total Time

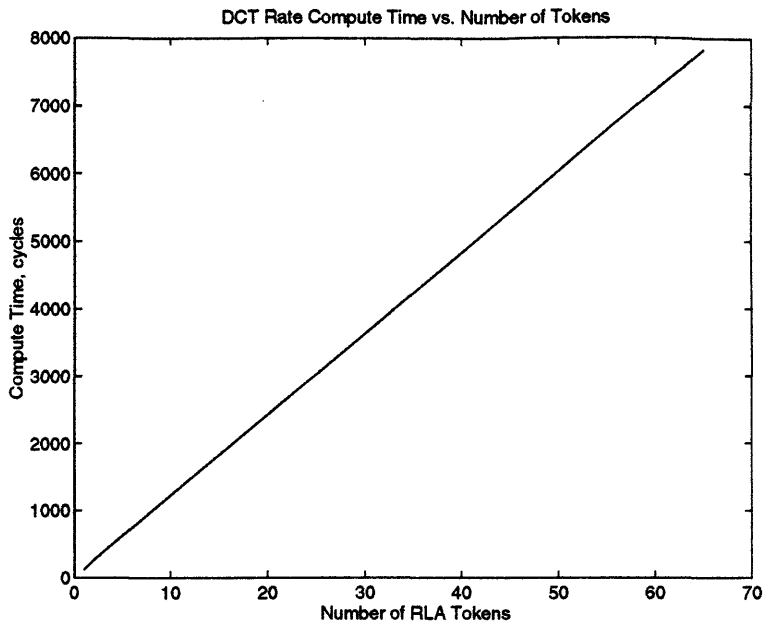


Figure 5.3: DCT Rate Compute Time

5.2.11 Codebook Adaptation

The codebook adaptation algorithm is quite simple and requires very little computation. For every scalar quantized block in the bitstream, if the block affords more than a certain amount of improvement, it gets added to the codebooks. The appropriate location must be found, and the block is then DMA'd into the codebook. The thresholding implies that only blocks with significant improvement will be added. At most, this can be around 30 blocks in a frame. For each block, a linear search is done to find the block it is to replace. Since there are 256 entries in each codebook, this search can take as long as 2500 cycles (roughly 10 cycles for each comparison in the search). The replacement of the entry in the codebook requires a DMA of an 8x8 block, which is 24 cycles for each of the 30 blocks. Multiplying out, and adding 15% for overhead, the total time comes to 87.07×10^3 cycles.

Codebook Adaptation	87.07×10^3 cycles	3.1% Total Time

5.2.12 Bitstream Formatting

Bitstream formatting is also a very simple process. Every field in the bitstream requires a table lookup to identify its code, followed by a shift and addition to insert it into the bitstream. In a worst case situation, each field is 1 bit wide. Assuming a table lookup requires 10 cycles, and that a shift and add requires 5 cycles, the total time needed is 24×10^3 cycles for a 1600 bit/frame coder. Adding 15% for overhead, the total comes to 27.6×10^3 cycles.

Bitstream Formatting	27.6×10^3 cycles	0.9% Total Time

5.3 Results

Figure 5.4 summarizes the above results, and tallies them up.

Decimation	112.70×10^3 cycles	4.0% Total Time
Motion Estimation	3.60×10^6 cycles	129.0% Total Time
Motion Compensation	56.02×10^3 cycles	2.0% Total Time
Error Computation	28.01×10^3 cycles	1.0% Total Time
Sorting	180.80×10^3 cycles	6.5% Total Time
MV Rate Computation	1.73×10^6 cycles	61.7% Total Time
Vector Quantization	360.00×10^3 cycles	12.8% Total Time
VQ Rate Calculation	708.30×10^3 cycles	25.2% Total Time
Scalar Quantization	101.11×10^3 cycles	3.6% Total Time
SQ Rate Computation	192.00×10^3 cycles	6.8% Total Time
Codebook Adaptation	87.07×10^3 cycles	3.1% Total Time
Bitstream Formatting	27.60×10^3 cycles	0.9% Total Time
TOTAL:	**7.18×10^6 cycles**	**256% Total Time**

Figure 5.4: Summary of Timing Estimates

Figure 5.4 clearly indicates that there are three major computational problems: Motion Estimation, Motion Vector Rate Computation, and Vector Quantization Rate Computation. If each of these tasks can be reduced in computation time, than the coder can work in real time without a resource manager. Chapter 6 deals with Motion Vector Rate Computation, Chapter 7 with Vector Quantization Rate Computation, and Chapter 8 with Motion Estimation.

Chapter 6

Motion Vector Rate Computation

The results of the timing analysis in Chapter 5 clearly indicate that the motion vector rate computation is one of the most time consuming operations. Most of the cycles in computing the rate are spent regenerating the local Huffman table. One obvious way to reduce computation is to eliminate the local Huffman table entirely. If its coding gains are small, it may not be worthwhile to include it in the real-time system. The results of section 6.1 indicate, however, that the local Huffman table is quite useful. Section 6.2 describes an algorithm which can quickly compute the change in coding rate from discarding a motion vector. Section 6.3 describes an efficient algorithm for determining the total coding rate for all the motion vectors (which is needed for initialization). Section 6.4 describes an algorithm for actually generating the Huffman code that does not require dynamic memory management.

6.1 The Local Huffman Table

To determine the utility of the local table, models of motion statistics will be used to generate a rough estimate of when the local Huffman code is more efficient than the global Huffman code. One would expect that the local Huffman code becomes more and more useful as the statistics of the motion vectors deviate from those used to generate the global Huffman table. The results of section 6.1.1 verify this fact. Section 6.1.2 examines actual data , and 6.1.3 will draw the conclusion that the local Huffman table should be kept.

6.1.1 Models of Motion Statistics

To develop a model for the local Huffman table usage, many simplifications will be made. First, we will assume that motion vectors are one dimensional (and not two). The analysis that follows can easily be extended to two dimensions. Second, we assume that all statistics are discrete Gaussian in nature. A discrete version of a Gaussian can be obtained by:

$$G[n] = \int_{n-\frac{1}{2}}^{n+\frac{1}{2}} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

$$G[n] = \frac{1}{2}\operatorname{erf}\left(\frac{n+\frac{1}{2}-\mu}{\sqrt{2\sigma^2}}\right) - \frac{1}{2}\operatorname{erf}\left(\frac{n-\frac{1}{2}-\mu}{\sqrt{2\sigma^2}}\right)$$

Where erf(x) can be defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2} dt$$

This definition of a discrete Gaussian has the important property that:

$$\sum_{n=-\infty}^{\infty} G[n] = 1$$

If we assume that the statistics used to generate the global table were Gaussian in nature, it is not unreasonable to expect that the bits required for each motion vector will be an inverted Gaussian of the form A-B(G[n]) for some A and B. The actual Huffman code can be approximated using the parameters $A = 14$, $B = 91.32$, $\mu = 0$, and $\sigma^2 = 9$. Figure 6.1 is a plot of this function, which we shall call L[n].

We can also model the statistics of the motion vectors in the current frame as a discrete Gaussian. However, these statistics have a different mean and variance than those used to generate the global Huffman table. Let p[n] be the probabilities of occurrence of the motion vectors in the frame, and let N be the total number of motion vectors in the frame. Thus, Np[n] can be considered the rate of occurrence for each motion vector.

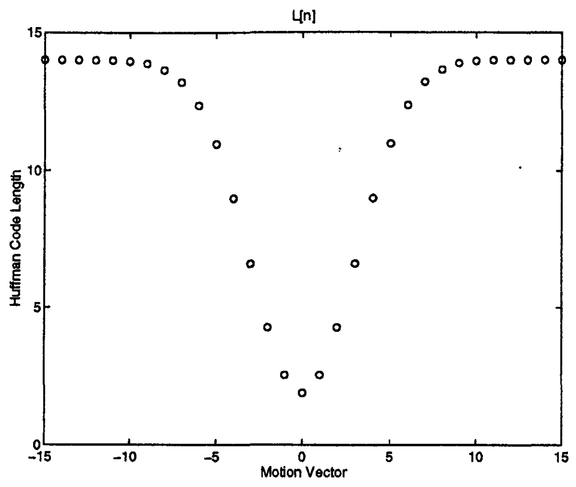


Figure 6.1: L[n], the global Huffman code lengths

Using these models, we can now develop an equation for the bitrate required using the global Huffman table. This rate will be the length of the symbol for each motion vector, times its rate of occurrence. This rate, G, is then:

$$G = N\sum_{n=-\infty}^{\infty} L[n]p[n]$$

The rate required using the local Huffman table is more complicated. It consists of the Huffman codes themselves, and the motion list. The list consists of each unique motion vector, represented by its code in the global Huffman table, and its rate of its occurrence in the frame. We can estimate the number of bits for the code itself using the entropy. If H is the entropy of p[n], then we can estimate the rate required as HN. The entropy, H, of the discrete Gaussian source is difficult to compute in closed form. However, it can easily be estimated as follows [27]. Assume that $f_x(x_0)$ is a continuous Gaussian, and that we define a discretized version of it as $p_2[n] = f_x(n\Delta x)\Delta x$. For Δx equal to one, this approximates p[n] as originally defined. We can then compute the entropy as:

$$H = \sum_{n=-\infty}^{\infty} f_x(n\Delta x)\Delta x \log_2\left(\frac{1}{f(n\Delta x)\Delta x}\right)$$

$$H = \sum_{n=-\infty}^{\infty} f_x(n\Delta x)\Delta x \log_2\left(\frac{1}{f(n\Delta x)}\right) - \log_2 \Delta x \sum_{n=-\infty}^{\infty} f_x(n\Delta x)\Delta x$$

$$H = \sum_{n=-\infty}^{\infty} f_x(n\Delta x)\Delta x \log_2\left(\frac{1}{f(n\Delta x)}\right) - \log_2 \Delta x$$

Since we are assuming that $f_x(n\Delta x)$ is roughly p[n], the sum on the right converged to 1. However, the Riemann sum on the left will not converge to its integral with Δx equal to one. However, we will assume that the sum is reasonably approximated by its integral, and therefore we can further estimate H as:

$$H = \int_{-\infty}^{\infty} f_x(x_0)\log_2\left(\frac{1}{f_x(x_0)}\right) = h$$

Where h is the *differential entropy* of a continuous source. It is well known that the differential entropy of a continuous Gaussian source is:

$$h = \frac{1}{2}\log_2(2\pi e\sigma^2)$$

We are therefore justified in using this as a first order estimate of the entropy H, of p[n].

