Scalable Video Coding in a Reconfigurable Universal Compression System

Trevor Spiteri

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering

Merchant Venturers School of Engineering

September 2012

30000 words

Abstract

This work investigates how scalable video coding can be incorporated as part of a universal reconfigurable compression system. This universal compression system can adapt its architecture for different compression algorithms suitable for different kinds of data, such as general data, image data, and video data. Although these algorithms are fundamentally different, they have common components, which means that savings in hardware complexity, energy and overall costs are possible with the use of reconfiguration and shared blocks.

Scalable video coding is one of the compression modes of the universal compression system, and generates a video bitstream such that quality can be sacrificed for a lower bit rate after a video sequence has already been encoded, without the need of further processing. This work investigates such a scalable video coding algorithm that can exploit the block sharing and the specialization required in the reconfigurable architecture, with the constraints of offering fine-grained scalability and low hardware complexity.

The video coding algorithm makes use of motion vectors, vectors that describe the motion of blocks within a video frame relative to another frame. This work presents the design of an algorithm to encode these motion vectors in a scalable manner using multi-layer motion vector palettes, allowing the video bitstream to be scaled to low bit rates. This scheme is used with a wavelet-based video coding system. The compression performance is analysed and the results obtained compare favourably to the JSVM reference encoder of the state-of-the-art SVC extension to H.264. The proposed scheme provides finer-grained scalability over a wider range of bit rates, and requires less processing and memory.

The suitability of the algorithm for implementation in reconfigurable hardware systems is investigated. This work demonstrates the suitability of the algorithm for encoding high-definition video sequences in real time.

to David, for Bristol

I would like to thank my supervisor, Dr. José Núñez-Yáñez, for his guidance, patience, and insights, my co-supervisor, Prof. Dave Bull, and my colleagues in our research group, mostly George Vafiadis, Xiaolin Chen, Atukem Nabina, and Arash Farhadi Beldachi.

I would also like to acknowledge several free software projects used both during the project development and during the production of this dissertation, mainly GNU Emacs, GCC, Valgrind, GHDL, LaTeX, gnuplot and Inkscape.

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Signed:  Date: **25 September 2012**

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

2D+T two-dimensional spatial filtering followed by temporal filtering.

2D+T+2D two-dimensional spatial filtering followed by temporal filtering followed by two-dimensional spatial filtering.

720p a video format with a resolution of 1280×720.

ASIC application-specific integrated circuit.

ASIP application-specific instruction-set processor.

BAPME backward adaptive pixel-based fast predictive motion estimation [35].

BTC Block Truncation Coding [39].

CIF Common Intermediate Format, a video format with a resolution of 352×288.

CORDIC an efficient algorithm to calculate hyperbolic and trigonometric functions [72].

DCT discrete cosine transform.

DWT discrete wavelet transform.

EBCOT embedded block coding with optimal truncation [11].

EZW embedded zero-tree wavelet [9].

FF flip flop.

FPEU fractional-pel execution unit.

FPGA field-programmable gate array.

fps frames per second.

FSM finite-state machine.

H.264/AVC Advanced video coding for generic audiovisual services [3].

HD high definition video with a resolution of 1920×1080.

ICAP internal configuration access port.

ICER a progressive wavelet image compressor [12].

IDE integrated development environment.

IPEU integer-pel execution unit.

ISA instruction set architecture.

JPEG lossy image compression developed by the Joint Photographic Experts Group.

JPEG 2000 wavelet-based compression system from JPEG [26].

JSVM Joint Scalable Video Model.

LMMIC lossless multi-mode interband image compression.

LPS least probable symbol.

LUT lookup table.

MCTF motion-compensated temporal filtering.

MER Mars Exploration Rover.

MPEG Moving Picture Experts Group.

MPS most probable symbol.

MZ-coder modified Z-coder.

PPMH prediction by partial matching in hardware.

PSNR peak signal-to-noise ratio, given by

$$\text{PSNR} = 10 \log_{10} \frac{\text{maximum}^2}{\text{mean square error}}.$$

QP quantization parameter.

RGB a colour model with red, green and blue components.

RTL register transfer level.

RVC Reconfigurable Video Coding.

SAD sum of absolute differences.

SEU single event upset.

SPIEU sub-pel interpolator execution unit.

SPIHT set partitioning in hierarchical trees [10].

SRAM static random-access memory.

SVC scalable video coding.

T+2D temporal filtering followed by two-dimensional spatial filtering.

VHDL VHSIC hardware description language.

VHSIC very-high-speed integrated circuits.

YUV a way to encode RGB colours using a luma component (Y) and two sub-sampled chroma components (U and V).

Z-coder A binary arithmetic coder [79].

Chapter 1

Introduction

1.1 Aim

The aim of this work is to investigate how scalable video coding can be incorporated as part of a reconfigurable universal compression system.

1.2 Scalable video coding

Scalable video coding (SVC) systems are video coding systems that produce one video bitstream from which several embedded bitstreams can be extracted without the need of further processing. The source video sequence is encoded only once, generating an output video bitstream with the maximum available quality. From this output bitstream, it is possible to extract several subsets that have different qualities, without the need of further coding. This is suitable for providing multiple video bitstreams of different quality, and requires less processing than the alternative which is to encode each of these needed bitstreams independently.

SVC is becoming more popular in recent years, with emerging systems using either hybrid schemes based on current video standards [1] or using approaches based on wavelets for both the spatial and temporal dimensions [2].

The H.264/AVC [3, 4] standard adopted a scalable video extension in its version 8 [3]. There is also a reference implementation of this extension, the Joint Scalable Video Model (JSVM) software for SVC [5]. This scheme is called a hybrid scheme, since it uses predictive coding for the tempo-

ral dimension, that is, to remove redundancy in the time dimension, and transform coding for spatial redundancy.

Since the early 1990s, work on video coding based on motion compensation for wavelets has been present in the literature [6–8]. Wavelet coding has been used for scalable coding of both images [9–12] and video [13–17]. While video coding using wavelets could be hybrid as well, by using predictive coding between frames and the wavelet transform for spatial coding within frames, most wavelet coding schemes use wavelet coding for both the temporal and spatial dimensions.

Most current video coding systems make use of motion estimation. Motion estimation generally involves dividing a video frame into macroblocks, and then, for each macroblock, searching for a similar macroblock in a reference frame that is available to both the encoder and the decoder. The offset of the matching macroblock in the reference frame is known as a motion vector. The motion vectors are encoded as side information to the frame data. At low bit rates, the bit rate allocated to the motion vectors becomes a significant fraction of the total bit rate, so scalable coding has been used for the coding of the motion vectors themselves [18, 19].

1.3 Reconfigurable universal compression system

Reconfigurable hardware has become a popular platform for signal processing. For a long time, there have been reconfigurable hardware systems for applications that are intensive on computation [20–22]. Moving the most intensive parts of algorithms from the general processor to hardware implementations can result in average speedups by a factor of three to five, and in average energy savings of 35% to 70% [23].

As an example of the use of reconfigurable hardware for signal processing and compression, a lossy hyper-spectral image compression system based on SPIHT [10] is presented in [24], and a lossless image compression based on predictive coding is presented in [25]. Both of these works are aimed as space systems such as satellites, where the flexibility of reconfigurable hardware to be modified to use newer algorithms and fix bugs is critical. Another lossy image compression algorithm, EBCOT [11], has become popular recently; the JPEG 2000 [26] standard is based on EBCOT. Hardware systems have been designed to execute the algorithm efficiently [27–29]. Recent work

on image compression systems for space applications uses reconfigurable hardware as well [12, 30]. This shows the increasing use of reconfigurable hardware for digital signal processing applications.

Reconfigurable hardware has been used for compression. This work aims to complement an existing body of work developed within our research group towards a universal compression system. The system includes generic compression [31, 32], lossless predictive image compression [33, 34] and lossless predictive video compression [35]. The lossless video compression system includes a motion estimation engine [36], which can be used by the developed SVC system as well.

It is worth mentioning the MPEG Reconfigurable Video Coding (RVC) standard [37]. This standard provides a framework for different components in a video coding system to be used with other components from other video coding systems. This means that for RVC only video coding is involved. On the other hand, the work presented here is different, since it is aimed to form part of a universal compression system with other kinds of data.

1.4 Objectives

This work aims to provide a hardware-amenable SVC system for use with the existing universal compression system. The main objectives are:

1. to investigate the existing universal compression system, and how an SVC algorithm can be incorporated into it,

2. to investigate the scalable encoding of motion vectors such that they can be included in the video bitstream generated by the scalable video system,

3. to investigate and analyse an SVC algorithm that generates a video bitstream which is scalable through a large range of bit rates, from low bit rates to high bit rates, without the need of reencoding, and that is compatible with the existing universal compression system,

4. to analyse the compression performance of the algorithm to ensure it has good compression performance comparable to other SVC systems,

5. to design for low complexity so that the algorithm is amenable to

hardware implementations, and demonstrate the suitability of the algorithm for hardware implementations.

The presented SVC system is based on wavelets, and is thus scalable through a large range of bit rates. As mentioned in Section 1.2, at low bit rates, the motion vectors themselves need to be encoded in a scalable way. A system to scale the motion vectors by making use of a multi-layer motion vector palette is presented in this work, enabling scalability to low bit rates.

Throughout the design of the proposed SVC scheme, effort was made to keep the complexity as low as possible so that the algorithm is suitable for hardware implementation. The suitability of the algorithm for hardware implementation was demonstrated by means of VHDL implementation and simulation of the main components. The control logic itself is not the most intensive task of the compression scheme, so it can be implemented using either a separate general purpose processor, or a soft-core processor on top of the reconfigurable system itself.

As an example of the use of such an SVC system, consider a space application which has to transmit video. In space applications, it is usual to require very high quality representation of important parts in the camera's view. However, there may be large parts of the image which are not important. Transmission bandwidth is expensive in space applications. To reduce the bandwidth requirements, a scalable compression system can be used, where the data can be compressed retaining all of the original data obtained from the camera. Then, a lower quality subset of the compressed video bitstream can be produced and transmitted using much less bandwidth than would be required for the lossless data. If a part of the video sequence is deemed to be important, a request can be sent back to ask for more data. Using a scalable compression scheme has two major advantages for such a scheme. Firstly, the lower quality subset can be obtained from the lossless video bitstream without the need of further processing. Secondly, data that has already been transmitted does not need to be retransmitted, as the refinement data builds on the lower quality subset that was already transmitted. Processing and bandwidth are both very expensive in space applications, making scalable coding an attractive line to pursue.

1.5 Outline

Chapter 2 includes background material required for the understanding of later chapters. It also presents existing work in the area of image and video compression, and existing work related to the hardware components designed within this project.

Chapter 3 presents the existing reconfigurable universal compression system. The scalable video coding algorithm developed within this project was designed to be compatible with this universal compression system.

Chapter 4 investigates a method to encode the motion vectors produced by the motion estimation engine in a multi-layer motion vector palette. This enables the motion vectors to be encoded in a scalable manner.

Chapter 5 investigates the next steps in the scalable coding of video, which are motion-compensated temporal filtering (MCTF), two-dimensional wavelet transforms, and entropy coding. The chapter includes an analysis of the memory requirements of the system.

In Chapter 6, the performance of the proposed scalable video coding is analysed and compared to Motion JPEG 2000 and to JSVM. The chapter shows that the performance matches, and sometimes beats, the performance of JSVM for high-definition video sequences while the processing and memory requirements are much less than those of JSVM.

Chapter 7 demonstrates that the SVC system is suitable for reconfigurable hardware systems. It presents the design of hardware components to be used with the reconfigurable universal compression system, and includes details of how the hardware cost of some components was reduced.

Finally, Chapter 8 highlights the achievements of the project, and gives some indications for future developments.

Chapter 2

Background and Related Work

This chapter presents the background required for the understanding of later chapters, and describes existing related work. It also points out the limitations in the existing work that this work tries to address.

Section 2.1 introduces image compression systems which form the basis of our video coding scheme. Section 2.2 introduces wavelet coding for both the spatial dimensions and the temporal dimension. Section 2.3 describes progressive coding of images using wavelets, and Section 2.4 describes video coding techniques such as motion estimation and scalable video coding. Section 2.5 describes some existing work related to the design of hardware components of the scalable video components designed in this work.

2.1 Image compression

Image compression can be either lossless or lossy. In lossless compression, the original image is restored exactly bit by bit after decoding. The minimum compressed size is bound by the entropy of the original image. Lossy compression is used when there is no need to retain an exact bit-by-bit copy, and some information can be lost. This does not necessarily mean that the quality will be poor. For example, the original image can be noisy, and if the noise introduced by lossy compression is less than the original noise, the quality will not be affected. Of course, when using lossy compression, some quality can be sacrificed for a further reduction in the size of the compressed

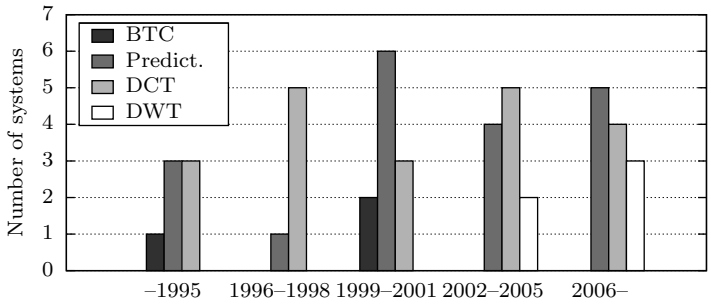


Figure 2.1: Theoretical basis against launch year for image compression in satellite systems [38, Figure 8].

bitstream.

2.1.1 Satellite image compression

An overview of image compression systems used on board satellites is presented in [38]. The systems are separated into systems using Block Truncation Coding (BTC) [39], those that are based on predictive coding, those that are based on the discrete cosine transform (DCT) [40], and those that are based on discrete wavelet transforms (DWTs) [41]. Prediction-based systems are used more widely for lossless compression, while the transform-based systems are used more when lossy compression is required. Figure 2.1 shows what basis is used for compression systems according to the satellite launch year. Prediction based systems were and still are popular, since they are very effective for lossless compression, which is sometimes a requirement. DCT systems, which suffer from blocking artefacts, are losing popularity to wavelet systems, which are gaining popularity due to their very good performance in compressing at low bit rates. Figure 2.2 shows what implementation approach is used for compression systems. ASICs are quite popular, however FPGAs are becoming popular in recent years. This may be because they are cheaper and easier to develop than ASICs, since to develop an ASIC takes a long time and has to be done in large volumes to be economically feasible. However, regular FPGAs are susceptible to single event upsets (SEUs) in space, creating a need for radiation-hardened FPGAs.

One of the aims of this work is to provide a system which is suitable for space applications. This survey justifies the choice of the wavelet transform

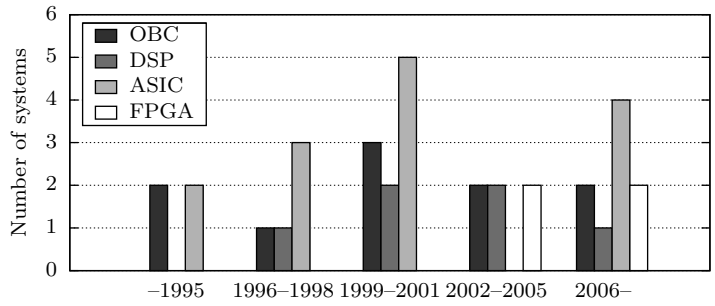


Figure 2.2: Implementation approaches against launch year for image compression in satellite systems [38, Figure 10].

for lossy image compression, and for the selection of an FPGA platform, by showing their increasing use.

2.1.2 Lossy image compression

The most popular techniques for lossy image compression are based on transform coding. The image is transformed using a transform such as the DCT or the wavelet transform. Information is typically lost when quantizing the transformed image.

When JPEG was introduced, wavelet coding was still new, and the DCT was well established. At the time, the performance of wavelets was not better than that of the DCT, so the JPEG committee adopted the DCT for the lossy JPEG standard. However, new wavelet techniques led to the adoption of wavelet-based coding for the newer JPEG 2000 [26] standard [42]. JPEG 2000 is being used by the GEZGIN image compression payload on the Turkish statellite BilSAT-1. It will also be used by the GEZGIN-2 payload for the Turkish RASAT and by the Parallel Processing Unit for Singapore's X-SAT [38]. In 2005, the CCSDS published a recommended standard for image compression that uses the wavelet transform [43]. This standard, which supports both lossy and lossless image compression, has not yet been used in space [38]. Again, these works indicate the growing popularity of wavelet transforms in video coding applications.

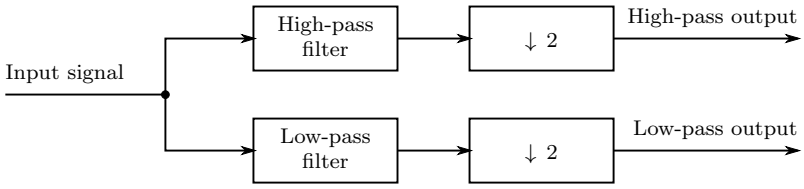


Figure 2.3: One level of one-dimensional wavelet decomposition.

2.2 Wavelet coding

Wavelets are a tool for decomposition of signals into bands of different frequencies [41]. Wavelets can be used for decomposition in both the spatial dimensions and the temporal dimension. The spatial dimensions refer to the two-dimensional space domain of an image. The temporal dimension refers to the time dimension of a video sequence. Wavelet transforms can be obtained by filtering the original signal for as many times as required.

2.2.1 Wavelet filtering

One level of one-dimensional wavelet decomposition is obtained by subjecting the signal to a low-pass filter and a high-pass filter. Since the output of each of the filters has a frequency range that is half the frequency range of the original signal, it is typical to downsample the output of the two filters by a factor of two as shown in Figure 2.3 [41]. The decimation does not lose any information since the output of both filters has a frequency range that is half that of the original signal. If there are n input values, there will be n output values, $n/2$ from the low-pass filter and $n/2$ from the high-pass filter. The filters in the decomposition stage are called analysis filters.

The reverse process is similar. One level of one-dimensional wavelet reconstruction is obtained by upsampling the low-pass signal by two and passing it through a low-pass filter, upsampling the high-pass signal by two and passing it through a high-pass filter, and adding the two resulting signals, as shown in Figure 2.4 [41]. The filters in the reconstruction stage are called synthesis filters.

Multiple levels of the transform can be made by repeating the filtering and decimation on the output of the low-pass filter while leaving the output of the high-pass filter unchanged [41].

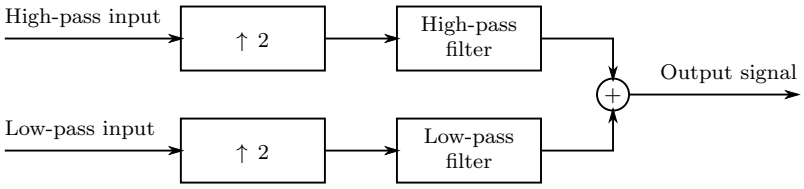


Figure 2.4: One level of one-dimensional wavelet reconstruction.

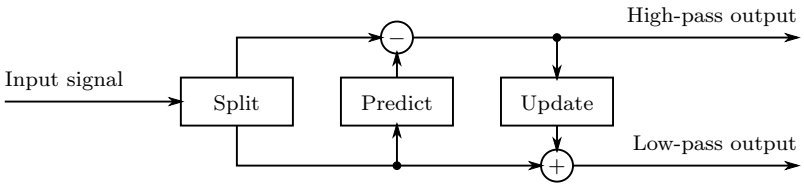


Figure 2.5: One level of wavelet filtering using lifting.

2.2.2 Wavelet filtering using lifting

A fast implementation of the wavelet transform using a technique called lifting is presented in [44]. Lifting is an efficient construct to obtain the wavelet transform which gives the same results as those obtained using the traditional filters shown in Figures 2.3 and 2.4. Figure 2.5 shows how the analysis filter of Figure 2.3 can be implemented using lifting. The Predict and Update blocks in the figure are not stateless. The filtering is performed in two stages, the first stage being the predict step and the second stage being the update step. Decimation is achieved by splitting the input into alternate values before the predict and update steps. Since from every two values one is passed to one branch and the other value is passed to the other branch, there is no need to decimate the output, which means that no values are calculated only to be decimated in the next step.

The high-pass output is available after the prediction step, and the low-pass output is available after the update step. These outputs can be written as

$$h_{2k+1} = x_{2k+1} - \alpha(x_{2k} + x_{2k+2}) \tag{2.1}$$

$$l_{2k} = x_{2k} \quad + \beta(h_{2k-1} + h_{2k+1}), \tag{2.2}$$

where h_{2k+1} are the high-pass outputs and l_{2k} are the low-pass outputs. Notice that the high-pass and low-pass output values alternate for even and

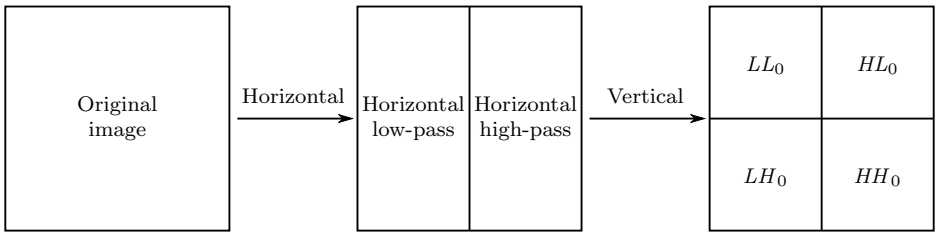


Figure 2.6: One level of two-dimensional wavelet decomposition obtained by a horizontal transform followed by a vertical transform.

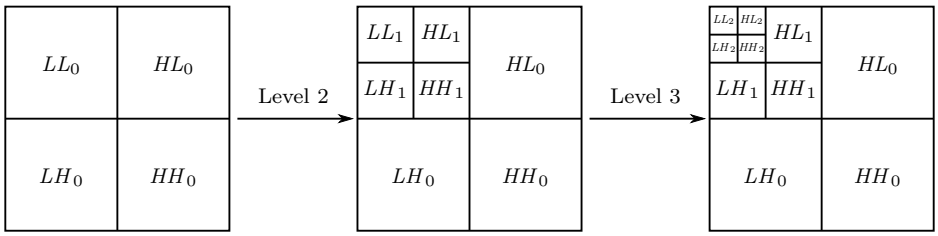


Figure 2.7: Second-level and third-level two-dimensional wavelet decomposition.

odd input values.

2.2.3 Two-dimensional wavelet transforms

For the wavelet transform in the spatial domain, we need a two-dimensional transform. One level of the two-dimensional transform can be obtained by first applying a one-dimensional transform to all the rows of the input signal, and then repeating on all the columns [41]. After doing this, the output consists of four parts as shown in Figure 2.6: (a) LL_0 for horizontal low-pass and vertical low-pass, (b) HL_0 for horizontal high-pass and vertical low-pass, (c) LH_0 for horizontal low-pass and vertical high-pass, and (d) HH_0 for horizontal high-pass and vertical high-pass.

For multiple levels of the two-dimensional transform, the transform is repeated on the horizontal low-pass and vertical low-pass output only, while leaving the other three parts unchanged. Figure 2.7 shows two more levels. In the second level, LL_0 is decomposed into LL_1, HL_1, LH_1 and HH_1, and in the third level, LL_1 is decomposed into LL_2, HL_2, LH_2 and HH_2.

Usually, to perform the wavelet transform, the whole image to be transformed is stored in memory and then the transform is performed on the

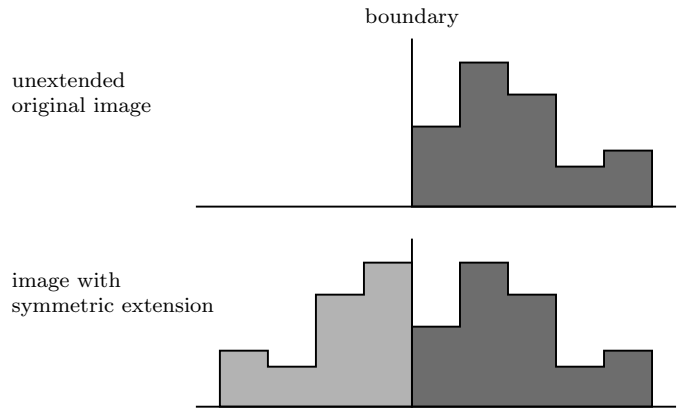


Figure 2.8: An example of symmetric extension at an image boundary.

whole image. A line-based wavelet image compression method that uses less memory is presented in [45]. In line-based wavelet image compression, the horizontal transform is performed in the usual manner. The vertical filtering is performed as soon as there are enough transformed lines. The compression performance can suffer slightly because less information is available to the encoder at any time; whereas usually the whole image is available for analysis by the encoder, when using the line-based transform less lines are available. Synchronization is another issue that has to be given care.

Line-based lifting wavelet transforms are also used in [46] for a hardware architecture for motion JPEG 2000. The scheme includes symmetric extension [47] along the edges. Symmetric extension is commonly used in wavelet transforms since it does not suffer from boundary effects. Figure 2.8 shows an example of a symmetric extension. The values on the left are the extended values which are used during filtering of the actual values to remove boundary effects.

In [46], however, the symmetric extension is only applied in the horizontal direction, where only a few pixels need to be stored to reflect an edge. In the vertical direction no symmetric extension is applied, because whole lines would be required to be stored in memory. Our work aims to avoid this limitation and to find a solution which provides symmetric extension for both the horizontal and vertical direction without a penalty in complexity or memory requirements.

The wavelet transform performance depends on the filter coefficients. In JPEG 2000, there are two sets of filter coefficients which may be used, one

TABLE 2.1: COEFFICIENTS FOR THE DAUBECHIES 9/7 ANALYSIS AND SYNTHESIS FILTERS [49].

	Analysis coefficients		Synthesis coefficients	
n	Low-pass h_n	High-pass g_n	Low-pass h_n	High-pass g_n
0	0.602949	0.557543	0.557543	0.602949
±1	0.266864	−0.295636	0.295636	−0.266864
±2	−0.078223	−0.028772	−0.028772	−0.078223
±3	−0.016864	0.045636	−0.045636	0.016864
±4	0.026749			0.026749

TABLE 2.2: COEFFICIENTS FOR THE LE GALL 5/3 ANALYSIS AND SYNTHESIS FILTERS [50].

	Analysis coefficients		Synthesis coefficients	
n	Low-pass h_n	High-pass g_n	Low-pass h_n	High-pass g_n
0	6/8	2/2	2/2	6/8
±1	2/8	−1/2	1/2	−2/8
±2	−1/8			−1/8

for an irreversible transformation suitable for lossy compression, and one for a reversible transformation suitable for both lossless and lossy compression [48]. The irreversible transform is obtained using the Daubechies 9-tap/7-tap filter [49], and the reversible transform is obtained using the Le Gall 5-tap/3-tap filter [50]. Table 2.1 shows the filter coefficients for the 9/7 filter and Table 2.2 shows the filter coefficients for the 5/3 filter.

As well as being reversible, the 5/3 filter, has the advantage that it can be implemented using shift and add operations in hardware. In hardware, the 9/7 filter would require fixed point multiplication to be implemented. This renders the 5/3 filter more attractive than the 9/7 filter for hardware solutions.

For standard wavelets filters, few wavelet bases can compete with the performance of the 9/7 filter, and it is hard to improve the performance it achieves significantly [51]. For certain image processing applications, such as denoising, wavelet frames with nearly shift-invariant properties show promise [52], but they are expansive, that is, the number of wavelet coefficients produced is larger than the number of input samples [53], which complicates their possible use in compression applications.

Early wavelet coding used sub-band coding techniques, but this had

several problems [42]. The method was not suitable for low bit rate applications. It was also difficult to encode an input to an exact target bit rate. New wavelet coding techniques were then developed, starting with EZW coding published in 1993 [9]. EZW coding uses successive approximation quantization: after the wavelet transform stage, the most significant wavelet transform outputs are encoded first. This requires the position of the most significant coefficients to be encoded, and this is done efficiently using zero-trees, data structures designed for EZW.

2.3 Progressive image coding

EZW has an important property resulting from its successive approximation quantization; it generates an embedded code representation. This means that during coding, if the number of bits generated is the required amount to reach the desired bit rate, the coding process can stop and the coding is complete. For a higher bit rate, the coding process can simply continue until the required number of bits are generated. To generate a lower bit rate from an existing coded image, all that is needed is the truncation of the bits at the end. The same general idea applies to scalable video coding (SVC), which is the video equivalent to progressive coding for images.

In 1996, SPIHT [10], a new implementation based on set partitioning in hierarchical trees, was introduced. SPIHT is an improvement on EZW coding that does not need to transmit the position data explicitly. If the entropy coding stage is omitted, SPIHT shows only a small loss in performance. Similar to EZW, in SPIHT coding the most significant information is coded first, producing a progressive code.

In 2000, the EBCOT [11] algorithm was published. EBCOT is based on independent embedded block coding with optimized truncation of the embedded bitstreams. Apart from the good compression performance, EBCOT produces a bitstream with many features, such as resolution scalability. For progressive coding, EBCOT uses bit plane coding. With bit plane coding, the output values of the wavelet transform are not encoded value by value, but bit by bit. All the bits with a significance of 2^k are in the same bit plane and are encoded together before the bits from the next bit plane with significance 2^{k-1} are encoded. Thus, the bit plane containing the most significant bits is encoded first, and the bit plane containing the least significant

bits, that is, the bit plane with significance $2^0 = 1$, is encoded last. If the bitstream is truncated, only the least significant bits are lost.

Context modelling is tied to bit plane coding in EBCOT. The context of a pixel in the bit plane is determined from the value of the bits of the same pixel and its neighbours in more significant bit planes. EBCOT uses arithmetic coding for the entropy coding stage. The EBCOT algorithm was adopted with modifications as the basis of the JPEG 2000 image compression standard [11].

The Mars Exploration Rover (MER) space mission in 2004 used another image compression algorithm, ICER [12]. In many ways, ICER is similar to EBCOT and JPEG 2000. ICER supports seven different invertible filters for the wavelet transform. It uses a bit plane coding technique based on that used in EBCOT. For entropy coding, ICER uses interleaved entropy coding [54].

The standard recommended by the CCSDS for image compression [43] supports progressive coding as well. It uses a three-level two-dimensional wavelet transform with a 9/7 filter, followed by a bit plane encoder. The simple entropy coder in the standard uses variable-length binary codes. This standard is aimed at high-rate instruments such as those on board spacecraft, aiming for low complexity to allow fast and low-power hardware implementation.

Our work takes inspiration from EBCOT and ICER for the spatial compression component. This is because ICER is a good fit for our requirements, since one of its original goals was to provide image compression for space applications and so it has low complexity.

2.3.1 Bit plane coding and context modelling

We have already mentioned that in EBCOT, the outputs values of the wavelet transform are not encoded one value at a time, but one bit plane at a time. If the value with the largest amplitude in one code block has K significant bits, there are K bit planes to encode. The most significant bit plane contains all the bits with significance 2^{K-1} from all the values in the code block. The least significance bit plane contains all the bits with significance $2^0 = 1$ from all the values in the code block.

Before encoding a bit plane, the plane is divided into sub-blocks and the significance of each sub-block is encoded. A sub-block is significant if any of

its pixels is significant. If an eight-bit pixel has the binary value `00010010`, it is not significant for the three most significant bits, then, starting from the first `1` it becomes significant, and remains significant for all the five least significant bits.

The significance of the code block is encoded in a hierarchical manner. For example, if the code block contains 256×256 pixels, it is first split into four sub-blocks of 128×128 pixels each, and the significance of each sub-block is encoded. If a sub-block is not significant, the process ends there, but if a sub-block is significant, it is divided into four further sub-blocks of 64×64 pixels each and the significance of each sub-block is encoded. This is repeated down to sub-blocks of 16×16 pixels each, for a maximum of $4 + 16 + 64 + 256 = 340$ bits.

For the next bit plane, this process will be repeated, but if a sub-block was significant in a previous bit plane, it is still significant, so its significance does not need to be reencoded.

We have seen that when a sub-block has just become significant, it is divided into four sub-blocks and the significance of each is encoded. If the first three are not significant, then the fourth must be significant, and does not need to be encoded.

In all, there are three cases where significance is not encoded:

1. If the parent sub-block is not significant, then the sub-block cannot be significant.

2. If the sub-block was already encoded as significant in a previous bit plane, then it is still significant.

3. If the parent sub-block is significant, and the three sibling sub-blocks are not significant, then the sub-block must be significant.

The bit plane coding and significance coding described above is from EBCOT. EBCOT also has a context modelling scheme, which is slightly modified in ICER. In our work, we use the bit plane coding and significance coding of EBCOT, and the context modelling of ICER, which will be described now.

The context modelling makes use of the significance of the pixel being encoded and of its neighbouring pixels. Each pixel is given a category, which can take a value from 0 to 3. If a pixel is not significant, it has category 0.

If a pixel has just become significant, its category remains 0, then becomes 1 for the next bit plane, 2 for the bit plane after that, and 3 for all the bit planes after that. Thus, if a pixel has the binary value `00010010`, it has category 0 for the three most significant bit planes. The fourth bit plane is encoded using category 0, but since the fourth bit is a `1`, the category will be changed to 1. The fifth bit is encoded using category 1, the sixth bit is encoded using category 2, and all the remaining bits are encoded using category 3.

The most important category to encode well is category 0, and it has nine contexts assigned to it. The context depends on which of its neighbouring pixels are significant at the time of encoding. Category 1 has two contexts, and category 2 has one context. The bits in category 3 are nearly incompressible, so they are not encoded using arithmetic coding and need no contexts [12].

Other than the twelve contexts described above, there are five contexts for the sign bit. The sign bit is encoded just after the first significant bit of a pixel is encoded. For our number `00010010`, in the fourth bit plane, the `1` bit is encoded using one of the contexts for category 0, then the sign bit (which is not shown in the representation) is encoded in one of the contexts for the sign bit. The context for the sign bit depends on the significance of the neighbouring pixels, and to the sign of the significant neighbouring pixels.

2.4 Video compression

Video compression has some similarities to image compression, as each frame in a video sequence is an image. But video has another dimension, the temporal dimension. To compress video effectively, we need to cater for temporal redundancy as well as spatial redundancy.

Modern coding standards such as VC-1, AVS and H.264/AVC [3, 4] use motion estimation and compensation to compress across frames. In H.264/AVC, a frame is split into macroblocks, and for each macroblock the best match is found from reference frames. Once a match is found, the difference between it and the macroblock is found, and this residual is encoded instead of the macroblock. Since the residual usually has pixels with smaller amplitudes than those of the original macroblock, it can be encoded

using fewer bits. The motion vector, that is, the coordinates of the matching block, has to be encoded as side information.

2.4.1 Motion estimation

Motion estimation is the process of searching for a matching macroblock in a reference frame. It is called motion estimation since in effect it is estimating the motion of objects from one frame to another. The relative offset of a macroblock in the reference frame is called a motion vector.

Previous work [55] shows that motion estimation is the most expensive operation in the H.264/AVC encoder, representing up to 90% of the total complexity. This indicates that to design a video coding system where complexity is an important consideration, it is important to consider the motion estimation search.

Full search motion estimation algorithms have gained popularity in hardware implementations owing to their regularities, which make it possible to implement motion estimation hardware using systolic arrays [56]. Many other approaches, such as the hexagonal search algorithm [57] and the unsymmetrical multi-hexagonal search algorithm [58], do not perform an exhaustive search on full point regions. The use of these block-matching algorithms can make the estimation process faster by requiring less computations than a full search. Although the full-search algorithm is usually believed to yield optimal rate-distortion performance, it has been shown that a well-designed fast block-matching algorithm can provide better rate-distortion performance owing to its ability to track real motion more accurately [59].