Computing the rate required for sending the list is also tricky. For each unique motion vector, we must send the symbol for it from the global table along with 5 bits for its rate of occurrence. In the model, the probabilities of occurrence of the motion vectors are nonzero from $-\infty$ to ∞. As a result, the model provides no concept of a unique set of motion vectors. A reasonable definition, however, is to assume that each motion vector from μ-σ to μ+σ occurs at least once, and none of the motion vectors outside of that range occurs. We can then write the bits required as

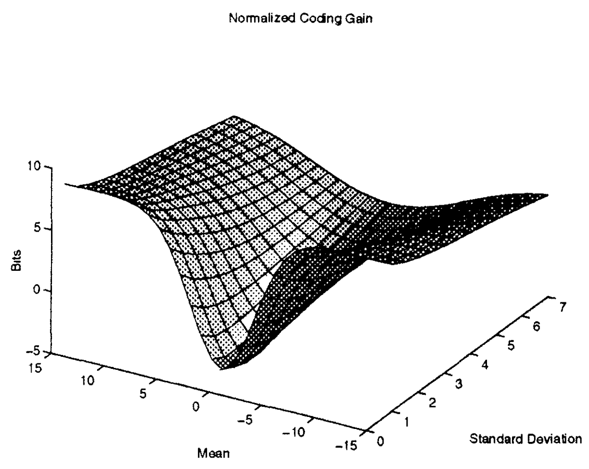
$$\sum_{n=-\infty}^{\infty} w[n]L[n] + 5N$$

Where w[n] is a rectangular window function from μ-σ to μ+σ, with a height of one. We can approximate w[n] with a Gaussian of mean μ and variance σ^2. To make sure that the windows have the same total area, we must multiply the Gaussian by 2σ. Since the Gaussian has a mean of μ and a variance of σ^2, it is equal to p[n]. The bits for the list can thus be estimated as:

$$2\sigma\sum_{n=-\infty}^{\infty} p[n]L[n] + 5N$$

The total bits, L, required for the local table is thus:

$$L = 2\sigma\sum_{n=-\infty}^{\infty} p[n]L[n] + 5N + \frac{N}{2}\log_2(2\pi e\sigma^2)$$



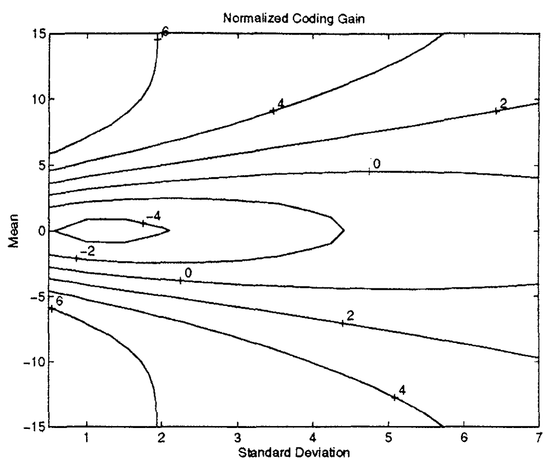


Figure 6.2 and 6.3: 3D and contour plot of the Normalized Coding Gain

We can then define the *normalized coding gain* as (G-L)/N:

$$\mathrm{NCG} = \left(1 - \frac{2\sigma}{N}\right)\sum_{n=-\infty}^{\infty} L[n]p[n] - 5 - \frac{1}{2}\log_2(2\pi e\sigma^2)$$

This quantity represents an estimate of the average number of bits that the local table will save, as compared to the global table. Clearly, the local table will *never* be used if $N < 2\sigma$. For purposes of analysis, we will assume that $N >> 2\sigma$. This implies that the list requires less space than the codewords. The results which follow can easily be extended to include cases where this limit is not true.

If we plot the NCG as a function of σ and μ, we can determine when to expect the local table to be used. Figure 6.2 is a 3D plot of the function, and Figure 6.3 is a contour plot. The local table will be more useful when the NCG is positive. The plots indicate that the NCG increases when either the mean gets larger in magnitude, or the variance gets smaller. These results are intuitively satisfying. As the mean gets farther from 0 (the mean of the statistics used for the global Huffman table), the local statistics become less matched to the global, so the local table should be more useful. As the variance gets smaller, any differences in mean become amplified, improving the NCG of the local table.

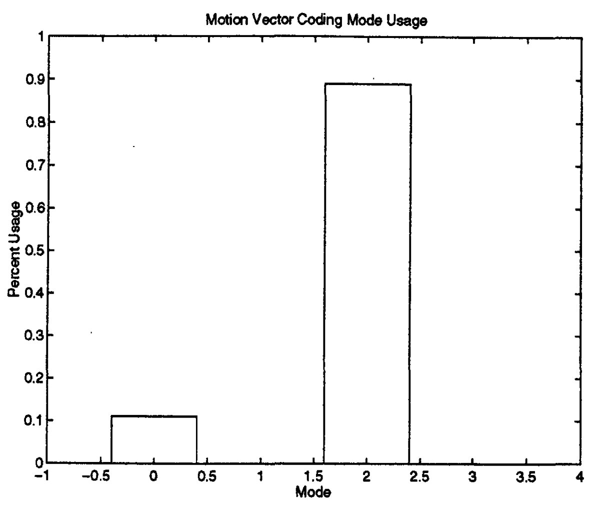
6.1.2 Data

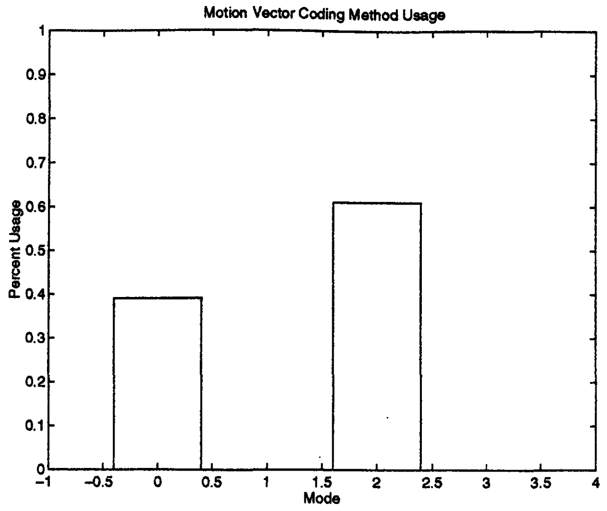
The above model is impractical for determining the actual utility of the local table for real sequences, as the statistics aren't always Gaussian. Instead, two figures of merit are determined. The first is the probability of usage, P_i, which reflects how often the local table is more efficient than either the PCM code or the global Huffman table. The other figure of merit is the gain, G_i, which represents the difference in bits between the rate required by the local table and the next best method.

Clearly, both G_i and P_i will depend heavily on the sequence. More specifically, they will depend on how closely the motion statistics of the sequence match those used to generate the global Huffman table. The results of the previous section indicate that the local table is most often used when the variance in the motion statistics is small, and when the mean is large. This will be the case when a large object moves uniformly. Such motions are common of "busy" sequences, where the individual in the sequence tends to move a lot. A typical motion, for example, would be a back and forth swaying in a chair.

P_i and G_i were both measured for two sequences. A "quiet sequence" which had little motion, and a "busy" sequence, where the person in the sequence swayed back and forth gently in a chair while talking. Figure 6.4 is a plot of the percentage of times the various motion vector coding methods were used in the "quiet" sequence. A "0" represents the local Huffman table, with the global Huffman table being used to represent the motion list. A "1" represents the local Huffman table with the PCM Code being used to represent the motion list. A "2" represents the global Huffman code, and a "3" represents the PCM code. Figure 6.5 represents the same plot for the "busy" sequence. Notice than in both cases, only the global Huffman code and the local Huffman code with the global Huffman list were ever used. The data was collected over 300 frames of a 10fps coder. For the quiet sequence, P_i averaged .11 over all 300 frames, and the busy sequence P_i averaged .39.

Figure 6.6 is a plot of G_i for the quiet sequence. G_i was set to zero for those frames for which the local Huffman table was not used. Figure 6.7 is a plot of the same data for the busy sequence.



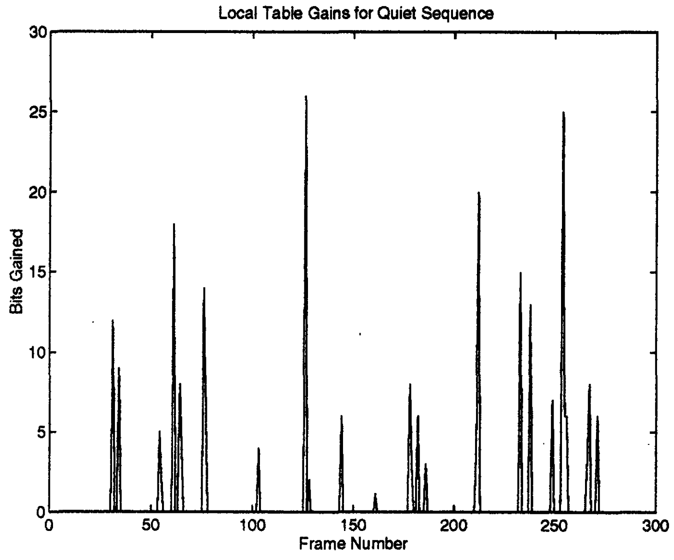


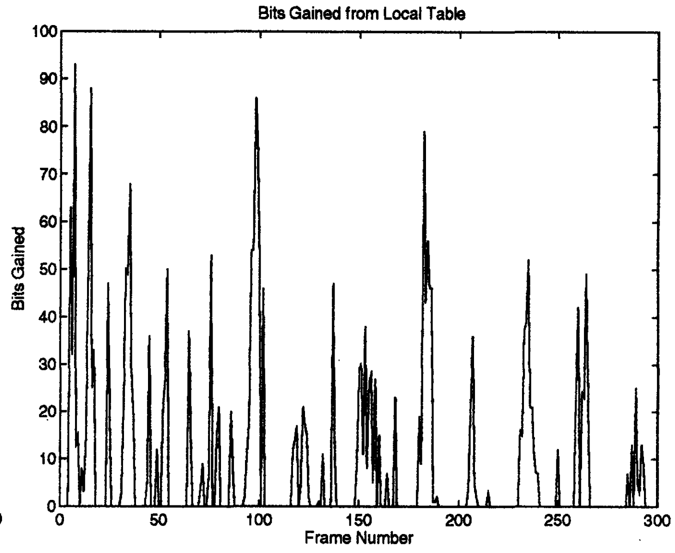
Figures 6.4 and 6.5: Usage of the motion vector coding modes for the quiet sequence (left) and busy sequence (right).

For the quiet sequence, G_i was 7.57 bits when averaged over the entire sequence. For the busy sequence, G_i was 23.73 bits.

6.1.3 Results

Clearly, the advantages of the local Huffman table depend heavily on the sequence.





Figures 6.6 and 6.7: Coding gain from local Huffman table for quiet sequence (left) and busy sequence (right)

However, the results of the previous subsections indicate that for many typical sequences, the local table can gain as many as 90 bits in a frame, with an average of roughly 25. In very busy sequences, such as those where the coding gain may be very high, the extra bits are almost always needed to send scalar quantized blocks. Therefore, it would seem that the local table is quite useful and should remain part of the real-time coder.

6.2 Modifying the Rate Computation

Since the local Huffman table can achieve good coding gains in many instances, it should be kept, and thus the method used for computing the rate of the Huffman code must be altered.

6.2.1 Analysis

Clearly, recomputing the Huffman table every time a motion vector is discarded is very wasteful, for two reasons. First, the statistics don't change that much from iteration to iteration, and secondly, the code itself is not needed, only the total rate needed using the code. In fact, every iteration, what is needed is the *change* in rate obtained by discarding a motion vector.

Modeling the rate required for a Huffman code is actually quite simple. The entropy provides an excellent measure of determining the number of bits required by a Huffman code. Assume that there are N motion vectors, M of which are distinct, and that the rate of occurrence for each one is R_i. By definition:

$$\sum_{i=1}^{M} R_i = N$$

To determine the entropy of the alphabet of M symbols, we can define the probability of symbol i as $\frac{R_i}{N}$. The entropy, H, is then:

$$H = \sum_{i=1}^{M} -\left(\frac{R_i}{N}\right)\log_2\left(\frac{R_i}{N}\right)$$

Using the R_i to generate a Huffman code will yield M symbols (one symbol for each distinct motion vector). The length of a given symbol can be defined as L_i. If that Huffman code is used to represent every motion vector, the total number of bits required (B) will be:

$$B = \sum_{i=1}^{M} L_i R_i$$

We can also define $\bar{L}$, the average length of a symbol in the Huffman code, as:

$$\bar{L} = \sum_{i=1}^{M} \left(\frac{R_i}{N}\right) L_i$$

We can immediately derive the important relationship:

$$B = \bar{L}N$$

It is well known that there is a strong relationship between $\bar{L}$ and H, which is:

$$H \le \bar{L} \le H + 1$$

We can then arrive at the desired result by multiplying all sides of the above inequality by N, and substituting in B:

$$HN \le B \le (H+1)N$$

We have now derived a relationship between the total number of bits needed to encode an alphabet using a Huffman code (B), and the entropy (H). When N is large, this range can actually be quite substantial. However, we will assume that a Huffman coder generally performs very close to the lower bound, and thus:

$$B \approx HN$$

If we let H_1 represent the entropy of the motion vectors before discarding one, and H_2 represent the entropy after discarding one, we can express the difference in bits as:

$$\Delta B = H_1 N - H_2 (N-1)$$

The entropies only differ in a single term in the sum, since the only change has been in the rate of occurrence for one motion vector, which has decreased by one. Let R_i' represent the rates of occurrence after discarding a motion vector. We then have the $R_i = R_i'$ for all i except one, the Mth, for example, for which $R_M' = R_M - 1$.

Plugging in the formulas for the entropies into the equation for ΔB:

$$\Delta B = N\sum_{i=1}^{M} -\left(\frac{R_i}{N}\right)\log_2\left(\frac{R_i}{N}\right) - (N-1)\sum_{i=1}^{M}\left(\frac{R_i'}{N-1}\right)\log_2\left(\frac{R_i'}{N-1}\right)$$

$$\Delta B = \sum_{i=1}^{M} -R_i \log_2\left(\frac{R_i}{N}\right) - \sum_{i=1}^{M} -R_i' \log_2\left(\frac{R_i'}{N-1}\right)$$

$$\Delta B = -\sum_{i=1}^{M} R_i \log_2 R_i + \sum_{i=1}^{M} R_i \log_2 N + \sum_{i=1}^{M} R_i' \log_2 R_i' - \sum_{i=1}^{M} R_i' \log_2 (N-1)$$

$$\Delta B = -\sum_{i=1}^{M} R_i \log_2 R_i + \log_2 N \sum_{i=1}^{M} R_i + \sum_{i=1}^{M} R_i' \log_2 R' - \log_2 (N-1) \sum_{i=1}^{M} R_i'$$

Given that $\sum_{i=1}^{M} R_i = N$ and that $\sum_{i=1}^{M} R_i' = N-1$, the equation can be further simplified:

$$\Delta B = -\sum_{i=1}^{M} R_i \log_2 R_i + N \log_2 N + \sum_{i=1}^{M} R_i' \log_2 R_i' - (N-1)\log_2 (N-1)$$

Most of the terms in the two sums cancel, except for the M^{th}. The result, which we will call the *entropy estimation equation*, is:

$$\boxed{\Delta B = -R_M \log_2 R_M + N \log_2 N + [R_M - 1]\log_2 [R_M - 1] - [N-1]\log_2 [N-1]}$$

ΔB can thus be expressed as the sum of four terms. Each term is of the form AlogA. Unfortunately, there is still a problem. Since the MIPS-X is a fixed point processor, it cannot compute logs, and cannot multiply efficiently either. However, great improvements can be made by realizing that R_M and N are both integral, and range from 0 to 140. Given that this is the case, a lookup table of AlogA can be stored in an array with 140 elements. Then, ΔB can easily be computed by using four table lookups and four additions, an extremely rapid operation.

ΔB only gives the change in the number of bits required to transmit the motion vectors using the local Huffman code. Every time a motion vector is discarded, we must also compute the change in rate for the other two methods. Even for the local Huffman table, we must also compute the change in rate from altering the motion vector list. However, all of these are simple computations:

1. *Local Huffman Code, Global Huffman List:* The change in rate using the code is ΔB, from above. The list may or may not change in rate. If the rate of

occurrence of a vector was one before it was discarded, then discarding it will also remove it from the list. This will gain 5 bits for the count, plus the length of the symbol for that vector in the global Huffman table.

2. *Local Huffman Code, PCM Coded List*: The change in rate is exactly the same as for the Local Huffman Code with the Global Huffman List. The only difference is that a fixed 10 bits are gained from discarding the symbol for the vector in the list (as opposed to a variable amount).
3. *Global Huffman Code*: When the global Huffman table is being used to code the motion vectors, discarding a vector will increase the number of bits available by the length of the symbol for that motion vector.
4. *PCM Code*: When a motion vector is discarded, the number of bits gained is always the same, 10.

All the encoder needs to do, then, is to keep track of the total rate required to code the motion vectors using all four methods. Every time a vector is discarded, the change in rate for each one is computed (or estimated, using ΔB), and a new rate for each method is

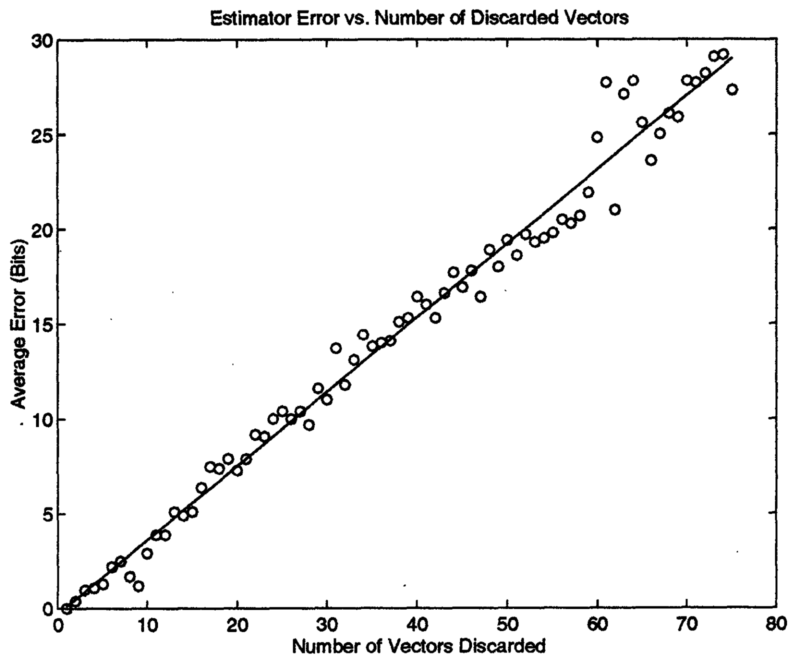


Figure 6.8: Error in Entropy Estimator

obtained. The least of all four methods is the new motion vector coding rate.

6.2.2 Results

Unfortunately, the entropy estimator is just that, an estimator, and will have some error associated with it. Additionally, this error will tend to accumulate every time a new estimate is created. Since the entropy is a conservative estimator, these errors will all be of the same sign, and thus the error will always increase every time a vector is discarded and the new rate is estimated.

Figure 6.8 is a plot of the error, in bits, vs. the number of iterations (i.e., the number of vectors discarded). The error has a very nice, linear characteristic. This implies that the

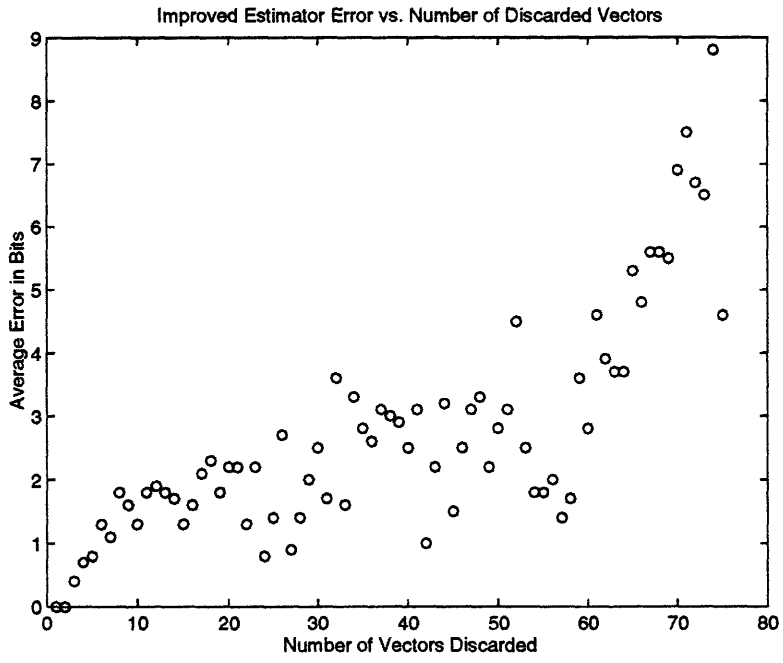


Figure 6.9: Error in Improved Estimator

error is not a strong function of the total number of motion vectors, or their rates of occurrence. Instead, the error added is roughly constant each iteration. The error can be obtained by measuring the slope of the line which corresponds to the least-squares fit to the data. This slope is .3903, meaning that every iteration, on average, accumulates .39

bits of error. To better estimate the number of bits, we can add this amount to ΔB every iteration of the loop. The result is an *improved entropy estimator.* Figure 6.9 is a plot of the error in this estimator as a function of the number of vectors discarded. The improved entropy estimator is quite accurate. After all the vectors have been discarded, the Huffman code needs to be generated. When this occurs, the actual rate can then be computed. This way, the errors never get past the motion vector limiting. The only way the errors surface is if the limiter discards one or two too many vectors, or one or two too few. Either way, the effect is small, and the savings in computation is enormous.

6.3 Initial Rate Computation

In order for the entropy estimator to work, it needs to know the *exact* number of bits for encoding to begin with. However, it is unnecessary to generate the entire Huffman code, since it is only the rate that is needed. Therefore, it would be desirable to determine an algorithm which can quickly determine the rate for such a Huffman code without actually generating the code itself.

There is in fact, a very simple way to determine the rate. Assume that there are N distinct symbols, and that the rate of occurrence for each one is R_i. We seek the rate required to code all the symbols, which is:

$$\sum_{i=1}^{N} L_i R_i$$

At each stage in the Huffman tree generation procedure, two nodes are combined to form a new node, whose total rate of occurrence is the sum of the two nodes beneath it. This implies that the Huffman code for each distinct symbol beneath the new node increases in length by one. The number of times each symbol will be transmitted is equal to the rate of occurrence of that symbol. If all the symbols beneath the new node have their Huffman codes increase in length by one, the number of additional bits required will be equal to the sum of the rates of occurrences of those symbols. This sum is, by recursive definition, equal to the rate of occurrence for the new node. Thus, *at every step in the tree* *generation process, the total number of bits required to encode the symbols increases by the rate of the new node.* Figure 6.10 illustrates this concept.

There are three symbols, A,B, and C, with rates of occurrence of 15,12 and 8, respectively. At stage 0, no code has yet been generated, and the total number of bits required is zero. At stage 1, symbols B and C are combined into a new node with a rate of

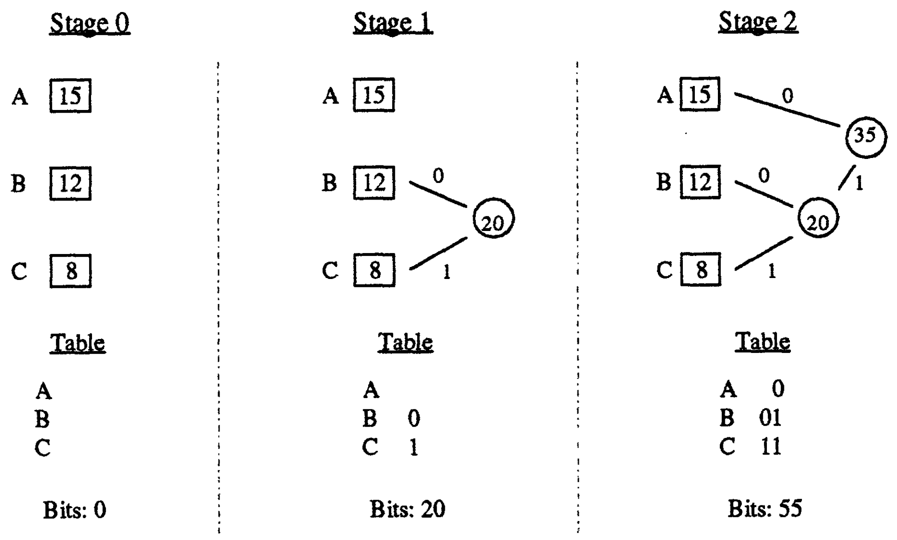


Figure 6.10: Huffman Rate Computation

20. This implies that the Huffman code for symbols B and C will increase in length by 1. Since there are 12 occurrences of B, and 8 of C, 20 bits will be required for transmitting the symbols with the code so far. At stage 2, the new node and symbol A are combined. This increases the length of the Huffman code for symbols A, B, and C by 1. The rate of the newest node (35), is also equal to the sum of the rates of all the symbols below it, A,B, and C. Therefore, the total number of bits required increases by 35, bringing the total to 55. This number can now be checked. $L_A = 1$, $L_B = 2$, and $L_C = 2$. Plugging into the equation for B:

$$B = \sum_{i=1}^{B} L_i R_i$$
$$B = 1 \times 15 + 2 \times 12 + 2 \times 8$$
$$B = 15 + 24 + 16$$
$$B = 55$$

Which is correct.