There are various hardware implementations of motion estimation algorithms. Processors with instruction set architectures (ISAs) tailored for block-matching search algorithms, are presented in [60] and [61]. Xilinx have a motion estimation engine [62] that computes the sum of absolute differences (SAD) for a set of 120 search locations within a 112×128 search window in parallel. None of these cores offer the possibility of matching the hardware architecture and the search algorithm to optimize performance. In this work, we use a motion estimation engine developed within our research group that is an application-specific instruction-set processor (ASIP) [63]. This engine can be tuned to a particular application domain so that it uses the least possible hardware cost.

The motion estimation process can be performed in various different

ways, and it is up to the designer to choose the strategy. Other than the search strategy itself, other choices include whether to use multiple motion vector candidates in the search, the number of reference frames to which to compare the macroblocks, whether the macroblocks are split into partitions, whether to perform sub-pixel interpolation and search, and whether to include the cost of encoding the motion vector itself during estimation. Because of the number of design parameters and their complexity, the design space can be very large, and exploring this design space to find design parameters that are optimal can be complex and ultimately application dependent.

As part of this work, tools were developed for the configuration of the motion estimation processor. A toolset was developed for the optimization and generation of configuration data for a high-performance motion estimation processor. The toolset makes the process of finding the optimal hardware configuration and software parameters faster. A cycle-accurate simulator is included, making it possible to change the parameters and test the configuration in a short time without requiring hardware access.

The idea of configurable processors itself is not new, there is also similar work on configurable generic processors, like the Xtensa configurable processor [64] from Tensilica. Designers can choose configuration parameters and generate a custom processor optimized for their needs. The options include support for 16×16-bit multiplication, a floating point unit, a barrel shifter, zero overhead looping, and others. Tensilica also provides the XPRES compiler, a tool for design space exploration.

2.4.2 Scalable video compression

Progressive image coding, which was presented in Section 2.3, provides a bitstream that can be used to generate a lower quality bitstream by simply using only a subset of the bits. Scalable video compression is the counterpart of progressive image coding for video coding.

In some scalable video coding algorithms, the encoder keeps track of the bitstream being decoded by the decoder. Different decoders can receive different subsets of the same bitstream, and will thus have a different state, so the encoder has to maintain multiple motion compensation loops to accommodate all the different decoders. This makes the encoder complex, and limits the number of possible output scales [65].

As an alternative to this strategy, motion adaptive temporal wavelet transforms are presented as a means to deal with temporal redundancy in [65]. Using a 5/3 wavelet kernel for temporal filtering is shown to have better compression performance than using the Haar wavelet or a 1/3 wavelet kernel, which are the most basic wavelet filters.

Scalable quantization after the temporal and spatial transforms can be obtained using embedded block coding as in EBCOT, which was presented in Section 2.3.

2.4.3 Hybrid video coding

Hybrid video coding schemes are called hybrid because they use different techniques for the spatial and temporal dimensions, typically predictive coding in the temporal dimension and transform coding in the spatial dimensions.

The SVC extension to the H.264/AVC video compression standard is such a hybrid video coding scheme [1]. Scalability is achieved on three different levels, temporal scalability, spatial scalability and quality scalability. Temporal scalability is achieved using hierarchical prediction structures as in H.264/AVC. Frames are not encoded in chronological order. The frames in the temporal base layer are encoded first, then the frames in the next layers, which may be interleaved within the frames in upper layers. This does not add complexity to H.264/AVC, which already uses hierarchical temporal prediction structures. But H.264/AVC is already very complex, especially for low complexity systems like space applications which have more stringent complexity constraints than other platforms.

For spatial scalability, the lowest resolution layer is encoded first, then the higher resolution layers. Macroblocks in a layer can be predicted using matching blocks in the same layer, or matching blocks in layers with a lower resolution. For a block in a lower resolution to be used in prediction, it has to be scaled up. Also, for a block in a lower resolution to be usable, it has to be decodable without needing access to a block with yet a lower resolution. This last condition is to limit the decoder complexity, limiting the amount of computation required to decode one macroblock.

Our work uses wavelet transforms in both the temporal and spatial domain. As shown in Figure 2.7, the two-dimensional wavelet transform has spatial scalability intrinsically. If the higher-level bands are discarded, the

image is automatically scaled down in the spatial dimensions. The same can be done in the temporal dimension.

The SVC extension for H.264/AVC also supports quality scaling, with a number of scales available, with low quality layers being encoded first followed by quality refinement layers. This has to be done carefully, as discarding quality refinement packets will cause drift between the motion compensation loops of the encoder and the decoder. To leave the refinement packets totally out of the motion compensation loop would result in a loss in compression performance. The concept of key pictures is introduced in SVC to control the drift, where some key frames do not depend on any refinement packets to be decoded. These key pictures help to bring drift back down whenever they occur in the bitstream. It is worth noting that the quality scalability in SVC for H.264/AVC is limited to a fixed number of possible output scales, that is, it is not finely-scalable. Our work uses schemes similar to EBCOT for quality scalability across different frames and does not suffer from drift, so there is no need for complexity to handle drift.

One of the objectives of our work is to provide a simpler algorithm that is less complex than H.264/AVC. Also, SVC for H.264/AVC achieves coarse-grained scalability using multiple layers, the quality of which is specified in the encoding process. The transform coefficients in each layer can then be further split into several quality layers, providing medium-grained scalability. Our work aims to offer fine-grained scalability, such that no quality specification is required during the encoding process, that is, the encoded bitstream will have a quality ranging from the lowest to the highest obtainable quality.

2.4.4 Systems based on wavelets

Hybrid video coding schemes use different strategies for dealing with temporal and spatial redundancy, most commonly predictive coding for temporal redundancy and transform coding for spatial redundancy. Systems based on wavelets use the same strategy, specifically wavelet transform coding, for both temporal and spatial redundancy.

Wavelet-based SVC systems can have one of several architectures for motion compensation. The most popular is the T+2D architecture, which stands for temporal + two-dimensional spatial, where motion-compensated temporal filtering (MCTF) precedes the two-dimensional spatial DWT. This

system is the simplest and most intuitive, with its drawback being difficult spatial scalability.

Some systems use the 2D+T architecture, where the two-dimensional spatial transform is done first, but this makes MCTF difficult because of the shift variant nature of the two-dimensional DWT. A proposed solution for this is to transform the complete DWT to an over-complete DWT [66], but the complexity and memory requirements increase significantly with the number of spatial wavelet decomposition levels. Even so, the coding efficiency of 2D+T systems is worse than that of T+2D systems, especially at high resolutions [2]. There are also adaptive architectures that propose to select 2D+T or T+2D according to the content, and multi-scale pyramid architectures, sometimes referred to as 2D+T+2D, that try to combine the two [2]. The system proposed in this paper is based on the T+2D architecture because of the complexity of the other architectures.

Early MCTF work used the Haar filter [6–8,13,14] for filtering in the temporal dimension. Later works demonstrate the advantages of using wavelet filters with longer kernels [15–17], mainly the 5/3 filter. The system presented in this dissertation uses the 5/3 filter for its MCTF stage.

Wavelet filtering using lifting makes use of a prediction step and an update step as described in Section 2.2.2. For temporal filtering, we can treat frames in a similar manner to the way we treat values in (2.1) and (2.2). If the input frames are $F_0,\ F_1,\ F_2,\ \ldots$, then the prediction step and update step can be written as

$$H_{2k+1} = F_{2k+1} - \alpha(F_{2k} + F_{2k+2}) \tag{2.3}$$

$$L_{2k} = F_{2k} \quad + \beta(H_{2k-1} + H_{2k+1}). \tag{2.4}$$

These two equations are not catering for motion compensation yet. For the prediction step (2.3), the pixels in F_{2k} and F_{2k+2} have to be mapped to their matching pixels in F_{2k+1}. For the update step (2.4), H_{2k-1} and H_{2k+1} from the prediction steps have to be mapped back to match the pixels in F_{2k}.

In MCTF, a motion-estimation search is typically performed to generate motion vectors that are used for motion compensation in the prediction step. The reverse of these motion vectors are then used in the update step. Since multiple macroblocks may be matched to the same macroblock in the reference frame, motion estimation yields a many-to-one mapping, and this

has to be taken into account when reversing the motion vectors for the update step. In [16], when there are more than two pixels in a prediction frame H_{2k-1} that are mapped onto the same pixel in a frame F_{2k}, only one of the pixels is used, and the others are discarded. In [17], the motion compensation in both the prediction and update steps is modified so that for one pixel, instead of using one other pixel from each adjacent frame, multiple pixels are used with a weighting function, in a technique named barbell lifting. In [15], instead of using motion vectors, motion compensation is done by warping the frames to account for the motion; this method assumes a unique motion trajectory and so has no pixel collisions. Our work aims to use a more accurate value than the single value used in [16], while avoiding the complexity of the systems in [17] and [15].

2.4.5 Scalable motion vector coding

At high bit rates, the bit rate taken by the motion vectors themselves is insignificant. At low bit rates, however, the bit rate taken by the motion vectors becomes significant. In [18], motion vectors are split into several layers for motion vector scalability. A motion estimation search is performed for each layer, and in each refinement layer, the motion vectors from the previous layer are used as motion vector candidates in the motion estimation search. This system has the disadvantages that motion-estimation searches have to be performed more than once, and each refinement layer will only use the motion vectors from previous layers as motion vector candidates. When a new motion estimation search is performed, the starting point can be the motion vector from the previous layer, but a complete search is still performed. Recall that motion estimation is one of the most time consuming parts in typical video coding algorithms, so this strategy is not very attractive when low complexity is a goal.

The work in [65] and [19] achieves scalable motion vector encoding without requiring multiple motion estimation passes. Our work takes this route to avoid the need for multiple motion estimation searches, which are very time consuming. In [65] and [19], scalable motion vector encoding is achieved by quantizing the motion vectors themselves. The motion vectors are transformed along the temporal domain and in the spatial domain using reversible DWTs, and then the coefficients are encoded using bit plane coding in the same way as image data is encoded in EBCOT. The errors introduced by

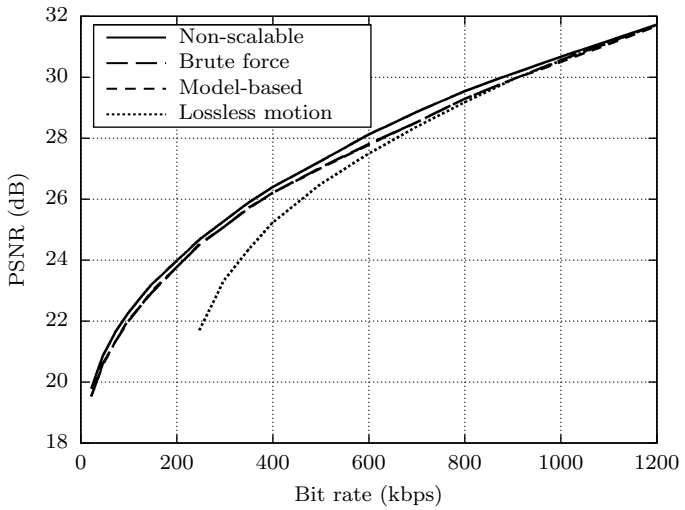


Figure 2.9: Rate-distortion performance of various rate-allocation strategies for motion vectors [19, Figure 7].

quantizing the motion parameters are analysed, and a framework is presented that scales both the motion parameters and the samples obtained from the wavelet transforms jointly. Two techniques to distribute the bits between motion parameters and samples are presented: a brute force search method, and a model-based rate allocation strategy. Figure 2.9 shows a typical result from the presented experiments. The non-scalable curve shows the achievable quality if no scalability is used, that is, for each point in the curve, the encoder had to be run. The lossless motion curve, on the other hand, retains all the motion vector information and only scales the image data. This shows the problem we are trying to solve very well: at low bit rates, the bit rate used by the motion vectors themselves becomes the major part of the total bit rate, and motion vectors alone do not provide a signal, so the PSNR suffers. The other two curves show a scalable system where the motion vectors are scaled together with the image data. The brute force method produces a slightly better output than the model-based method, but requires more computation. Both methods are better than using lossless encoding of the motion parameters and lossy encoding of the samples, especially at low bit rates.

The schemes just seen treat the motion vectors as if they were images. Our algorithm aims to achieve better performance by not treating the motion

vectors as images, but as vectors, which should make it possible to tailor the algorithm specifically to the required use case. This can be achieved by encoding motion vectors using a motion vector palette. This makes use of the fact that many motion vectors will have similar values; if a large object is moving, the macroblocks showing the object would have similar values, and can be mapped onto the same motion vectors in the motion vector palette. A divisive clustering technique can be used to split the set of motion vectors into clusters, which in the end are used to generate a palette of motion vectors. This palette can then be split into layers.

Palettes of motion vectors have already been proposed for motion vector coding in [67], but that scheme is not aimed for scalable coding of the motion vectors, and no layers for scalable encoding are used, such that each set of motion vectors is encoded independently. Our work aims to overcome this limitation and use multi-layered motion vector palettes. In the designed multi-layer coding scheme, each refinement layer for the motion vector encoding would add to the number of possible values in the palette, that is, each layer would enlarge the motion vector palette.

In [67] motion vectors are split into clusters using a popularity algorithm, that is, counting which motion vectors are more popular. This does not lend itself well to splitting the final palette into layers. This work investigates the use of a divisive clustering technique, where each cluster division can be recorded and the information used later when splitting the final palette into layers.

A divisive clustering technique that is suitable for adoption by such a scheme is presented in [68]. The work in [68] is targeted at colour quantization with palettes, while in this work palettes would be used for motion vector quantization. Also, the divisive clustering technique would have to be adapted to support multi-layered coding.

Once the palette of possible motion vector values is selected, the palette and motion vectors would need to be encoded. In the refinement layers for the motion vector palettes, rather than encoding the motion vector values themselves, the difference in the motion vector values from the values for a lesser-quality layer are encoded.

Generating the palette is only the first step. After the palette generation and encoding, the motion vectors have to be encoded as index values pointing to the motion vector values in the palette. A method to encode such index

values is presented in [69], where colour pixels are encoded for palette images using two-dimensional context models. The work in [69] uses 512 different contexts for this encoding process. This number of contexts can be too high for this work, so the use of a smaller number of contexts is investigated in this work. A smaller number of contexts may be required for the adaptive encoder to settle earlier, as the number of macroblocks for a typical frame is much smaller than the number of pixels in a typical image, and using too many contexts means that the contexts take longer to adapt, making the use of a large number of contexts counter-productive. Also, the method would need modifications to support the encoding of the values in layers for scalable encoding.

2.5 Hardware amenability

The scalable video compression system presented in this work can be divided into two parts: motion-compensated temporal filtering, and image compression of the temporally filtered frames. Temporal filtering removes the temporal redundancy between frames, and then the temporally filtered frames are individually encoded using an image compression algorithm based on wavelets. Hardware systems for image compression based on predictive coding, such as [25] and [30], are not suitable for scalable video compression, as predictive coding is only suitable for lossless compression.

There are hardware systems, such as [27], for progressive image compression, which would be suitable for scalable video coding. The ICER image compression algorithm is hardware amenable, as it was designed for reconfigurable hardware implementation for space missions. The image compression subsystem used in this work is very similar to ICER. Although it is similar to ICER, the system presented in this dissertation has some differences. It is designed in such a way that it can be used as one component of a system which is dynamically reconfigurable to handle different kinds of data, with the use of other components that handle generic data [31], lossless image and video compression [34, 35], and motion estimation [36].

In this work, we present hardware components that include the encoding of motion vectors using palettes. The motion vector palette encoding algorithm makes use of divisive clustering, which in turn requires the computation of eigenvalues. A system for hierarchical K-means clustering system

is presented in [70]. While that system is flexible and suitable for a range of problems, it is quite complex. This work investigates the use of a more specialized clustering technique that is sufficient for motion vector palette encoding at a lower hardware cost, since there is no need for the flexibility provided by [70].

As mentioned above, the clustering algorithm requires the calculation of eigenvalues. A generic system for the calculation of eigenvalues is presented in [71], but again, the generic nature of the system makes complexity an issue, and it uses two CORDIC [72] units to calculate the eigenvalues. In this dissertation, a hardware component tailored to the specific algorithm requirements will be presented. Since it does not need to be generic, it can have a lower hardware cost; for example, the operations are pipelined in a way that only one CORDIC unit is required.

The scalable image compression component of this paper is based on the discrete wavelet transform with lifting, similar to the system described in [46]. In [46], the wavelet transform uses symmetric extension at the edges in the horizontal direction, but not in the vertical direction, in order to avoid buffering extra whole lines. In the hardware design for our work, symmetric extension at the edges in the vertical direction is achieved without the need of extra buffering.

2.6 Conclusion

This chapter presented existing work in image and video coding and in scalable video coding, as well as hardware components related to the scalable video coding algorithm investigated in this work. The areas in which this work aims to provide improvements were pointed out.

Chapter 3

Reconfigurable Universal Compression

This chapter presents a reconfigurable universal compression system developed within our research group. The various components of the universal compression system, and how they work together, are presented. This chapter also indicates where the new components of this work fit in the universal compression system.

Section 3.1 describes how dynamic reconfiguration can be used in a universal compression system. Section 3.2 introduces the different stages in the universal compression system, and which stages can be shared for multiple kinds of data. Section 3.3 talks about some kinds of data that can be compressed using this system. Section 3.4 describes the architecture of the reconfigurable universal compression system, including the scalable video coding components which are proposed in this work. Section 3.5 gives a brief overview of the different stages in the proposed scalable video coding scheme.

3.1 Dynamic reconfiguration

To enable compression of different kinds of data without the need to reconfigure the whole system for each kind of data, the compression process is partitioned into multiple stages, some of which are common to all kinds of data.

One reconfigurable device can be used to encode different kinds of data.

Some parts are common to different kinds of data, and some other parts can be reconfigured for the specific kind of data that is being compressed at a particular time. The reconfigurable parts can be reconfigured dynamically on modern FPGAs.

On Xilinx dynamically reconfigurable FPGAs, the configuration bitstream of a part of the FPGA can be changed using the internal configuration access port (ICAP), while the rest continues to operate [73]. The ICAP enables these devices to reconfigure themselves without the need of an external configuration controller.

Reconfiguring the FPGA dynamically introduces some configuration delay. This means that the number of reconfigurations should be kept as low as possible. If possible, data streams of the same kind should be compressed one after the other, without a different kind of data in between, to reduce the number of reconfigurations [74, 75]. The reconfiguration bitstream can be optimized using bitstream compression techniques developed within our research group and presented in [76].

This dissertation presents a scalable video coding (SVC) algorithm that reuses components of the existing universal compression system. For example, the motion estimation engine required by the SVC is already a part of the universal compression system when used for lossless video coding, reducing the amount of reconfiguration required to go from the lossless video mode to the scalable video mode.

3.2 Statistical compression in three stages

The universal compression process can be partitioned into three stages [32]:

1. context data modelling,
2. probability estimation, and
3. arithmetic coding.

Some kinds of data may require some preprocessing to prepare them for these stages. The preprocessing stage can be considered as a part of the context modelling for the scope of these three stages.

Figure 3.1 shows an overview of the universal lossless compression system described in [32]. The first stage, context data modelling, is a dynamically

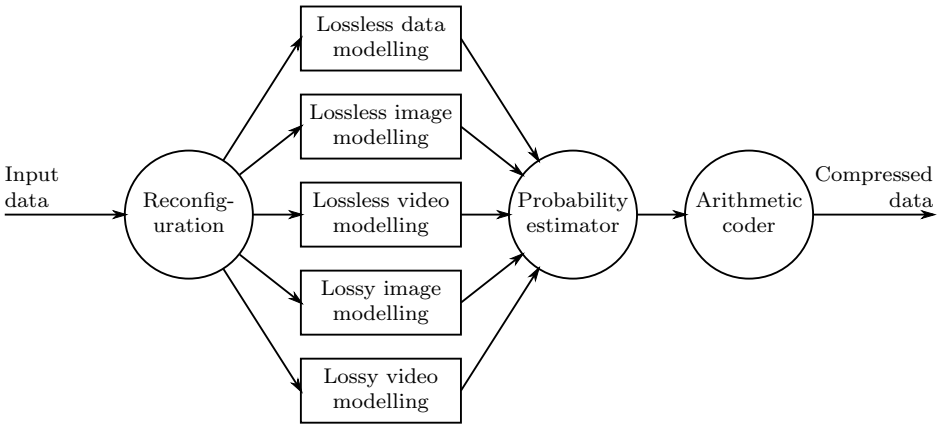


Figure 3.1: Overview of universal lossless compression system.

reconfigurable stage: a different configuration will be used for each kind of data. The second and third stages are statically configured, and are common to multiple kinds of data.

In arithmetic coding [77], a message is encoded as a probability interval. In the beginning of the process, the interval is $[0, 1)$. Each time a symbol is to be encoded, the current interval is split into subintervals, one for each possible symbol, and the subinterval corresponding to the symbol to encode is selected as the new current interval.

For memoryless sources, the symbol probabilities of the symbols are the same throughout the message. However, most information sources are not memoryless, that is, the symbol probabilities often depend on the previously encoded symbols. We can say that the symbol probabilities depend on the context in which the symbols are.

The job of the first stage, context modelling, is to determine which context to use to encode a symbol. From the message symbols that have already been encoded, a context is identified. Each context will hold information on how probable each symbol is, and the second stage, probability estimation, splits the probability interval into subintervals. The third stage, arithmetic coding, encodes the probability interval into a number of bits.

3.2.1 Context modelling

Context modelling is the process of identifying a context for the symbol to be encoded. Each context has its own set of symbol probabilities. When

a symbol is to be encoded, the context is identified using only the symbols preceding it. The symbol itself may not be used in determining the context, otherwise the decoder will have less information than the encoder, and will not be able to identify the context.

The PPMH [31] algorithm is a generic compression algorithm that is suitable for hardware implementation. This algorithm can be used to compress generic data using the three stages above. In the context modelling stage, the n preceding symbols are used in an n-order model. These symbols are used to identify a context to encode a symbol. This is achieved using a tree structure in memory, where each tree node corresponds to a context.

As another example, when compressing two-dimensional images, the value of a pixel will depend on all previous pixels which are in the vicinity of the pixel to compress. These will not necessarily be the previous n encoded pixels. LMMIC [34] is a predictive lossless scheme for compressing images. In this scheme, the context is identified using a number of symbols in the vicinity of the symbol to be encoded. Unlike the PPMH scheme, which uses only the symbol values in identifying a context, the LMMIC scheme does not identify a context using only pixel values. The context is chosen using various techniques, including image segmentation, run length coding, and image gradients.

It is worth noting that the algorithms used for context modelling can be very different. For example, the context modeller for PPMH uses a tree structure to store the different contexts, while the context modeller for LMMIC does not.

3.2.2 Probability estimation

An implementation of the probability estimation stage is presented in [31]. Once the context modelling stage identifies a context for a particular symbol, the probability estimator will use a memory area associated with the context. The estimator uses the symbol to determine the subinterval to pass to the arithmetic coder. In an adaptive system, the estimator also has to update the context information with the symbol, so that if a symbol is very common and needs to be encoded multiple times, the number of bits required to encode it will become smaller every time.

For each context, the probability estimator uses a balanced binary tree to store the symbols of the alphabet and their frequency counts. An additional

symbol, the escape symbol, is used when the symbol being encoded has a frequency count of zero in the tree. If the alphabet size, that is, the number of different possible symbols that can be encoded, is 2^k, the depth of the required binary tree is $k + 1$. To fully encode a symbol using this binary tree, it is enough to encode one binary decisions at each level of the tree. This means that after at most $k + 1$ binary decisions, the symbol is fully encoded.

Using the binary tree of depth $k + 1$ from [31] to encode a symbol in a 2^k-size alphabet has two main advantages. Firstly, the arithmetic coding stage does not need to be a complex multi-bit symbol arithmetic coder; a binary arithmetic coder is sufficient. Secondly, updating the frequency counts is achieved with a single update operation for each visited node while traversing the tree. Traditional arithmetic coders maintain frequency counts in a cumulative form. In cumulative form, updating frequency counts for symbols at the bottom of the range affects the cumulative values of all symbols higher in the range. The presented tree requires only one update for each tree node visited, so a constant cycle count is obtained when the design is moved to hardware.

The proposed SVC scheme makes use of bit plane coding. A probability context will be found for every bit in each bit plane. Since the scheme works on bits rather than k-bit symbols, it does not need the tree required for a k-bit probability estimator, and can use a simpler one-bit probability estimator.

3.2.3 Arithmetic coding

The final stage of the PPMH algorithm mentioned in Section 3.2.1 is the arithmetic coder. A suitable arithmetic coding engine is described in detail in [78].

As described in Section 3.2.2, the arithmetic coder does not need to be a complex multi-bit symbol coder; only a binary arithmetic coder is required. The algorithm used is based on the Z-coder [79], which is a generalization of the Golomb-Rice coder [80] for lossless compression. The Golomb run-length coder for binary symbols has some limitations which make it unsuitable for adaptive codes; it is not suitable to encode events when the event probability changes within the message. The Z-coder adapts Golomb coding to provide a coding system that can be used for adaptive entropy coding of binary

symbols.

The Z-coder itself is further modified to be hardware amenable in [78]; the modified algorithm is the MZ-coder. The MZ-coder balances the complexity of coding the most probable symbol (MPS) and the least probable symbol (LPS). This balancing is useful since the reconfigurable hardware implementation used encodes one binary symbol per clock cycle. Having low complexity for the MPSs at the cost of higher complexity for the LPSs can be a good idea for a program running on a traditional processor, but it can lead to higher complexity in the slowest data path on an FPGA, leading to a lower clock frequency. The MZ-coder also simplifies the precision of the arithmetic and allows for a fully pipelined architecture. The proposed work uses this encoder since it fits well with the requirement of a binary arithmetic encoder.

3.3 Different kinds of data

The ability to dynamically reconfigure a universal compression system is desirable, but its use would be limited if the compression performance suffered in order to have reconfiguration capabilities. For the scheme described in Section 3.2 to be attractive, the compression performance needs to be comparable to the performance for stand-alone compression algorithms. To demonstrate that compression performance does not need to be sacrificed, this section gives details about existing lossless image and video coding algorithms within this reconfigurable framework.

3.3.1 Images

The LMMIC scheme presented in [34] is a lossless image compression scheme developed within our group. It uses the three-stage architecture, models images using multiple modes. Once the mode and context are determined from the image, the probability estimation and arithmetic coding stages will compress the data. The compression performance of this algorithm is compared to the performance of other lossless image compression schemes in [33]. The algorithm is aimed at high-speed space applications, so relevant test data was used. The system outperforms other schemes in terms of bits required per pixel, and processes one bit per clock cycle. This translates to a throughput of 100 Mbit/sec on a Xilinx Virtex-4 SX35 FPGA.

The performance of the LMMIC scheme demonstrates that it is possible to design algorithms that fit the three-stage architecture of Section 3.2 and still obtain a compression performance comparable to the best available techniques.

3.3.2 Video

In the same way that LMMIC compresses images using the three-stage architecture, there are video coding schemes that use such an approach in stages.

A lossless video compression algorithm that uses adaptive prediction is presented in [81]. The algorithm uses both spatial and temporal prediction. The prediction stage is followed by a context-based arithmetic coder. This algorithm could be fitted into the reconfigurable three-stage architecture; the preprocessing and context identification sections can become the reconfigurable context modelling stage, while the static probability estimator and arithmetic coder can be used for the final stages of the algorithm. The presented experimental results demonstrate that this algorithm performs well compared to other available lossless video compression schemes.

Another lossless video compression scheme, which was developed in our research group, is BAPME [35]. The BAPME lossless video compression scheme is built on top of the LMMIC lossless image compression scheme, with motion estimation techniques similar to [81], and gives good results as well. Once again, this demonstrates that using the three-stage architecture for compression does not result in a loss in compression performance.

The scheme of [81] performs lossless video coding without encoding any motion vectors to reduce the amount of side information required. The motion compensation stage of this scheme performs a motion estimation search for each individual pixel instead of trying to find a match for a whole macroblock. The search is performed using a window of pixels close to the pixel that is being processed. Suppose that the pixel $p_i(x, y)$ in frame i is being processed. A corresponding pixel $p_{i-1}(x+m, y+n)$ in frame $i-1$ needs to be found. To find m and n, a window of neighbouring pixels $w_i(x, y)$ that have already been encoded and that are close to the pixel $p_i(x, y)$ is used. A match $w_{i-1}(x+m, y+n)$ for this window is found in the reference frame, giving us the values of m and n. Then the value of $p_{i-1}(x+m, y+n)$ is used in predicting the value of $p_i(x, y)$. When the decoder is trying to decode

$p_i(x, y)$, it will have the values of frame $i-1$ and of $w_i(x, y)$, so the decoder can perform the same search and get the value of the pixel $p_{i-1}(x+m, y+n)$. The decoder does not need the values m and n as side information, since it can repeat the motion estimation search itself.

This technique is used in BAPME, which is a lossless video coding scheme, as well. This work investigates whether this behaviour is suitable for SVC.

The motion estimation search in the algorithm by [81] is an exhaustive search. In BAPME, a fast motion estimation search is used for motion estimation in pixel-based predictive lossless video compression. A fast diamond search technique is used to find the matching window in the reference frame. Since our algorithm is in the same reconfigurable system as BAPME, we reuse the same fast motion estimation search that is used by BAPME, which gives a speed improvement over full search algorithms without loss in compression performance.

Whereas the two schemes described above are for lossless video coding, the scheme presented in this work is a lossy video coding scheme. For lossy video compression, similar to the case of lossy image compression, transform based coding is more widely used than predictive coding. This work aims to provide a compression performance that matches other lossy video coding algorithms, which would indicate that compression in stages does not require a loss in performance

3.4 System architecture

Figure 3.2 shows the reconfigurable system for universal compression. The encoder at the top is for lossless data compression. It treats input as one-dimensional data, and uses statistical Markov modelling for context modelling. Such a system is presented in [31], together with details about a probability estimator for k-bit words and arithmetic coding.

The second encoder is for lossless image compression using predictive coding which makes use of the same probability estimation and arithmetic coding techniques: the LMMIC system mentioned in Section 3.3.1. When changing the configuration of the system from one-dimensional data compression to lossless image compression, the probability estimator and arithmetic encoder may be retained, and only a part of the system has to be

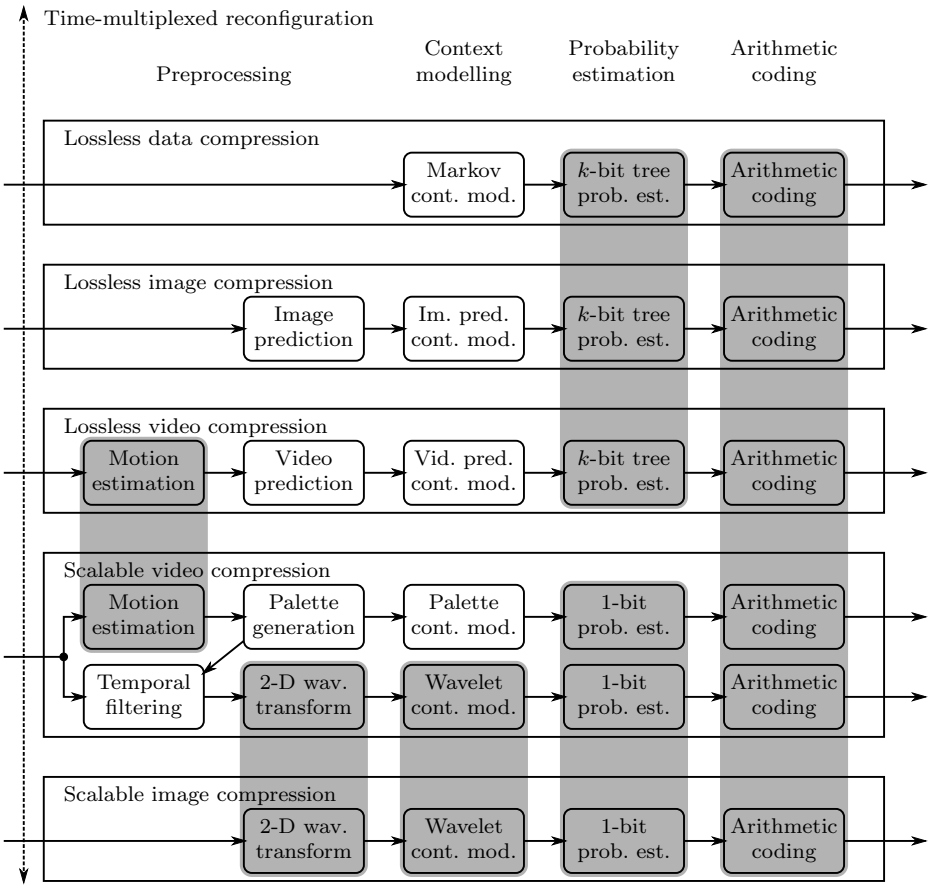


Figure 3.2: A universal reconfigurable compression system, with shaded areas representing components that can be retained for different kinds of data.

reconfigured. The third encoder is for lossless video compression, and uses the BAPME coding system mentioned in Section 3.3.2. Motion estimation is required for video compression, so a motion estimation block is included in the diagram. Details about the suitable hardware motion estimation block developed by our group are presented in [36].

The last two blocks shown in Figure 3.2 are for scalable video compression and scalable image compression, sometimes called progressive image compression. Whereas predictive coding is most suitable for lossless compression, lossy compression of image and video is typically obtained using transform coding, such as the discrete wavelet transform (DWT). As indicated by the figure, the scalable video compression system is a superset of the scalable image compression system. Also, the scalable video compression system shares the motion estimation engine and the arithmetic encoder with other compression systems.

3.5 Scalable video coding

Figure 3.3 shows a basic overview of the encoding process for the SVC method proposed in this dissertation. It corresponds to the scalable video compression block in Figure 3.2. The system is a T+2D system, that is, temporal filtering followed by two-dimensional spatial filtering. The resultant video bitstream is scalable in time, in space, and in quality.

If three levels of temporal decomposition are used, the frame rate can be reduced to 1/2, 1/4, or 1/8 of the original frame rate by dropping frames. If three levels of spatial decomposition are used, the spatial dimensions can be reduced to 1/2, 1/4, or 1/8 of the original dimensions, with the area thus begin reduced to 1/4, 1/16, or 1/64 of the original area. The quality of the output can be reduced using fine-grained scalability. If a reversible two-dimensional DWT is used, a video sequence can be scaled all the way from lossless to the least acceptable quality. If a non-reversible two-dimensional DWT is used, the encoder cannot encode losslessly, but the sequence can still be scaled from very high quality to the least acceptable quality.

3.5.1 Scalable motion vector coding

The motion estimation stage generates motion vectors describing the motion between frames. The motion estimation processor searches for the motion

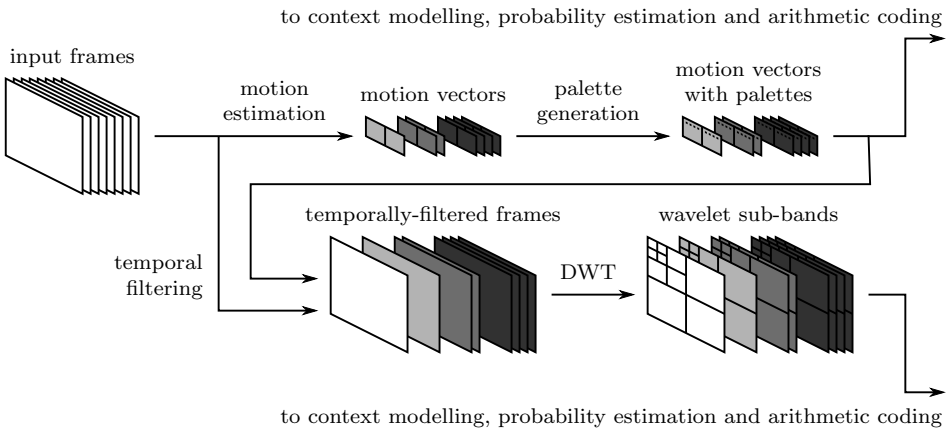


Figure 3.3: An overview of the encoding process for T+2D SVC with motion vector palettes.

vectors that best describe the motion of the video, which are then encoded as part of the compressed video bitstream. In order to have scalable coding of the motion vectors themselves, the vectors are encoded in multiple layers. This is done using motion vector palettes, which will be described in Chapter 4.