Implementing the algorithm is much easier than figure 6.10 would indicate. At each stage in the process, only the rates need be stored, not the connections between the nodes. At every stage, the two smallest rates are added, and the sum is added to the total number of bits. The two rates which were summed are removed from the list, and the new rate is added to the list. The list is then sorted (or, the new rate can be inserted in the proper location), and the procedure iterates until there is one node.

Since the procedure must be passed a list of N rates of occurrence, and since the only data that needs to be stored are the rates at every successive stage, the algorithm requires no additional memory, that is, in can occur *in-place*. Typical implementations of Huffman code generators require the creation of 2N-1 additional storage locations. Although it still requires $O(N^2)$ in time to compute (the same for generating the code), there is no overhead associated with dynamic memory management and pointer manipulations. As a result, it is much faster than the algorithm to generate the code itself.

6.4 Huffman Code Generation

After enough motion vectors have been discarded to bring the rate below the threshold, the actual Huffman code must be generated (along with the rate required, in order to correct for errors in the estimator). Typically, Huffman codes are generated by building a Huffman tree, and then traversing the tree to build the actual Huffman table. In typical C implementations, nodes in the tree are often created by *malloc*, which will allocate space for the node and return a pointer to it. Since the connections between nodes is data-dependent, dynamic memory management, using malloc, is an efficient way of creating the tree.

Unfortunately, the MIPS-X, which must be used to generate the Huffman code, does not have the ability to perform dynamic memory management via *malloc* and *free*. In order to implement the Huffman code generation algorithm, a different procedure needs to be used. One possibility is to simply declare an array which is large enough to handle all 2N-1 nodes, and then manage the chunk of memory manually. However, this is both wasteful and time consuming. A better way is to come up with an algorithm for generating a Huffman code which does not require dynamic memory management.

It is quite simple to come up with such an algorithm by realizing that the Huffman tree is an intermediate step to the final, desired product, which is the Huffman table. The Huffman table is a fixed size, and does not require dynamic memory management to maintain. Therefore, if the Huffman table can somehow be generated without using the tree, the memory problems will be solved.

The procedure to generate the table without the tree operates in a similar fashion to the algorithm for computing the rate without the tree. At every stage in the process, when two nodes are combined, the length of every symbol below the new node increases in length by one. In addition, all the symbols associated with one of the children of the new node has a 0 appended to its Huffman code, and all the symbols associated with the other child have a 1 appended to their Huffman code. Every top node (a node with no parents), say node i, in a given stage in the Huffman tree is associated with a set of symbols, R_i. This set is a subset of the set of all symbols, Z. Since no symbol is beneath more than one top node, the R_i are disjoint. Additionally, since all symbols are beneath a top node, the union of the R_i is Z. When two nodes are combined (node a and b, for example), the new node, c, becomes associated with $R_a \cup R_b$. To generate the Huffman code, a zero is appended to the codeword of every member of R_a, and a one is appended to every member of R_b. This process can then be iterated until the code is generated.

At every stage in the process, only two pieces of information need to be maintained. The first is a list of all the top nodes, along with their rates of occurrence and a pointer to the set of symbols R_i with which they are associated. The second piece of information is the set of R_i. Since the R_i are always disjoint and always union to Z, they can be stored in a fixed space. The notion of a set can be maintained by linking every symbol to the next symbol in the set. The last symbol in a set is linked to nothing. Thus, to join two R_i, the pointer at the end of one set is linked to the first symbol in the next set. Figure 6.11 illustrates the procedure. We start with four symbols, A,B,C, and D. At the first stage, there are four nodes, corresponding to the four symbols. The rate of occurrence for each node is 2, 4, 5, and 7, respectively. At stage 1, nodes 2 and 4 are combined. The set of symbols associated with node 2, {A}, has a zero appended to their codewords, and the set associated with node 4, {B}, has a one appended to their codewords. The two sets of symbols are then linked together, and become associated with the new node. At stage 2, nodes 6 and 5 are combined. All the symbols associated with node 6, {A,B} have a 0 appended to their Huffman codewords, and all the symbols associated with node 5, {C}, have a 1 appended to their codewords. The two sets are then joined and associated with the new node. At stage three, the final two nodes are combined. The set of symbols {A,B,C} associated with node 11 have a 0 appended to their Huffman code, and the set of symbols {D} associated with node 7 have a 1 appended to their code. Then, the two sets are combined, and associated with the new node. With only one node remaining, the procedure is complete.

As with the algorithm described in section 6.2, the Huffman code generating algorithm does not require that the parent-child relationships be maintained. The only relationships that need to be kept are the association of the top nodes with their symbols. Since the number of top nodes starts at N and decreases by one at every stage, and since the lists require O(N) space, the entire algorithm requires only O(N) space as well, *without* any dynamic memory management. The big advantage is in computational savings. All of the overhead associated with maintaining all the parent-child relationships, along with the overhead of traversing the tree to generate the table, are eliminated. Even though the computation is still $O(N^2)$ in time, it is much faster than the procedure outlined in *Numerical Recipes in C* [22].

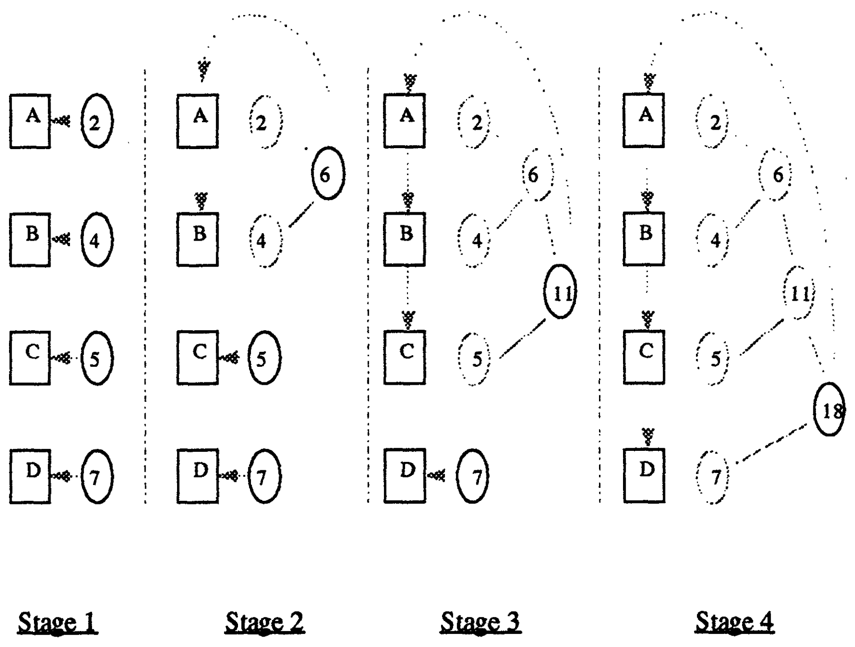


Figure 6.11: Huffman Code Generation procedure

6.5 Summary

The results of this chapter indicate that on-the-fly Huffman code generation does not pose an unreasonable strain on computational resources. By initializing the rate required, using the results of Section 6.3, then using the entropy estimator to estimate the rate, as described in Section 6.2, and finally generating the code using the procedure outlined in Section 6.4, the computation can be greatly reduced.

Chapter 7

Vector Quantization Rate Calculation

As with motion vector rate computation, the vector quantization rate computation can be very time consuming. Every time a VQ is added or discarded, the coding rate must be recomputed. The coding rate consists of several factors:

1. 8 bits to address into the codebook
2. 2 bits to indicate the type of the MV/VQ instance
3. The coded block position of the MV/VQ instance

It is computing the rate for the CBP (coded block positions) that can be the most time consuming. The coded block positions are transmitted in one of three ways: with a 140 bit mask, direct addressing, or Huffman coding the zero-run length codes for the mask. The third method is the one that requires the most time to compute. If the discarding or addition of a VQ deletes or adds an MV/VQ instance, the entire mask must be run-length coded, and then each of the run-length tokens must be looked up in a Huffman table. In and of itself, the operation is not time consuming. However, it can be iterated as many as 280 times in a single frame.

Section 7.1 describes a mathematical formulation for determining the cases for which the Huffman coding will be used. Section 7.2 describes a modification to the coding routine which can reduce computation time dramatically.

7.1 Bounding the Usage of the Huffman Code

When transmitting the coded block positions, the total rate can never exceed 140 bits. This is because the transmission of the mask always requires 140 bits. If either the Huffman RL coding or the direct addressing require more bits, the mask will be used. The method used is highly dependent on the number of MV/VQ instances. When the number is small, direct addressing or Huffman RL-coding are often used. When the number is large, Huffman coding or mask coding will be used. For very large numbers of MV/VQ instances, the mask is almost always used.

Since most of the computation time is spent determining the coding rate for the Huffman codes, it would be very efficient to determine exactly when the Huffman coding will or won't be used. If, for example, the Huffman coding is never used when there are more than 76 MV/VQ instances, 128 of the 280 iterations of the VQ rate computation routine will not need to check the Huffman coding rate.

7.1.1 Mathematical Formulation

Determining this range is the same as answering the following mathematical question:

Given a Huffman Code H for coding zero-run lengths in a sequence of M binary digits, and given that N of those binary digits (N<M) are 1, there are $\binom{M}{N}$ *possible binary sequences. Which sequence can be coded with H and result in fewer bits than any other sequence, and how many bits will it require?*

By answering this question, we can find the set of N for which the number of bits is greater than 140. This set of N will be the range over which the Huffman code will never be used. The proof that follows will show that the sequence which requires the fewest bits to code is the sequence that has runs of length zero and one only. Any sequence of this form will require 13N-840 bits to code, where N is the number of MV/VQ instances. As a result, if there are more than 76 MV/VQ instances, the Huffman code will never be used.

7.1.2 Proof

To solve this problem, a few terms and operations will need to be defined:

α_N : This is a sequence of M binary digits (M is a universal constant), N of which are 1. It is known as an N-Vector. Alpha can be any alpha-numeric identifier, i.e. x,y,z, etc.
R_β : Is a vector containing the run-lengths of an N-Vector. It is known as an R-Vector. The number of run lengths in an R-Vector is always less than M+1. Beta is any numeric identifier.
H(a): H(a) is the length (number of bits) of the Huffman code for a run of length a.

For example, assume X_3 is an N-Vector, and X_3 = [0 1 1 0 0 1]. The R-Vector, R_1, which corresponds to this N-Vector would then equal [1 0 2]. The number of bits required to Huffman code these run lengths is then equal to H(1) + H(0) + H(2). In general, the number of bits required to Huffman code an R-Vector is:

$$\text{Bits} = \sum_i H(R[i])$$

First, we define a transformation T_A which operates on an N-Vector. The result of this transformation is another N-Vector which has bit positions A and A+1 swapped, and all other bits unchanged. For example:

$$X_4 = [\mathbf{0}\ \mathbf{1}\ 1\ 0\ 1\ 1]$$

$$Y_4 = T_1(X_4) = [\mathbf{1}\ \mathbf{0}\ 1\ 0\ 1\ 1]$$

The first and second bits of Y_4 (which are bold for clarity) are the second and first bits of X_4. All the other bits in Y_4 are the same as those in X_4. Clearly, this transformation preserves the total number of ones in a sequence.