3.5.2 Motion-compensated temporal filtering

The next step is to use the motion vectors in motion-compensated temporal filtering (MCTF) using the 5/3 filter with lifting. This makes it necessary for a number of frames to be held in memory. The details for this will follow in Section 5.1.

3.5.3 Spatial wavelet filtering

The temporally-filtered frames are then encoded individually like images. A two-dimensional DWT is performed on each frame. There are two options of wavelet filters to use, the 5/3 filter for a reversible integer transform suitable if the video sequence needs to be encoded losslessly, or the 9/7 filter.

3.5.4 Entropy coding of frames

Following the DWT step, each resulting wavelet band is encoded in turn. The transformed bit planes are encoded starting from the most significant bit plane. No fractional bit planes are used. Before context modelling, each

bit plane is split into sub-blocks of 16×16 pixels, and the significance of each sub-block is encoded like in EBCOT [11]. The significant sub-blocks are then processed using the two-dimensional context modelling scheme used in ICER [12], which is similar to EBCOT but is less complex, lending itself better to hardware implementations. The context modelling uses 17 contexts, which are then used in adaptive arithmetic coding. The arithmetic coding system used is the modified Z-Coder [82] that is suitable for hardware implementations.

3.6 Conclusion

This chapter described the universal reconfigurable compression system and indicated how scalable video coding can be incorporated into it. The common blocks that can be reused were indicated.

Chapter 4

Scalable Motion Vectors

The video stream will consist mainly of two components, the motion vector information, and the video frame information. This chapter analyses the scalable encoding of motion vector.

Section 4.1 discusses the overhead incurred by encoding motion vectors as side information and methods to reduce this overhead. Section 4.2 discusses the generation of motion vector palettes as a viable way to encode motion vectors. Section 4.3 illustrates how the motion vector palette can be encoded, and Section 4.4 illustrates how the motion vectors themselves can then be encoded as indices pointing into this motion vector palette. Section 4.5 discusses why the motion vector palette is suitable for scalable coding. Section 4.6 shows how the use of the layered motion vector palettes improves the rate-distortion characteristic at low bit rates. Section 4.7 shows the gain obtained by using palettes to encode the motion vector rather than using wavelets as suggested by previous work.

4.1 Motion vectors as side information

In Section 3.3.2, we mentioned BAPME [35], which is used for lossless video compression in the universal compression system. In BAPME, motion vectors are not transmitted as side information. Instead, a pixel-based motion estimation search is performed both at the encoder and at the decoder. Since BAPME is lossless, the encoder and the decoder will be synchronized.

4.1.1 No side information

An attempt was made to avoid side information with scalable video coding (SVC), but this lead to the decoding failing whenever the full quality of the video bitstream is not transmitted. Reduction in the quality of the video bitstream leads to discrepancies between the data at the encoder and at the decoder. A small reduction in quality will change the decoded pixels. Since the lossy decoded pixels are different from the encoded pixels, a motion estimation search at the decoder will give motion vectors that can be completely different from the motion vectors found at the encoder. This will lead to larger discrepancies in the next decoded pixels. Thus, decoding was found to fail catastrophically even with slight losses in quality, showing that the technique used by BAPME to avoid transmitting motion vectors as side information is not suitable for scalable video coding.

This suggests that using no side information does not work for scalable video, and is suitable only for lossless video compression.

4.1.2 Extracting motion vectors from a base layer

As an alternative solution, an attempt was made to use two layers for the video coding. The first layer, or base layer, is encoded first. Since it is always transmitted in full, the decoder will have the exact same base layer as the encoder, and the motion vectors can be extracted from this layer without using any side information, just like in BAPME. Then, these motion vectors can be used by the second layer, which is the scalable layer.

However, this did not give satisfactory results. When the base layer has a low quality, the motion vectors extracted from the base layer are not accurate, and lead to poor compression performance. For the motion vectors to be accurate using this method, the base layer had to be of a high quality. However, having a base layer of high quality is not suitable for SVC, since then the quality cannot be scaled down sufficiently for low bit rates, and the overhead can be actually worse than transmitting the motion vectors as side information in the first place.

Figure 4.1 shows some typical rate-distortion curves obtained using this method. The best curve in the set is obtained when intra-coding is used, that is, when no base layer and no motion estimation are used and each frame is encoded independently. When the base layer is of poor quality, at

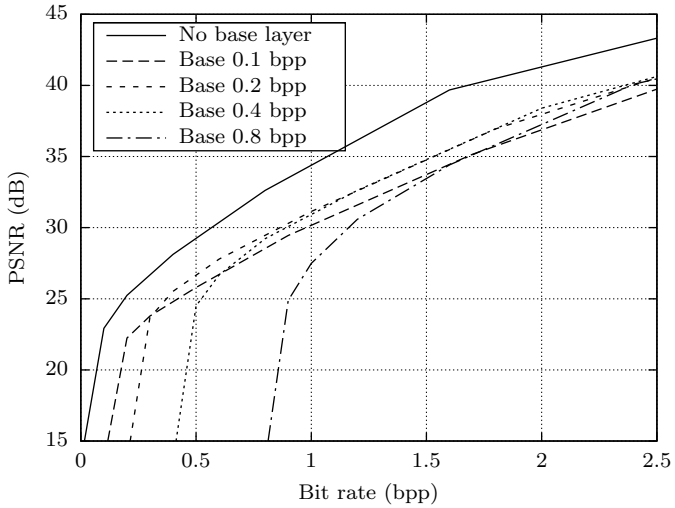


Figure 4.1: Typical rate-distortion curves obtained when extracting motion vectors from base layers.

0.1 bits per pixel, the rate-distortion curve is much poorer, indicating that the motion vectors obtained using a low quality base layer are of such bad quality that they hinder the encoding process, making the curve lag the intra-coding curve by more than 0.1 bpp. The curves for base layers with better quality are not good either, they never recover the high bit rate spent on the base layer for the motion vectors.

4.1.3 Scalable encoding of the motion vectors

The problem with encoding the motion vectors as side information is at low bit rates. The bit rate of motion vectors is insignificant at high bit rates, where the bit rate of the frame data for each frame will be much higher, but at low bit rates this is not the case. A number of video sequences from [83] at different resolutions were used to obtain bit rates for the motion vectors and for the frame data. Table 4.1 gives some details on the sequences.

Table 4.2 shows the bit rate of the motion vectors and the bit rate of the full quality frame data for the SVC scheme presented in this work. Notice that at full quality, the motion vector bit rate is less than 1% for practically all video sequences shown. However, if the frame data is scaled to a lower quality, the situation changes and the bit rate used for motion vectors becomes significant. If, for example, we need to scale the *crowdrun*

Table 4.1: The video sequences used in experiments [83].

Sequence	Resolution	Details
crowdrun	1920×1080	crowd of people running
riverbed	1920×1080	riverbed seen through water, hard to code
rush hour	1920×1080	many cars moving slowly, fixed camera
tractor	1920×1080	chaotic movement, camera follows tractor
shields	1280×720	man walking in front of detailed shields
stockholm	1280×720	detailed houses, water and moving cars
coastguard	352×288	boats moving on a river
foreman	352×288	man talking and showing buildings

Table 4.2: Bit rate used by motion vectors and by frame data for various video sequences.

Sequence	Resolution	Motion vectors (bpp)	Frame data (bpp)
crowdrun	1920×1080	0.0055	2.84
riverbed	1920×1080	0.0185	1.98
rush hour	1920×1080	0.0073	1.37
tractor	1920×1080	0.0076	1.73
shields	1280×720	0.0015	2.35
stockholm	1280×720	0.0017	2.39
coastguard	352×288	0.0102	2.08
foreman	352×288	0.0218	1.79

sequence down to a bit rate of 0.02 bits per pixel, which is about 1.5 Mbps, the bit rate of the motion vectors would be around 25% of the whole bit rate.

When motion estimation is performed, a video frame is split into a number of macroblocks, and a matching macroblock is found for each one in a reference frame. The offset of the matching macroblock in the reference frame relative to its location in the current frame is the motion vector of the macroblock.

In order to decrease the overhead of the motion vectors at low bit rates, a layered scheme for the motion vectors themselves was designed. The layered scheme makes use of motion vector palettes.

The motion vector palette is a palette consisting of a limited number of motion vectors that can be used for motion compensation in one frame. For scalability, this motion vector palette is split into layers so that sections of

it can be removed later without the need of re-encoding.

The size of the video bitstream used to encode the motion vectors is smaller than the size of the bitstream used to encode the temporally filtered frames. Scaling the motion vector information can have a large effect on the quality. Thus, motion vector information is scaled only at low bit rates, when the size used for the frames is reduced so much that it is comparable to the size of the motion vectors, otherwise the loss in quality due to the scaling of motion vectors is larger than the loss in quality due to the scaling of frames by the same amount.

It is important to note that with wavelet temporal filtering, the effects introduced into a frame will only propagate to a fixed number of frames. For example, if one level of a 5/3 filter is used, the quantization errors in a frame will propagate to a maximum distance of two frames. For two levels, the maximum propagation distance is increased by a further four frames to become six frames. For three levels, the maximum distance is increased by eight frames to 14 frames. Thus, the drifting effects are limited and do not go on indefinitely.

Using colour palettes for image coding usually shows visible artefacts, since this quantizes the pixel values in the output image. Using motion vector palettes does not imply that similar artefacts will be a problem in the decoded bitstreams. Motion vectors are only used in reducing the temporal redundancy before the final encoding of the image data. Quantization of the motion vectors before the temporal filtering does not lead to quantization errors in the output pixel values. In fact, it is possible to quantize the motion vectors using motion vector palettes and obtain lossless encoding when using the 5/3 reversible filter for the spatial dimensions.

4.2 Generating motion vector palettes

A method for palette generation for colour quantization is presented in [68]. This method was adopted to generate palettes of motion vectors instead of colours, and it was modified to allow the splitting of the resulting palette into a number of layers.

The method is essentially a divisive clustering technique starting with a set containing all the motion vectors, and dividing into subsets as required.

If the number of macroblocks is n_1, there are n_1 motion vectors of the

form

$$\mathbf{v}_{1,j} = \begin{pmatrix} \delta x_{1,j} \\ \delta y_{1,j} \end{pmatrix}, \qquad 1 \leq j \leq n_1. \tag{4.1}$$

These motion vectors may be repeated, that is, two or more macroblocks may have identical motion vectors. Let S_1 be a set containing all the original n_1 motion vectors. We need to partition S_1 into disjoint sets of the form S_i, where each S_i is a set containing n_i motion vectors $\mathbf{v}_{i,j}$, with $1 \leq j \leq n_i$. As for the set S_1, motion vectors may be repeated, however, identical motion vectors will always be grouped into the same subset during partitioning, that is, the sets will be disjoint. Each set S_i has a mean value

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{v}_{i,j}. \tag{4.2}$$

The final aim of the whole process is to obtain p distinct motion vectors from the original n_1 motion vectors. This is achieved by partitioning S_1 into p sets. The mean of each of these sets will be used as one of the p motion vectors in the palette. The motion vectors of the frame will then be encoded as indices pointing to one of the p motion vectors in the palette. That is, there will be p motion vectors in the palette, and n_1 indices which each point to one of the p motion vectors in the palette.

The partitioning has the structure of a proper binary tree. Figure 4.2 shows such a binary tree. The root node has an index $i = 1$ and represents set S_1. After one division, S_1 is replaced by S_2 and S_3, which in Figure 4.2 is indicated by the root node 1 having two child nodes 2 and 3. The motion vectors in the set represented by a parent node are split between the two sets represented by the child nodes. Next, S_2 is split into S_4 and S_5, and this divisive clustering process continues until in the end there are p nodes. In Figure 4.2, there are $p = 9$ leaf nodes representing the final nine sets. The total number of nodes is $2p - 1 = 17$.

During the partitioning, the choice of which set to split, and how to split it, is made to minimize the total squared error *TSE* between the original motion vectors and the resultant motion vectors, which is given by

$$TSE = \sum_{\substack{\text{all leaf} \\ \text{nodes } i}} \sum_{j=1}^{n_i} \|\mathbf{v}_{i,j} - \mathbf{m}_i\|^2. \tag{4.3}$$

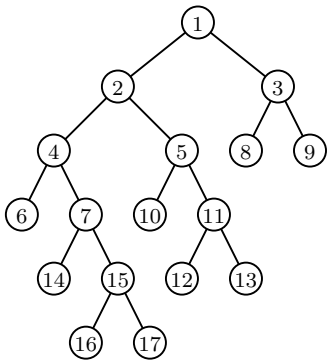


Figure 4.2: A binary tree with nodes representing sets of motion vectors.

To decide which set to split and how to split it, we need to calculate the estimator $\mathbf{R}_i$ and the variance $\tilde{\mathbf{R}}_i$, which are given by

$$\mathbf{R}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{v}_{i,j} \mathbf{v}_{i,j}^{\mathrm{T}} \tag{4.4}$$

$$\tilde{\mathbf{R}}_i = \mathbf{R}_i - \mathbf{m}_i \mathbf{m}_i^{\mathrm{T}} \tag{4.5}$$

for all leaf nodes i. $\mathbf{R}_i$ and $\tilde{\mathbf{R}}_i$ are 2×2 symmetric matrices.

As described in [68], the cluster variation in a set S_i is greatest in the direction of the vector $\mathbf{e}_i$ which maximizes the expression

$$\frac{1}{n_i} \sum_{j=1}^{n_i} \big((\mathbf{v}_{i,j} - \mathbf{m}_i)^{\mathrm{T}} \mathbf{e}_i \big)^2 = \mathbf{e}_i^{\mathrm{T}} \tilde{\mathbf{R}}_i \mathbf{e}_i. \tag{4.6}$$

Since $\tilde{\mathbf{R}}_i$ is a symmetric matrix, this expression is maximized when $\mathbf{e}_i$ is an eigenvector corresponding to the principal eigenvalue λ_i of $\tilde{\mathbf{R}}_i$. This principal eigenvalue λ_i and a corresponding eigenvector $\mathbf{e}_i$ are calculated for each set S_i.

In each division, we increase the number of leaf nodes by one by splitting one set represented by a leaf node into two subsets. Each time we need to split a set, the leaf node i corresponding to the set S_i with the largest absolute value of λ_i is selected. If the number of leaf nodes in the binary tree before the split is k, the total number of nodes is $2k-1$. After the split there will be a total of $2k+1$ nodes, of which $k+1$ will be leaf nodes. The

set S_i will be split into sets S_{2k} and S_{2k+1} according to

$$S_{2k} = \{\mathbf{v} \in S_i : \mathbf{e}_i^{\mathrm{T}}\mathbf{v} \leq \mathbf{e}_i^{\mathrm{T}}\mathbf{m}_i\} \tag{4.7}$$

$$S_{2k+1} = \{\mathbf{v} \in S_i : \mathbf{e}_i^{\mathrm{T}}\mathbf{v} > \mathbf{e}_i^{\mathrm{T}}\mathbf{m}_i\}. \tag{4.8}$$

Clearly, any identical motion vectors will go into the same set. After $p-1$ divisions, the binary tree will have a total of $2p-1$ nodes, of which p are leaf nodes.

4.3 Encoding the motion vector palette

Encoding the motion vector palette entails encoding many integers, the magnitude of which is not known in advance. Section 4.3.1 shows a method to encode these integers.

4.3.1 Encoding integers

Algorithm 4.1 shows an algorithm similar to exponential-Golomb coding [84] that is used to encode the integers. A number of experiments on different video sequences was performed, and on average, the modification produces 4.5% less bits than exponential-Golomb for integers typical to the palette encoder of Sections 4.3 and 4.4. Algorithm 4.1 encodes a non-negative integer n. The decoder will not have information about how large n is, so the exact number of bits required to represent n cannot be used by the encoder, but an estimate e is used, and the better the estimate, the less bits are used to encode n. For binary, the number of bits required to represent n is

$$r = \lceil \log_2(n+1) \rceil. \tag{4.9}$$

The algorithm is equivalent to exponential-Golomb coding when $r \leq e$, but differs when $r > e$.

As an example, let us encode the number 13 with an estimated number of bits $e = 6$. The required number of bits is $r = \lceil \log_2(13+1) \rceil = 4$. In fact, 13 is represented as '`1101`' in binary, which is four bits wide. Since $r \leq e$, we encode 13 using the first branch, that will be '`1`' (Line 3) followed by '`001101`' (Line 4). Thus, 13 is encoded as '`1001101`', using seven bits.

If the estimated number of bits is $e = 2$, we encode 13 using the second

Algorithm 4.1 An algorithm to encode a non-negative integer n with e as the estimated number of bits required.

1: $r \leftarrow \lceil \log_2(n+1) \rceil$ {required bits to encode n}
2: **if** $r \leq e$ **then**
3: write '1'
4: write n using e bits
5: **else**
6: write $r - e$ '0's
7: write n using r bits
8: **end if**

TABLE 4.3: THE NUMBER OF BITS USED TO ENCODE A NON-NEGATIVE INTEGER n USING ALGORITHM 4.1 AND USING EXPONENTIAL-GOLOMB CODING, WITH AN ESTIMATED NUMBER OF BITS e.

Range of n	r	Bits used by Algorithm 4.1	Bits used by exponential-Golomb
$0 \leq n/2^e < 1$	$\leq e$	$e+1$	$e+1$
$1 \leq n/2^e < 2$	$e+1$	$e+2$	$e+3$
$2 \leq n/2^e < 3$	$e+2$	$e+4$	$e+3$
$3 \leq n/2^e < 4$	$e+2$	$e+4$	$e+5$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2^{i-1} \leq n/2^e < 2^i - 1$	$e+i$	$e+2i$	$e+2i-1$
$2^i - 1 \leq n/2^e < 2^i$	$e+i$	$e+2i$	$e+2i+1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

branch, that will be '00' (Line 6) followed by '1101' (Line 7). In this case, 13 is encoded as '001101', using six bits.

For the general case, if $r \leq e$, the number of bits used to encode n will be 1 for Line 3 plus e for Line 4, for a total of $e+1$ bits. If $r > e$, the number of bits used to encode n will be $r - e$ for Line 6 plus r for Line 7, for a total of $2r - e$ bits.

Table 4.3 compares the number of bits used by this algorithm to the number of bits used by exponential-Golomb coding. In our case, excluding the equivalent case of $r \leq e$, the most common case is when $r = e+1$. This explains why the modified algorithm gives better results than exponential-Golomb in our use case, as it uses $e+2$ bits instead of the $e+3$ bits required by exponential-Golomb.

The error E in the estimate e is $E = |e - r|$. If $r \leq e$, then $1 + e$

Algorithm 4.2 An algorithm to encode an integer n with e as the estimated number of bits required for its magnitude.

```
1: encode magnitude |n| with e estimated bits {Algorithm 4.1}
2: if n > 0 then
3:     write '0'
4: else if n < 0 then
5:     write '1'
6: end if
```

bits are used instead of the r bits required for simple binary representation, resulting in an overhead of $1 + E$ bits. For example, to encode $n = 13$ with an estimated bit size $e = 6$, the number of bits used is $1 + e = 7$ instead of the $r = 4$ bits required to encode 13.

If $r > e$, then $2r - e$ bits are used instead of the minimum required r, resulting in an overhead of E bits. For example, to encode $n = 13$ with $e = 2$, the number of bits used is $2r - e = 6$ bits instead of the required $r = 4$ bits.

The overhead is $1 + E$ when $r \leq e$ and E when $r > e$. In both cases, for every extra bit of error in the estimate, one extra bit is wasted in the encoding.

To encode integers that can be positive or negative, the algorithm is extended as shown in Algorithm 4.2. First the non-negative magnitude of the integer is encoded, then, a sign bit is written only if $n \neq 0$. This has the advantage that unlike popular exponential-Golomb negative extensions, no overhead is present when $n = 0$. This is particularly important for our algorithm, which encodes differences in motion vectors between successive layers, and for these differences, the case $n = 0$ is not uncommon.

4.3.2 Encoding the palette

To encode a motion vector palette with p entries, the number p is encoded using Algorithm 4.1, with an estimated number of bits $e_p = 4$. Although this looks like a small number (a palette with more than 16 motion vectors would require more than four bits), this is intended for multi-layer palettes, where an initial value of $e_p = 4$ is not bad. As shown in Section 4.3.1, an error in the estimate e_p will lead to higher bit overheads, so a system was devised to adapt e_p dynamically. If we need to write more than one palette, as will be the case in Section 4.5 below, e_p is updated after each value of p

is encoded using

$$r_p = \lceil \log_2(p+1) \rceil \tag{4.10}$$

$$\text{new } e_p = \frac{\lfloor 3e_p + 1/2 \rfloor + r_p}{4}. \tag{4.11}$$

e_p has a precision of $1/4$, and its fractional part is discarded when used in the algorithm that expects e to be an integer.

After encoding the number of motion vectors in the palette, we must encode the p motion vectors themselves. Recall that each motion vector has two components, δx and δy. The values δx and δy can be negative, so we use Algorithm 4.2 for each component. To write the motion vector components δx and δy of each motion vector, we start with an estimated number of bits $e_v = 1$. The value of this initial estimate is not really important, as after each component of a motion vector is encoded, e_v is updated in the same way as e_p in (4.10) and (4.11).

4.4 Encoding the motion vector indices

After the motion vector palette with p entries is encoded, the motion vectors are encoded with reference to the palette, that is, each motion vector is encoded as an index n in the range $0 \leq n < p$. The details of the designed scheme for context modelling of the indices follow below. The scheme is similar to [69], which is designed for colour palettes, but uses a much smaller number of contexts. The small number of contexts is essential for the adaptive encoder to settle quickly: using more contexts means that the contexts would take longer to adapt.

The indices are encoded using adaptive two-dimensional contexts. Figure 4.3 shows neighbouring motion vectors that are used in obtaining the context for the current motion vector O. Each vector is encoded as the answer to four questions:

1. Is the current motion vector O equal to W?
2. Is the current motion vector O equal to N?
3. Is the current motion vector O equal to NE?
4. What is the current motion vector O?

$NNWW$	NNW	NN	NNE
NWW	NW	N	NE
WW	W	O	

Figure 4.3: Motion vectors for macroblocks neighbouring the current motion vector O.

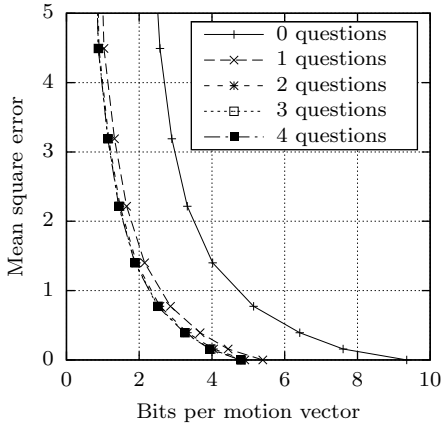
Table 4.4: The context number for encoding a motion vector.

Question 1 Is $O = W$?		Question 2 Is $O = N$?		Question 3 Is $O = NE$?	
$W = WW$	1	$W = NW$	1	$W = N$	1
$N = NW$	2	$N = NN$	2	$N = NNE$	2
$NW = NWW$	4	$NW = NNW$	4	$NW = NN$	4
Extra	0	Extra	8	Extra	16

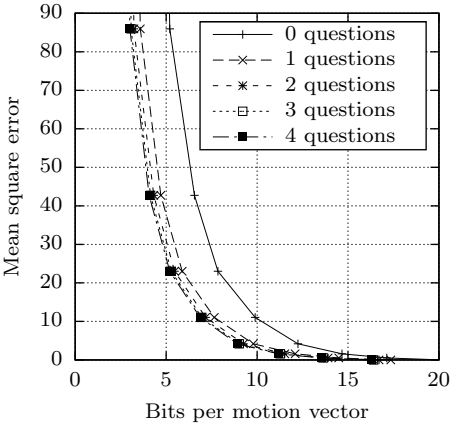
If the answer to any of the first three questions is true, the remaining questions are redundant and skipped. Question 4 is the fall-back question in case none of the tested neighbours match O, and does not have a binary answer.

Figure 4.4 shows how the number of questions to ask before the fall-back question affects the compression performance. The motion vectors encoded were obtained from four different sequences. The curves for zero questions, that is, when only the fall-back question is used, are the poorest in all cases. There are marked improvements when one or two questions precede the fall-back question, and a further slight improvement for three questions. However, there is no visible improvement in performance for a further fourth question, indicating that three questions are sufficient before the fall-back question.

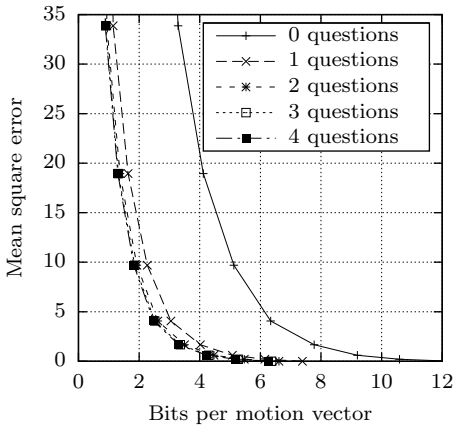
The answers to the first three questions are encoded using a binary arithmetic coder with adaptive contexts. The contexts used are derived according to Table 4.4.



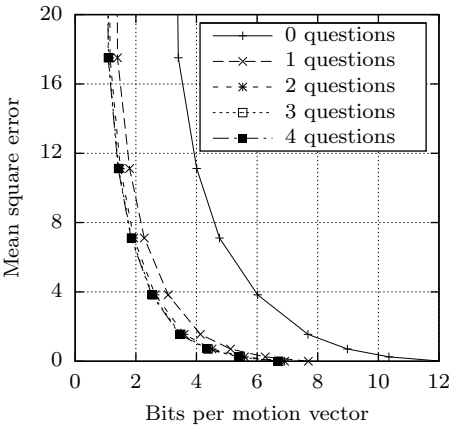
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)



(c) *rush hour* (1920 × 1080)



(d) *tractor* (1920 × 1080)

Figure 4.4: The mean square error of the motion vectors when encoded using palettes with a different number of questions.

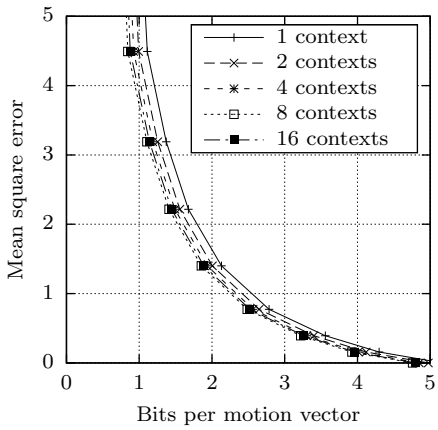
$NNWW$ $= 2$	NNW $= 7$	NN $= 7$	NNE $= 8$
NWW $= 2$	NW $= 7$	N $= 7$	NE $= 8$
WW $= 3$	W $= 3$	O $= 7$	

Figure 4.5: Example values for neighbours of the current motion vector O.

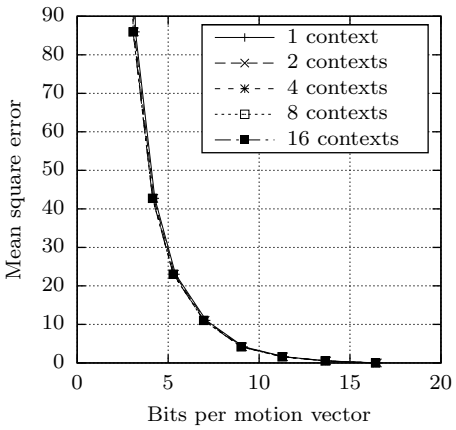
For example, suppose we have the values of Figure 4.5. For Question 1, we have $W = WW$, $N = NW$, and $NW \neq NWW$, so the context to use will be $1(1)+1(2)+0(4)+0 = 3$. Since $O \neq W$, the answer to Question 1 is false, or '0'. Then, we have to encode bit '0' using context 3. For Question 2, we have $W \neq NW$, $N = NN$, and $NW = NNW$, so the context to use will be $0(1) + 1(2) + 1(4) + 8 = 14$. Since $O = N$, the answer to Question 2 is true, so we encode bit '1' using context 14. Since the value of O can be obtained from the answer to this question, no further questions are required.

There are a total of 24 possible contexts, eight for each of Questions 1–3. Figure 4.6 shows how the compression performance varies according to the number of contexts for each question. Increasing the number of contexts improves compression until the number of contexts reaches eight for each question. There is no improvement when using 16 contexts. This is because with too many contexts, each context will require more motion vector to adapt to a good probability estimation. The number of motion vectors is not very large, it is much smaller than the number of pixels in a frame, thus the number of contexts has to be limited for effective context modelling. Note that for the (b) *riverbed* sequence, there is almost no visible improvement from one context to eight contexts per question. This is because the moving water in the sequence results in dissimilar motion vectors in neighbouring macroblocks, which makes the context modelling scheme ineffective.

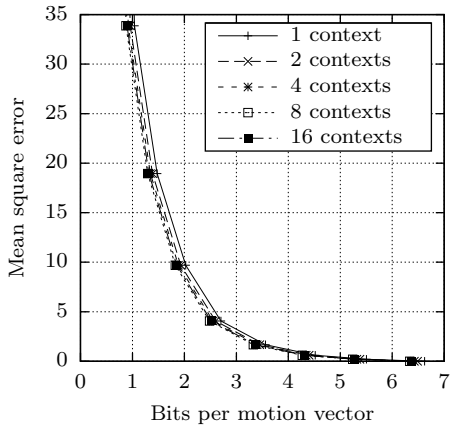
When O lies on an edge, some of the neighbours in Figure 4.3 can be missing, so some questions make no sense and are skipped. For example, if O lies on the left edge of the image, there are no W neighbours, so Question 1 is skipped. Similarly, Question 2 is skipped if O lies on the top edge, and Question 3 is skipped if O lies on the right edge or the top edge.



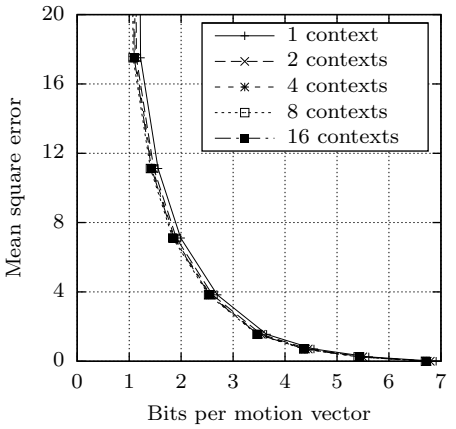
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)



(c) *rush hour* (1920 × 1080)



(d) *tractor* (1920 × 1080)

Figure 4.6: The mean square error of the motion vectors when encoded using palettes with a different number of contexts per question.

Algorithm 4.3 An algorithm to encode an integer n in the range $0 \leq n < p$ using a binary arithmetic encoder.

```
1: mid ← largest power of 2 that is < p
2: while mid ≥ 1 do
3:    encode n < mid, with prob(n < mid) = mid/p
4:    if n < mid then
5:       p ← mid
6:    else
7:       n ← n − mid
8:       p ← p − mid
9:    end if
10:   while mid ≥ p do
11:      mid ← mid/2
12:   end while
13: end while
```

Also, when O lies on the top edge of the image, some of the conditions for Question 1 in Table 4.4 need to be adjusted, as there are no N, NW and NWW neighbours. In this case, for the purpose of obtaining a context number, we assume that $N = NW$ and $NW = NWW$.

Sometimes Questions 2 and 3 can be skipped because the answer is already known from the previous questions. For example, if $W = N$, Question 2 is equivalent to Question 1 and thus redundant. Similarly, if $W = NE$, Question 3 is equivalent to Question 1, and if $N = NE$, Question 3 is equivalent to Question 2, both cases making Question 3 redundant.

If none of the answers to the first three questions are true, the index of the motion vector in the motion vector palette is encoded. This can be achieved using the same binary arithmetic coder with Algorithm 4.3. In this algorithm, all integers n in the range $0 \leq n < p$ are assumed to be equally probable, so the arithmetic encoder uses $\log_2 p$ bits to encode the integer n.

The encoding of Question 4 can be optimized if we consider the answers to Questions 1–3. Suppose that

$$O = 7, \qquad W = 0, \qquad N = 8, \qquad NE = 6, \qquad p = 10.$$

In this case, we could naively encode O by encoding the value $n = 7$ with $p = 10$ using $\log_2 10$ bits. But both the encoder and the decoder already have the answers to Questions 1–3, and thus can deduce that n cannot take the values 0, 6 and 8, meaning that we can remove some redundancy by

adjusting the values of n and p. The value of n is decreased by the number of distinct checked values less than O. In this case,

$$W < O, \qquad N \not< O, \qquad NE < O.$$

There are two distinct checked values, W and NE, that are less than O, so $n' = n - 2 = 5$. The value of p is decreased by the number of distinct questions asked, in this case three, so that $p' = p - 3 = 7$. Instead of encoding $n = 7$ in the range $0 \leq n < 10$, we can thus encode $n' = 5$ in the range $0 \leq n' < 7$. This saves $\log_2 10 - \log_2 7 = 0.51$ bits in the arithmetic coder. This saving is more pronounced for smaller values of p, which will make it particularly important when splitting the motion vector palette into layers, which will follow in Section 4.5.

Further to this, if originally $p = 1$, n has to be 0 and no questions are required. If $p = 2$, and one question is asked, the question essentially decreases p to $p' = 1$, and no further questions are required. The same happens with two questions when $p = 3$ and with three questions when $p = 4$.

4.5 Splitting the motion vectors into layers

To support scalable motion vector encoding, the motion vectors are encoded in layers. In Section 4.2 we obtained a binary tree with p leaf nodes and a total of $2p - 1$ nodes. Figure 4.7 shows such a binary tree with $p = 9$ leaf nodes and a total of 17 nodes. The set S_1 at the root node contains all the motion vectors. If we want to truncate the binary tree so that we have only t leaf nodes instead of p, all we need to do is to remove all the nodes $i > 2t - 1$. In the example, if we want only $t = 4$ leaf nodes, we remove all the nodes $i > 7$, which are dashed in Figure 4.7. Nodes 3, 5 and 7 are not leaf nodes in the complete tree, but become leaf nodes in the truncated tree.

Suppose we want to encode the motion vectors in two layers. The first layer has only four values: nodes 3, 5, 6 and 7. The second layer has nine values: nodes 6, 8, 9, 10, 12, 13, 14, 16 and 17. These two layers can be represented by the tree shown in Figure 4.8. Note that the root node in this case is 0, not 1. This is to indicate that if neither of the two layers is encoded, no motion vector information is transmitted at all. Whereas $\mathbf{m}_1$,

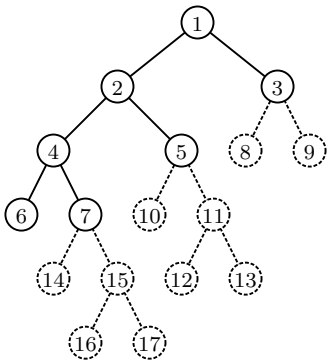


Figure 4.7: The binary tree of motion vectors which can be truncated.