Swapping two bits can have one of three effects on the run lengths of an N-Vector. If T_A is applied to X_N, and $X_N[A] = X_N[A+1]$, then the swap will have no effect on the run lengths in X_N, since $T_A(X_N) = X_N$. If $X_N[A] = 0$ and $X_N[A+1] = 1$, then the run lengths of the transformed sequence will be different from the run lengths in the original. The run which formerly ended at bit position A+1 will now end at bit position A, decreasing the run length by 1. The run which formerly began bit position A+1 will now begin at bit position A, increasing the run length by 1. The situation is exactly flipped if $X_N[A] = 1$ and $X_N[A+1] = 0$. The run which ended at A will now end at A+1, and thus increase in length by 1. The run which started at A will now start at A+1, and thus decrease in length by one. For example, let $X_3 = [0\ 0\ 1\ 0\ 0\ 1\ 1]$. By run length coding X_3, the resulting R-Vector will be [2 2 0]. If T_3 is applied to X_3, the result, Y_3, will be $[0\ 0\ 0\ 1\ 0\ 1\ 1]$. Run length coding Y_N will result in the R-Vector [3 1 0]. Notice that one of the run lengths has increased by one, and the one next to it has decreased by one.

Once again, a T_A transformation will have one of three effects on the run-lengths. The transformation will either do nothing (i.e., when $X_N[A] = X_N[A+1]$), increase the length of the run ending at A, and decrease the length of the run starting at A+1, or decrease the length of the run ending at A+1, and increase the length of the run starting at A+2. We can define the last two transformations as $T_+[X,Y]$ and $T_-[X,Y]$ respectively. When $T_+[X,Y]$ is applied to an N-Vector whose R-Vector has a run of X followed by a run of Y, the result is another N-Vector whose R-Vector has a run of X+1 followed by a run of Y-1. When $T_-[X,Y]$ is applied to an N-Vector, the result will have an R-Vector with a run of X-1 followed by a run of Y+1. $T_+[X,Y]$ and $T_-[X,Y]$ will leave all other runs unchanged. Clearly, $T_+[X,Y]$ will only be defined if the N-Vector it operates on corresponds to an R-Vector which *has* a run of X followed by a run of Y.

We will now state and prove Theorem I:

Theorem I: Define X_N as an N-Vector whose R-Vector has a run of length 0 next to any other run of length A (A>1). Define Y_N as $T_+[0,A]$ applied to X_N. Given that $H(0) - H(1) > H(A-1) - H(A)$ for all A>1, Y_N will always require fewer bits than X_N when coded by H.

The proof of the theorem is as follows. First, we define R_1 as the R-Vector corresponding to X_N, and R_2 as the R-Vector corresponding to Y_N. R_1 and R_2 differ only in two places, say the pth and p+1th. R_1 has a 0 in the pth element followed by an A in the p+1th, and R_2 has a 1 in the pth followed by an A-1 in the p+1th. All other elements of R_1 and R_2 are identical.

Define H_1 as the number of bits required to code R_1, and H_2 as the number of bits required to code R_2. Then,

$$H_1 = \sum_i H(R_1[i])$$
$$H_2 = \sum_i H(R_2[i])$$

We expand the above sums as:

$$H_1 = \sum_{i \neq p,p+1} H(R_1[i]) + H(0) + H(A)$$
$$H_2 = \sum_{i \neq p,p+1} H(R_2[i]) + H(1) + H(A-1)$$

Since R_1 and R_2 are identical in every position except the pth and p+1th,

$$\sum_{i \neq p,p+1} H(R_1[i]) = \sum_{i \neq p,p+1} H(R_2[i])$$

This implies that $H_1 - H_2$ can be written as:

$$H_1 - H_2 = H(0) + H(A) - H(1) - H(A-1)$$
$$H_1 - H_2 = H(0) - H(1) - \left(H(A-1) - H(A)\right)$$

We are given that H(0) - H(1) > H(A-1)-H(A), which implies that H_1-H_2 is positive for all A>1. Since X_N requires H_1 bits when coded by H, and Y_N requires H_2 bits, X_N requires more bits to code than Y_N, and the theorem is proven.

It is important to discuss the implications of the theorem. It is only true when the Huffman code has the property that H(0)-H(1) > H(A-1)-H(A). A Huffman code with this property is one for which coding a run of zero costs many more bits than just about every other run. It therefore makes sense that changing the run of zero to a run of one reduces the bits required to code the sequence.

As it turns out, this property is generally true of Huffman run length codes. Huffman run length codes are efficient at coding *sparse* vectors. This implies that there are few ones,

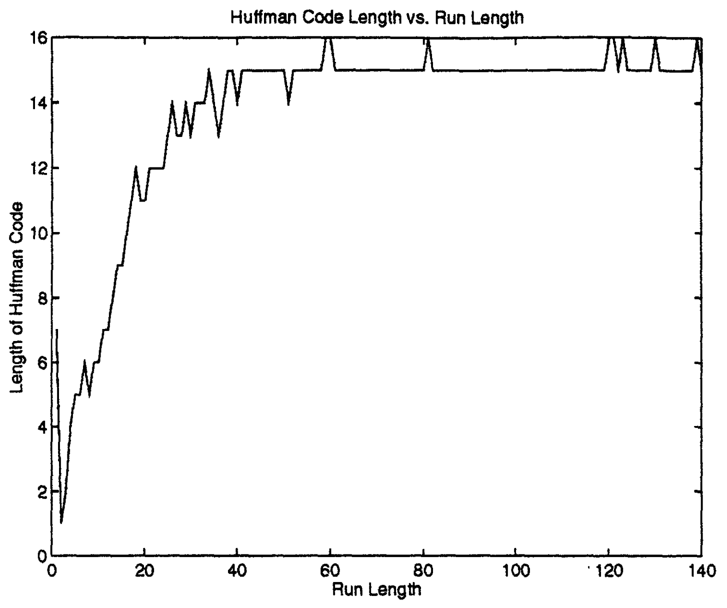


Figure 7.1: Huffman Code Lengths

and therefore there are very few cases where a one is right next to another one (corresponding to a run-length of zero). Therefore, if a Huffman run length code is generated from statistics based on sparse vectors, we would expect that a run of zero is the most costly to code.

This property is possessed by the Huffman code used in the Scene Adaptive Coder. Figure

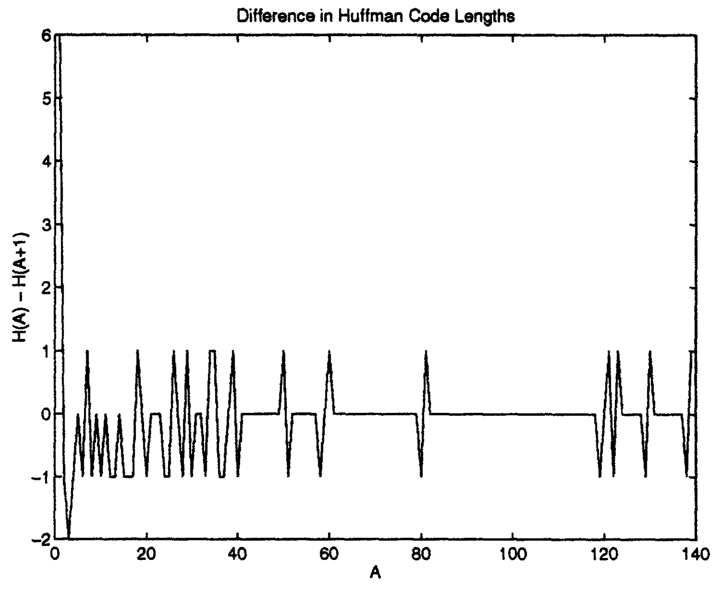


Figure 7.2: Difference in Huffman Code Lengths

7.1 is a plot of the run lengths vs. the number of bits required to code that run length. Note that a run of zero requires the most amount of bits to code. Figure 7.2 is a plot of the forward difference of the code, i.e. H(A-1) - H(A). Notice that H(0) - H(1) is greater than all the other differences.

We now state and prove Theorem II:

Theorem II: Define X_N as an N-Vector and R_1 as the corresponding R-Vector. There always exist a series of T_+ or T_- transformations that can be applied to X_N, resulting in an N-Vector Y_N with an R-Vector R_2 such that $R_1[i] = R_2[i]$ $i \neq p, p+1$, $R_1[p] = R_2[p+1]$, $R_1[p+1] = R_2[p]$.

In other words, we can "swap" two run lengths by applying successive T_+ or T_- transformations. Let $A=R_1[p]$, and $B=R_1[p+1]$. Clearly, if A>B, then (A-B) T_- transformations will accomplish the swap, and if A<B, then (B-A) T_+ transformations will accomplish the swap. This "swapping" will also produce an R_2 which requires the same number of bits to code as R_1. This is because the set of runs in R_1 and R_2 is different only in their order, which does not affect the coding rate.

With these theorems, we can now prove the main result, Theorem III:

Theorem III: If N>M/2, the N-Vector which requires the fewest number of bits when coded by H has no run lengths greater than 1.

We prove by contradiction. Assume that X_N is the N-Vector (N>M/2) with the smallest Huffman code, and it has a run length greater than one. By the *pigeonhole principle*, since N>M/2, there must be two 1's in X_N that are next to each other, which corresponds to a run length of zero. If the run length of zero is not next to the run of length greater than one, by Theorem II we can perform successive swaps which can bring the run length of zero next to the run length of greater than one. These swaps will not alter the number of bits required to code the R-Vector. Since this R-Vector then has a run length of zero next to a run length greater than one, then by Theorem I, there also exists a transformation T_+

which can then be applied, resulting in a new N-Vector with a Huffman code less than that of X_N's. However, this is a contradiction, since X_N is the vector with the smallest Huffman code. Therefore, the theorem is proven.

Now, we have answered the first part of the original mathematical question, i.e., *what is the sequence which will require the fewest number of bits to code with H*? We can now answer the second question, and determine how many bits are required to code it.

We know that a run of length zero occupies one bit in X_N, and a run of length 1 occupies 2 bits. The total number of bits must be M. In addition, since N is the number of ones in X_N, and there is a single 1 at the end of each run, the sum of the number of runs in any X_N must be N. Let A_0 be the number of runs of length 0, and A_1 the number of runs of length 1. Solving:

$$A_0 + 2A_1 = M$$
$$A_0 + A_1 = N$$

results in:

$$A_1 = M - N$$
$$A_0 = 2N - M$$

As Figure 7.1 illustrates, the Huffman code for a run length of zero requires 7 bits, and the Huffman code for a run length of one requires 1 bit. Additionally, M=140. Plugging in, we can obtain the number of bits required for coding, B, as a function of N:

$$B = 1 \cdot (M - N) + 7 \cdot (2N - M)$$
$$B = 13N - 6M$$
$$B = 13N - 840$$

We are interested in the case where the number of bits is greater than 140. Plugging in for B, and solving for N:

$$N = 75.38$$

We have therefore solved the mathematical problem posed at the beginning of the section. The sequence which requires the fewest number of bits to code is the sequence with runs of length zero and one only, and any sequence of that form will require 13N-840 bits to code. Therefore, any sequence with more than 76 ones will always require more than 140 bits to code.