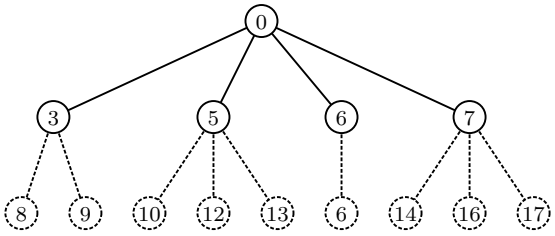


Figure 4.8: A layered tree with nodes representing sets of motion vectors.

the mean of all the motion vectors in S_1, may be non-zero, $\mathbf{m}_0$ is always zero.

After the motion vectors are split into layers, each layer is encoded in turn. The first layer can be encoded using the methods described in Sections 4.3 and 4.4, with $p = 4$ possible motion vectors.

The second layer is encoded next. As before, this is done by encoding the palette first and the indices later. For this second layer, we encode four motion vector palettes, one for each sub-tree, each using the method of Section 4.3 with one small modification. Instead of encoding the motion vectors, which for the sub-tree rooted at node 3 would be $\mathbf{m}_8$ and $\mathbf{m}_9$, the differences between the nodes and their parent is encoded, in this case $\mathbf{m}_8 - \mathbf{m}_3$ and $\mathbf{m}_9 - \mathbf{m}_3$. These differences are typically very close to zero.

After the motion vector palettes are encoded, the motion vector indices have to be encoded using the method of Section 4.4. The method is modified to make use of the previous layer, which will be known to both the decoder and the encoder. Suppose the current motion vector O from the previous layer is the motion vector represented by node 3. Then, for this particu-

lar macroblock, only two motion vectors are possible, those represented by nodes 8 and 9, so we have $p = 2$. Also, if W for the previous layer is not the motion vector for node 3, that is, if for the previous layer $O \neq W$ and Question 1 was false, then Question 1 is skipped for this layer, as $O \neq W$ still. Similarly, Question 2 is skipped if $O \neq N$ for the previous layer, and Question 3 is skipped if $O \neq NE$ for the previous layer. Notice that when the motion vector O from the previous layer is 6, we have $p = 1$, which, as mentioned in Section 4.4, does not need any bits to encode.

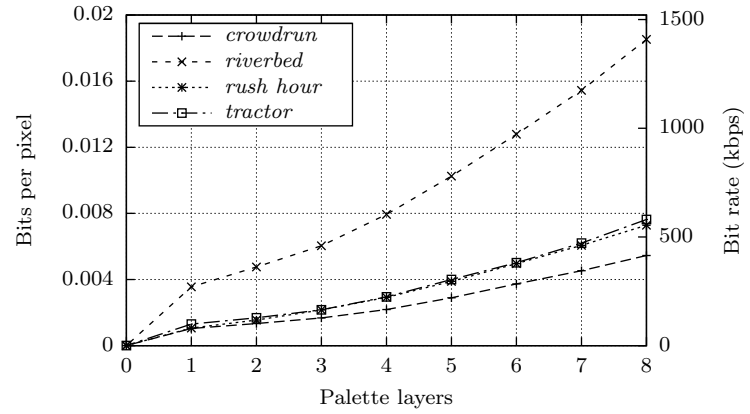
4.5.1 Where to split the motion vector palette layers

In the example of Figures 4.7 and 4.8, we saw how to split a palette of nine motion vectors in two layers by truncating five values. When encoding a motion vector palette, we need a method to decide how many palette values to retain in each layer.

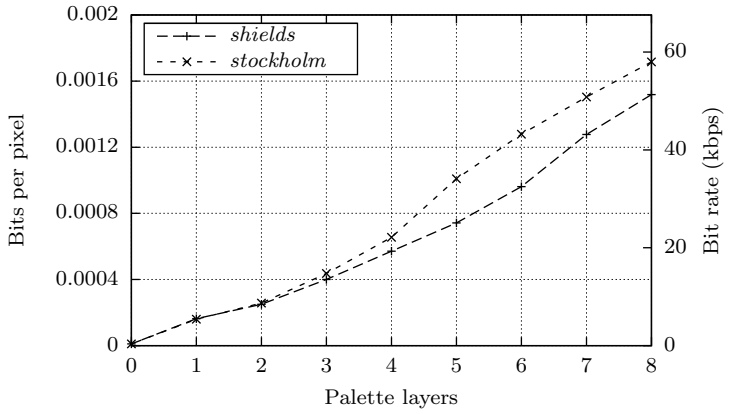
We start with the original binary tree, like the one shown in Figure 4.7. The tree has p leaf nodes, which means there are p motion vectors in the palette. These are split into eight layers. Eight layers was found to be a reasonable value experimentally; using less layers gives us less scalability, and using more layers will require more bits to encode the motion vectors without adequate compensation. The eight layers will have a number of motion vectors $p_i, 1 \leq i \leq 8$, with p_1 being the number of motion vectors in the first layer, and $p_8 = p$ being the number of motion vectors in the final layer. The values p_i are calculated using

$$p_i = \lceil p \times f_i \rceil \,, \tag{4.12}$$

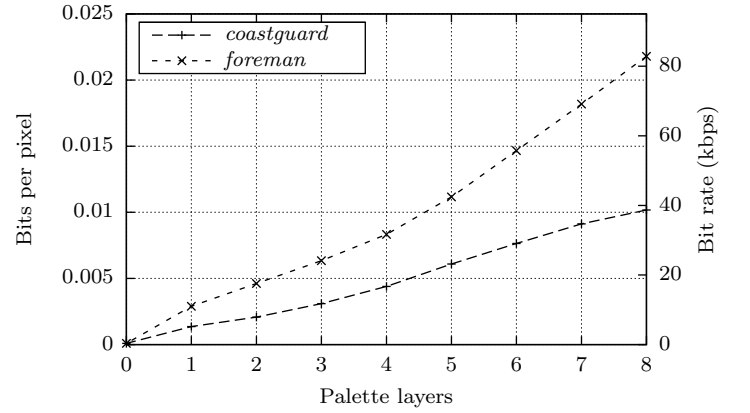
with the values f_i found in Table 4.5. The values in the table were found experimentally to split the palette into layers that require a similar number of bits to encode for a wide range of video sequences. Figure 4.9 shows how the bit rates required for the motion vectors increases gradually as the number of palette layers increases for different video sequences at different resolutions. The gradual increase in all the sequences validates the choice of values in Table 4.5.



(a) 1920×1080



(b) 1280×720



(c) 352×288

Figure 4.9: The bit rates of the motion vectors against different number of palette layers for video sequences of different resolutions.

TABLE 4.5: FACTORS TO SPLIT THE MOTION VECTOR PALETTE INTO EIGHT LAYERS.

Layer i	Fraction f_i
1	$^2/_{64}$
2	$^4/_{64}$
3	$^7/_{64}$
4	$^{12}/_{64}$
5	$^{20}/_{64}$
6	$^{31}/_{64}$
7	$^{45}/_{64}$
8	$^{64}/_{64}$

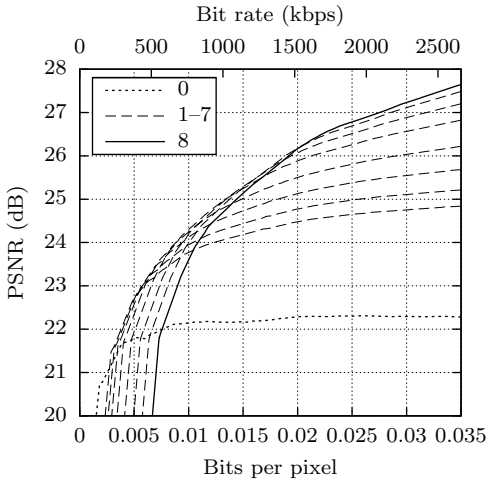
4.6 The effect of using motion vector layers

This section shows results for experiments related to motion vector scalability.

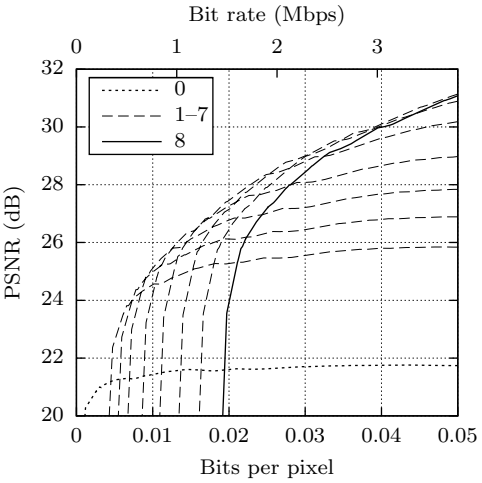
Table 4.2 in Section 4.1.3 has already given us an indication of the bit rate used for the motion vectors and for the frame data in the resultant bitstreams. The bit rate for the motion vectors is much smaller than that for the frame data for the full quality bitstream, but when the bitstream is scaled down to low quality, the bit rate for the motion vectors can become a significant part of the total bitstream. In this case, discarding some motion vector layers becomes essential, otherwise we can have more bits dedicated to the motion vectors than to the frame data itself.

The four high definition (HD) sequences of Table 4.1 were encoded using eight motion vector layers as described in Section 4.5.1. At high bit rates, all the eight layers should be retained. However, at low bit rates, using a large proportion of the bitstream for motion vectors will not give optimal results. Figure 4.10 shows how discarding motion vector layers affects the quality of the decoded sequences. Figure 4.11 shows the same thing for lower-resolution sequences, two 720p sequences and two CIF sequences.

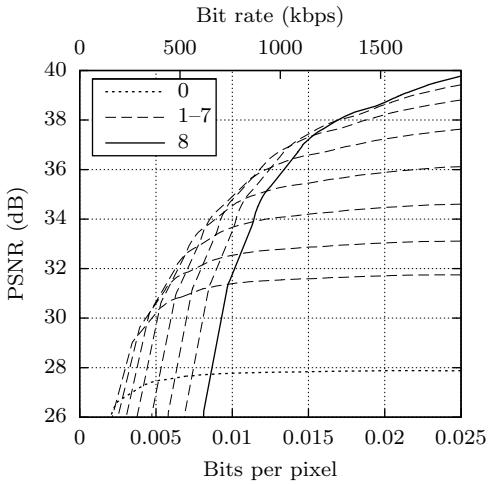
The curves labelled 8 show the quality when all eight layers are retained. In this case, when scaling the quality down to a certain point, almost all of the bitstream will be used up by the motion vectors, leading to very low values of PSNR. The curves labelled 0 show the quality when all motion vector layers are removed. In this case, the quality will never rise above a low value, because the temporal filtering is not aligned. The curves labelled



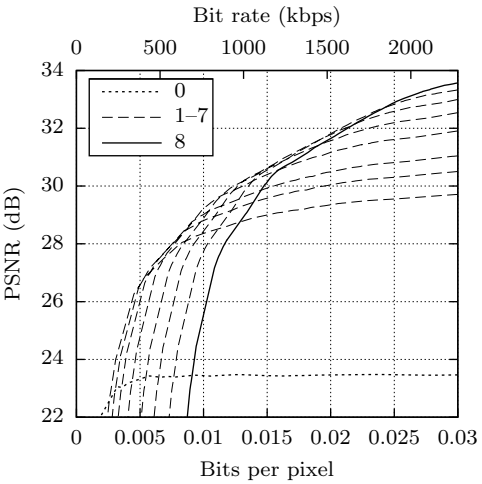
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)

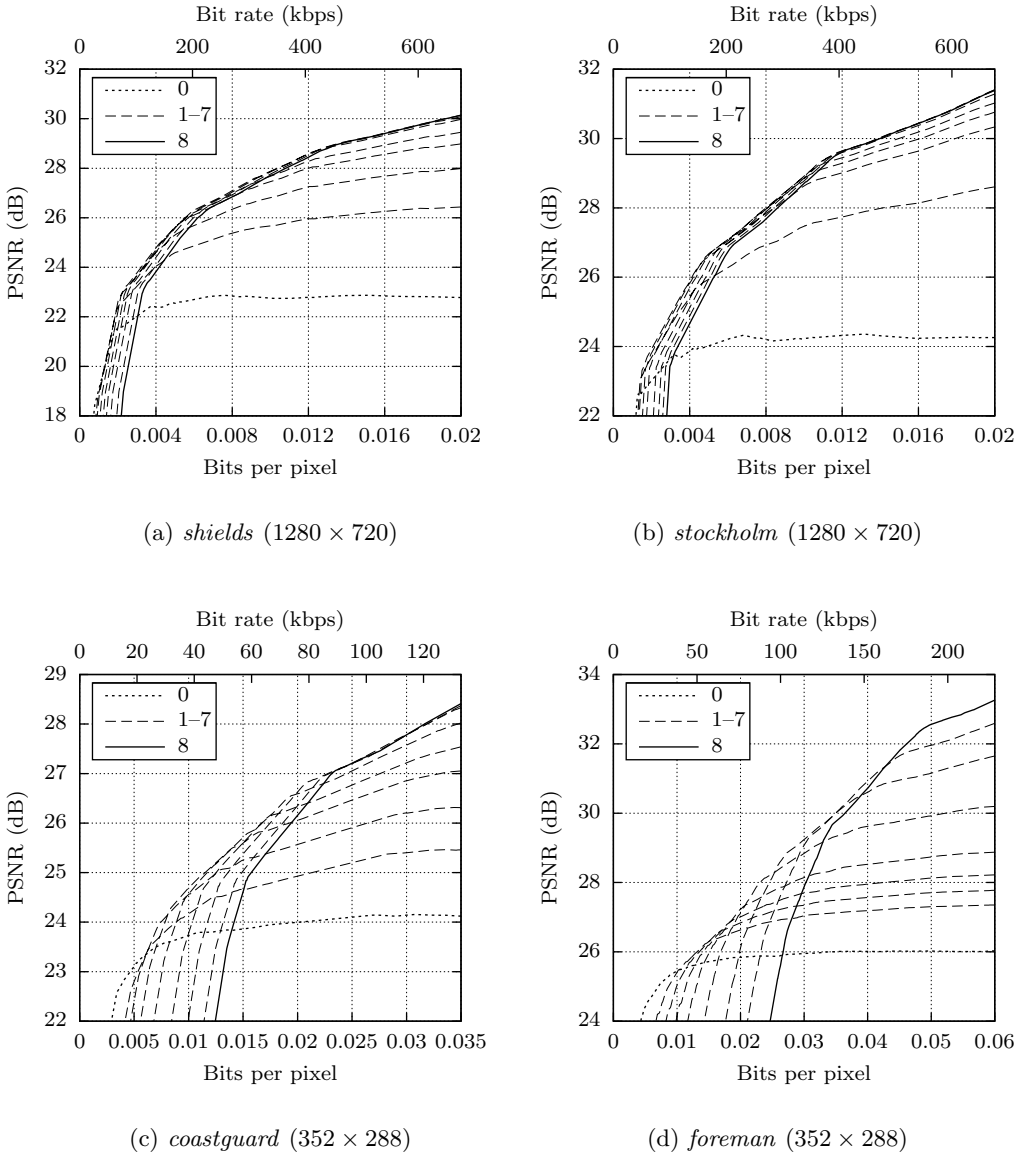


(c) *rush hour* (1920 × 1080)



(d) *tractor* (1920 × 1080)

Figure 4.10: Rate-distortion curves for the proposed SVC scheme with multi-layer motion vector palettes when retaining 0, 1, ..., 8 palette layers for HD video sequences.



(a) *shields* (1280 × 720)

(b) *stockholm* (1280 × 720)

(c) *coastguard* (352 × 288)

(d) *foreman* (352 × 288)

Figure 4.11: Rate-distortion curves for the proposed SVC scheme with multi-layer motion vector palettes when retaining 0, 1, ..., 8 palette layers for lower-resolution video sequences.

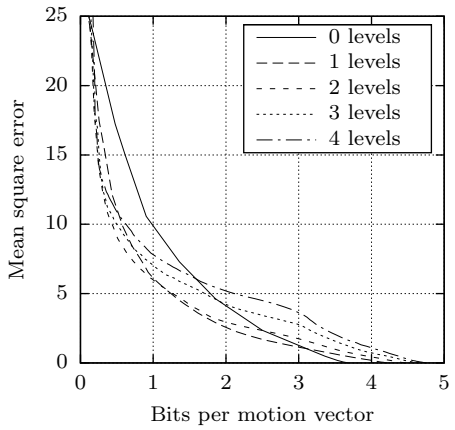
1–7 lie between the two extremes. Choosing the appropriate number of motion vector layers will thus give a resulting curve that encloses all the nine curves.

These curves show that eight layers are enough to obtain a smooth resultant curve which encloses the nine curves of each plot for a wide range of resolutions and for different kinds of video sequences. Using more layers will not give a curve which is much smoother, however, it will be more expensive in two ways. Firstly, it will need more processing to encode more layers, and secondly, using more layers will increase the bit rate overhead caused by the layering.

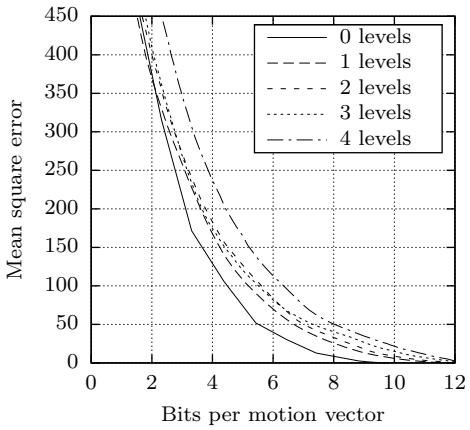
4.7 Palettes versus wavelets for motion vectors

In [65] and [19], motion vector scalability is achieved by quantizing the motion vectors using wavelets in a similar way to how images are encoded in EBCOT [11]. The effect of scaling motion vectors encoded using wavelets was investigated. Figure 4.12 shows how the mean square error of the motion vectors increases when the motion vector bitstream is scaled to a smaller size. The different curves are for different numbers of levels of wavelet decomposition. The best results are obtained for two levels of wavelet decomposition, except in the case of the (b) *riverbed* sequence. Again, this sequence behaves differently because neighbouring motion vectors are dissimilar owing to the nature of the video sequence, where the riverbed is seen through moving water. This dissimilarity makes the wavelet transform counter-productive, and that is why the best results are obtained when using no levels of wavelet decomposition.

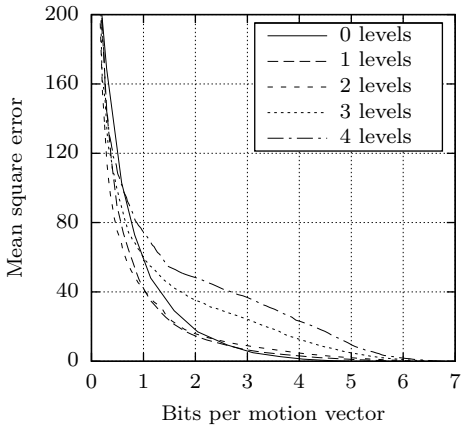
Figure 4.13 shows a comparison of this wavelet scheme to the motion vector palettes scheme presented in this chapter. For both, scaling down the motion vector bitstream increases the mean square error, but this increase is significantly smaller for our palette scheme. This is because scaling down the motion vectors using the wavelet scheme will affect all the motion vectors, while scaling down the motion vectors when using the palettes will not affect the most common motion vector values. The proposed algorithm has the advantage that it has more flexibility in choosing which motion vectors to be approximated (or quantized). If, for example, a particular motion vector is very common, it can be retained at a very high precision, while other motion



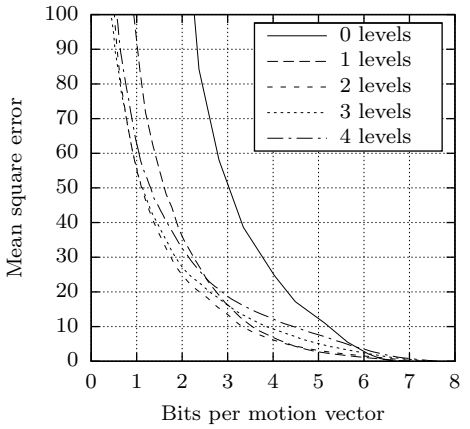
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)

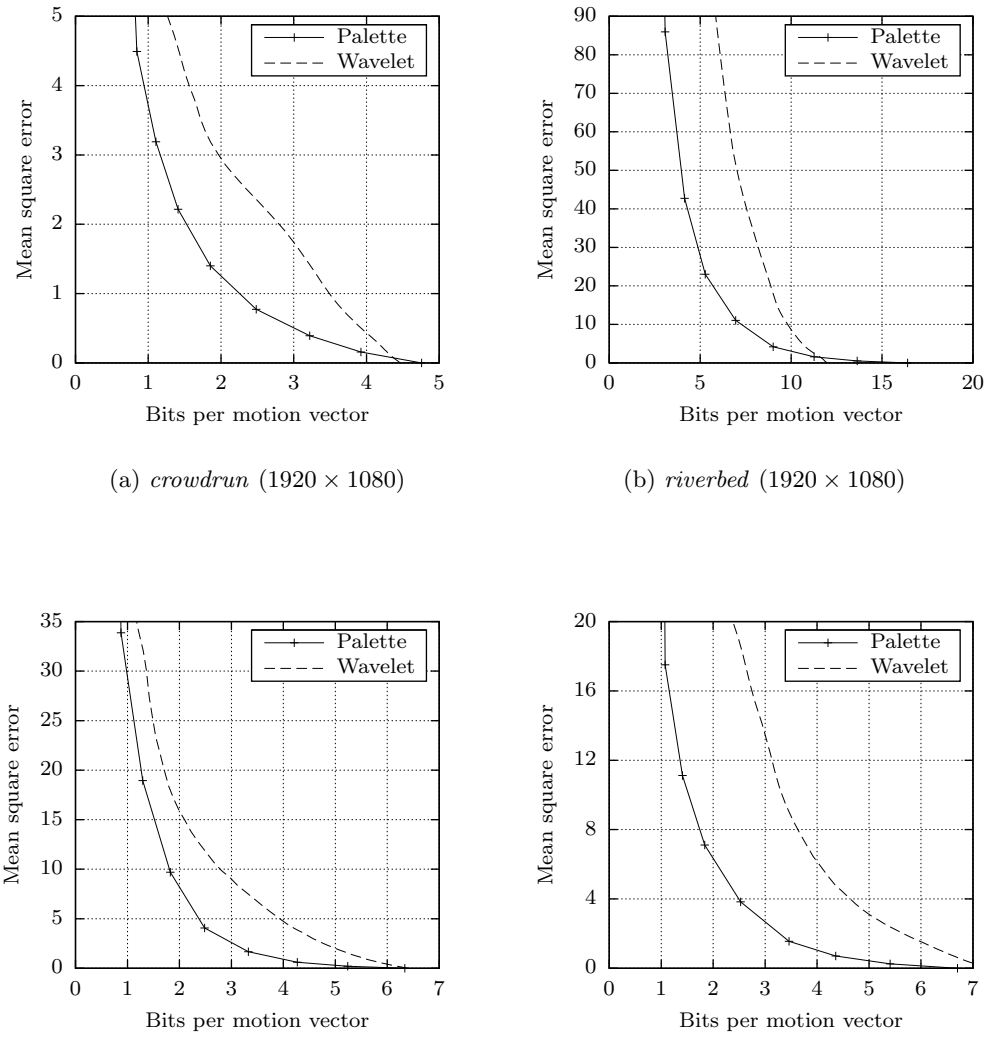


(c) *rush hour* (1920 × 1080)



(d) *tractor* (1920 × 1080)

Figure 4.12: The mean square error of the motion vectors when encoded using wavelets with a different number of levels.



(a) *crowdrun* (1920 × 1080)

(b) *riverbed* (1920 × 1080)

(c) *rush hour* (1920 × 1080)

(d) *tractor* (1920 × 1080)

Figure 4.13: Comparison of the mean square error of the motion vectors when encoded using palettes and wavelets.

vectors which are used very little are retained at a lower precision. Having a motion vector common to a high number of macroblocks is not unlikely; when a large visual object is moving, for instance, a lot of macroblocks will have similar motion vectors.

4.8 Conclusion

In this chapter, we have shown that while omitting motion vectors in the final video bitstream is a viable solution for lossless video coding, it is unsuitable for scalable video coding. A scheme for the scalable encoding of motion vectors using palettes was presented, together with analysis of its design and an investigation of its design parameters. The effect that scaling the motion vectors has on the rate distortion characteristics of the scalable video was investigated. The motion vector palette scheme was shown to outperform scalable motion vector encoding that uses wavelets as suggested by previous work.

Chapter 5

Scalable Video Coding

After finding motion vectors using the motion estimation engine, motion vector palettes are generated. Then, wavelet transforms are used to filter the frames in the temporal dimension and in the spatial dimensions. This chapter analyses a scalable video coding scheme that includes motion-compensated temporal filtering, wavelet spatial filtering, and the steps that follow.

Section 5.1 presents an analysis of the motion-compensated temporal filtering, including details about memory requirements and about the latency introduced. Section 5.2 describes how the temporally filtered frames are encoded using two-dimensional wavelet transforms, and investigates the effect of different wavelet filter types and different wavelet transform parameters.

5.1 Motion-compensated temporal filtering

The temporal filtering is performed using the 5/3 wavelet filter. As stated in Section 2.4.2, the 5/3 filter performs better than the simpler 1/3 filter. Larger filters will increase the complexity, the memory requirements, and the frame latency. Figure 5.1 shows one level of temporal filtering with lifting. When using lifting [85], the 5/3 filter at a level l, with decimation taken into account, is given by

$$H_{l,k} = L_{l-1,2k+1} - \left\lfloor \frac{L_{l-1,2k} + L_{l-1,2k+2}}{2} \right\rfloor \tag{5.1}$$

$$L_{l,k} = L_{l-1,2k} \quad + \left\lfloor \frac{H_{l,k-1} + H_{l,k} + 2}{4} \right\rfloor, \tag{5.2}$$

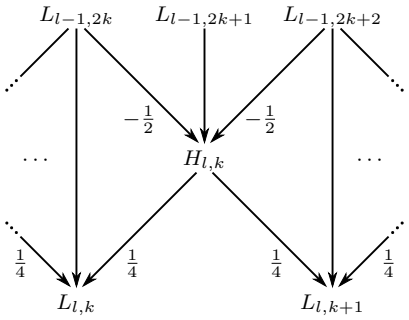


Figure 5.1: One level of temporal wavelet filtering using the 5/3 filter.

where $H_{l,k}$ is the kth high-pass coefficient and $L_{l,k}$ is the kth low-pass coefficient. The high-pass step (5.1) is the prediction step and the low-pass step (5.2) is the update step. $H_{l,k}$ and $L_{l,k}$ refer to whole frames, and motion compensation is needed to align the pixels for filtering.

5.1.1 Memory requirements for temporal filtering

Figure 5.2 shows three levels of temporal wavelet decomposition, with the original frames denoted as F_0, F_1, ... Computing $L_{2,0}$ depends on $H_{2,0}$, which in turn depends on $L_{1,2}$, which depends on $H_{1,2}$, on $L_{0,6}$, on $H_{0,6}$ and on F_{14}. This shows that one level of temporal wavelet decomposition requires a latency of two frames, two levels require a latency of six frames, and three levels require a latency of 14 frames. Also, buffering a number of frames is necessary. For one level, a buffer with capacity for four frames can be used, a buffer with capacity for eight frames can be used for two levels, and a buffer with capacity for 16 frames can be used for three levels.

Figure 5.3 shows how a buffer size of 16 frames is sufficient for three levels of wavelet filtering. Each group of steps processes eight frames, and consists of five steps:

a. Input: eight frames are read into the buffer,

b. Level 1: the first level of wavelet decomposition processes the eight frames and produces four low-pass and four high-pass frames,

c. Level 2: the second level of decomposition processes the four low-pass frames from the previous step and produces two low-pass and two high-pass frames,

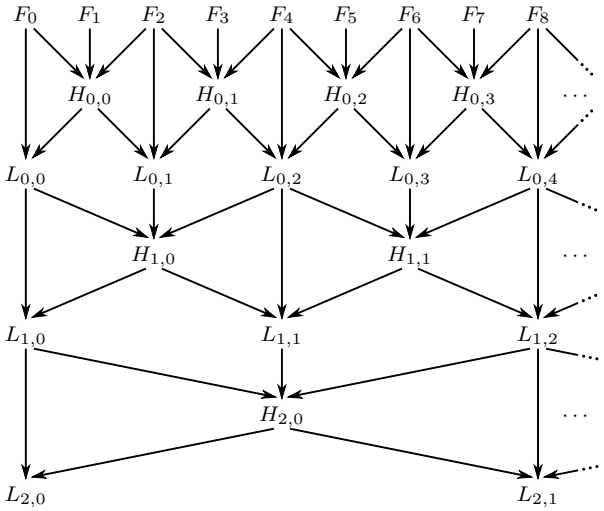


Figure 5.2: Three levels of temporal decomposition for the first frames.

d. Level 3: the third level of decomposition processes the two low-pass frames from the previous step and produces one low-pass and one high-pass frame, and

e. Output: eight temporally decomposed frames are passed on to the next stage.

In Step 1a, the first eight frames F_0, ..., F_7 are read into the buffer. In Step 1b, the first level of filtering produces $H_{0,0}$, $H_{0,1}$, $H_{0,2}$ and $L_{0,0}$, $L_{0,1}$, $L_{0,2}$. No new buffer space is required for the H_0 and L_0 frames, as $H_{0,0}$ can replace F_1, $L_{0,0}$ can replace F_0, and so on, such that F_0, ..., F_5 are replaced. In Step 1c, the second level replaces $L_{0,0}$, $L_{0,1}$ with $H_{1,0}$ and $L_{1,0}$. For Step 1d, the third level does not have enough inputs yet, and, for Step 1e, there are not enough frames to output.

Next, in Step 2a, the eight frames F_8, ..., F_{15} are read into the last eight frame locations of the buffer. In Step 2b, the first level of filtering replaces the eight frames F_6, ..., F_{13} with $H_{0,3}$, ..., $H_{0,6}$ and $L_{0,3}$, ..., $L_{0,6}$. In Step 2c, the second level replaces the four frames $L_{0,2}$, ..., $L_{0,5}$ with $H_{1,1}$, $H_{1,2}$ and $L_{1,1}$, $L_{1,2}$. In Step 2d, the third level replaces the two frames $L_{1,0}$, $L_{1,1}$ with $H_{2,0}$ and $L_{2,0}$. Now, as shown in Step 2e, the eight temporally filtered frames $L_{2,0}$, $H_{2,0}$, $H_{1,0}$, $H_{1,1}$, $H_{0,0}$, $H_{0,1}$, $H_{0,2}$, $H_{0,3}$ are ready for the next coding stages, that is, spatial wavelet filtering, bit plane coding, context modelling, and arithmetic coding.

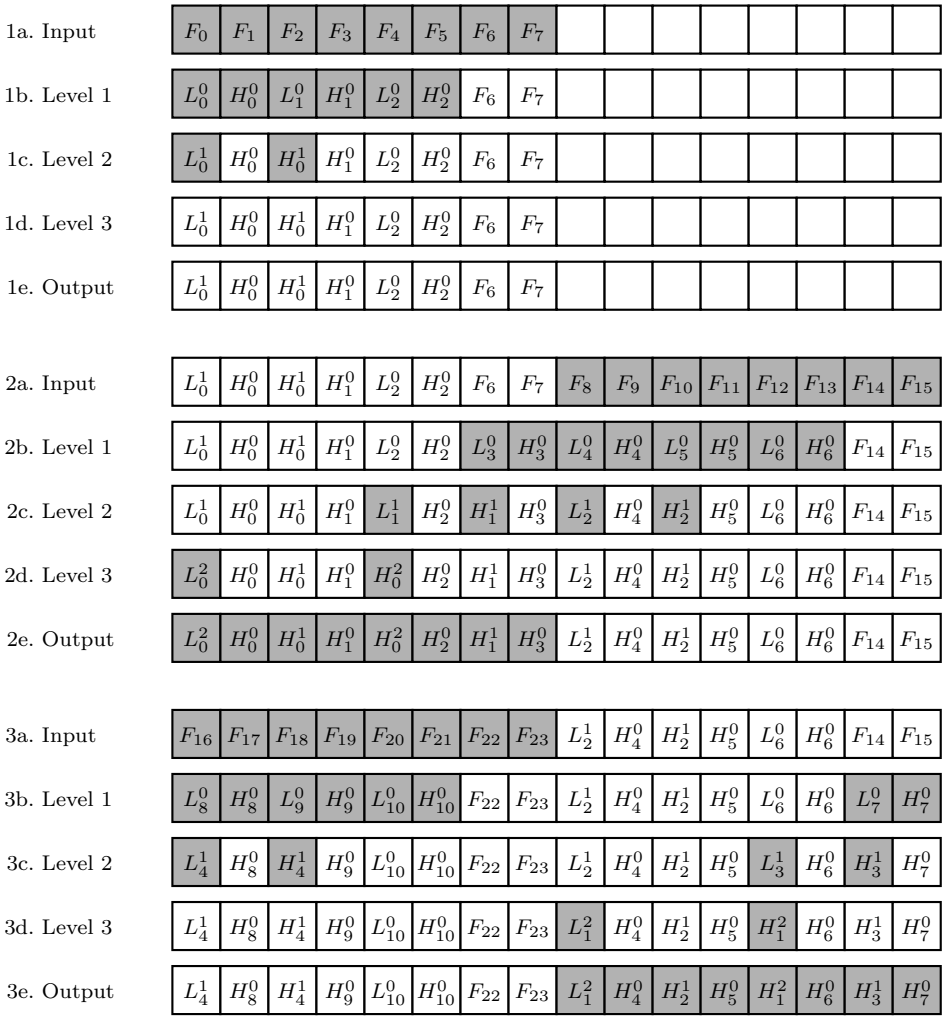


Figure 5.3: The contents of the 16-frame buffer for three levels of temporal wavelet decomposition.

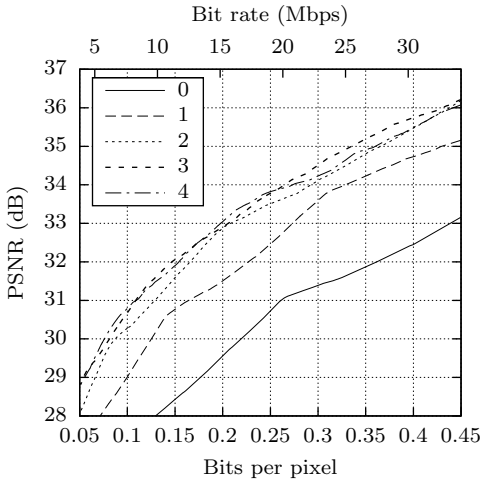
Steps 3a–3e show the next steps, when frames F_{16} ..., F_{23} are read and filtered. The buffer is used as a circular buffer.

We have seen that a buffer with capacity for 16 frames is required for three levels of temporal filtering. In general, for L levels, 2^{L+1} frames need to be held in memory, and memory requirements grow exponentially with L. In practice, choosing three levels was found to give an adequate balance between memory requirements and compression performance; using four levels does not improve compression performance enough to compensate for using twice as much memory as three levels, and doubling the latency.

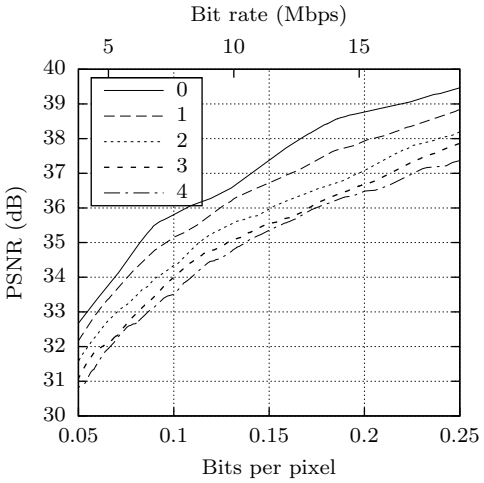
Figure 5.4 shows rate-distortion curves for various different levels of temporal wavelet decomposition, beginning from 0 for no temporal filtering, and ending at 4 for four levels of temporal decomposition. As can be seen from the graphs, three levels of decomposition is a good choice. Note that the (b) *riverbed* sequence shows degradation on temporal filtering, indicating that the sequence is difficult to encode using temporal filtering. The sequence consists of a riverbed as seen through moving water, which makes it difficult to match macroblocks from a frame to another frame, and thus, temporal filtering is counter-productive in this case.

Other than buffering the pixel data, the motion vectors generated by the motion estimation need to be buffered as well. Motion estimation is performed before the prediction and update steps so that the original unfiltered frames can be used in the search. For the first eight frames, motion estimation is performed between Steps 1a and 1b of Section 5.1.1. For the first level, motion estimation is performed on F_1 with F_0 as a backwards reference frame and F_2 as a forwards reference frame. These searches are independent, and motion vectors from each have to be stored. Similarly, motion estimation is done on F_3 with reference frames F_2 and F_4, on F_5 with reference frames F_4 and F_6, and on F_7 with reference frames F_6 and F_8. For the second level, the motion estimation search is performed on F_2 with reference frames F_0 and F_4 and on F_6 with reference frames F_4 and F_8. For the third level, motion estimation is performed on F_4 with reference frames F_0 and F_8. All of these motion estimation searches are performed between Steps 1a and 1b. Note that frame F_8 was required for some of these, so although it is not included in Step 1a in Figure 5.3, one more frame has to be stored before motion estimation.

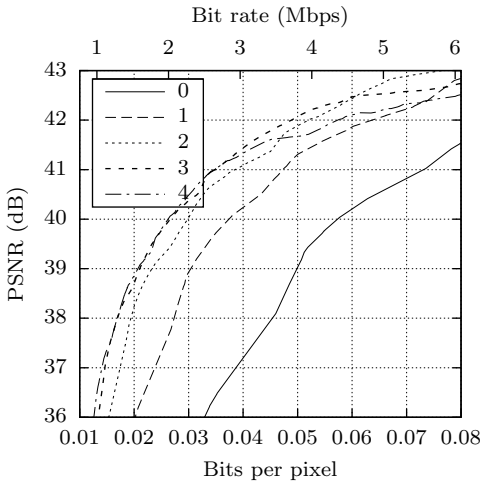
For each pixel frame, we need to buffer two motion vector frames. So



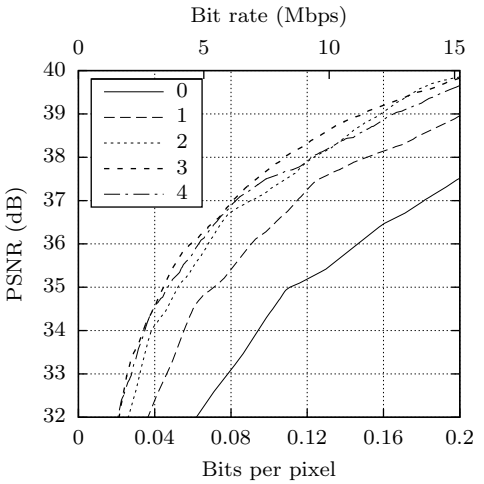
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)



(c) *rush hour* (1920 × 1080)



(d) *tractor* (1920 × 1080)

Figure 5.4: Rate-distortion curves for different levels of temporal wavelet decomposition.

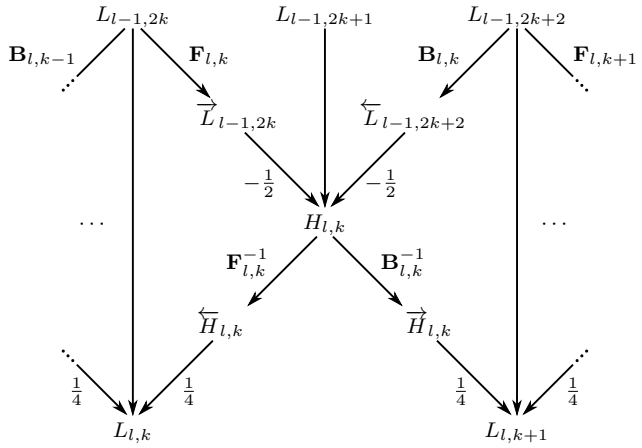


Figure 5.5: One level of temporal wavelet filtering using motion compensation and the 5/3 filter.

for L levels, as well as the 2^{L+1} pixel frames, we need 2^{L+2} motion vector frames. Although there are twice as many motion vector frames as pixel frames, motion vectors take much less space than the pixels. For example, using 16×16 macroblocks, there will be one motion vector every 256 pixels for one frame of values.

5.1.2 Motion compensation for the prediction step

If we introduce motion compensation into Figure 5.1, we get Figure 5.5. To compute $H_{l,k}$ in the prediction step, motion compensation needs to be applied to two frames, the previous frame $L_{l-1,2k}$ and the next frame $L_{l-1,2k+2}$, in order to align their pixels with corresponding pixels in $L_{l-1,2k+1}$. $\mathbf{F}_{l,k}$ denotes the operation that is applied to the previous frame $L_{l-1,2k}$ (moving it Forwards), and $\mathbf{B}_{l,k}$ denotes the operation that is applied to the next frame $L_{l-1,2k+2}$ (moving it Backwards). To compute $L_{l,k}$ in the update step, the operation $\mathbf{B}_{l,k-1}^{-1}$, which is the inverse of $\mathbf{B}_{l,k-1}$, is applied to $H_{l,k-1}$, and $\mathbf{F}_{l,k}^{-1}$ is applied to $H_{l,k}$. Similarly, to compute $L_{l,k+1}$, $\mathbf{B}_{l,k}^{-1}$ is applied to $H_{l,k}$, and $\mathbf{F}_{l,k+1}^{-1}$ is applied to $H_{l,k+1}$.

If

$$\overrightarrow{L}_{l-1,2k} = \mathbf{F}_{l,k}(L_{l-1,2k}) \tag{5.3}$$

$$\overleftarrow{L}_{l-1,2k+2} = \mathbf{B}_{l,k}(L_{l-1,2k+2}), \tag{5.4}$$

Algorithm 5.1 An algorithm to generate $\overrightarrow{L}_{l-1,2k} = \mathbf{F}_{l,k}(L_{l-1,2k})$.

1: **for all** (x, y) **do**
2: $(x', y') \leftarrow (x, y) + (\delta x_{\mathbf{F}}, \delta y_{\mathbf{F}})$
3: $\overrightarrow{L}_{l-1,2k}(x, y) \leftarrow L_{l-1,2k}(x', y')$
4: **end for**

then motion compensation can be introduced into (5.1) by rewriting it as

$$H_{l,k} = L_{l-1,2k+1} - \left\lfloor \frac{\overrightarrow{L}_{l-1,2k} + \overleftarrow{L}_{l-1,2k+2}}{2} \right\rfloor. \tag{5.5}$$

For each location (x, y) in the frame $L_{l-1,2k+1}$, we search for a forwards motion vector $(\delta x_{\mathbf{F}}, \delta y_{\mathbf{F}})$ such that the pixel at $(x + \delta x_{\mathbf{F}}, y + \delta y_{\mathbf{F}})$ in the previous frame $L_{l-1,2k}$ corresponds to the pixel (x, y) in $L_{l-1,2k+1}$. Similarly, $(\delta x_{\mathbf{B}}, \delta y_{\mathbf{B}})$ is the backwards motion vector to map pixels from the next frame $L_{l-1,2k+2}$. Notice that we do not perform one motion estimation search for each pixel (x, y), but only once for each macroblock, so that for all the values of (x, y) in one macroblock, there will be only one $(\delta x_{\mathbf{F}}, \delta y_{\mathbf{F}})$ and one $(\delta x_{\mathbf{B}}, \delta y_{\mathbf{B}})$. Once we have the motion vectors, the operations $\mathbf{F}_{l,k}$ and $\mathbf{B}_{l,k}$ in (5.3) and (5.4) can be written as

$$\overrightarrow{L}_{l-1,2k}(x, y) = L_{l-1,2k}(x + \delta x_{\mathbf{F}}, y + \delta y_{\mathbf{F}}) \tag{5.6}$$

$$\overleftarrow{L}_{l-1,2k+2}(x, y) = L_{l-1,2k+2}(x + \delta x_{\mathbf{B}}, y + \delta y_{\mathbf{B}}). \tag{5.7}$$

The process to generate $\overrightarrow{L}_{l-1,2k}$ is shown in Algorithm 5.1, and a similar process can be used to generate $\overleftarrow{L}_{l-1,2k+2}$.

5.1.3 Motion compensation for the update step

To compute $L_{l,k}$, we need to apply the operation $\mathbf{B}^{-1}_{l,k-1}$ to $H_{l,k-1}$ and the operation $\mathbf{F}^{-1}_{l,k}$ to $H_{l,k}$. If

$$\overrightarrow{H}_{l,k-1} = \mathbf{B}^{-1}_{l,k-1}(H_{l,k-1}) \tag{5.8}$$

$$\overleftarrow{H}_{l,k} = \mathbf{F}^{-1}_{l,k}(H_{l,k}), \tag{5.9}$$

then motion compensation can be introduced into (5.2) by rewriting it as

$$L_{l,k} = L_{l-1,2k} + \left\lfloor \frac{\overrightarrow{H}_{l,k-1} + \overleftarrow{H}_{l,k} + 2}{4} \right\rfloor . \tag{5.10}$$

The operations $\mathbf{B}^{-1}$ and $\mathbf{F}^{-1}$ are not as straightforward as the operations $\mathbf{F}$ and $\mathbf{B}$. The process to generate $\overleftarrow{H}_{l,k}$ is shown in Algorithm 5.2. The algorithm maps the values in $H_{l,k}$ back to match the pixels in $L_{l-1,2k}$ using the motion vectors $(\delta x_{\mathbf{F}}, \delta y_{\mathbf{F}})$. However, different macroblocks can be mapped back to the same area. That is, there can be two pixels (x_a, y_a) and (x_b, y_b) that satisfy

$$(x_a, y_a) + (\delta x_{a\,\mathbf{F}}, \delta y_{a\,\mathbf{F}}) = (x_b, y_b) + (\delta x_{b\,\mathbf{F}}, \delta y_{b\,\mathbf{F}}). \tag{5.11}$$

To handle this many-to-one mapping, all the values in $\overleftarrow{H}_{l,k}$ are initially marked as invalid, as shown in the loop at Lines 1–3. When a pixel is mapped back to a value that is still invalid, the pixel value replaces the invalid value as shown in Line 7. When a pixel is mapped back to a value that is already valid, the mean of the current valid value and the pixel value replaces the current valid value, as shown in Line 9. At the end, any invalid values are set to zero, as shown in the loop at Lines 12–16. A similar process can be used to generate $\overrightarrow{H}_{l,k-1}$.

Let us look back at how collisions are handled in the algorithm. If, for example, two pixels in $H_{l,k}$ with values a and b are mapped to the same pixel in $\overleftarrow{H}_{l,k}$, a will be stored directly using Line 7, and b will be processed using Line 9. When b is processed, Line 9 will replace the current value a with the value $^{(a+b)}/_2 = {}^{a}/_2 + {}^{b}/_2$.

Now suppose that three pixels with values a, b and c are mapped to the same pixel in $\overleftarrow{H}_{l,k}$. As before, a will be stored directly using Line 7, and when b is processed, the value a will be replaced by ${}^{a}/_2 + {}^{b}/_2$ using Line 9. When c is processed, the value is replaced by $^{({}^{a}/_2+{}^{b}/_2+c)}/_2 = {}^{a}/_4 + {}^{b}/_4 + {}^{c}/_2$, which is only an approximation of the real average value ${}^{a}/_3 + {}^{b}/_3 + {}^{c}/_3$. This approximation reduces the complexity of the implementation without having much effect on the result, especially when considering that most many-to-one collisions that happen will only be two-to-one, which is handled accurately. This approximation is particularly useful for hardware implementation, where division by two is a simple shift operation, and division

Algorithm 5.2 An algorithm to generate $\overleftarrow{H}_{l,k} = \mathbf{F}_{l,k}^{-1}(H_{l,k})$.

1: **for all** (x, y) **do** {loop to initialize all values as invalid}
2: $\quad\overleftarrow{H}_{l,k}(x, y) \leftarrow \wedge$ {$\wedge$ is the invalid value}
3: **end for**
4: **for all** (x, y) **do** {the main loop}
5: $\quad(x', y') \leftarrow (x, y) + (\delta x_{\mathbf{F}}, \delta y_{\mathbf{F}})$
6: $\quad$**if** $\overleftarrow{H}_{l,k}(x', y') = \wedge$ **then**
7: $\quad\quad\overleftarrow{H}_{l,k}(x', y') \leftarrow H_{l,k}(x, y)$
8: $\quad$**else**
9: $\quad\quad\overleftarrow{H}_{l,k}(x', y') \leftarrow \left\lceil \dfrac{H_{l,k}(x, y) + \overleftarrow{H}_{l,k}(x', y')}{2} \right\rceil$
10: $\quad$**end if**
11: **end for**
12: **for all** (x, y) **do** {loop to set all invalid values to 0}
13: $\quad$**if** $\overleftarrow{H}_{l,k}(x, y) = \wedge$ **then**
14: $\quad\quad\overleftarrow{H}_{l,k}(x, y) \leftarrow 0$
15: $\quad$**end if**
16: **end for**

by other numbers can be relatively expensive.

5.1.4 Temporal filtering after motion estimation

After the pixel matching steps of Sections 5.1.2 and 5.1.3 are performed, wavelet decomposition is performed using regular wavelet lifting as introduced in Section 2.2.2. This modifies the contents of the 16-frame buffer as described in Section 5.1.1. The Output steps of Figure 5.3 will pass the data to the next steps, that is, to spatial filtering, context modelling, and probability estimation and arithmetic coding.

5.2 Encoding the temporally filtered frames

The temporally filtered frames need to be encoded using wavelet transforms, context modelling, probability estimation and arithmetic coding.

5.2.1 Spatial filtering

The first step after motion compensation is the spatial filtering of the frames. There are two possible wavelet filters, the 5/3 filter and the 9/7 filter. The

5/3 filter uses integer arithmetic and is reversible, such that it can provide lossless quality. The 9/7 filter uses fixed point or floating point arithmetic and consequently has rounding errors, making it non-reversible and unsuitable for lossless encoding. The 5/3 filter outperforms the 9/7 filter at high bit rates where the rounding errors are significant, while the 9/7 filter is better at lower bit rates where the rounding errors are not significant and its better filter characteristics are more significant. Figure 5.6 shows how the 9/7 filter outperforms the 5/3 filter at low bit rates, and Figure 5.7 shows how the situation is reversed at high bit rates.

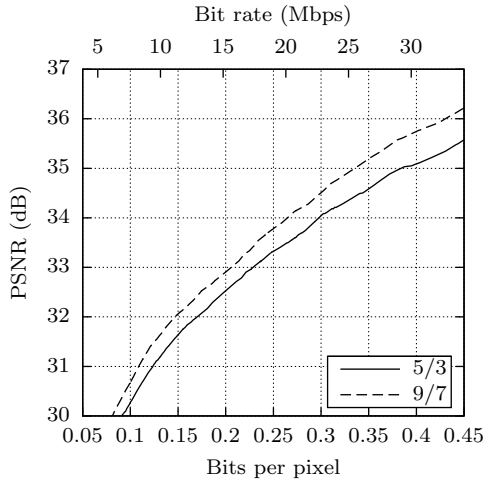
Figure 5.8 shows the performance of the scalable video coding (SVC) scheme for different levels of spatial decomposition, starting from zero levels where no spatial decomposition is performed, up to four levels of decomposition. Increasing the levels from zero to one, to two, and to three gives significant gains in the rate-distortion curves. However, there is no clear gain for using four levels of decomposition instead of three. In this work, three levels of spatial decomposition are used.

5.2.2 Context modelling

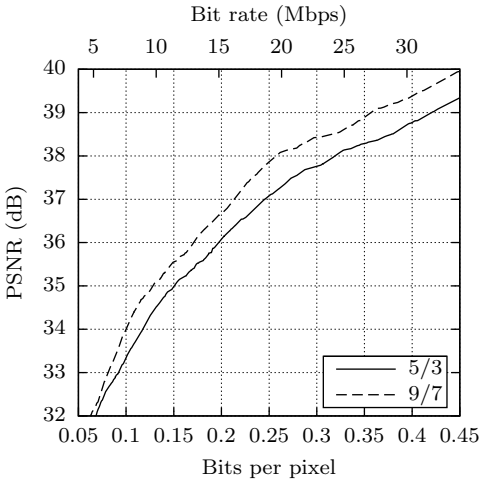
After the spatial wavelet transform, bit plane coding and context modelling are performed. These steps include encoding the significance of code blocks and bit plane coding techniques from EBCOT [11], together with context modelling as used by ICER [12]. The details of these parts can be seen in Section 2.3.1.

5.2.3 Probability estimation

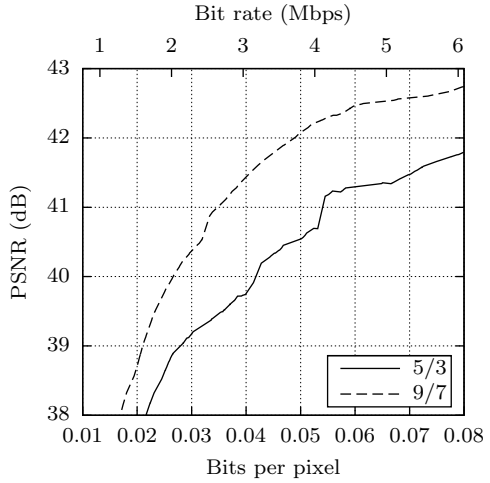
The wavelet coding algorithm uses bit plane coding, which only needs a one-bit probability estimator. For probability estimation, a one-bit adaptive probability estimator is used. The probability estimate for the next bit will depend on the previous bits. We need to find the probability of the next bit being '0' or '1'. Let m be the number of previous '0's, n be the number of previous '1's, and x be the probability that the next bit will be '0', $x = P(\text{'0'})$, $0 \leq x \leq 1$. If we assume that the value x remains constant throughout the bit sequence, we can use Bayes' theorem to write



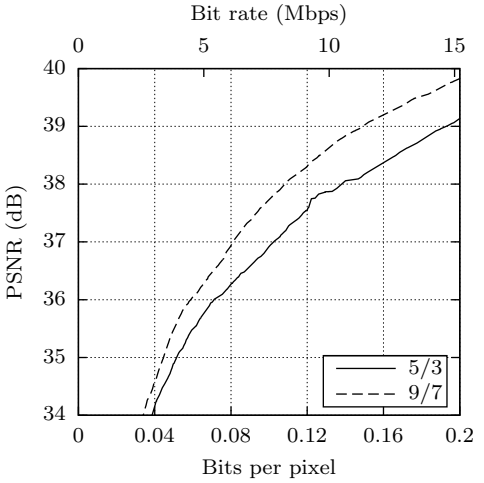
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)

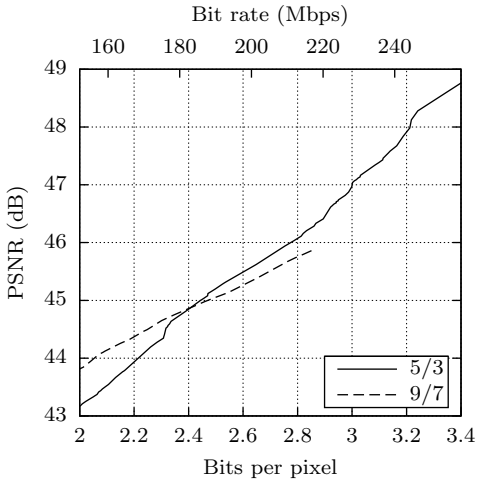


(c) *rush hour* (1920 × 1080)

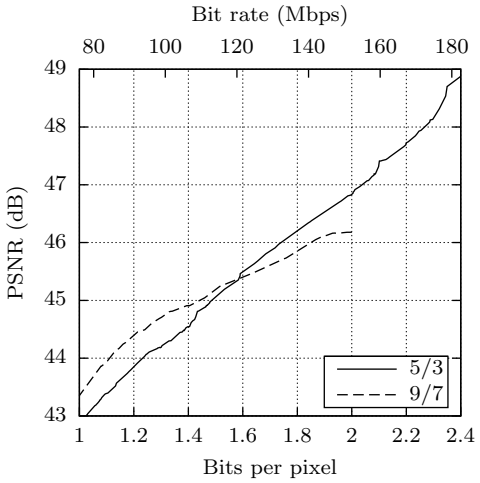


(d) *tractor* (1920 × 1080)

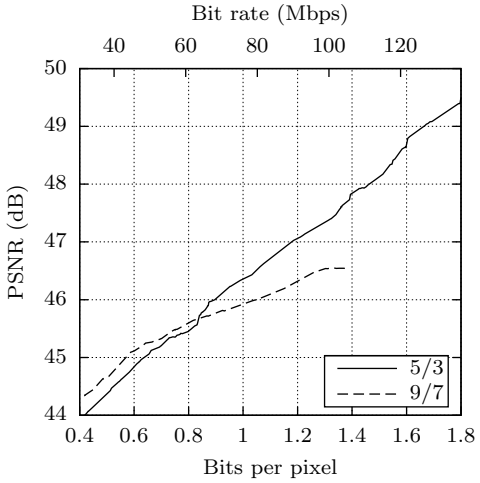
Figure 5.6: Rate-distortion curves for the 5/3 and 9/7 wavelet filters for low bit rates.



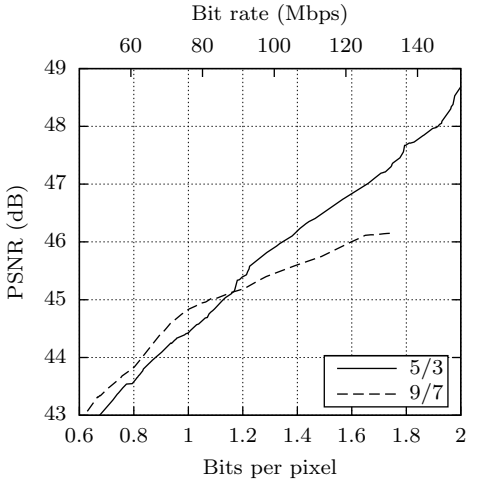
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)

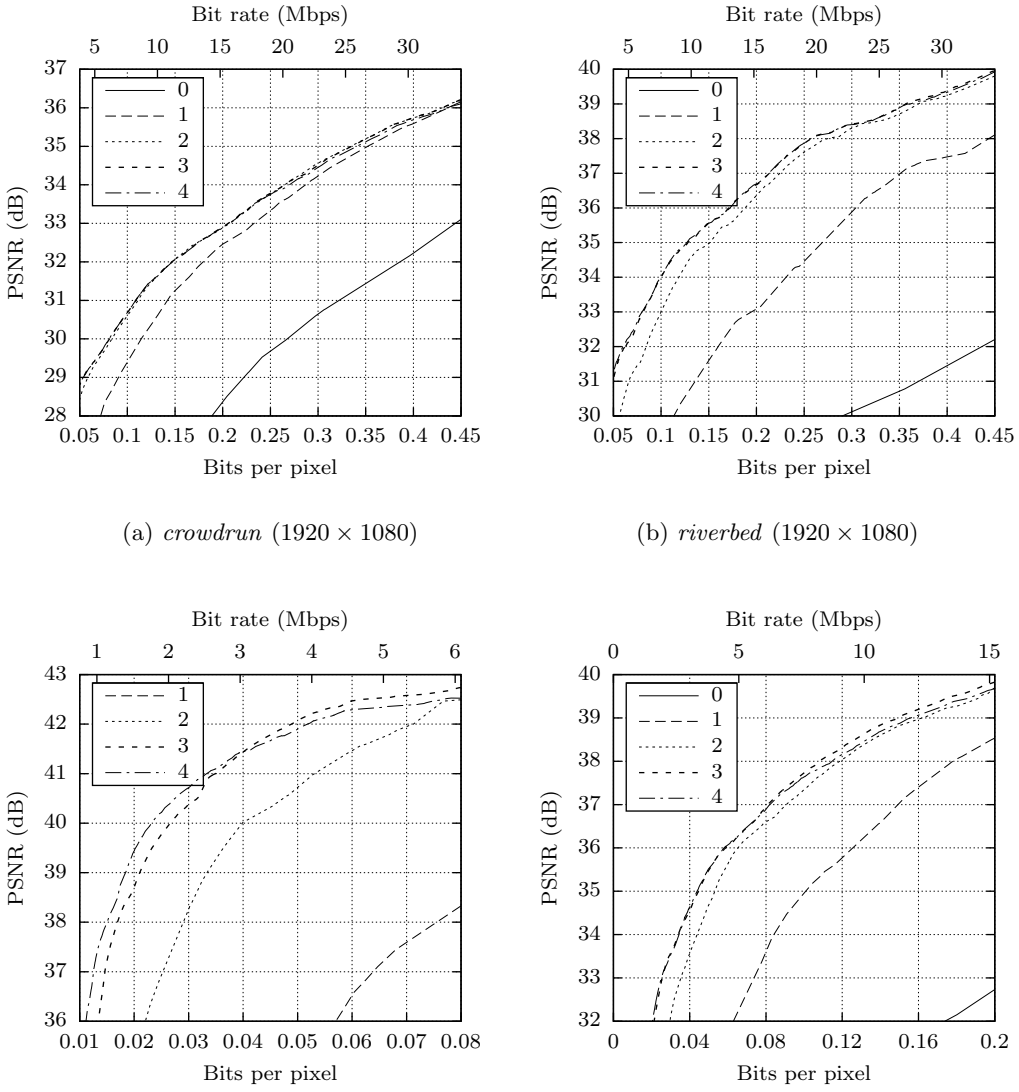


(c) *rush hour* (1920 × 1080)



(d) *tractor* (1920 × 1080)

Figure 5.7: Rate-distortion curves for the 5/3 and 9/7 wavelet filters for high bit rates.



(a) *crowdrun* (1920 × 1080)

(b) *riverbed* (1920 × 1080)

(c) *rush hour* (1920 × 1080)

(d) *tractor* (1920 × 1080)

Figure 5.8: Rate-distortion curves for different levels of spatial wavelet decomposition.

the probability distribution for x given the values m and n as

$$P(x \mid m, n) = \frac{P(x)P(m, n \mid x)}{P(m, n)}, \tag{5.12}$$

where $P(x)$ is the a priori probability of x, $P(m, n \mid x)$ is the a posteriori probability of m and n, and $P(m, n)$ is a normalizing factor. We can assume that the a priori probability $P(x)$ is constant within the range $0 \leq x \leq 1$. The a posteriori probability $P(m, n|x)$ is given by

$$P(m, n \mid x) = x^m (1 - x)^n \binom{m + n}{m}. \tag{5.13}$$

Since $P(x)$, $P(m, n)$ and $\binom{m+n}{m}$ are all constant, we can rewrite (5.12) as

$$P(x \mid m, n) = k x^m (1 - x)^n. \tag{5.14}$$

where k is a constant. Further to this,

$$\int_0^1 P(x \mid m, n)\, dx = 1. \tag{5.15}$$

From (5.14) and (5.15), we can find

$$P(x \mid m, n) = \frac{(m + n + 1)!}{m!\, n!} x^m (1 - x)^n. \tag{5.16}$$

The probability that the next bit is '0' is thus

$$\begin{aligned} P(\text{`0'} \mid m, n) &= \int_0^1 x P(x \mid m, n)\, dx && (5.17) \\ &= \int_0^1 \frac{(m + n + 1)!}{m!\, n!} x^{m+1} (1 - x)^n\, dx && (5.18) \\ &= \frac{m + 1}{m + n + 2} && (5.19) \end{aligned}$$

To implement this probability estimator, all we need to do is to keep two counters; m for the number of '0's and n for the number of '1's. If the counters overflow, both are divided by two. This scaling down has the advantage that it keeps the probability estimation up do date with the adapting input, giving better results as described in [35].

5.3 Conclusion

In this chapter, we have presented the motion-compensated temporal filtering employed by our scalable video coding scheme. The memory requirements of the temporal filtering were analysed, and the effect of the number of levels of temporal filtering on memory requirements, latency and compression performance was investigated, showing that using three levels of temporal filtering gives a good balance of memory requirements, latency and compression performance. Spatial wavelet filtering was investigated as well, demonstrating the differences between different filter types and the effect of the number of levels of spatial decomposition on video sequences of different resolutions. The chapter concludes with an analysis of the probability estimation scheme that follows the context modelling.

Chapter 6

Performance Analysis

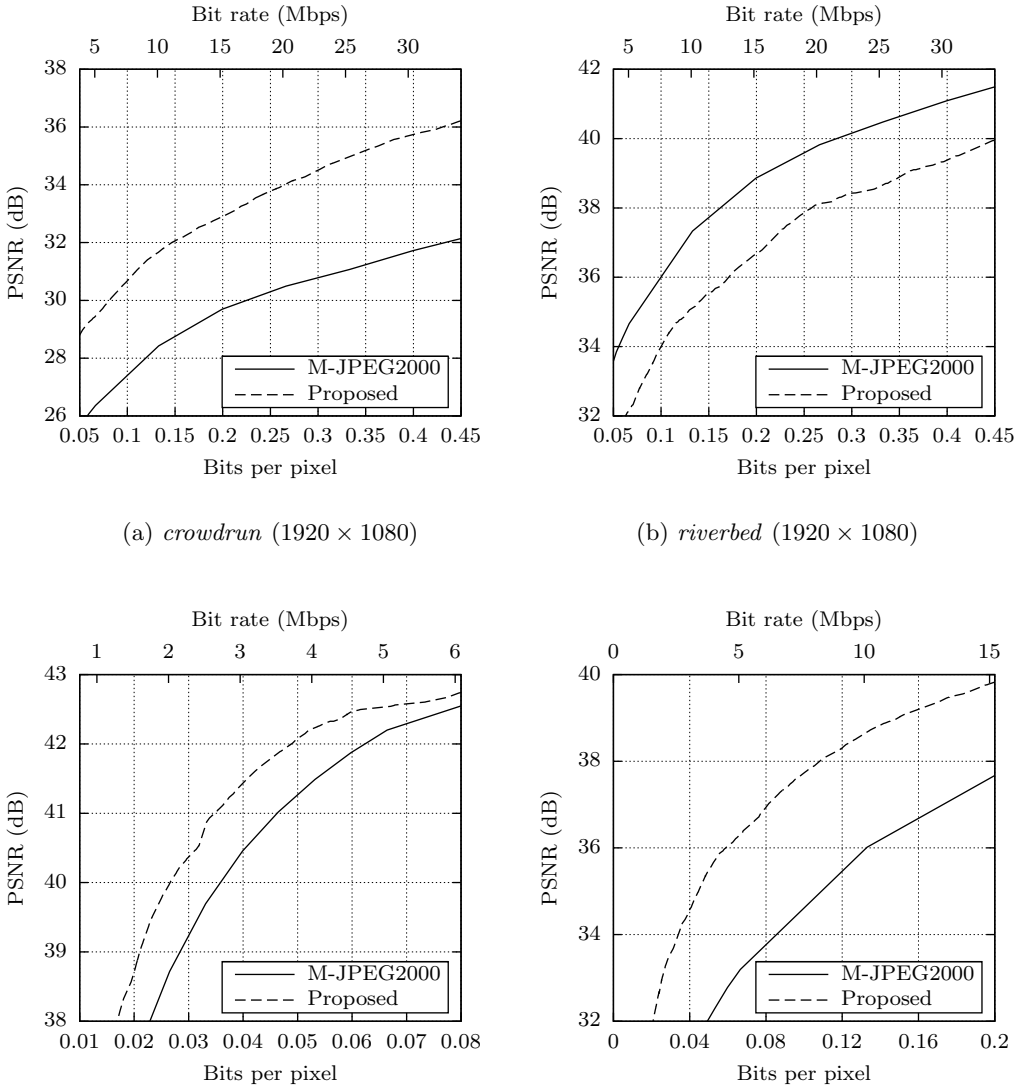
This chapter analyses the performance of the proposed scalable video coding scheme.

The performance is first compared to Motion JPEG 2000, which uses similar spatial coding but does not include motion compensation, in Section 6.1. Section 6.2 presents a comparison of the scalable video coding (SVC) scheme presented to Joint Scalable Video Model (JSVM), including memory requirements, execution duration, and compression performance comparisons.

6.1 Comparison to Motion JPEG 2000

In this section, the proposed scalable video coding scheme is compared to Motion JPEG 2000 using the Kakadu encoder [86], a wavelet-based video encoder that does not include motion compensation. Figure 6.1 shows the rate-distortion curves for the two schems for the four high definition (HD) sequences of Table 4.1. The proposed system outperforms Motion JPEG 2000 for almost all the video sequences, demonstrating the gains obtained using motion compensated temporal filtering. On (b) *riverbed*, however, Motion JPEG 2000 outperforms the proposed scheme. This is because the sequence shows the riverbed through moving water, which makes motion compensation less effective.

Figure 6.2 compares the proposed scheme to Motion JPEG 2000, this time for video sequences with a lower resolution. Motion JPEG 2000 exhibits poorer performance for all of these sequences by a larger margin than for



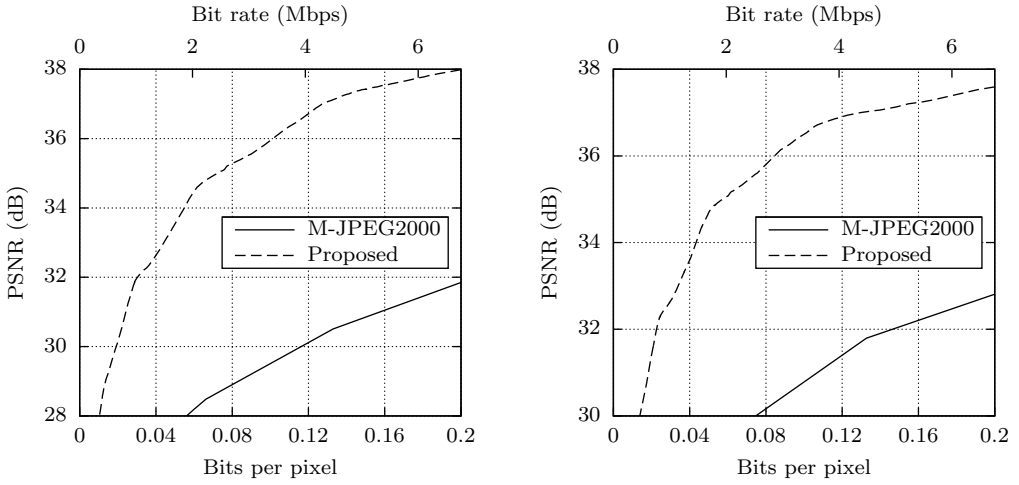
(a) *crowdrun* (1920 × 1080)

(b) *riverbed* (1920 × 1080)

(c) *rush hour* (1920 × 1080)

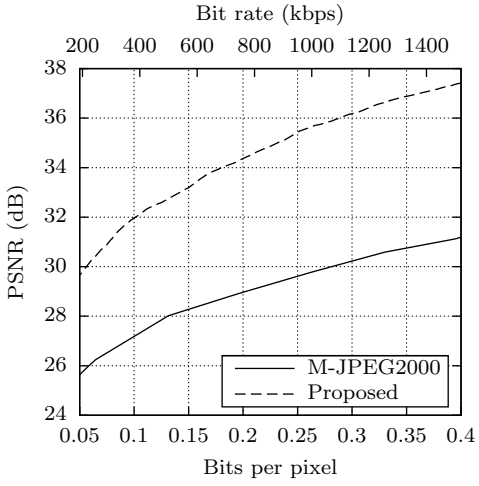
(d) *tractor* (1920 × 1080)

Figure 6.1: Rate-distortion curves for Motion JPEG 2000 and the proposed method.