7.1.3 Application

The results of the previous section are not unexpected, but are very helpful. Given that there are more than 76 MV/VQ instances in the frame, the Huffman coding of the coded block positions *must* required more than 140 bits. This important result greatly reduces computation. The encoder can easily keep track of the number of MV/VQ instances in the frame. During the vector quantization, as VQ's are added, the coder can stop trying the Huffman coding as soon as more than 76 MV/VQ instances are present. As VQ's are discarded during the dynamic bit allocation, the Huffman coding does not need to be tried until the number of MV/VQ instances falls below 76. Therefore, out of 280 possible iterations of the MV/VQ rate computation, only 152 are actually necessary, which is a reduction in computation by almost a factor of two.

Further gains can be made, however. During the vector quantization, blocks are VQ'd until the total rate has been exhausted. The rate available before vector quantization begins is the total number of bits in a frame, minus the number of bits required for motion estimation. However, the number of bits for motion estimation is fixed at 15% of the total bits in a frame, leaving 85% for the vector quantization. If the bitrate is high enough, the encoder may be able to vector quantize *all* of the blocks. The rate required is 140 +10N bits, where N is the number of blocks that have been quantized. If we set N to 140, we can solve for the minimum number of bits per frame:

$$\frac{\text{Bits}}{\text{Frame}} \times .85 > 1540$$

$$\frac{\text{Bits}}{\text{Frame}} = 1811$$

At the 10fps, 16kb/s coder configuration, there will be 1600 bits per frame. All other configurations involve higher bitrates and smaller frame rates. This implies that the coder will never work with less than 1600 bits per frame. Even for this worst case configuration, 121 VQ's can be done. The odds that there are 121 codebook matches, out of a maximum of 140, is quite small. This suggests an alternate way of performing the VQ. Figure 7.3 is a

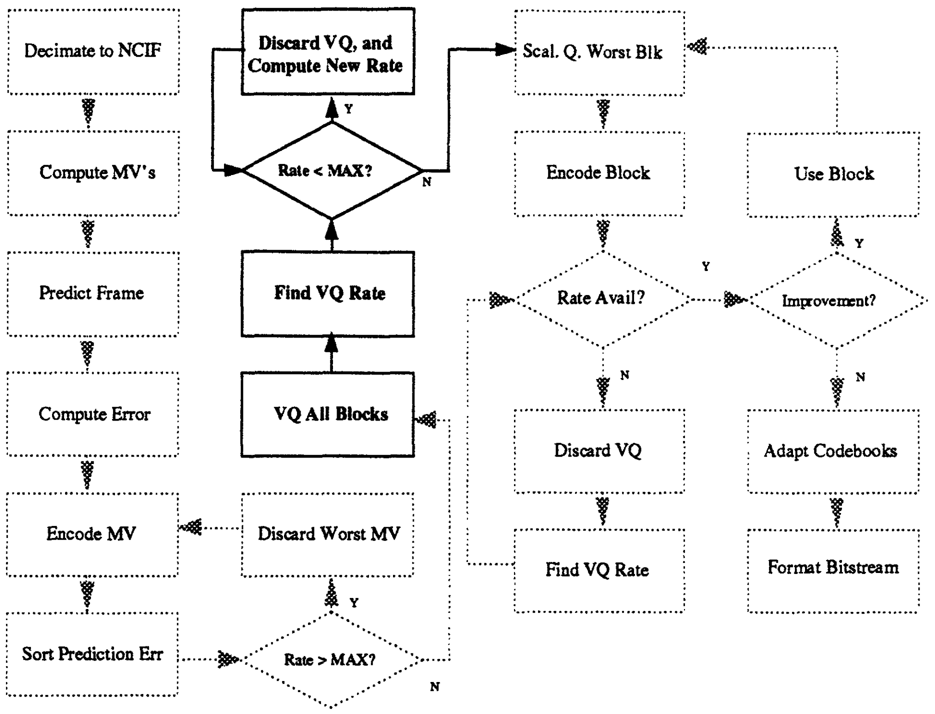


Figure 7.3: New VQ Algorithm

block diagram of the improved VQ routine. Instead of checking the rate every time a VQ is added, all VQ's are done, and then the rate is computed. If this rate is greater than the maximum, VQ's will be discarded until it is below the maximum. In most cases, no discarding will need to be done at all. The result is that the VQ rate does not need to be computed as VQ's are added (unless the bitrate is very low), and only needs to be computed 76 times as they are discarded. This reduces the computation by another factor of two.

7.2 Improving the Rate Calculation

As VQ's are discarded, the rate required will need to be computed. However, it is inefficient to go and Huffman code the entire MV/VQ mask, especially since it only changes by a single bit in every iteration. As with motion vector rate encoding, it is much more efficient to compute the *change* in rate every time a VQ is discarded. Unlike the motion vectors, however, this change needs to be determined exactly, not estimated.

7.2.1 Computing the Rate Change

Every time a VQ is discarded, 8 bits are gained for the address into the codebook. If the block did not have a motion vector either, then the entire instance is deleted, and 2 bits more are gained for the MV/VQ type information, plus however many bits are gained by removing a 1 from the mask of coded block positions.

Computing the change in the coding rate of the coded block positions is simple, and depends on how it is coded:

1. 140 Bit Mask: No change in coding rate.
2. Direct Addressing: 8 bits for the address.
3. Huffman RL Coding: When a one is removed, the run length to the left is removed, and the run length to the right is removed. It is replaced by a run which is the sum of the two plus one..

Figure 7.4 illustrates the concept of changing the coding rate for Huffman RL encoding. Every time an MV/VQ instance is discarded, the new coding rate can be computed by subtracting the number of bits required to code the run to the left, subtracting the number

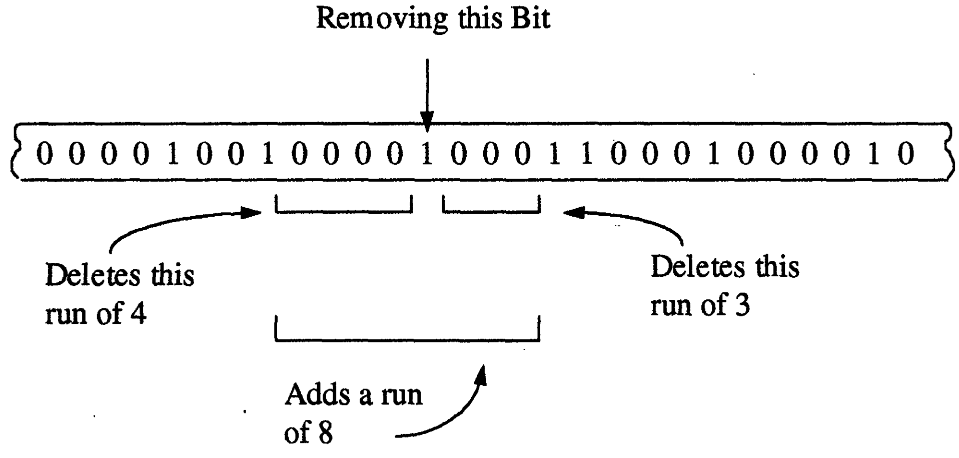


Figure 7.4: Changing the Run Length Coding

of bits required to code the run to the right, and adding the number of bits required to code the new run in the middle.

Unfortunately, this algorithm is still O(N) in time, since it must search for the closest 1 to the left and to the right of the 1 removed. Recomputing the rate all over again is also O(N). However, the worst case running time of the new implementation is proportional to the sum of the two largest runs in the mask, whereas the worst case running time of the original algorithm is proportional to the number of runs in the mask.

7.2.2 Modeling the Running Time

Simple models can be developed for the running time of the old and new versions of the routine. These models can assist in determining whether the new version of the algorithm can be expected to run faster.

Assume that there are N ones in the MV/VQ mask. The average running time of the new

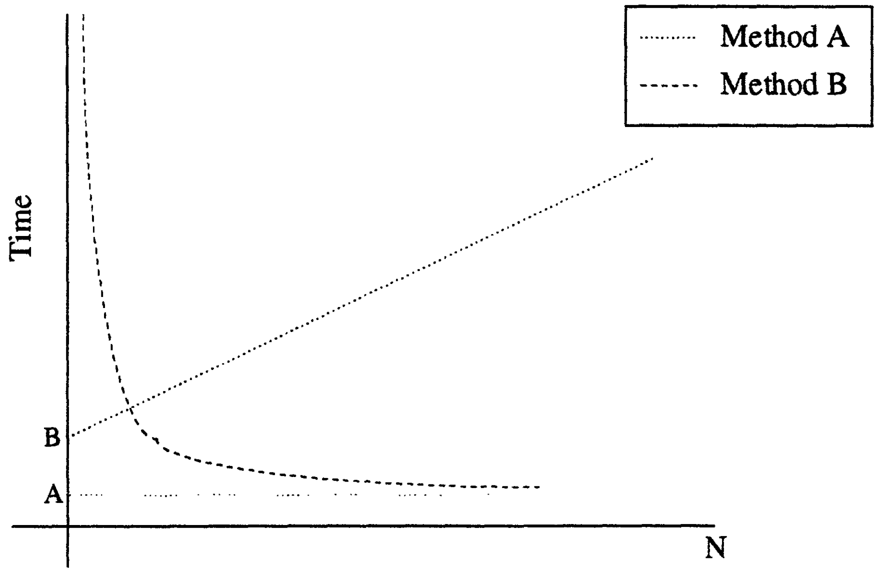


Figure 7.5: Running times of RL Coding Methods

algorithm is proportional to the average length of a run in the mask. Since the sums of the runs in the mask must be 140, the average run length is 140/N. Assume then, that the constant of proportionality is α, and that A cycles are required for setup and initialization. The compute time for the new method is then A +140α/N. For the old version of the algorithm, the running time is proportional to the average number of ones in the mask, which is N. Assume that the constant of proportionality is β, and that B cycles of initialization are required. The compute time for the old method is then B + βN. Figure 7.5 is a plot of the running time vs. N for the old algorithm (method A), and the new one (method B). The utility of the new algorithm depends on the values of α, β, A, and B. Since it is impossible to estimate these quantities, they were measured from actual data. Figure 7.6 is a plot of the actual average running times vs. the number of MV/VQ instances. Clearly, B >> A, and the new method is always faster than the old one, often by a factor of nearly ten.

7.3 Summary

The results of 7.1 define a range of the number of MV/VQ instances over which the Huffman run-length codes will not be used. This result can be used in two ways. First, the

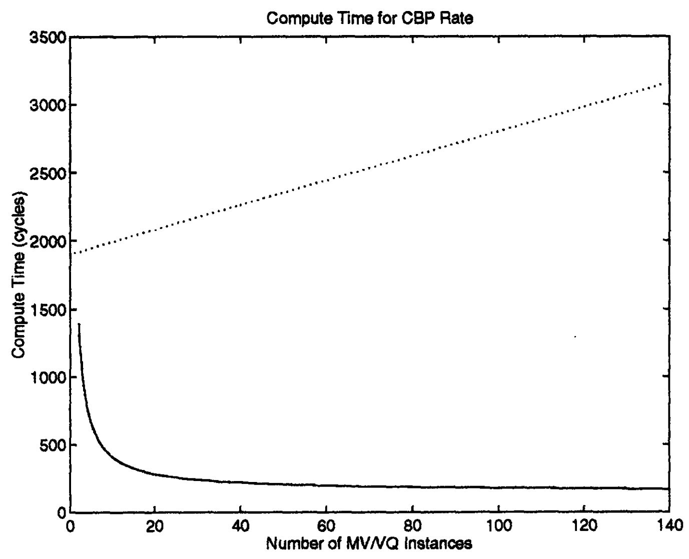


Figure 7.6: Running times for Method A (solid line) and Method B (dotted line) vs. the number of MV/VQ instances.

Huffman codes can be ignored whenever there are more than 76 MV/VQ instances. Secondly, as VQ's are added, there will always be more than 76 of them, so the rate is known. Using this information, it turns out that it is more efficient to perform all the VQ's and *then* check the rate, discarding VQ's if necessary. Despite these changes, the Huffman RL coding rate must be checked during dynamic bit allocation when there are less than 76 MV/VQ instances. In this situation, the modified rate computation routine suggested in section 7.2 can dramatically reduce computation further.