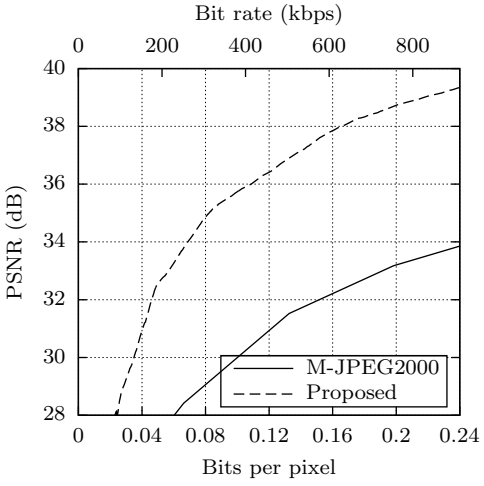


(a) *shields* (1280 × 720)

(b) *stockholm* (1280 × 720)



(c) *coastguard* (352 × 288)



(d) *foreman* (352 × 288)

Figure 6.2: Rate-distortion curves for Motion JPEG 2000 and the proposed method for lower resolution sequences.

the sequences in Figure 6.1, indicating that motion compensation is more important for lower resolutions.

6.2 Comparison to JSVM

In this section, the proposed scheme is compared to the reference JSVM software for SVC version 9.19.12 [5], as implementations for other wavelet methods are not as readily available. The video sequences of Table 4.1 were used. Only quality scalability is compared here. The JSVM software was configured so that its motion estimation search is similar to that currently used by our scheme. The JSVM software only supports coarse-grained and medium-grained scalability, and it was configured with three coarse-grained scalability layers. The base layer, Layer 0, and the refinement layers, Layers 1 and 2, were configured to have a quantization parameter (QP) of 38, 32 and 26 respectively. The refinement layers were each configured to have four levels of medium-grained scalability. The parameters used for the JSVM software are listed in Table 6.1. Our scheme supports fine-grained scalability and does not need configuration of different layers. For both systems, three temporal layers were used. For the proposed scheme, three levels of the 9/7 two-dimensional wavelet filter were used, as suggested in Section 5.2.1. Notice that while JSVM needs to be given some detailed configuration about the encoding process, the proposed scheme only needs a few parameters.

6.2.1 Memory requirements and execution duration

The memory requirements and processing time of the algorithm was measured and compared. Both the proposed scheme and JSVM were compiled using the GCC C++ compiler version 4.7.0 [87] with full optimization and run on an Intel Core i5-760 at 2.8 GHz [88]. Memory usage was measured using the Massif heap profiler, a tool from the Valgrind suite of tools, version 3.7.0 [89]. Table 6.2 shows the memory requirements for JSVM and the proposed method. Our method uses less memory by a factor of more than 7 for both high resolution sequences and low resolution sequences. Table 6.3 shows the processing time. Again, our method outperforms JSVM by a large factor ranging from 25 to over 100.

Table 6.1: The parameters used for the JSVM encoder.

Profile	
GOPSize	8
BaseLayerMode	2
SearchMode	4
SearchRange	128
CgsSnrRefinement	1
EncodeKeyPictures	1
MGSControl	1
NumLayers	3

Layer 0	
MGSVectorMode	0
LowComplexityMbMode	1
MeQPLP	36
MeQP0–MeQP5	36
QP	38

Layer 1	
InterLayerPred	1
MGSVectorMode	1
MGSVector0	2
MGSVector1	2
MGSVector2	4
MGSVector3	8
LowComplexityMbMode	1
MeQPLP	30
MeQP0–MeQP5	30
QP	32

Layer 2	
InterLayerPred	1
MGSVectorMode	1
MGSVector0	2
MGSVector1	2
MGSVector2	4
MGSVector3	8
LowComplexityMbMode	1
MeQPLP	24
MeQP0–MeQP5	24
QP	26

Table 6.2: Memory usage for JSVM and the proposed method.

Sequence	Resolution	JSVM	Proposed	Factor
crowdrun	1920×1080	2.41 GiB	320 MiB	7.7
riverbed	1920×1080	2.40 GiB	321 MiB	7.7
rush hour	1920×1080	2.40 GiB	320 MiB	7.7
tractor	1920×1080	2.40 GiB	320 MiB	7.7
shields	1280×720	1.10 GiB	146 MiB	7.7
stockholm	1280×720	1.10 GiB	146 MiB	7.7
coastguard	352×288	156 MiB	17.2 MiB	9.1
foreman	352×288	156 MiB	17.2 MiB	9.0

TABLE 6.3: PROCESSING TIME FOR JSVM AND THE PROPOSED METHOD.

Sequence	Resolution	JSVM	Proposed	Factor
crowdrun	1920×1080	2110 s	82.5 s	26
riverbed	1920×1080	9090 s	73.2 s	120
rush hour	1920×1080	2420 s	51.2 s	47
tractor	1920×1080	4270 s	59.0 s	72
shields	1280×720	1340 s	34.7 s	39
stockholm	1280×720	834 s	33.9 s	25
coastguard	352×288	115 s	3.87 s	30
foreman	352×288	112 s	3.73 s	30

6.2.2 Rate-distortion characteristics

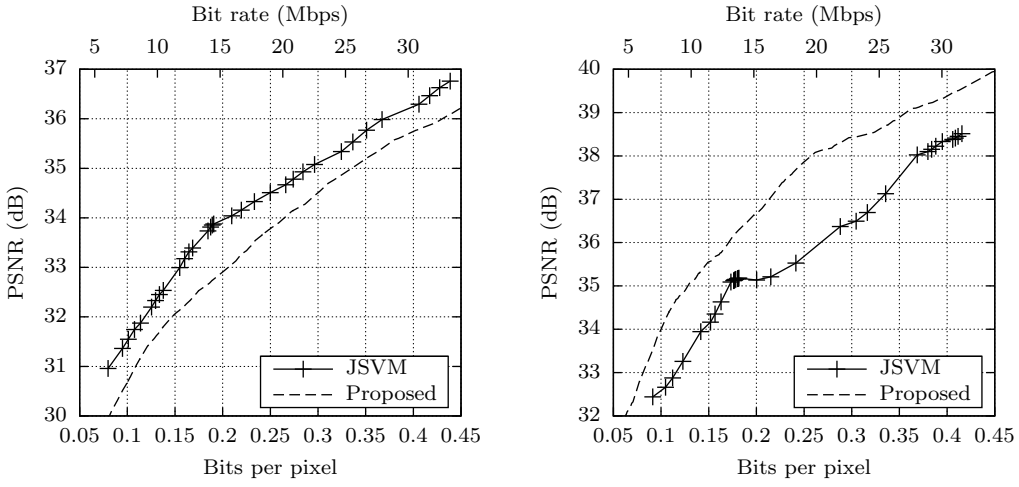
Figures 6.3 shows the rate-distortion curve for the two systems for the four HD sequences. For (a) *crowdrun*, JSVM performs better than our scheme, but for the other sequences our scheme is comparable to, or better than, JSVM. At the lower end of the bit rate, JSVM is harder to match. This may be because the lowermost point for JSVM corresponds to its base layer, which has no scalability restrictions, whereas our scheme is scalable to even lower bit rates than those shown in the plots. The JSVM curves have markers to indicate the possible operating points, whereas the proposed scheme supports fine-grained scalability.

The full range

The range of the curves in Figure 6.3 are limited to the range of the JSVM bitstream. In Figure 6.4, the same curves are shown, but this time covering the range of the single video bitstream obtained from our method. We can see that the scalability range of JSVM is much smaller than the range for the proposed scheme. It is possible to add the number of layers for JSVM to increase its range, but the amount of processing will increase with each refinement layer, which is not the case with the proposed scheme, where one encoding pass will be sufficient for the whole range shown.

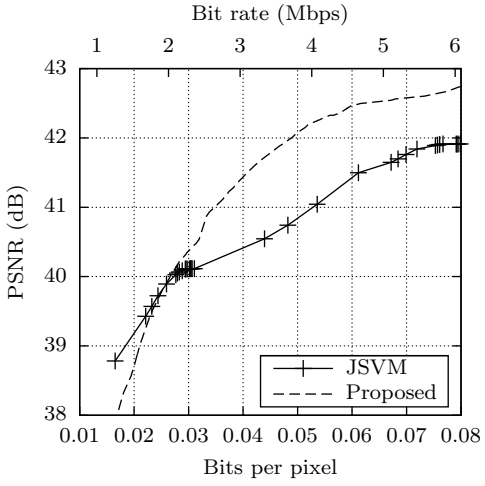
Multiple JSVM bitsreams

Figure 6.5 compares the proposed scheme to JSVM again, this time using three separate bitstreams for the JSVM method. The JSVM curves for Figure 6.3 were obtained using a QP of 38 for the base layer, 32 for the

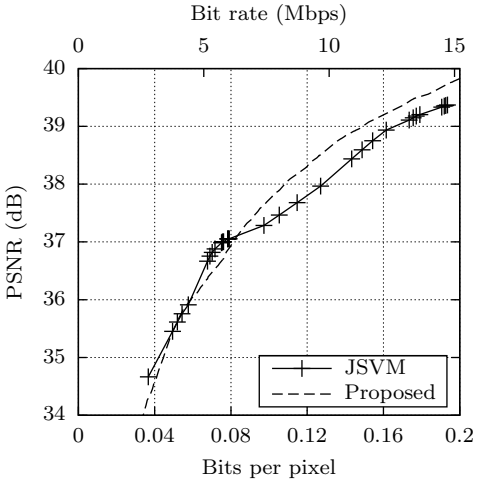


(a) *crowdrun* (1920 × 1080)

(b) *riverbed* (1920 × 1080)

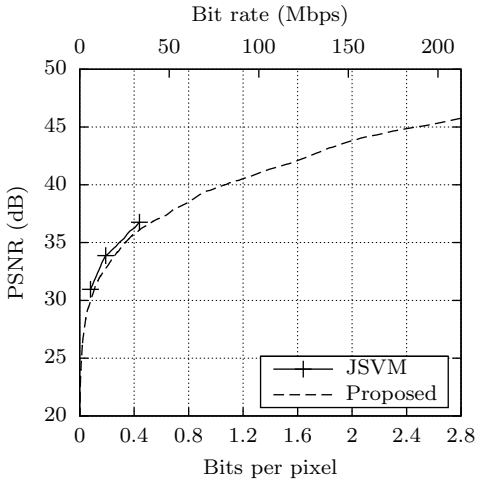


(c) *rush hour* (1920 × 1080)

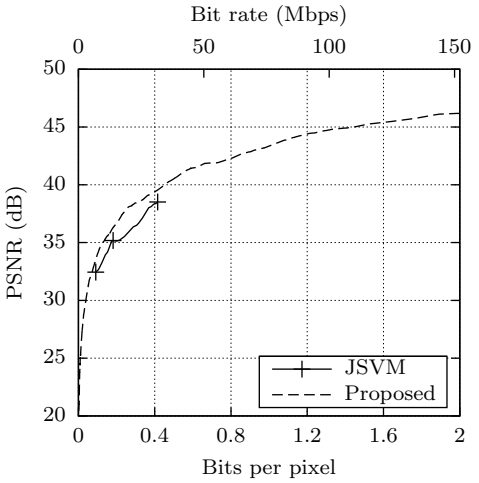


(d) *tractor* (1920 × 1080)

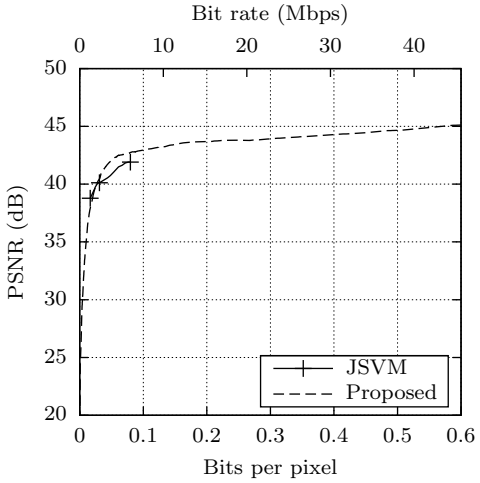
Figure 6.3: Rate-distortion curves for JSVM and the proposed method, for the bit rate range of the JSVM bitstream.



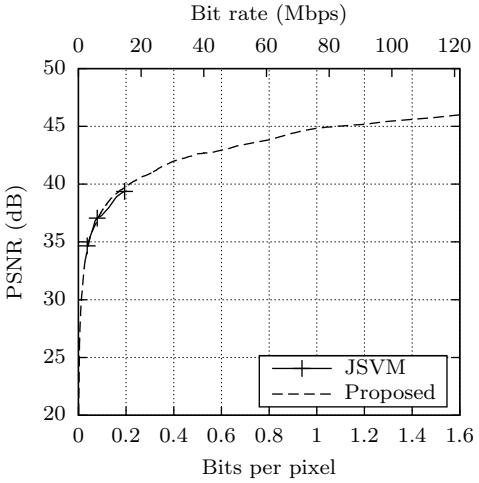
(a) *crowdrun* (1920 × 1080)



(b) *riverbed* (1920 × 1080)

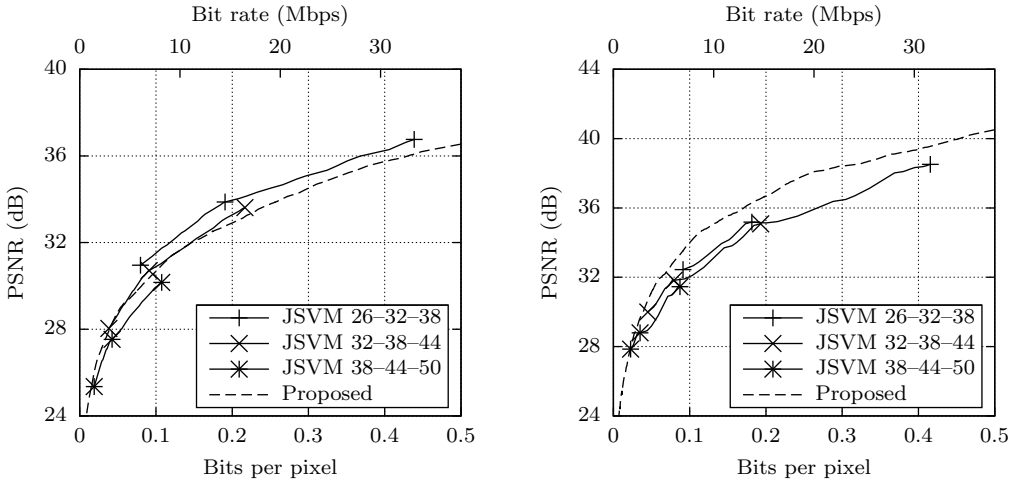


(c) *rush hour* (1920 × 1080)



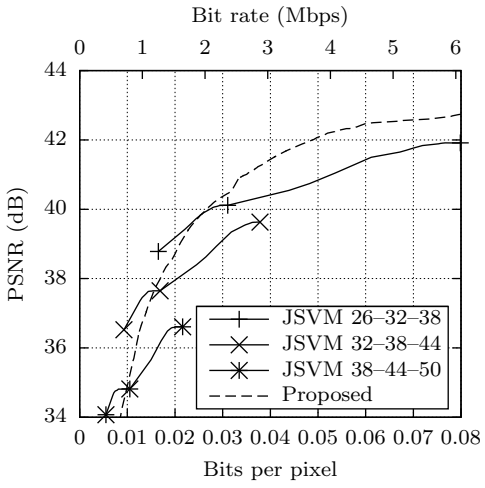
(d) *tractor* (1920 × 1080)

Figure 6.4: Rate-distortion curves for JSVM and the proposed method, for the bit rate range of the bitstream from the proposed method.

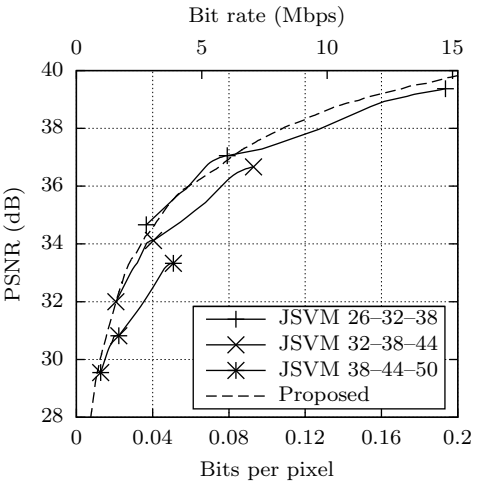


(a) *crowdrun* (1920 × 1080)

(b) *riverbed* (1920 × 1080)



(c) *rush hour* (1920 × 1080)



(d) *tractor* (1920 × 1080)

Figure 6.5: Rate-distortion curves for JSVM and the proposed method, with three different ranges for JSVM.

second layer, and 26 for the complete bitstream, covering a QP range of 26–32–38. In Figure 6.5, three JSVM bitstreams are used for each video sequence. Other than the 26–32–38 range of QP, there are curves for the two other ranges 32–38–44 and 38–44–50. The figure shows that the point for a QP of 38, which is available for the three bitstreams, lies on different points on the graphs. The best of these three points is when the 38 point the QP of the base layer, and the worst is when the 38 point is the QP of the complete bitstream. This demonstrates why it is more difficult to match the performance of JSVM towards the lower bit rates supported by a bitstream, where the scalability of the bitstream does not affect the rate distortion characteristic. This can be seen clearly in the curves for (c) *rush hour*, where JSVM seems to outperform our scheme at the points for the base layers for each of the three JSVM bitstreams, but our scheme outperforms JSVM for the rest of the scalability range.

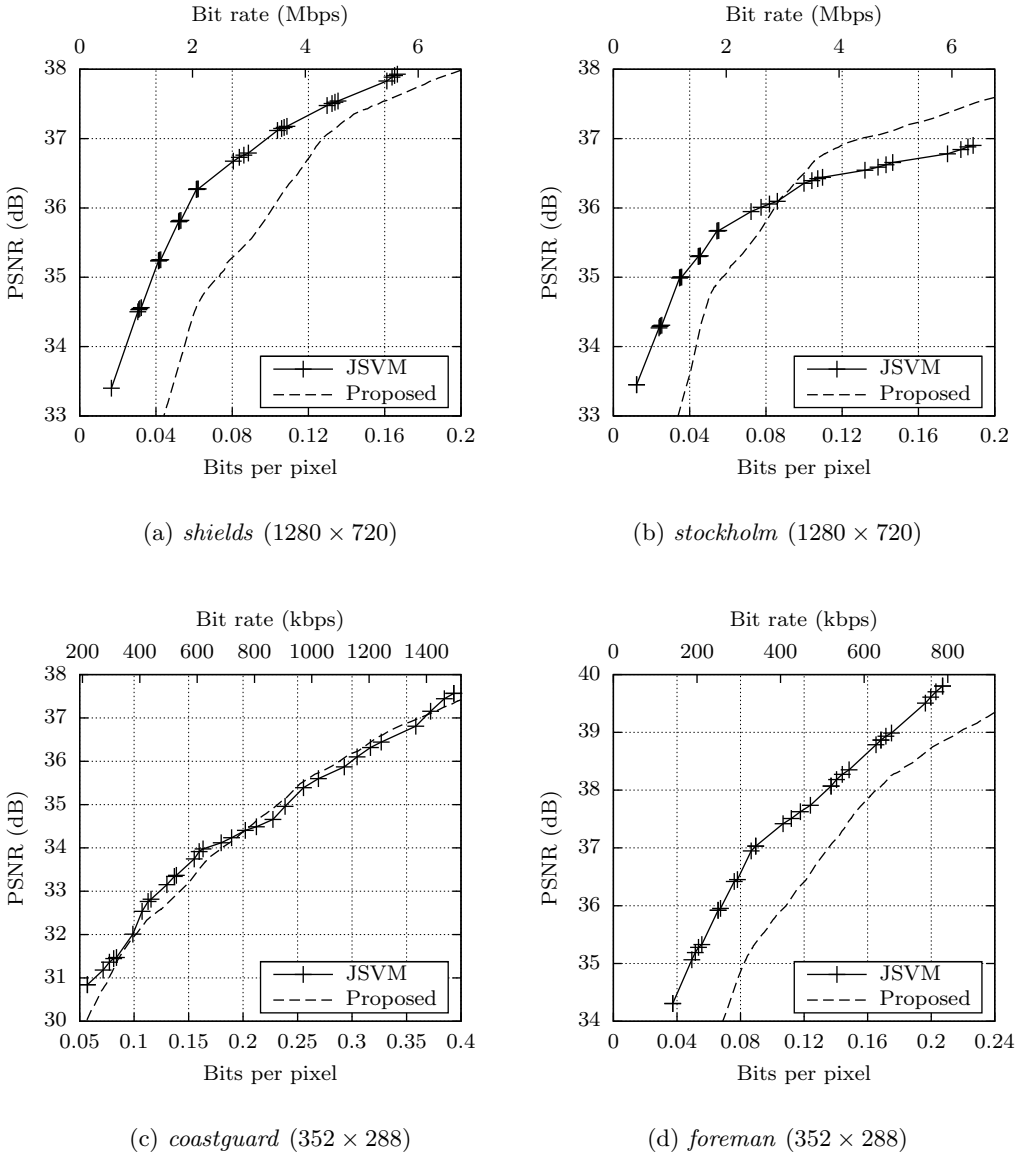
Lower resolutions

Other than the HD sequences, the algorithm was tested on 720p sequences and CIF sequences. In some of these lower resolution sequences, the curves for the proposed methods are close to JSVM, and in some of them, the performance of our method is poorer. Figure 6.6 shows some typical results obtained. At 720p, there is (a) *shields* which shows poorer performance and (b) *stockholm* which is close to JSVM. At CIF resolution, the performance for (c) *coastguard* is very similar to JSVM, but for (d) *foreman*, the performance is poorer.

Generally, wavelet video coding performs better than predictive or hybrid video coding at high resolutions. This explains why the proposed method outperforms JSVM for HD sequences, but not for 720p and CIF sequences.

6.3 Conclusion

In this chapter we have analysed the performance of the presented scalable video coding scheme. The benefits of the temporal filtering were demonstrated by comparing the scheme to Motion JPEG 2000, which employs similar methods for spatial redundancy but has no motion compensation. The memory requirements and execution duration of the proposed scheme and of the state-of-the-art JSVM algorithm were investigated, showing that



(a) *shields* (1280 × 720)

(b) *stockholm* (1280 × 720)

(c) *coastguard* (352 × 288)

(d) *foreman* (352 × 288)

Figure 6.6: Rate-distortion curves for JSVM and the proposed method for lower resolution sequences.

our scheme requires much less memory than JSVM and executes much faster. The compression performance analysis shows that our scheme generally outperforms JSVM for HD video sequences and gives similar performance on video sequences of lower resolution, while providing finer scalability over a larger range of bit rates.

Chapter 7

Hardware Amenability

A reconfigurable universal compression system developed within our research group was presented in Chapter 3. This chapter investigates the suitability of the scalable video scheme presented in previous chapters for hardware implementation as a part of the universal compression system.

Section 7.1 describes the motion estimation engine, presents some tools developed to analyse and configure this engine, and also includes analysis of motion estimation algorithms. Section 7.2 investigates the hardware amenability of the algorithms that generate and encode the motion vector palettes. Section 7.3 investigates the hardware amenability of the components that encode the temporally filtered frames. Section 7.4 analyses the hardware cost of the designed components. Section 7.5 gives some detail about how the designed hardware components were validated.

7.1 Motion estimation

As mentioned in Section 2.4.1, motion estimation is an expensive part of video coding in terms of complexity and hardware cost. To design for low complexity, motion estimation must be given its due consideration. This section describes the configurable motion estimation engine used in this work, and investigates how it can be configured for low complexity.

7.1.1 Motion estimation engine

A reconfigurable application-specific instruction-set processor (ASIP) developed in our group is to be used for motion estimation. It is designed

to execute user-defined block-matching motion estimation algorithms optimized for hybrid video codecs such as MPEG-2, MPEG-4, H.264/AVC [3, 4] and Microsoft VC-1. It is also used for BAPME [35], and is the engine for the scalable video coding (SVC) scheme proposed by this work.

The core offers scalable performance dependent on the features of the chosen motion estimation search algorithm and the number and type of execution units to be implemented in hardware. Hardware configuration can typically be achieved at compile time by adapting the architecture to the chosen algorithm, and in an FPGA implementation, it is possible to precompile a range of hardware bitstreams with different configurations from which one can be chosen to match the current video processing requirements. The microarchitecture can be easily scaled to high definition (HD) video even when using low cost FPGAs such as the Xilinx Spartan-3. The ability to program the search algorithm to be used, together with the ability to reconfigure the underlying hardware that it will execute on, give an extremely flexible video processing platform. A base configuration consisting of a single 64-bit integer pipeline, capable of processing a hexagonal motion estimation algorithm, such as the one implemented in the x264 [90] video encoder, over a search window of 112×128 pixels in real-time for high-definition video, can be implemented in 2300 logic cells on a Xilinx FPGA. In contrast, a complex configuration supporting motion vector candidates, sub-blocks, motion vector costing using Lagrangian optimization, four integer-pel execution units (IPEUs) and one fractional-pel execution unit (FPEU) plus sub-pel interpolator execution unit (SPIEU) will need around $14,600$ logic cells.

Table 7.1 compares the hardware cost and the performance of the processor core implementation to that of other implementations. The IPEUs and FPEUs have been carefully pipelined, and all the configurations can be implemented to achieve a clock rate of 200 MHz when targeting the Virtex-4 Xilinx family. More details can be obtained in [91].

7.1.2 Motion estimation algorithms

In order to target the motion estimation ASIP, a custom language developed within the group is used. The Estimo C language is a high-level C-like language that is aimed at designing a broad range of block-matching algorithms. The code can be developed and compiled in the SharpeEye Studio [92], an

Table 7.1: Comparison of the hardware cost for different implementations for a diamond search pattern.

Processor impl.	Cycles per MB	FPGA slices	Virtex-II clock	Memory (BRAMS)
Intel P4 assembly	~ 3000	N/A	N/A	N/A
Dias et al. [60]	4532	2052	67 MHz	4 (external reference area)
Babionitakis et al. [61]	660	2127	50 MHz	11 (1 ref. area, 48×48 pixels)
Proposed, 1 IPEU	510	1231	125 MHz	21 (2 ref. areas, 112×128 pixels)
Proposed, 2 IPEUs	287	2051	125 MHz	38 (2 ref. areas, 112×128 pixels)

IDE for motion estimation.

The language contains a preprocessor for macro facilities that include conditional (*if*) and loop (*for*, *while*, *do*) statements. The language also has facilities directly related to the motion estimation processor's instruction set, such as checking the sum of absolute differences (SAD) of a pattern consisting of a set of points, and conditional branching depending on which point from the last pattern check command had the best SAD. The compiler converts the program to a program memory file containing instructions and a point memory file containing patterns.

Figure 7.1 shows an example block-matching algorithm written in Estimo C and excerpts from the target files. The algorithm is a diamond search pattern executed for up to five times for a radius of eight, four, two and one pixels, followed by a small full search at fractional pixel level. The first set of *check* and *update* commands create the first search pattern, which consists of five points. Each *check* command adds a point to the search pattern being constructed, and the *update* command completes the pattern. This pattern is compiled into the instruction at program address `00`, which uses the five points available in the point memory at addresses `00`–`04`. The preprocessor goes through the *do while* loop three times, with s taking the values four, two and one. Each time, a four-point pattern is checked for up to five times. The $\#if(WINID == 0)\ \#break$ command ensures that if a pattern search does not improve the motion vector estimate, it is not repeated. The final

Estimo C source code

```
s = 8; // initial step size

check(0, 0);
check(0, s);
check(0, −s);
check(s, 0);
check(−s, 0);
update;

do {
    s = s/2;
    for (i = 1 to 5 step 1) {
        check(0, s);
        check(0, −s);
        check(s, 0);
        check(−s, 0);
        update;
        #if (WINID == 0)
            #break;
    }
} while (s > 1);

for (x = −0.5 to 0.5 step 0.25)
    for (y = −0.5 to 0.5 step 0.25)
        check(x, y);
update;
```

Program memory

```
00: 0 05 00   check 5 points, offset 00
01: 0 04 05   check 4 points, offset 05
02: 2 00 0b   if WINID is 0, goto 0b
03: 0 04 05   check 4 points, offset 05
              ...
0b: 0 04 09   check 4 points, offset 09
0c: 2 00 15   if WINID is 0, goto 15
              ...
15: 0 04 0d   check 4 points, offset 0d
16: 2 00 15   if WINID is 0, goto 1f
              ...
1f: 1 04 0d   chk 25 frac points, offset 11
   ↑
 opcode
   0          integer check pattern
   1          fractional check pattern
   2          conditional jump
```

Point memory

```
00: 00 00   integer (0, 0)
01: 00 08   integer (0, 8)
02: 00 f8   integer (0, −8)
03: 08 00   integer (8, 0)
04: f8 00   integer (−8, 0)
05: 00 04   integer (0, 4)
06: 00 fc   integer (0, −4)
            ...
11: fe fe   fractional (−0.5, −0.5)
12: fe ff   fractional (−0.5, −0.25)
            ...
29: 03 03   fractional (0.5, 0.5)
```

Figure 7.1: The Estimo C code for a motion estimation algorithm and excerpts of the target files generated by the compiler.

lines create a 25-point fractional pattern search.

7.1.3 Cycle-accurate simulation

The configurable hardware system has a number of design parameters, and it can be complex to configure. Since it has many design parameters, it has a large design space, and consequently, exploring this design space to find optimal configuration parameters can be hard. Doing this exploration on the actual hardware can be complicated and time consuming.

During motion estimation algorithm design, the time a particular algo-

rithm takes to determine the motion estimation vectors could be needed. It would also be very useful to be able to choose configuration parameters for the motion estimation processor depending on the particular requirements of the design.

Doing this analysis on the actual processor can be complicated and time consuming. The tasks required would include synthesizing the hardware with specific configuration parameters, configuring the FPGA board, and measuring the time used by the processor to perform the motion estimation. An alternative is to use a cycle-accurate simulator of the hardware system that can speed up the development cycle significantly by reducing the number of tasks required for the analysis of a particular hardware configuration. This has the added advantage that there is no need to access the hardware when using the simulator.

For these reasons, a cycle-accurate simulator was developed for the design space exploration of the motion estimation processor. This enables the configuration and analysis of the motion estimation search as part of the universal compression system. x264 [90], a free software library for encoding H.264/AVC, was modified to use the cycle-accurate model for its motion estimation search; the motion estimation code in x264 was replaced by an engine that uses the cycle-accurate model when searching for the motion vectors.

The cycle-accurate simulator can be used directly from the SharpEye IDE introduced in Section 7.1.2. Designers can design a motion estimation algorithm and test it using different processor configuration parameters. Figure 7.2 shows a sample session.

The simulator takes several parameters as inputs. The inputs which determine the processor configuration are

- the program and point memories generated by the Estimo compiler,
- the number of IPEUs and FPEUs,
- the minimum size for block partitioning,
- whether to use motion vector cost optimization, and
- whether to use multiple motion vector candidates.

The simulator takes other options which do not affect the processor configuration itself, which are the video file to process and its resolution, the

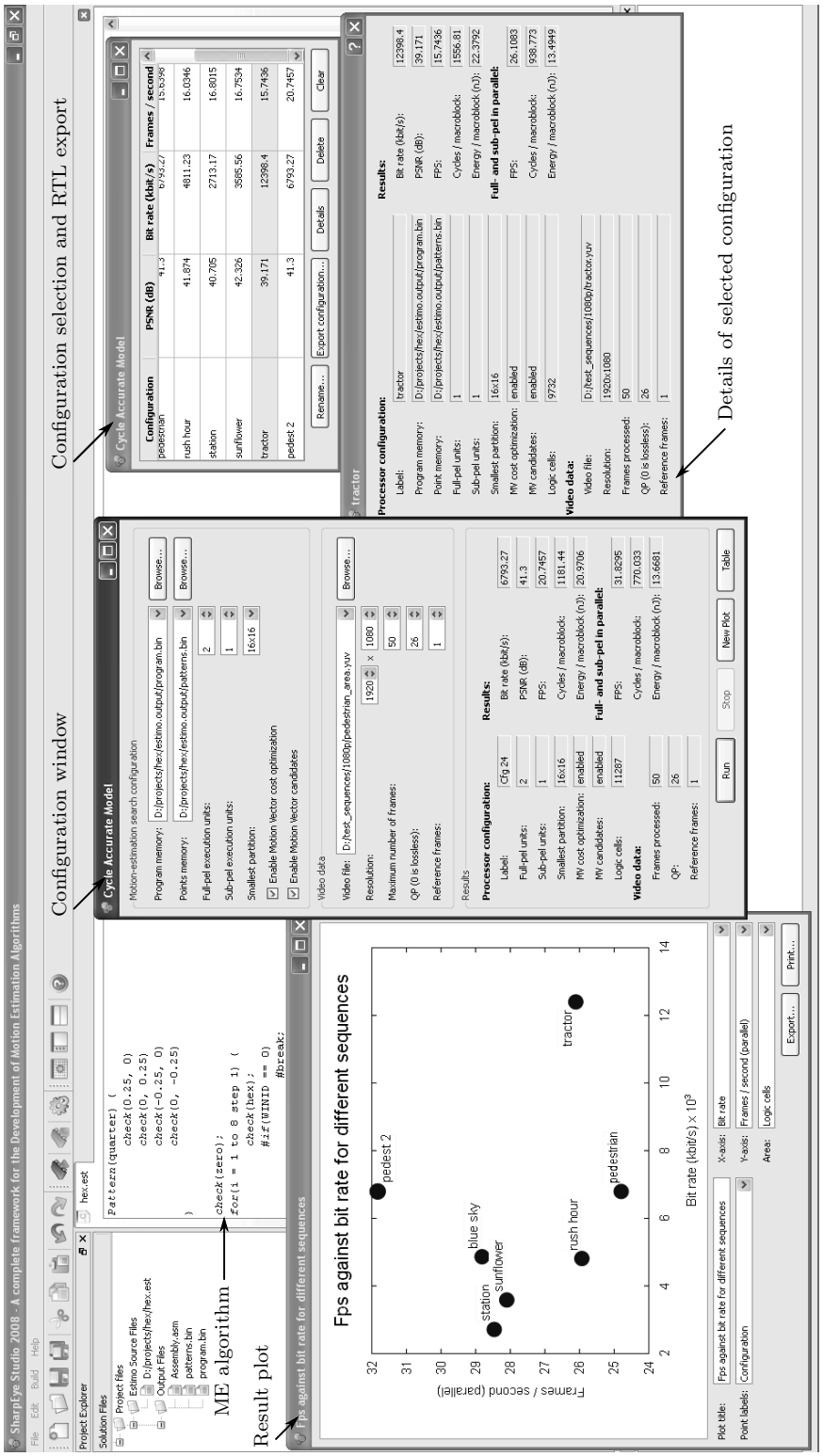


Figure 7.2: Screenshot of the SharpEye IDE used to analyse motion estimation algorithms and processor configurations.

maximum number of frames to process, and the quantization parameter (QP).

The simulator will then process the video file using the selected search algorithm and processor configuration, and give the following outputs:

- the bit rate of the compressed video,
- the PSNR,
- the number of frames processed per second (fps) assuming a clock rate of 200 MHz,
- the number of clock cycles required per macroblock, and
- the energy requirements per macroblock.

Designers can simulate and analyse various configurations by using the simple controls in the configuration window, and then generate plots or view the results in a table. When they are satisfied with a particular configuration, they can generate a VHDL file which can be used together with the rest of the core hardware register transfer level (RTL) library to synthesize the motion estimation processor.

Using the cycle-accurate simulator developed in this work, different block matching strategies can be evaluated with video sequences that are suitable for the problem domain of the final application. For example, if the aim is to develop a system for space applications, satellite video data can be used to test the system.

For a particular required level of performance, the minimum possible hardware cost can be found easily by testing several configurations and picking the least complex one which satisfies the timing and performance requirements. The bitstream for the reconfigurable hardware can then be used in the reconfigurable system. It is also possible to generate separate configurations for different use cases, in order, for example, to save power for certain conditions that require less complexity.

7.1.4 Analysis of motion estimation algorithms

For analysis, a number of test video sequences from [83] were used. Each sequence has a frame rate of 25 fps. The analysis was performed with a QP of 26.

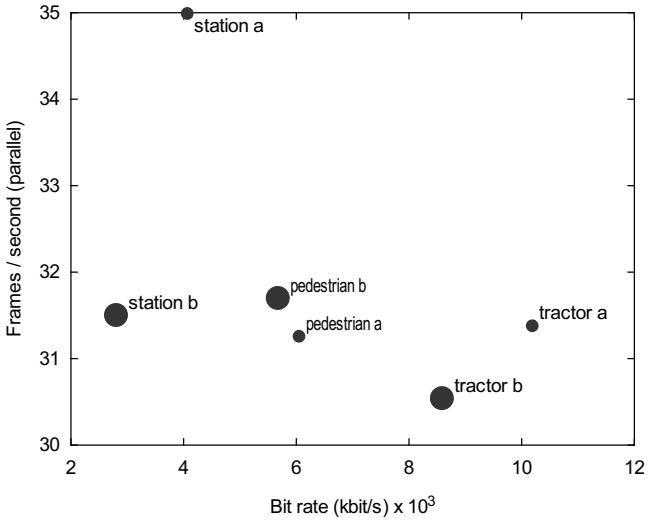


Figure 7.3: Graph of fps against bit rate for the *station*, *pedestrian area* and *tractor* sequences, with (a) one IPEU and no sub-pel estimation, and (b) two IPEUs and one FPEU.

Three 1920×1080 sequences, *pedestrian area*, *station* and *tractor*, were processed and the number of frames that can be processed per second (fps) was plotted against the bit rate. Figure 7.3 shows the results. The graph of Figure 7.3 was generated directly by the tools developed as part of this work. Each of the files was processed twice, (a) without sub-pel estimation and (b) with sub-pel estimation. In each case, Langrangian optimization and multiple motion vector candidate optimization were used. With no sub-pel estimation, the files can be processed at a rate larger than 30 fps with only one IPEU, requiring only 2900 logic cells. In order to have a similar frame rate when sub-pel estimation is used, two IPEUs and one FPEU were used, raising the number of required logic cells to $11,000$. The bit rate is reduced by 6% for *pedestrian area*, 31% for *station*, and 16% for *tractor*, showing the benefits of sub-pel motion estimation. The area of the plot points is proportional to the number of logic cells.

The processor supports operating the IPEUs and FPEUs in parallel. In case (a), since no sub-pel estimation was used, this does not affect the results. In case (b), the fps plotted is for the case of using this parallelization. The advantage of parallel operation is that for the same fps rate, less logic cells are required than in the case of sequential operation. For example, if the

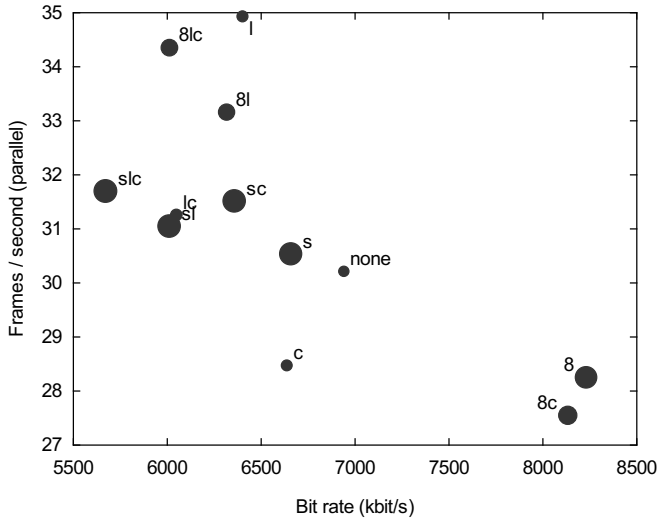


Figure 7.4: Different configurations for the *pedestrian area* sequence. The labels contain a list of optimizations used: (8) 8×8 partitioning, (s) sub-pixel estimation, (l) Lagrangian optimization, (c) multiple motion vector candidates. The point area is proportional to the number of logic cells.

integer-pel and sub-pel execution units operate sequentially instead of in parallel, we will have to use three IPEUs and two FPEUs for similar frame rates, further raising the number of required logic cells to $14,600$.

It is important to remember that different kinds of video may give different results, so the video files used in the simulation should be representative of the final application. The *station* sequence has very little motion, so a bit rate of 2700 kbit/s is enough to encode it. The *pedestrian area* sequence has more motion and requires a bit rate of around 6800 kbit/s for the same QP. The *tractor* sequence is the most difficult of the three to encode and requires a bit rate of $12,400$ kbit/s. More complex motion estimation strategies could be deployed to reduce the bit rate, illustrating the advantages of a flexible motion estimation core.

Figure 7.4 shows the effect of different configurations when processing the 1920×1080 sequence *pedestrian area*. The area of the points in the figure is proportional to the number of logic cells required. Some configurations had an fps smaller than 25, which is the minimum for real-time processing, so the number of execution units was increased. This can be seen by their larger area requirements. The point labelled *none* supports none of the

TABLE 7.2: BIT RATE OBTAINED BY USING HEXAGONAL AND FULL SEARCHES, WITH AND WITHOUT SUB-PEL MOTION ESTIMATION.

Sequence	Resolution	No sub-pel		Sub-pel	
		Hex (kbit/s)	Full (kbit/s)	Hex (kbit/s)	Full (kbit/s)
pedestrian area	1920×1080	6048	5980	5671	5532
station	1920×1080	4064	4047	2800	2579
tractor	1920×1080	10187	10140	8584	8739
park run	1280×720	10963	10947	8611	9284
shields	1280×720	5854	5812	3612	4252
stockholm	1280×720	4440	4422	2188	2917

optimizations.

When the multiple motion vector candidate optimization is used (points having c in the label), the bit rate is reduced, and the processing speed changes. When 8×8 partition sizes are used (8) with no Lagrangian optimization (points having no l in the label), the bit rate is actually worse than when no sub-block partitions are used, indicating that Lagrangian optimization is essential when using sub-block partitions. Lagrangian optimization has the advantage of both reducing the bit rate and increasing the processing speed in all the cases. The processing speed is increased because the number of search points is reduced owing to faster convergence.

The hexagonal-search algorithm was compared to the full-search algorithm in another experiment. The hexagonal search used consists of up to eight iterations, with a hexagon radius of two pixels; it can select points up to 16 pixels in each direction from the initial point. The full search used can span the same range, 16 pixels in each direction from the initial point. The full search was performed by replacing the integer-pel search in the cycle-accurate simulator with a full search while leaving everything else unchanged. Table 7.2 shows the bit rates produced when processing sequences using these two searches both with and without sub-pel refinement.

With no sub-pel refinement, the full search produces a marginally better bit rate. With sub-pel refinement, the bit rates are very similar for the 1920×1080 sequences, but the hexagonal search performs better in the 720×576 sequences by 8% for *park run*, 18% for *shields*, and 33% for *stockholm*. This can be because the hexagonal search is a more logical search which has a higher chance of corresponding to the real object motion in the video

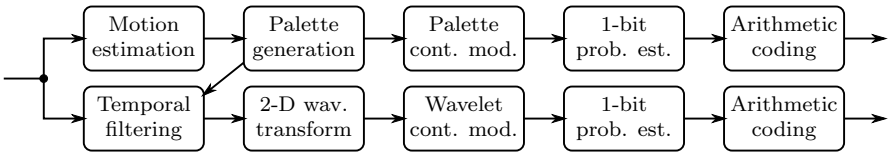


Figure 7.5: The scalable video compression components required for the universal reconfigurable compression system.

sequence, while the full search is much more likely to select a point which does not correspond to the real object motion; the motion vectors given by the full search are more susceptible to noise. This result confirms that a well-designed fast block matching algorithm can provide better rate-distortion performance than the full search algorithm as shown by [59].

These experiments have demonstrated how for a particular type of video sequences, we can configure the motion estimation processor. The processor's features can be added until the required compression performance is obtained. Then, the number of processing units can be added until the timing constraints are met, that is, until the number of frames processed per second is sufficient. This allows us to configure the processor to have the least possible hardware cost while meeting the design constraints.

7.2 The motion vector palettes

In Section 3.4, an architecture for universal compression was presented in Figure 3.2. Figure 7.5 shows the subset of Figure 3.2 that is used for SVC. Figure 7.6 shows the data flow through the components of this subsystem.

The motion estimation engine presented in [36] is used to perform the motion estimation search and generate one motion vector for each macroblock. The motion vector palette generator and encoder takes these motion vectors and generates a motion vector palette, which it then encodes. The number of motion vectors in the palette can be smaller than the number of distinct original motion vectors, so for the motion-compensated temporal filter, only the motion vectors available in the palette should be used. This means that the temporal filter should use the quantized motion vectors from the palette generator and encoder, not the motion vectors from the motion estimator.

Figure 7.7 shows an overview of the architecture of the motion vector

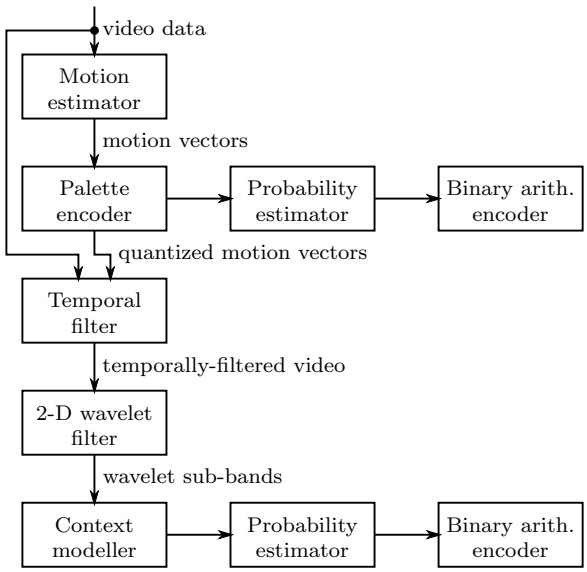


Figure 7.6: The data flow through the components of the scalable video compression system.

palette generator and encoder. The complete process can be divided into three main parts:

1. Input: the initialization process that reads the original motion vectors into one cluster of motion vectors (Section 7.2.1),

2. Clustering: divisive clustering of the motion vectors to generate a motion vector palette (Section 7.2.2), and

3. Output: the output of the quantized motion vectors to the temporal filter (Section 7.2.3) and of context modelling data to the probability estimator (Section 7.2.4).

Together with these three blocks, there are two support components: the split accumulator (Section 7.2.5) and the statistics calculator (Section 7.2.6). There are also a number of arrays stored in memory, which will be described in the following sections.

7.2.1 Input of original motion vectors

The first stage is the initialization process. The motion vectors of the form (x, y) are read one at a time and stored in array (a) mv in memory. From

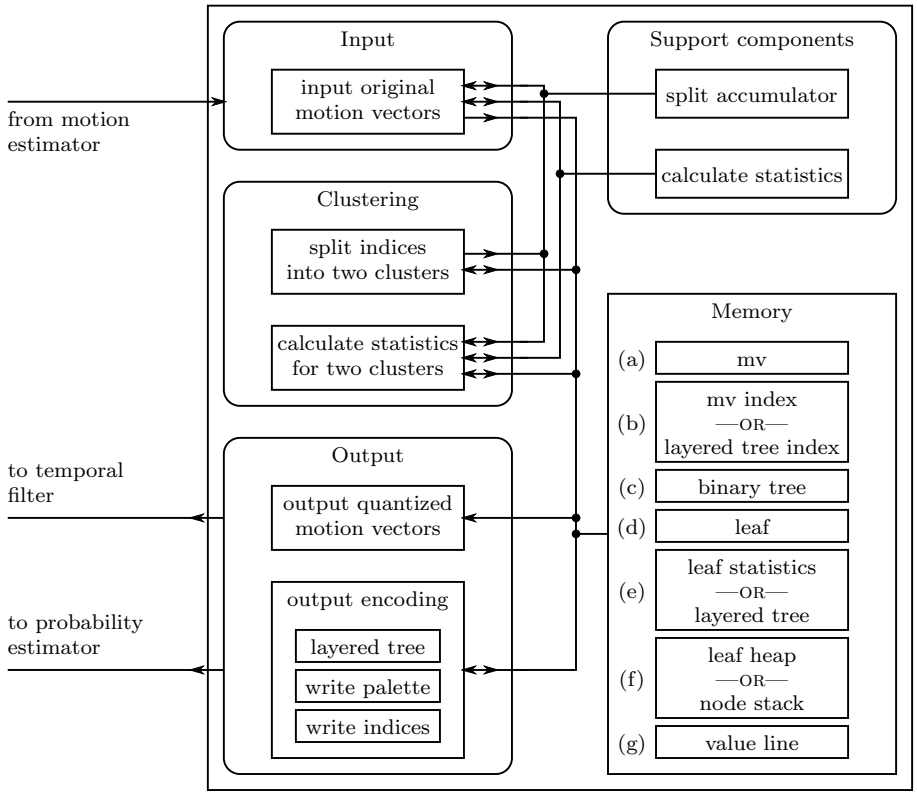


Figure 7.7: The architecture of the motion vector palette generator and context modeller.

Section 4.2, we know that we need some statistical values. In order to generate these statistics for the starting set of vectors S_1, each input vector is passed to the accumulator block which accumulates the sums Σx, Σy, Σx^2, Σy^2 and Σxy. These values are then passed to the statistics calculator block, which calculates statistical values needed by the divisive clustering described in Section 7.2.2 below. This stage also initializes the other memory arrays.

7.2.2 Divisive clustering of motion vectors

The divisive clustering method of Section 4.2 is used to split the set S_1 of all the motion vectors into a number of clusters equal to the size of the motion vector palette. A binary tree is built, with each node representing a cluster of motion vectors. For each cluster division, a cluster of motion vectors represented by one leaf node is split into two clusters. The leaf node in question becomes a parent node with two child nodes, one for each of the two new clusters.

To decide which leaf node to split, and how to divide its motion vectors into two clusters, some statistical values are required. Each cluster S_i has n_i motion vectors of the form (x, y). The values x, y, x^2, y^2 and xy are added over all the n_i vectors, and their means $\bar{x}$, $\bar{y}$, $\overline{xx}$, $\overline{yy}$ and $\overline{xy}$ are found.

As described in Section 4.2, we need to find the variance $\tilde{\mathbf{R}}_i$ in (4.5), which is a 2×2 symmetric matrix given by

$$\begin{pmatrix} a & b \\ b & d \end{pmatrix} = \begin{pmatrix} \overline{xx} - \bar{x}\bar{x} & \overline{xy} - \bar{x}\bar{y} \\ \overline{xy} - \bar{x}\bar{y} & \overline{yy} - \bar{y}\bar{y} \end{pmatrix}. \tag{7.1}$$

The values a and d cannot be negative, but b can be negative. Let

$$\Delta = \sqrt{(a-d)^2 + (2b)^2}. \tag{7.2}$$

We can find that the principal eigenvalue λ and the corresponding eigenvec-

tor $\mathbf{e}$ are

$$\lambda = \frac{a + d + \Delta}{2} \tag{7.3}$$

$$\mathbf{e} = \begin{cases} (1 \quad 0)^{\mathrm{T}} & \text{if } \Delta = 0, \\ (a - d + \Delta \quad 2b)^{\mathrm{T}} & \text{if } a > d, \\ (2b \quad d - a + \Delta)^{\mathrm{T}} & \text{otherwise.} \end{cases} \tag{7.4}$$

These statistical values are calculated using the statistics calculator component which will be presented in Section 7.2.6 below.

When a split is required, the set or cluster represented by the leaf node having the largest λ is split into two clusters, and the motion vectors that satisfy the inequality

$$(x \quad y)\,\mathbf{e} \leq (\bar{x} \quad \bar{y})\,\mathbf{e} \tag{7.5}$$

go into the first cluster, while the others go into the second cluster.

Figure 7.8 shows the contents of the used data structures when starting with nine motion vectors and generating a palette of four motion vectors. This means that at the end of the clustering process, there should be four clusters. This requires three divisions, or three splits.

The original motion vectors are shown in the bottom left corner. There are three data structures shown in Figure 7.8 apart from the table of original motion vectors: the *palette indices*, the *binary tree*, and the *heap of leaf nodes*. These data structures are stored using arrays in the actual component implementations.

After initialization, there is only one cluster, which holds all the motion vectors. Since there is only one cluster, all the values in the *palette indices* table of Figure 7.8 are 0. The mean of the nine motion vectors is $\mathbf{m}_1 = (-1, 0)$, and the binary tree consists of only one node, node 1, which is the root node and represents the cluster containing all the vectors. A heap data structure [93] is used to maintain an ordered list of all the palette indices sorted by their corresponding λ. After initialization, there is only one value in the heap, with $\lambda_1 = 116$.

Each time a cluster is to be split, the palette index at the top of the heap is the palette index of the cluster to split. The leaf node in the binary tree that corresponds to this palette will thus become a parent node with

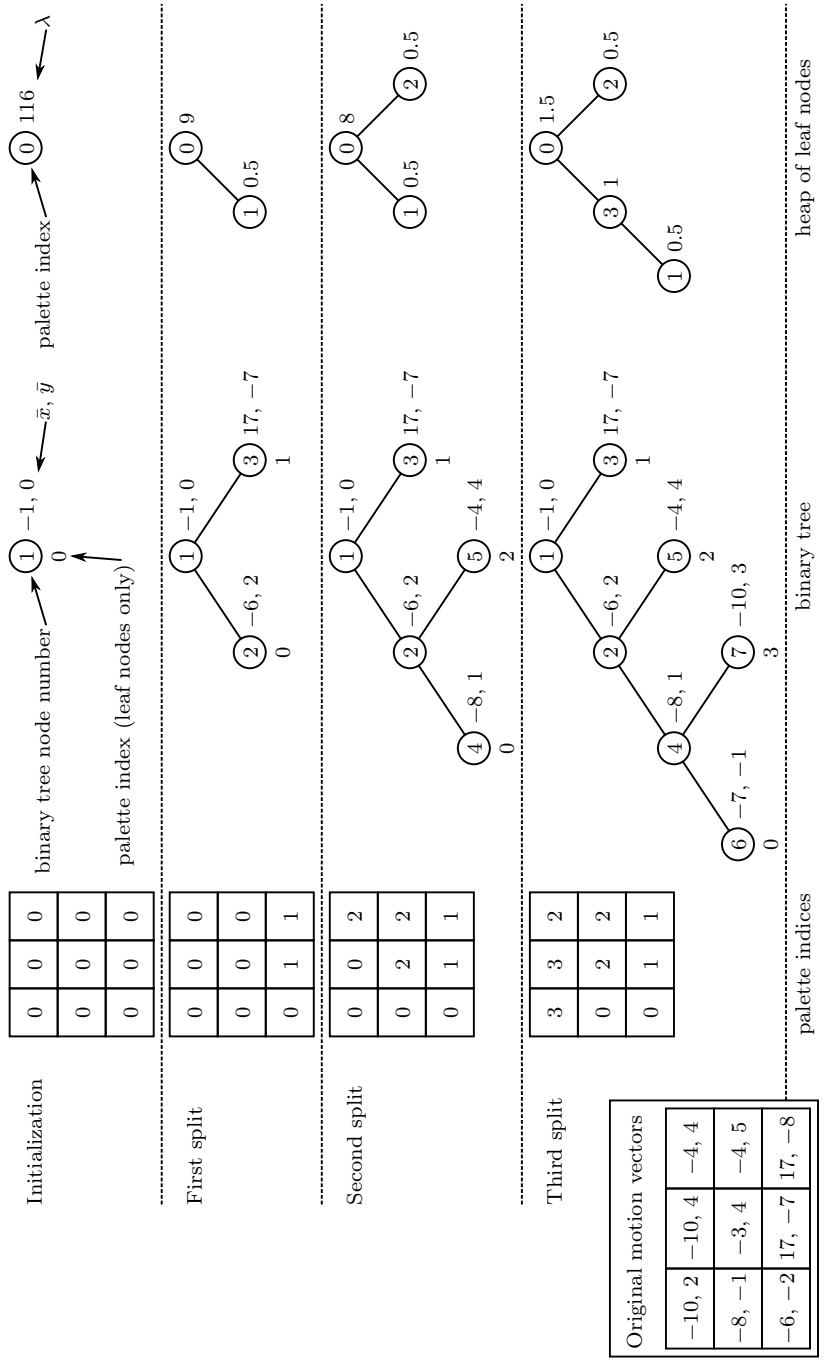


Figure 7.8: The data structures used in the divisive clustering of motion vectors.

two child nodes, the first child node will contain the motion vectors of the new parent which satisfy (7.5), and the second will contain the rest.

For the first cluster division, or the first split, the set S_1 of all motion vectors is split in two. The first set S_2, represented by node 2, contains seven vectors and has a mean $\mathbf{m}_2 = (-6, 2)$ and a principal eigenvector $\lambda_2 = 9$. The second set S_3, represented by node 3, contains two motion vectors and has a mean $\mathbf{m}_3 = (17, -7)$ and $\lambda_3 = 0.5$. The seven vectors in set S_2 have a corresponding palette index of 0, and the two vectors in set S_3 have a corresponding palette index of 1. Since $\lambda_2 > \lambda_3$, the palette index corresponding to S_2 is at the top of the heap, and the next set to split will thus be S_2. The first, second and third split are shown in Figure 7.8. For the second split, three vectors from palette index 0 are split into a new palette index 2. For the third split, two vectors from palette index 0 are split into a new palette index 3.

The data required by the clustering algorithm is held in a number of arrays. Table 7.3 shows the contents of all the arrays held in memory at the end of the third split for the example above.

Array (a) *mv* holds the actual motion vectors and their corresponding palette index. The motion vectors themselves are written in the initialization stage (Section 7.2.1) and are not changed during the rest of the algorithm. The palette index entries start as all 0s on initialization, and on each split, a new palette index is introduced. In the first split, palette index 0 is split into 0 and 1, and some entries are changed from 0 to 1. In the second split, palette index 0 is split into 0 and 2, and in the third split, palette index 0 is split into 0 and 3.

To avoid having to iterate through all the elements in array (a) *mv* when we need to iterate through the elements of one set, we use array (b) *mv index*, which is simply a list of the indices of array (a) *mv* ordered such that the motion vectors in the same set are contiguous. For example, the entries for node 5, which has palette index 2, are the motion vectors at locations 2, 4 and 5 in array (a) *mv*, which are not contiguous. So pointers to these three locations are stored contiguously in array (b) *mv index*. On each split, one set is split into two sets, so the entries belonging to that set in array (b) *mv index* are split into two; since they are already contiguous, the entries for the other sets are not touched, and only a small set of pointers has to be updated and reordered.

TABLE 7.3: MEMORY CONTENTS FOR THE DIVISIVE CLUSTERING OF MOTION VECTORS.

(a) *mv*

	x	y	palette index
0	−10	2	3
1	−10	4	3
2	−4	4	2
3	−8	−1	0
4	−3	4	2
5	−4	5	2
6	−6	−2	0
7	17	−7	1
8	17	−8	1

(b) *mv index*

	mv index
0	3
1	6
2	1
3	0
4	5
5	4
6	2
7	8
8	7

(c) *binary tree*

node	$\bar{x}$	$\bar{y}$	is leaf?	pointer	pointer meaning
1	−1	0	no	1	children: 2, 3
2	−6	2	no	2	children: 4, 5
3	17	−7	yes	1	palette index: 1
4	−8	1	no	3	children: 6, 7
5	−4	4	yes	2	palette index: 2
6	−7	−1	yes	0	palette index: 0
7	−10	3	yes	3	palette index: 3

(d) *leaf*

palette index	first mv index	size	bin tree
0	0	2	6
1	7	2	3
2	4	3	5
3	2	2	7

(e) *leaf statistics*

palette index	$\mathbf{e}_x$	$\mathbf{e}_y$	$\bar{x}_{\text{frac}}$ /256	$\bar{y}_{\text{frac}}$ /256
0	8	−4	0	−128
1	0	8	0	−128
2	−4	5	86	85
3	0	8	0	0

(f) *leaf heap*

	palette index	λ
1	0	1.5
2	3	1
3	2	0.5
4	1	0.5

Array (c) *binary tree* holds the binary tree shown in Figure 7.8. The meaning of the pointer field p depends on whether the node is a parent node or leaf node. If the node is a parent node, its children have node numbers $2p$ and $2p+1$, where p is the pointer value. If the node is a leaf node, the pointer value p refers to the palette index associated with the leaf node. The binary tree after the third split in Figure 7.8 is described completely by the contents of array (c) *binary tree* in Table 7.3.

Arrays (d) *leaf* and (e) *leaf statistics* hold the information about each palette entry. Each palette entry corresponds to a set of motion vectors which are contiguous in array (b) *mv index*, and the first index and number of entries is held in the first two fields of array (d) *leaf*. The palette entry also has a corresponding leaf node in the binary tree, and the node number is held in the third field. For example, palette index 2 has node 5, as can be seen in its third field, and it has three motion vectors starting at location 4 in array (b) *mv index*. Array (e) *leaf statistics* holds extra information about the set that is useful during clustering; it contains the eigenvector $\mathbf{e}$ and the fractional part of the mean $\mathbf{m}$ that is required for accurate calculation of (7.5).

Array (f) *leaf heap* holds the heap of leaf nodes that is used to select a set of motion vectors to split. The heap is sorted according to the principal eigenvalue λ of a set. To select a cluster for division, the top of the heap is taken. From the palette index obtained from the top of the heap, we can obtain the list of motion vectors by going to array (d) *leaf* and then to array (b) *mv index*.

For example, if we wanted one further division, the current top of the heap has palette index 0 and $\lambda = 1.5$. Form array (d) *leaf*, we can see that there are two vectors starting at motion vector index 0. With this information, we can use array (b) *mv index* to find that these two vectors are at locations 3 and 6 in the table or original motion vectors in array (a) *mv*. For each of these motion vector, inequality (7.5) is used to decide whether the vector goes into the first subset or the second subset. For the first motion vector, $(-8,-1)$, the inequality becomes

$$\begin{pmatrix} -8 & -1 \end{pmatrix} \begin{pmatrix} 8 \\ -4 \end{pmatrix} \leq \begin{pmatrix} -7 & -1\frac{1}{2} \end{pmatrix} \begin{pmatrix} 8 \\ -4 \end{pmatrix},$$

which is true. For the second vector, $(-6, -2)$, the inequality becomes

$$\begin{pmatrix} -6 & -2 \end{pmatrix} \begin{pmatrix} 8 \\ -4 \end{pmatrix} \leq \begin{pmatrix} -7 & -1\frac{1}{2} \end{pmatrix} \begin{pmatrix} 8 \\ -4 \end{pmatrix},$$

which is false. Thus, the first vector would go into the first subset, and the second vector would go into the second subset.

During the splitting process, while the values of these vectors are tested for inequality (7.5), the sums Σx, Σy, Σx^2, Σy^2 and Σxy are calculated on the fly for both subsets using the split accumulator, which will be described in Section 7.2.5. At the end of the splitting process, these two sets of sums are passed in turn to the statistics calculator, which calculates two sets of statistics. The statistics calculator also calculates the means $\bar{x}$ and $\bar{y}$ of the clusters. The statistics for all the leaf nodes are held in array (e) *leaf statistics*. The element at the top of the heap, which represents the new parent node, is replaced by two new elements which represent the two new leaf nodes. This means that we need three heap operations: removal of the parent node, and insertion of each of the child nodes. Usually, after each operation, the heap will have to be reordered. In our case, two of these operations can be combined. First, the parent node is replaced with one of the child nodes, and the heap is reordered once, then the second child node is added to the end of the heap and the heap is reordered once more. Thus, even though there are three operations on the heap, the heap needs to be reordered only two times.

Once the divisive clustering is complete, arrays (b) *mv index*, (e) *leaf statistics* and (f) *leaf heap* are no longer needed, so they are reused in the output stage described in Section 7.2.4. Arrays (a) *mv*, (c) *binary tree* and (d) *leaf* will still be required.

7.2.3 Output of the quantized motion vectors

The quantized motion vectors need to be passed to the temporal filter. For each motion vector, the palette index is read from array (a) *mv*, and then used to get the corresponding binary tree node from the array (d) *leaf*. Finally, the mean motion vector $\mathbf{m}$ is read from the $\bar{x}$ and $\bar{y}$ fields of array (c) *binary tree* and passed to the temporal filter.

For example, for the first motion vector, the palette index 3 is read from

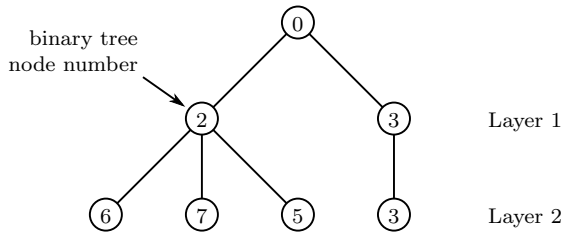


Figure 7.9: Layered tree for encoding motion vector palettes in layers.

the first location of array (a) *mv*. Then, the binary tree node number 7 is read from array (d) *leaf*. Finally, the mean value for the motion vectors in the cluster is read from (c) *binary tree*. This mean value, $(-10, 3)$, is the quantized motion vector, and is used instead of the original motion vector $(-10, 2)$.

7.2.4 Encoding of the motion vector palettes

While the quantized motion vectors are being passed to the temporal filter, the motion vector palettes can be encoded concurrently. This means that the processes of this section and Section 7.2.3 can be executed in parallel.

The encoding process of the motion vectors involves splitting the binary tree produced in Section 7.2.2 into a number of layers as described in Section 4.5. For each layer, the first thing to do is to determine the number of nodes from the binary tree to use.

For example, the values shown in Figure 7.8 can be split into two layers, the first with two palette entries, and the second with all the four palette entries. For the first layer, since only two palette entries are required, the binary tree nodes with number > 3 cannot be used, and the nodes with number ≤ 3 and with no children ≤ 3 are the nodes that are the effective leaf nodes. Generally, if we need k palette indices, the binary tree nodes i to be used are those that satisfy the two conditions

1. $i < 2k$ and

2. node i has no child nodes $< 2k$.

In this case, $k = 2$, and nodes 2 and 3 are used, as shown in the first level of the layered tree in Figure 7.9.

A stack is used to traverse the final binary tree of Figure 7.8 depth first. Since array (f) *leaf heap* of Section 7.2.2 is no longer used, the node stack

TABLE 7.4: MEMORY CONTENTS FOR THE FIRST LAYER OF THE LAYERED TREE.

(e) *layered tree*

	binary tree node	parent's binary tree node	sibling count	child index
0	2	0	2	0
1	3	0	–	1

(b) *layered tree index*

palette index	layered tree index
0	0
1	1
2	0
3	0

can reuse this area of memory. While traversing the binary tree, the two arrays in Table 7.4 are constructed. The binary tree nodes that satisfy the conditions above are stored in array (e) *layered tree*. A conversion table for palette indices from the whole binary tree to palette indices using the limited nodes of the first layer is stored in array (b) *layered tree index*. In this case, the palette indices 0, 2 and 3, which correspond to the leaf nodes 5, 6 and 7 in the binary tree, will all be assigned to the index 0 in the layered tree, while palette index 1, which corresponds to the leaf node 3 in the binary tree, will be assigned to index 1 of the layered tree. This first layer is encoded in two steps:

1. The palette itself is encoded as described in Section 4.3. This requires the encoding of a number of integers and does not require context modelling.

2. The motion vectors are encoded as indices to this palette as described in Section 4.4. This part requires a context modeller and probability estimator, which have been implemented as well.

For the second layer, the process is performed on parts of the binary tree instead of on the whole tree. Each of the binary tree nodes of the previous layer, in this case nodes 2 and 3, is treated as a root node in a sub-tree and the process performed on this sub-tree. In this case, array (e) *layered tree* will have four entries, the first three for the sub-tree with root 2, and the last entry for the sub-tree with root 3. After generating the layered tree, the second layer is encoded using the same two steps used to encode the first layer. If there are more than two layers, this process is repeated for all the

TABLE 7.5: MEMORY CONTENTS FOR THE SECOND LAYER OF THE LAYERED TREE.

(e) *layered tree*

	binary tree node	parent's binary tree node	sibling count	child index
0	6	2	3	0
1	7	2	–	1
2	5	2	–	2
3	3	3	1	0

(b) *layered tree index*

palette index	layered tree index
0	0
1	3
2	2
3	1

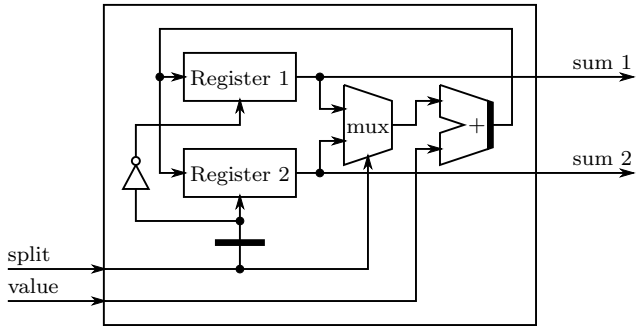


Figure 7.10: The split accumulator for each sum component required.

remaining layers.

7.2.5 Split accumulator

During the splitting of motion vectors into two clusters, each vector can either go to the first cluster, or to the second, according to the outcome of condition (7.5). For each of these two clusters, we will need the sums Σx, Σy, Σx^2, Σy^2 and Σxy.

One adder can be used to accumulate Σx for both clusters. In fact, for each of the five required sums, only one adder is required, so that in all five adders can be used instead of ten. Figure 7.10 shows the multiplexing done to achieve this, with the *split* input being a boolean input indicating whether the value belongs to the second cluster, that is, the cluster that is being split from the original cluster.