Chapter 8

Motion Estimation

The original non-real time system used a full search motion estimator to determine the motion vectors. However, this method is inappropriate for the real-time system, for two major reasons. First, the reference frame does not fit into the DM memory, and secondly, as Chapter 5 illustrated, the computation time is unreasonable. A much more efficient approach to motion estimation is to use a *coarse-fine* approach. In this method, a search is first done over a smaller range at a lower resolution. Then, the "coarse" estimate is polished by doing a fine search about the coarse estimate at higher resolutions. This chapter analyzes the decision to use coarse-fine motion estimation. Section 8.1 describes how a coarse fine approach might be implemented on the VP. Section 8.2 performs a timing analysis on the coarse-fine approach, and compares it to a full search. Section 8.3 analyzes the degradation incurred by using a coarse-fine approach, using actual data collected from typical sequences.

8.1 Implementing Coarse-Fine Estimation

To determine the motion vectors, the motion search is first done in a range of -5 to 5 pels at NCIF resolutions. Then, since both the current and previous SIF frames are available, a refinement search occurs. This search is done in a range of -2 to 2 pels at SIF resolutions (corresponding to -2/3 to 2/3 pels at NCIF), using a reference frame which has already been offset by the coarse motion vector. This refinement provides the fractional part of the motion vector.

8.1.1 Coarse Motion Estimation

The coarse search occurs over a range of -5 to 5 NCIF pels. The DM memory is used to hold the reference frame. Since the coarse motion search is in a range of -5 to 5 NCIF pels in both the x and y directions, a reference block of size 18x18 is needed. Once again, since DMA from the reference memory can occur in units of four pels horizontally, the reference block is enlarged to 24x18, although the extra 3 pels on each side are unused. The DP memory is used to hold the 8x8 NCIF block from the current frame. The coarse search routine then only needs to keep track of the x and y position of the best match, the x and y position of the current block, the SAD of the best match, and the SAD of the current block. These numbers can be stored and maintained by the risc core. The motion estimation unit can be used in parallel with the risc core, so some of the overhead (SAD comparisons, address increments, etc.) can occur along with the actual SAD computations. Once the best match is found, the motion vector can be transferred directly to the MIPS-X via the direct parameter transfer. The routine can also check the zero motion vector first, and compare the SAD to a threshold. If the SAD was less, the routine can just return a zero motion vector without searching further.

So, to determine the coarse motion vector, a 24x18 block from the previous NCIF frame is transferred from DRAM to the DM memory in the VP. Then, an 8x8 block from the current NCIF frame is transferred from DRAM to the DP memory in the VP. The coarse motion algorithm is called, and then the VP transfers the motion vector back to the MIPS via the direct parameter transfer. This procedure is then repeated for all 140 blocks.

8.1.2 Fine Motion Estimation

The procedure for implementing the fine motion estimation is nearly identical to that for coarse motion estimation. The MIPS-X will transfer an appropriately sized reference block to the DM, the 24x24 SIF block to the DP, call the fine motion search routine, and the VP will return the refined motion vector via direct parameter transfer. This time, the returned motion vector is between -2 and 2, and is the fractional amount to be added to the -5 to 5 returned from the coarse routine.

The main difference between the coarse and fine routines is overhead. The MIPS-X needs to use the coarse motion vector to select the appropriate reference block. However, the 4-pel alignment rule in DMA transfers means than some motion vectors will have extra pels in the reference block, and others will not. The MIPS-X must keep track of this problem and be sure to inform the VP, by calling the correct setup routine, where in the DM the reference block is actually located.

8.2 Timing Analysis

8.2.1 Coarse Motion Search

Like the full-search motion estimation, the coarse estimation computation time consists of four components: DMA the reference block to the DM, DMA the current block to the DP, perform the motion search, and pass the result back via direct parameter transfer. Ignoring the parameter transfer time, the total time can be written as:

$$T_{CMVEST} = T_{DMA1} + T_{DMA2} + T_{COMP}$$

The reference block is 24x18, which is 8x18 or 144 longwords. At 1.5 cycles per longword, T_{DMA1} = 144 longwords x 1.5 cycles/longword = 216 cycles. The block from the current frame is 8x8. Transferring it to the DP requires 24 cycles, so $T_{DMA2} = 24$.

Since the coarse search occurs over a range of -5 to 5 pels in both the x and y directions, there are a total of 11x11 = 121 motion vectors to check. Each motion vector comparison requires an 8x8 block to be compared to another 8x8 block. Each comparison requires 2 cycles. However, an additional 5 cycles are needed for performing the comparisons and address calculations. Therefore, T_{COMP} = 121 iterations x 7 cycles/iteration = 847 cycles.

Plugging in to T_{CMVEST}:

$$T_{CMVEST} = T_{DMA1}\ (216) + T_{DMA2}\ (24) + T_{COMP}\ (847)$$

$$T_{CMVEST} = 1087 \text{ cycles}$$

This search is repeated for all 140 blocks in the frame, bringing the computation time up to 152.18×10^3 cycles. Adding 15% for overhead, the total time is 175.01×10^3 cycles.

Coarse Motion Search	175.01×10^3 cycles	6.2% Total Time

8.2.2 Fine Motion Search

Fine motion search proceeds in a similar manner to coarse motion search. The 24x24 block from the current SIF frame is DMA'd to the DP memory, the 32x28 block from the previous SIF frame is DMA'd to the DM memory, and the search routine is called. The resulting motion vector is then passed back via direct parameter transfer. Ignoring the latter term,

$$T_{FMVEST} = T_{DMA1} + T_{DMA2} + T_{COMP}$$

The 24x24 block from the current SIF frame is 6x24 or 144 longwords. At 1.5 cycles per longword, T_{DMA1} = 144 longwords x 1.5 cycles/longword = 216 cycles. The 32x28 reference block is 8x28, or 224 longwords. Thus T_{DMA2} = 224 longwords x 1.5 cycles/longword = 336 cycles.

The fine search range is -2 to 2 pels in both the x and y directions. Therefore, a total of 5x5 or 25 comparisons must be made. Each comparison is of a 24x24 block, which is nine 8x8 blocks. Unlike coarse motion estimation, the overhead for comparisons and address computations can occur along with the use of the motion compensation unit. Therefore, T_{COMP} = 25 comparisons x 9 blocks/comparison x 2 cycles/block = 450 cycles. Plugging in for T_{FMVEST},

$$T_{FMVEST} = T_{DMA1}\,(216) + T_{DMA2}\,(336) + T_{COMP}\,(450)$$

$$T_{FMVEST} = 1002 \text{ cycles}$$

In the worst case, this must be iterated for all 140 blocks in the frame, which brings the time up to 140.28×10^3 cycles. Adding 15% for overhead, the total computation time comes to 161.32×10^3 cycles.

Fine Motion Search	161.32×10^3 cycles	5.7% Total Time

8.2.3 Comparison

The full search technique requires 3.6×10^6 cycles. Using a coarse-fine approach, the computation time has been brought down to 336×10^3 cycles, a reduction by over a factor of 10! The computational improvement comes at a penalty, however. The motion vectors resulting from a coarse-fine approach may no longer be the same as those resulting from the full search. The next chapter examines this problem.

8.3 Distortion Analysis

8.3.1 Mathematical Formulation

Assume that the results of the full-search motion estimation are a set of motion vectors $\mathbf{A}_i$, and the results of the coarse-fine motion estimation are another set of motion vectors, $\mathbf{B}_i$. These motion vectors may be different, such that $\mathbf{A}_i \neq \mathbf{B}_i$ for some i. Let $\mathbf{E}_i$ be the block predicted from $\mathbf{A}_i$, and $\mathbf{F}_i$ be the block predicted from $\mathbf{B}_i$. Since the same predictor is used in either case, we have $\mathbf{E}_i = \mathbf{F}_i$ if $\mathbf{A}_i = \mathbf{B}_i$. Let $\mathbf{C}_i$ be the block in the current frame which $\mathbf{E}_i$ and $\mathbf{F}_i$ are attempting to predict.

At first glance, it may seem that $\mathbf{E}_i$ is always a better prediction of $\mathbf{C}_i$ than $\mathbf{F}_i$. However, this is not true. First of all, the motion estimation is done by comparing with the previous input frame, not the previous reconstructed frame. Therefore, when the motion vectors are applied to the previous reconstructed frame, the resulting predicted frame may not be the best match for the current frame. Even if the prediction was made by comparing with the previous reconstructed frame, $\mathbf{E}_i$ may still not be a better prediction of $\mathbf{C}_i$. Both $\mathbf{A}_i$ and $\mathbf{B}_i$ minimize the difference between the current SIF frame and the previous SIF frame ($\mathbf{A}_i$ minimizes it globally, $\mathbf{B}_i$ locally). However, the prediction error is the difference between the decimated current frame and the decimated and *filtered* previous frame. Mathematically,

$$PE_i = \sum_{n_1,n_2 \in \text{Block}(i)} \left(C(3n_1,3n_2) * H(3n_1,3n_2) - P(3[n_1 - x],3[n_2 - y]) * H(3n_1,3n_2) * I(n_1,n_2) \right)^2$$

Where H is the decimating filter, I is the polyphase filter used for motion compensation (which depends on the fractional portion of the motion vector), C is the current SIF frame, P is the previous SIF frame, and x,y is the integral portion of the motion vector at NCIF resolutions. $\mathbf{A}_i$ and $\mathbf{B}_i$, in some sense, minimize:

$$\sum_{n_1,n_2 \in \text{Block}(i)} \left(C(n_1,n_2) - P([n_1 - x'],[n_2 - y']) \right)^2$$

where x and y prime is the motion vector at SIF resolutions. Clearly, the minimization of the latter equation will not necessarily minimize the former for either full-search or coarse-fine estimation.

The question then, is not how much better are the results of coarse-fine estimation as compared to full search, but how *different* are they. If we define $\mathbf{Z}_i$ as $\mathbf{E}_i$-$\mathbf{F}_i$, we can consider $\mathbf{Z}_i$ to be a measure of the difference between using full-search as compared to coarse-fine estimation. If the energy in $\mathbf{Z}_i$ is significant in comparison to the energy in the block it is trying to predict, then the different techniques will produce results which are noticeably different. Therefore, a measurement of the energy in $\mathbf{C}_i$ vs. the energy in $\mathbf{Z}_i$ can be considered a "signal to noise" ratio, in that the larger this SNR, the less noticeable the differences between full-search and coarse-fine estimation.

If we consider $\mathbf{Z}_i$ a stationary stochastic process, we can seek to measure its energy as:

$$\text{ENERGY} = E[\mathbf{Z}_i \mathbf{Z}_i^T]$$

Where E[] denotes the expectation operator. $\mathbf{Z}_i$ contains two pieces of information. First if it is zero, then $\mathbf{E}_i = \mathbf{F}_i$, which implies that $\mathbf{A}_i = \mathbf{B}_i$. If it is nonzero, then $\mathbf{A}_i \neq \mathbf{B}_i$. When $\mathbf{Z}_i$ is nonzero, its value indicates how different the predictions are. If we assume that the two variables are independent, we can model $\mathbf{Z}_i$ as $X_i\mathbf{G}_i$, where X_i is a Bernoulli random variable of mean P, and $\mathbf{G}_i$ is random variable which contains information about how different $\mathbf{E}_i$ is from $\mathbf{F}_i$, when they are different. Using this model, we can write the energy in $\mathbf{Z}_i$ as:

$$\text{ENERGY} = E[X_i^2 \mathbf{G}_i \mathbf{G}_i^T] = P \cdot E[\mathbf{G}_i \mathbf{G}_i^T]$$

P can be obtained by averaging the number of times $\mathbf{A}_i \neq \mathbf{B}_i$ in a frame, divided by the number of motion vectors. Figure 8.1 is a graph of the percentage of motion vectors which were different in each frame. These values were measured in an open-loop system, that is, the predicted frame was always generated from a perfect previous frame. This eliminates quantization noise and other coder errors from all computations. The sequence

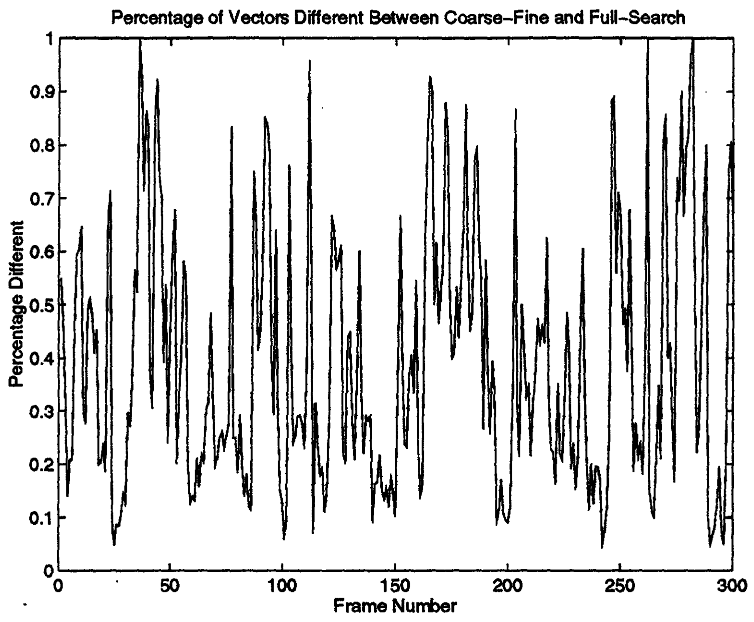


Figure 8.1: Percentage of Motion Vectors Different

was obtained from one minute of 5fps video of a typical head-and-shoulders scene. Notice that the percentage is quite substantial, oftentimes 100%. Averaging over all three-hundred frames, we can estimate P as .39.

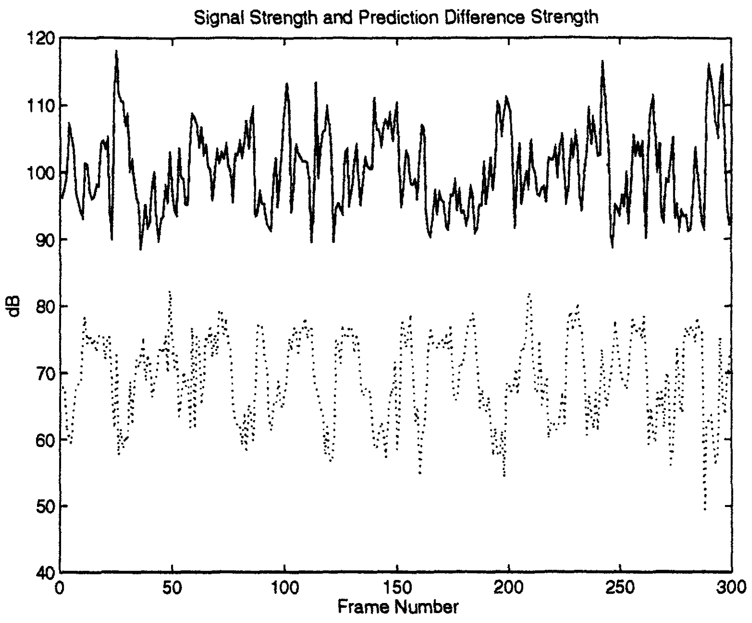


Figure 8.2: Signal energy (solid line) and difference energy (dashed line)

Figure 8.2 is a logarithmic plot of the average energy in the difference $\mathbf{Z}_i$ (when it is nonzero), vs. the energy in the current block $\mathbf{C}_i$. The solid line represents the energy in $\mathbf{C}_i$, and the dotted line the energy in $\mathbf{Z}_i$. Figure 8.3 represents the SNR of $\mathbf{C}_i$ compared to $\mathbf{Z}_i$. Averaging over all 300 frames, the SNR can be estimated at 39.32 dB.

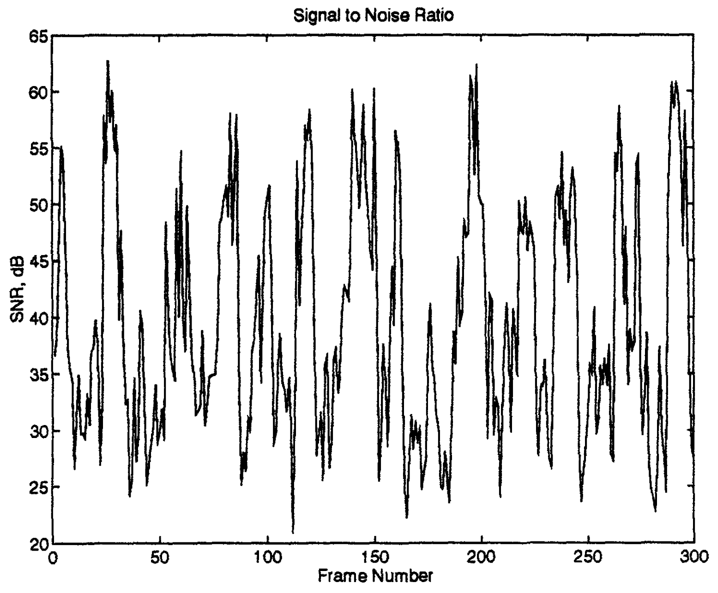


Figure 8.3: SNR of prediction difference

8.3.2 Discussion

At nearly 40 dB, this noise is small in comparison to the errors introduced by quantization noise and coder artifacts. Clearly then, whether full-search or coarse-fine is used will have little noticeable effect on the predicted frame. However, the results of section 8.2 indicate that the coarse-fine approach is over an order of magnitude more efficient, computationally. It therefore makes much more sense to use a coarse-fine approach in motion estimation.

Chapter 9

Conclusions

The research presented in this thesis clearly shows that the Scene Adaptive Video Coder can be recast for real time implementation. With relatively minor adjustments to motion estimation, motion vector rate computation, and vector quantization rate computation, along with intelligent ports of the rest of the algorithm, the SAVC can run in real time.

Routine	**Estimated Time**	**Actual Time**
Decimation	4.03 ms	4.90 ms
Motion Estimation	128.00 ms	12.86 ms
Motion Compensation	2.00 ms	2.31 ms
Error Computation	1.00 ms	2.14 ms
Sorting	6.46 ms	6.46 ms
MV Rate Computation	61.00 ms	1.88 ms
Vector Quantization	12.87 ms	12.87 ms
VQ Rate Calculation	25.30 ms	2.28 ms
Scalar Quantization	3.61 ms	5.00 ms
SQ Rate Computation	6.85 ms	6.85 ms
Codebook Adaptation	3.10 ms	1.99 ms
Bitstream Formatting	0.98 ms	1.22 ms
TOTAL:	**253.40 ms**	**60.76 ms**

Figure 9.1: Estimated and Actual Running Times

Figure 9.1 is a table of the actual worst case running times for the various routines. The worst case running time is now reduced to 61 ms, which implies that the coder can easily run at 10 frames per second. This would indicate that the changes described in Chapter 6,7, and 8 have reduced computation by a factor of 4, with almost no change in image quality. The motion estimation has been reduced from 128 ms to 12.86 ms (this figure includes coarse and fine motion estimation), which is nearly an order of magnitude. The motion vector rate computation has been reduced from 61ms to 1.88 ms, a reduction by over a factor of 30. The VQ Rate calculation has been reduced from 25.3ms to 2.28 ms, a reduction by over an order of magnitude.

The non-real time system requires roughly 180 seconds per frame, so the recast algorithm runs nearly 3000 times faster than the non-real time system. Clearly, this is due to intelligent use of the hardware (i.e., porting vector quantization effectively), in addition to changes in the algorithm.

Bibliography

[1] "ISO/IEC 11172-2, Information Technology - Coding of Moving Pictures and associated audio for digital storage media at up to about 1.5 Mbit/s - Part 2: Video". ITU Recommendation, May 1993.

[2] "Generic Coding of Moving Pictures and Associated Audio, Recommendation H.262, ISO/IEC 13818-2, Draft International Standard", ITU Recommendation, March 1994

[3] E. Petajan, J. Mailhot. "The Grand Alliance HDTV System". *SPIE Proceedings, Image and Video Compression.* Vol. 2186, pp. 2-15, Feb. 1994.

[4] "Draft Revised Recommendation H.261 - Video Codec for AudioVisual Services at Px64 kbit/s", CCITT Recommendation May 1992

[5] H. Li, A. Lundmark, R. Forchheimer. "Image Sequence Coding at very Low Bitrates: A Review". *IEEE Transactions on Image Processing*, Vol.3, No. 5 pp.589-609, September 1994.

[6] "First Draft of MPEG-4 Requirements", ISO/IEC Document, March 1994

[7] H.G. Musmann, P. Pirsch, H.J. Grallert. "Advances in Picture Coding*", Proceedings IEEE*, vol. 73, pp. 523-548, April 1985.

[8] R.J. Clarke. *Transform Coding of Images.* London:Academic, 1985.

[9] A. Gersho, R.M. Gray. *Vector Quantization and Signal Compression.* Boston:Kluwer, 1991.

[10] J.W. Woods. *Subband Image Coding.* Boston:Kluwer Academic Publishers, 1991

[11] M. Vetterli, J. Kovacevic. *Wavelet and Subband Coding.* Englewood Cliffs: Prentice Hall, 1995.

[12] A. Jacquin. "Fractal Image Coding: A Review", *Proceedings IEEE*, vol. 81, no. 10, pp. 1451-65, October 1993.

[13] H.G. Musmann, M. Hotter, J. Ostermann. "Object-Oriented Analysis-Synthesis Coding of Moving Images" Image Communications, vol. 1, no. 2, pp. 117-138, Oct. 1989.

[14] R. Forcheimer, O. Fahlander. "Low bit-rate Coding through Animation", *Proceedings of the Picture Coding Symposium*, pp. 113-114, May 1983.

[15] F. Parke. "Parameterized Models for Facial Animation", IEEE Computer Graphics Applications Magazine, vol. 12, pp. 61-68, November 1982.

[16] K. Aizawa, H. Harashima, T. Saito. "Model-based Synthesis Image Coding System - Modeling a Persons Face and Synthesis of Facial Expressions", *Proceedings of GLOBECOM-87*, pp. 45-49, November 1987.

[17] A. Jacquin, A. Eleftheriadis. "Automatic Face and Facial Feature Location Detection for Low Bit Rate Model-Assisted Coding of Video", submitted to the *Special Issue of Image Communication on Coding Techniques for Very Low Bit Rate Video.*

[18] K. Aizawa, H. Harashima, T.Saito. "Model-based Synthesis Image Coding (MBASIC) System for a Person's Face" *Signal Processing: Image Communication*, vol. 1, no. 2, pp. 139-152, October 1989

[19] J. Hartung. "Constant Frame Rate Scene Adaptive Video Coder for Very Low Bit Rate Applications", AT&T Bell Labs Patent Application, June 1994.

[20] L.G. Chen, personal communication, June 1994.

[21] L. Rabiner, R. Crochiere. *Multirate Digital Signal Processing.* Englewood Cliffs: Prentice-Hall, 1983.

[22] W. Press, S. Teukolsky, W. Vetterling, B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge:Cambridge University Press, 1992.

[23] K.Rao, P.Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications.* Boston: Academic Press Inc, 1990.

[24] "IIT VCP Preliminary Datasheet", IIT Inc., 1994.

[25] "DVC3 Board Users Guide", IIT Inc., 1994.

[26] "IIT Vision Processor", IIT Inc., 1992

[27] S. Haykin. *Digital Communications.* New York: John Wiley and Sons, 1988, pp. 42-44.

[illegible]