7.2.6 Calculating the statistics

The clustering algorithm in Section 7.2.2 requires the statistical values λ and $\mathbf{e}$ as given by (7.3) and (7.4). The calculation of these values requires division for the calculation of the means $\bar{x}$, $\bar{y}$, $\overline{xx}$, $\overline{yy}$ and $\overline{xy}$, multiplication for the calculation of $\bar{x}\bar{x}$, $\bar{y}\bar{y}$, $\bar{x}\bar{y}$, $(a-d)^2$ and $(2b)^2$, addition/subtraction, and a square root operation to calculate Δ from Δ^2. The square root operation can be obtained using the CORDIC [72] technique. For Xilinx Virtex 5, the divider, multiplier, and CORDIC square root operator can be generated using Xilinx CORE Generator.

Pipelining of the operations was used to use a single divider, a single multiplier, a single adder/subtractor and a single square root operator. Figure 7.11 shows the timing diagram used for the calculation of these values using pipelining. The total latency L depends on the divider latency D, the multiplier latency M, and the square root latency Q, and is given by

$$L = D + 2M + Q + 9. \tag{7.6}$$

Note that the adder/subtractor has a latency of one cycle. If the divider has a latency of 33 cycles, the multiplier has a latency of four cycles, and the square root operator has a latency of 10 cycles, the total latency is 60 cycles.

Each time a cluster is split in two, the statistical calculations have to be performed for each of the two new clusters. But the calculation for the second cluster can be started before the calculation for the first cluster is ready, as long as superimposing the two timing diagrams does not create any conflict. Thus, the divider, multiplier, adder/subtractor and square root operator can be shared between the two. Figure 7.12 shows the inputs to the mathematical operators for two parallel calculations with $D = 33$, $M = 4$ and $Q = 10$. If the calculation for the second cluster starts five cycles after the calculation for the first cluster, no collisions happen.

Figure 7.13 shows the architecture for the statistical calculator. Note that there are two FSMs, one for each of the two required calculations. The inputs to each mathematical operator are multiplexed from the two FSMs. There are four multiplexed operators: a divider, a multiplier, an adder/subtractor and a square root operator.

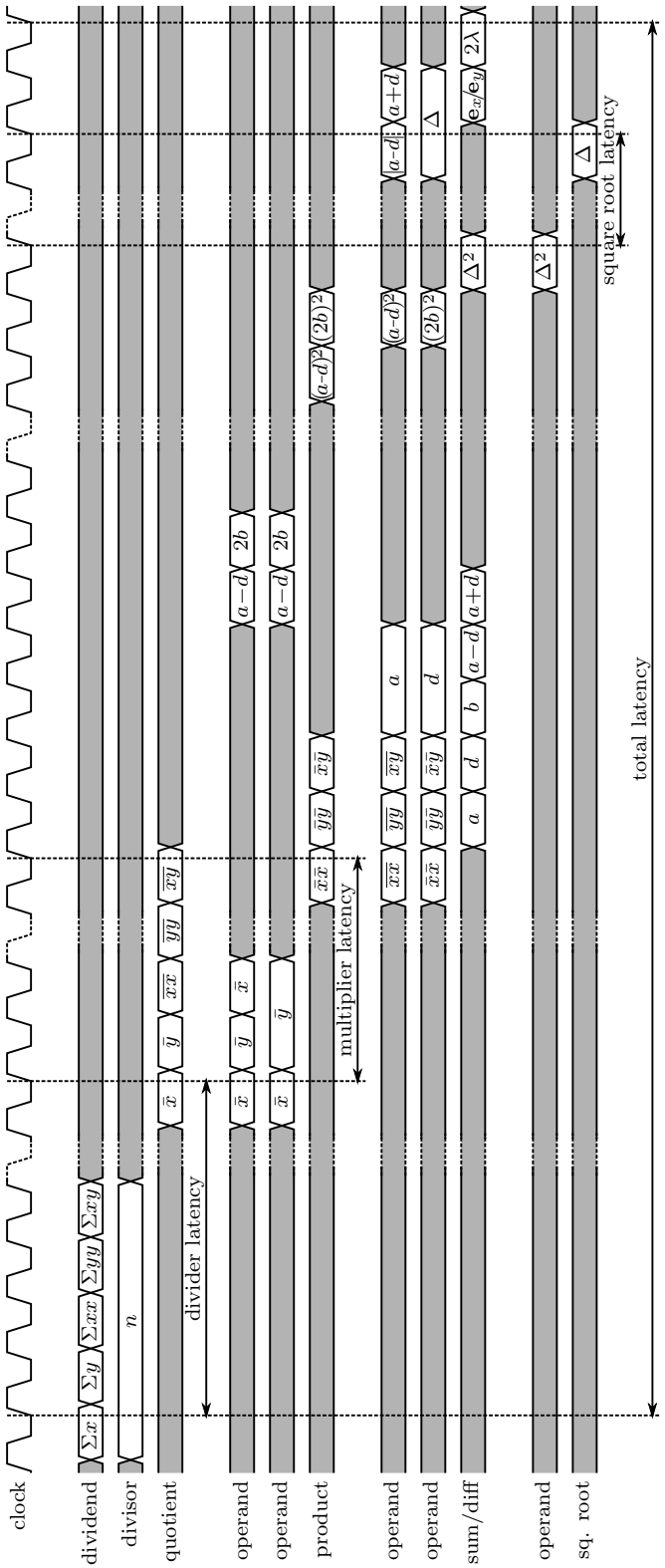


Figure 7.11: The timing diagram for the mathematical operators used in calculating the statistics.

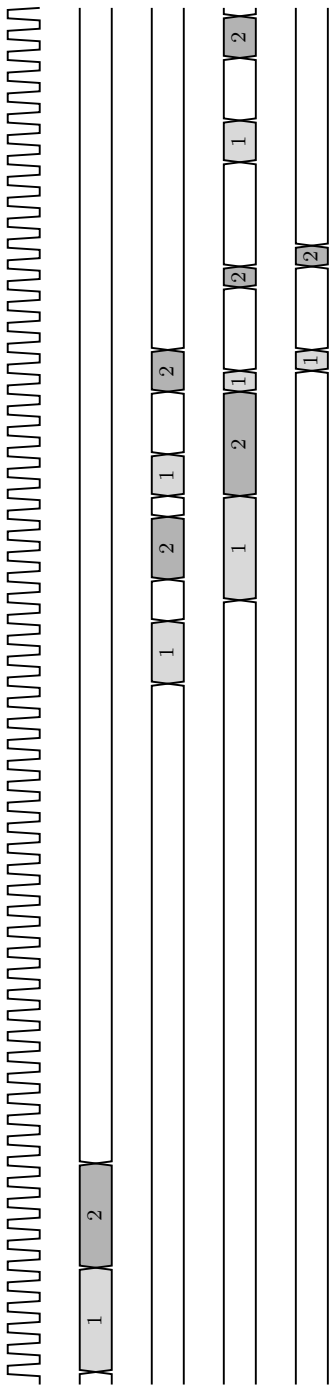


Figure 7.12: The inputs to the mathematical operators for calculating the statistics of two clusters (labelled 1 and 2) in parallel, with the divider latency $D = 33$, the multiplier latency $M = 4$, and the square root latency $Q = 10$.

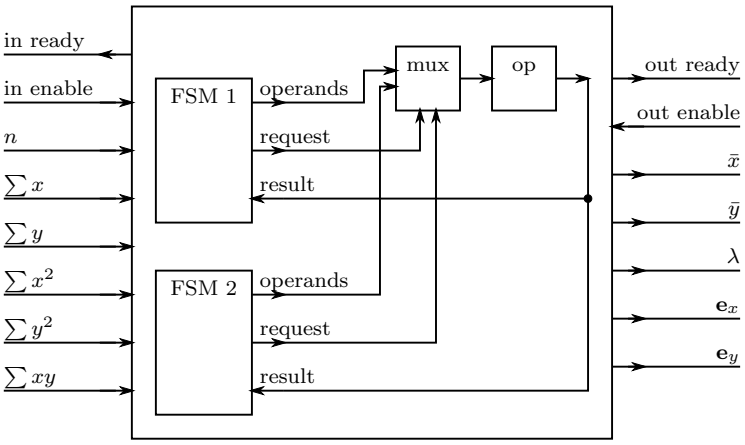


Figure 7.13: The architecture of the statistics calculator showing the multiplexing for one mathematical operator.

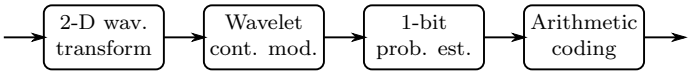


Figure 7.14: The scalable image compression components required for the universal reconfigurable compression system.

7.3 The temporally filtered frames

After the motion compensation steps, we are left with temporally-filtered frames that need to be encoded. Some components were implemented to do this. These components can be used to encode images as well. Figure 7.14 shows the subset of Figure 7.5 that is used to encode the temporally filtered frames. It is also the part of Figure 3.2 in Section 3.4 that processes scalable image compression.

7.3.1 Two-dimensional wavelet transforms

The temporally-filtered frames are first transformed using a two-dimensional discrete wavelet transform (DWT) [41], which generates a number of subbands. Figure 7.15 shows one level of the two-dimensional DWT. The wavelet filters use lifting [85]. The horizontal filter needs to buffer a number of values, and the vertical filter needs to buffer a number of horizontal lines. For the 5/3 filter, the horizontal filter needs to buffer two values, and the vertical filter needs to buffer two lines. If the width of the input image is w,

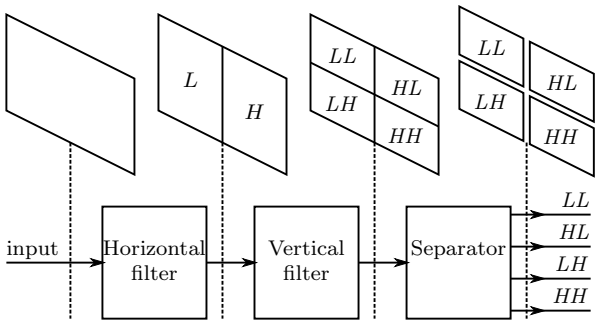


Figure 7.15: One level of the two-dimensional wavelet transform.

the filter introduces a latency of approximately $2w$, and one complete line of output will be produced for each of the sub-bands after three lines are input. After the first output line, another output line will be produced for every two input lines.

Let us denote the values before the DWT by $x_{0,k}$. If there are a total of n values, then $0 \leq k < n$. For the 5/3 filter, one stage of lifting is required, while for the 9/7 filter, two stages of lifting are required. The lifting process transforms the original values into low-pass values and high-pass values. The low-pass values are denoted by $x_{i,2k}$ and the high-pass values are denoted by $x_{i,2k+1}$, where $i = 1$ when one lifting stage is required and $i = 2$ when two lifting stages are required. The lifting process is given by

$$x_{i,2k+1} = x_{i-1,2k+1} - \alpha_i(x_{i-1,2k} + x_{i-1,2k+2}) \tag{7.7}$$

$$x_{i,2k} = x_{i-1,2k} \quad + \beta_i(x_{i,2k-1} + x_{i,2k+1}), \tag{7.8}$$

where (7.7) is the prediction step and (7.8) is the update step. For the 5/3 filter, $\alpha_1 = 1/2$ and $\beta_1 = 1/4$. It follows that both $x_{i,2k}$ and $x_{i,2k+1}$ depend on the values up to $x_{i-1,2k+2}$. So $x_{1,0}$ and $x_{1,1}$ depend on the values up to $x_{0,2}$. Similarly, $x_{2,0}$ and $x_{2,1}$ depend on the values up to $x_{1,2}$, which in turn depend on the values up to $x_{0,4}$.

If we use symmetric extension [47] at the edges of the input values, where $x_{0,0}$ is the first value and $x_{0,n-1}$ is the last value, we can write

$$x_{0,-k} = x_{0,k} \tag{7.9}$$

$$x_{0,n-1+k} = x_{0,n-1-k}. \tag{7.10}$$

From (7.7) to (7.10), it is trivial to show that

$$x_{i,-k} = x_{i,k} \tag{7.11}$$

$$x_{i,n-1+k} = x_{i,n-1-k}, \tag{7.12}$$

that is, symmetry at the edges of the input values leads to symmetry at the edges of the output values. This will have the important consequence that we do not need to generate or buffer any $x_{i,-k}$ and $x_{i,n-1+k}$ values at any time in the lifting process.

Let us consider the edge cases. At the beginning of the lifting process, that is, when $2k = 0$, we can rewrite (7.7) and (7.8) as

$$x_{i,1} = x_{i-1,1} - \alpha_i(x_{i-1,0} + x_{i-1,2}) \tag{7.13}$$

$$x_{i,0} = x_{i-1,0} + \beta_i(x_{i,-1} + x_{i,1}). \tag{7.14}$$

The reflected value $x_{i,-1}$ is used only together with $x_{i,1}$, and from (7.11) these two values are equal, so that no extra buffering is required.

At the end of the lifting process, we have two cases, the case when n is even and the case when n is odd. When n is even, the edge is at $2k+1 = n-1$, and we can rewrite (7.7) and (7.8) as

$$x_{i,n-1} = x_{i-1,n-1} - \alpha_i(x_{i-1,n-2} + x_{i-1,n}) \tag{7.15}$$

$$x_{i,n-2} = x_{i-1,n-2} + \beta_i(x_{i,n-3} + x_{i,n-1}). \tag{7.16}$$

The reflected value $x_{i-1,n}$ is used together with $x_{i-1,n-2}$, and from (7.12) these two values are equal, so no extra buffering is required.

When n is odd, the edge is at $2k = n - 1$, and we have to find the value of $x_{i,2k}$ only, and do not need to find the value of $x_{i,2k+1}$, which is only a reflection of $x_{i,2k-1}$. In this case, we can rewrite (7.8) as

$$x_{i,n-1} = x_{i-1,n-1} + \beta_i(x_{i,n-2} + x_{i,n}). \tag{7.17}$$

The reflected value $x_{i,n}$ is used together with $x_{i,n-2}$, and again from (7.12) these two values are equal, so no extra buffering is required here either.

Figure 7.16 shows how the two-dimensional DWT is cascaded for three levels of wavelet decomposition. The width of the input frame to the first level is w. The width of LL_0 is $w/2$, so the vertical filter in the second

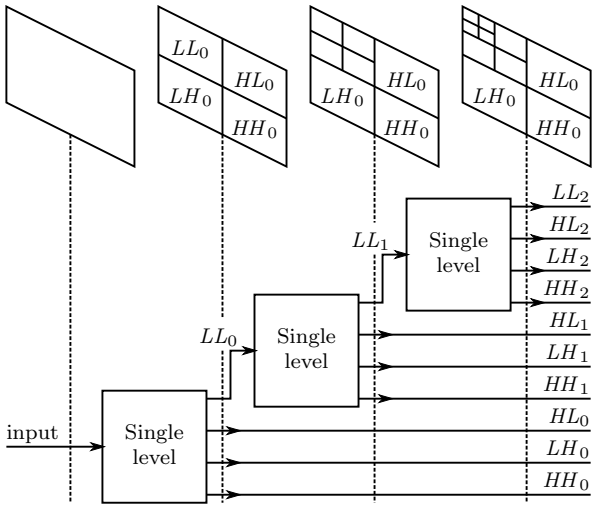


Figure 7.16: Three levels of the two-dimensional wavelet transform.

wavelet stage will need half the buffer space required by the first level.

For the 5/3 filter, after three lines of the original image are input into the first stage (think $x_{0,0}$, $x_{0,1}$ and $x_{0,2}$), the first line of LL_0 is produced ($x_{1,0}$). After two further lines are input ($x_{0,3}$ and $x_{0,4}$), the second line of LL_0 is produced ($x_{1,2}$, not $x_{1,1}$ which is high-pass). When seven lines of the original image are input into the first stage, three lines of LL_0 are produced. Just as three lines of the original image are needed to produce one line of LL_0, three lines of LL_0 are needed to produce one line of LL_1. Thus, when seven lines of the original image are processed, three lines of LL_0 are produced and input into the second stage, and the first line of LL_1 is produced.

Table 7.6 shows how many lines of the original image are required to output a given number of lines in the different wavelet sub-bands. For example, as described above, for the 5/3 filter, three input lines are required to output one LL_0 line, and seven input lines are required to output three lines. For the second level, three LL_0 lines are required to output one LL_1 line, which in turn require seven lines of the original image. For the third level, three LL_1 lines are required to output one LL_2 line, which in turn require seven LL_0 lines, which in turn require 15 lines of the original image. Similarly, for the 9/7 filter with two lifting stages, the output of one LL_2 line requires five LL_1 lines, which in turn require 13 LL_0 lines, which in turn require 29 lines of the original image.

It is important to note that this does not mean that 29 lines of the

TABLE 7.6: NUMBER OF WHOLE LINES OF INPUT REQUIRED TO OUTPUT A GIVEN NUMBER OF WHOLE LINES FOR THE GIVEN TWO-DIMENSIONAL WAVELET SUB-BANDS.

Output lines	5/3 LL_0	5/3 LL_1	5/3 LL_2	9/7 LL_0	9/7 LL_1	9/7 LL_2
1	3	7	15	5	13	29
2	5	11	23	7	17	37
3	7	15	31	9	21	45
4	9	19	39	11	25	53
5	11	23	47	13	29	61
⋮	⋮	⋮	⋮	⋮	⋮	⋮
n	$2n+1$	$4n+3$	$8n+7$	$2n+3$	$4n+9$	$8n+21$

original image need to be buffered. If the original image has a width of w, the two lifting stages of the first level will require $2w$ buffered values each, the two lifting stages of the second level will require $2w/2$ buffered values each, and the two lifting stages of the third level will require $2w/4$ buffered values each, for a total of $7w$ buffered values for three levels of the 9/7 filter. For the 5/3 filter, the requirements are half the requirements for the 9/7 filter, because only one lifting step is required as opposed to the two lifting steps required by the 9/7 filter.

7.3.2 Bit plane coding and context modelling

As described in Section 2.3.1, for each wavelet sub-band, the wavelet context modeller will process the bit planes in turn, so the output of the DWT are stored in SRAM by bit plane. This is so that when the context modeller is processing one bit plane, and it loads a word from SRAM, all the bits in the memory word are in the same bit plane, and no unnecessary bits are loaded. This means that the context modeller will not be reading from SRAM during all the cycles, as reading one word from memory will provide enough bits for several cycles. Thus, a number of wavelet sub-bands can be processed in parallel if required, with each context modeller reading from the SRAM in turn.

The context modelling used for the wavelet sub-bands is very similar to that used by ICER [12]. A wavelet sub-band is divided into code blocks with a limited size, this is done to limit the memory requirements of the context

modeller, and it gives the added advantage of error containment, where an error in the transmission of one code block does not affect the other code blocks.

For each bit plane, the code block is first divided into a number of sub-blocks. A sub-block is said to be significant if at least one pixel in it is non-zero. The significance of each sub-block is encoded in a hierarchical manner like in EBCOT [11]. If a 256×256 code block is subdivided into 16×16 sub-blocks, the number of bits required to store the significance hierarchically will be 4 for 128×128 sub-blocks, 16 for 64×64 sub-blocks, 64 for 32×32 sub-blocks and 256 for 16×16 sub-blocks, for a total of 340 bits. When a block is known to be significant from previous bit planes, its significance does not need to be encoded, and when a block is known to be insignificant from a coarser block, it does not need to be encoded either, so less than 340 bits are actually required.

After the significance of the sub-blocks in a bit plane is encoded, the significant sub-blocks are encoded with the context modelling detailed in [12]. The insignificant sub-blocks, which are all zero, are simply skipped. The context modeller needs to store three bits of state for each pixel, one for the sign and two for the category; these state bits are stored in SRAM.

As already mentioned, for each pixel we need the bit itself from the bit plane, and three bits for the modeller state. If the SRAM bus is 64-bit wide, four read cycles will give 64 bits from the bit plane and 64 context states of three bits each. The context modeller processes one pixel bit per cycle, so the four read cycles will provide enough data for the modeller to work for 64 cycles. One context modeller will thus use $4/64 = 1/16$ of the memory bandwidth, such that 16 context modellers can run in parallel. If eight context modellers are working in parallel, $8 \times 4/64 = 1/2$ of the memory read cycles are used by the context modellers.

7.4 Hardware cost

The algorithms for the motion vector palette encoder and for the wavelet transform and context modelling were implemented in VHDL, and simulated on ModelSim targeting a Xilinx Virtex 5 FPGA [94]. The clock period used in the timing constraints was 7.5 ns, leading to a frequency of 133 MHz, which is sufficient for encoding video at the required speed, as will be shown

Table 7.7: Dynamic power and resources required for a 133 MHz clock frequency implementation.

Component	Power (mW)	FFs	LUTs
Motion vector palette encoder	108.10	3417	7415
statistics calculator	*28.65*	*2252*	*4002*
FSM (2×)	*3.13*	*198*	*991*
divider	*16.15*	*1328*	*452*
multiplier	*0.00*	*271*	*402*
square root	*1.56*	*220*	*297*
2-D wavelet transform (3 levels)	123.74	2467	3457
Wavelet context modeller	88.48	184	780
Probability estimator	3.68	140	247
Arithmetic coder	1.41	163	462

below, while using higher clock frequencies would lead to higher dynamic power consumption. The dynamic power for the implementations was estimated using the Xilinx Power Analyzer. The power and resources required can be seen in Table 7.7. The motion vector palette encoder uses a total of about 108 mW. Three levels of the two-dimensional wavelet transform use 123 mW, while the context modeller, probability estimator and arithmetic coder combined use about 94 mW.

The wavelet transform uses one clock cycle for each input value, so that it can process 30 frames of size 2048×2048 per second at 133 MHz. Timing simulations show that the wavelet context modeller, probability estimator, and arithmetic coder can process a 256×256 code block in 5 ms. If required, a number of code blocks can be processed in parallel, since each code block can be encoded independently of other code blocks. The motion vector palette encoder can encode a set of 4096 motion vectors in 2.5 ms, which means that for bidirectional motion estimation, it will be able to process one frame every 5 ms, making it possible to process up to 200 frames per second. A 2048×2048 frame will give 4096 motion vectors using 32×32 macroblocks. For smaller frames, smaller macroblocks may be used.

The suitability of the designed components to process HD video sequences, with a 1920×1080 Y channel, and 960×540 U and V channels, at a rate of 25 frames per second, was also verified. One frame has to be processed every 40 ms. Eight wavelet context modellers in parallel will process

all the pixel data of a YUV frame in

$$\frac{1920 \times 1080 + 2 \times 960 \times 540}{256 \times 256} \times \frac{1}{8} \times 5 \text{ ms}$$

which is around 30 ms. Since only 30 ms are required for 40 ms of data, and since from Section 7.3.2 we know that eight context modellers in parallel will use 50% of the memory bandwidth for that time, the context modelling will require only 37.5% of the memory bandwidth. The wavelet transform and the motion estimation engine can use the remaining memory bandwidth.

7.5 Validation

The designs were validated by processing test data sets and comparing the output with output produced by the software implementation of the algorithm. For the wavelet transforms and wavelet context modelling, the output of the timing simulation and the output of a software implementation were identical. For the motion vector palette encoder, the results were not identical, but virtually equivalent. Since the motion vector palette encoder makes use of fixed point arithmetic in the hardware implementation, some of the intermediate values used in the divisive clustering algorithm are not identical to the corresponding values of the software algorithm, which makes use of double-precision floating point arithmetic. However, the resulting difference in the clustering of the motion vectors resulted in a negligible difference in the size of the compressed video bitstream after temporal filtering and wavelet compression.

7.6 Conclusion

In this chapter, the hardware amenability of the proposed scalable video coding algorithm was demonstrated. Tools for the analysis and configuration of the motion estimation engine were presented, and analysis of motion estimation algorithms which leads to the proper configuration of the engine was presented. Viable designs for the motion vector palette generator and encoder, as well as for spatial wavelet transforms and context modelling, were presented. The hardware cost of these components was analysed, and the algorithms' amenability to hardware implementation was demonstrated.

Chapter 8

Conclusion

This chapter summarizes the main achievements of this work. Section 8.1 shows how the objectives listed in Chapter 1 were achieved. Section 8.2 suggests some areas which can be investigated further in the future.

8.1 Achieved objectives

In Section 1.4, five main objectives were listed. The objectives follow below, with some discussion to show how the objectives were achieved. The objectives were

1. *to investigate the existing universal compression system, and how an SVC algorithm can be incorporated into it,*

   Chapter 3 investigated which blocks of the existing universal compression system can be used by the SVC algorithm and gave an overview of how the universal compression system can be reconfigured for SVC. The SVC algorithm follows the same basic stages of the existing components, that is, preprocessing and context modelling, probability estimation, and arithmetic coding. It also uses the same motion estimation engine as the lossless video component of the existing system. Thus, it can be used to extend the existing universal compression system.

2. *to investigate the scalable encoding of motion vectors such that they can be included in the video bitstream generated by the scalable video system,*

In order to enable scalability down to very low bit rates, the presented video compression algorithm makes use of layered motion vector palettes to encode the motion vector side information of the video in a scalable manner, as shown in Chapter 4. The multi-layer motion vector palettes provide a flexible system for scalable encoding of the motion vectors that can retain high precision for some motion vectors while using lower precision for other motion vectors according to their frequency. This makes it possible to scale the transmitted video bitstream to very low bit rates. At high bit rates, the overhead of encoding the motion vectors in layers only causes a negligible increase in the bit rate. At low bit rates, the layered motion vector palette scheme improves the rate-distortion characteristic of the coding scheme by allowing the bit rate of the side information to be decreased together with the bit rate of the frame information.

3. *to investigate and analyse an SVC algorithm that generates a video bitstream which is scalable through a large range of bit rates, from low bit rates to high bit rates, without the need of reencoding, and that is compatible with the existing universal compression system,*

   As well as its suitability for very low bit rates thanks to the scalable motion vector encoding, the algorithm includes scalable video coding methods suitable for a large range of bit rates as demonstrated in Chapter 5. Encoding the video sequence at the highest possible bit rate does not have any side-effects on the compression performance at low bit rates. If very high quality is required, the reversible 5/3 wavelet filter can be used in the spatial coding stage, which enables the compressed video bitstream to be scaled all the way up to lossless.

4. *to analyse the compression performance of the algorithm to ensure it has good compression performance comparable to other SVC systems,*

   The analysis in Chapter 6 showed that the scheme outperforms Motion JPEG 2000, which is a wavelet-based algorithm that does not make use of motion compensation. The proposed SVC scheme was also shown to offer compression performance that compares well to the state-of-the-art JSVM encoder, outperforming it when compressing a number of video sequences. In addition to this, the proposed scheme gives a finer and larger range of scalability.

5. *to design for low complexity so that the algorithm is amenable to hardware implementations, and demonstrate the suitability of the algorithm for hardware implementations.*

   While designing the algorithm, care was taken to keep the complexity in check so that the scheme is suitable for hardware implementations. Since the algorithm was kept as simple as possible, the processing and memory requirements of the designed method were much lower than the requirements of the JSVM encoder. As indicated in Chapter 7, the suitability of the developed algorithms for reconfigurable hardware was demonstrated by the design, implementation and simulation of all the main components of the system. The motion vector palette generator and encoder, and the wavelet transform, context modeller, probability estimator and arithmetic coder, were synthesized and simulated targeting a Xilinx Virtex 5 FPGA at a clock frequency of 133 MHz, and the algorithm was shown to be more than suitable for the real time encoding of 1920×1080 YUV video sequences.

   During the design, specialized components were designed to lower the hardware complexity and circuit size of the system. For example, a specialized divisive clustering component, a carefully pipelined statistics calculator, and an accumulator capable of accumulating two sums using only one adder, were developed. Using these specialized components instead of adopting generic equivalents led to lower hardware cost of the system.

8.2 Future work

EBCOT [11] and ICER [12] provide error containment by splitting an image into code blocks. Errors in a code block do not affect other code blocks in the same image. While the image is still split into code blocks, errors in a code block of a frame might still cause errors in different code blocks of a different frames owing to motion compensation. This effect could be investigated, and ways to maintain the error containment capabilities of the mentioned algorithms could be explored.

The contexts used for context modelling of wavelet sub-bands after spatial coding are borrowed from ICER, and are already optimized. However, the contexts used to encode the motion vector palettes are not optimized.

While their performance was demonstrated to be good, further study could be done to see if different context selection can lead to an improvement in the compression of motion vectors using the motion vector palettes.

In this work, motion vector palettes were used for wavelet-based SVC. The effect of using motion vector palettes as a method to encode the motion vector side information in other hybrid video coding systems can be investigated.

The hardware components designed in this work were meant to demonstrate that it is possible to implement the algorithms in hardware, but are not optimized. Future work might include optimizing the critical paths of the designed components such that the speed of the reconfigurable system is not limited by the SVC components.

Wavelet sub-bands can be encoded in parallel in hardware, which means that they are easy to parallelize. After the critical path optimization mentioned above, the other components can be investigated to see if there is room for more parallelization, which can lead to faster execution times after the.

References

[1] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1103–1120, Sep. 2007.

[2] N. Adami, A. Signoroni, and R. Leonardi, "State-of-the-art and trends in scalable video compression with wavelet-based approaches," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1238–1255, Sep. 2007.

[3] *Advanced video coding for generic audiovisual services*, International Telecommunication Union Rec. ITU-T H.264, Version 8: 11/2007. [Online]. Available: http://www.itu.int/rec/T-REC-H.264

[4] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi, "Video coding with H.264/AVC: Tools, performance and complexity," *IEEE Circuits and Systems Magazine*, vol. 4, no. 1, pp. 7–28, 2004.

[5] SVC reference software (JSVM software). [Online]. Available: http://ip.hhi.de/imagecom_G1/savce/downloads/SVC-Reference-Software.htm

[6] Y.-Q. Zhang and S. Zafar, "Motion-compensated wavelet transform coding for color video compression," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 2, no. 3, pp. 285–296, Sep. 1992.

[7] J.-R. Ohm, "Three-dimensional subband coding with motion compensation," *IEEE Transactions on Image Processing*, vol. 3, no. 5, pp. 559–571, Apr. 1994.

[8] S.-J. Choi and J. W. Woods, "Motion-compensated 3-D subband coding of video," *IEEE Transactions on Image Processing*, vol. 8, no. 2, pp. 155–167, Feb. 1999.

[9] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3445–3462, Dec. 1993.

[10] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 243–250, Jun. 1996.

[11] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Transactions on Image Processing*, vol. 9, no. 7, pp. 1158–1170, Jul. 2000.

[12] A. Kiely and M. Klimesh, "The ICER progressive wavelet image compressor," Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, The Interplanetary Network Progress Report 42-155, Jul.–Sep. 2003. [Online]. Available: http://ipnpr.jpl.nasa.gov/progress_report/42-155/155J.pdf

[13] D. Taubman and A. Zakhor, "Multirate 3-D subband coding of video," *IEEE Transactions on Image Processing*, vol. 3, no. 5, pp. 572–588, Sep. 1994.

[14] S.-T. Hsiang and J. W. Woods, "Embedded video coding using invertible motion compensated 3-D subband/wavelet filter bank," *Signal Processing: Image Communications*, vol. 16, no. 8, pp. 705–724, May 2001.

[15] A. Secker and D. Taubman, "Lifting-based invertible motion adaptive transform (LIMAT) framework for highly scalable video compression," *IEEE Transactions on Image Processing*, vol. 12, no. 12, pp. 1530–1542, Dec. 2003.

[16] L. Luo, F. Wu, S. Li, Z. Xiong, and Z. Zhuang, "Advanced motion threading for 3D wavelet video coding," *Signal Processing: Image Communications*, vol. 19, no. 7, pp. 601–616, Aug. 2004.

[17] R. Xiong, J. Xu, F. Wu, and S. Li, "Barbell-lifting based 3-D wavelet coding scheme," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1256–1269, Sep. 2007.

[18] R. Xiong, J. Xu, F. Wu, S. Li, and Y.-Q. Zhang, "Layered motion estimation and coding for fully scalable 3D wavelet video coding," in *International Conference on Image Processing*, vol. 4, Oct. 2004, pp. 2271–2274.

[19] A. Secker and D. Taubman, "Highly scalable video compression with scalable motion coding," *IEEE Transactions on Image Processing*, vol. 13, no. 8, pp. 1029–1041, Aug. 2004.

[20] J. Villasenor, C. Jones, and B. Schoner, "Video communications using rapidly reconfigurable hardware," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, no. 6, pp. 565–567, Dec. 1995.

[21] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, May 2000.

[22] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings Design Automation Conference*, Jun. 2000, pp. 507–512.

[23] G. Stitt, F. Vahid, and S. Nematbakhsh, "Energy savings and speedups from partitioning critical software loops to hardware in embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 1, pp. 218–232, Feb. 2004.

[24] T. W. Fry and S. Hauck, "Hyperspectral image compression on reconfigurable platforms," in *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2002, pp. 251–260.

[25] A. S. Dawood, J. A. Williams, and S. J. Visser, "On-board satellite image compression using reconfigurable FPGAs," in *Proceedings IEEE International Conference on Field-Programmable Technology*, Dec. 2002, pp. 306–310.

[26] *JPEG 2000 image coding system*, International Organization for Standardization/International Electrotechnical Commission and International Telecommunication Union Rec. ISO/IEC 15 444-1 and ITU-T T.800.

[27] C.-J. Lian, K.-F. Chen, H.-H. Chen, and L.-G. Chen, "Analysis and architecture design of block-coding engine for EBCOT in JPEG 2000," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 3, pp. 219–230, Mar. 2003.

[28] A. Tumeo, S. Borgio, D. Bosisio, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "A multiprocessor self-reconfigurable JPEG2000 encoder," in *IEEE International Symposium on Parallel and Distributed Processing*, May 2009, pp. 1–8.

[29] X. Zhao, A. T. Erdogan, and T. Arslan, "A hybrid dual-core reconfigurable processor for EBCOT tier-1 encoder in JPEG2000 on next generation of digital cameras," in *Conference on Design and Architectures for Signal and Image Processing*, Oct. 2010, pp. 84–89.

[30] G. Yu, T. Vladimirova, X. Wu, and M. N. Sweeting, "A new high-level reconfigurable lossless image compression system for space applications," in *NASA/ESA Conference on Adaptive Hardware and Systems*, Jun. 2008, pp. 183–190.

[31] J. L. Nunez-Yanez and V. A. Chouliaras, "A configurable statistical lossless compression core based on variable order Markov modeling and arithmetic coding," *IEEE Transactions on Computers*, vol. 54, no. 11, pp. 1345–1359, Nov. 2005.

[32] J. L. Nuñez-Yañez, X. Chen, N. Canagarajah, and R. Vitulli, "Statistical lossless compression of space imagery and general data in a reconfigurable architecture," in *NASA/ESA Conference on Adaptive Hardware and Systems, 2008*, Jun. 2008, pp. 172–177.

[33] X. Chen, N. Canagarajah, J. L. Nuñez-Yañez, and R. Vitulli, "Lossless compression for space imagery in a dynamically reconfigurable architecture," in *Workshop in Applied Reconfigurable Computing*, 2008, pp. 336–341.

[34] X. Chen, N. Canagarajah, and J. L. Nunez-Yanez, "Lossless multi-mode interband image compression and its hardware architecture," in *Conference on Design and Architectures for Signal and Image Processing*, Nov. 2008, pp. 208–215.

[35] X. Chen, N. Canagarajah, and J. L. Nunez-Yanez, "Backward adaptive pixel-based fast predictive motion estimation," *IEEE Signal Processing Letters*, vol. 16, no. 5, pp. 370–373, May 2009.

[36] J. L. Nunez-Yanez, T. Spiteri, and G. Vafiadis, "Multi-standard reconfigurable motion estimation processor for hybrid video codecs," *IET Computer and Digital Techniques*, vol. 5, no. 2, pp. 73–85, Mar. 2011.

[37] *Information Technology—MPEG systems technologies—Part 4: Codec configuration representation*, International Organization for Standardization/International Electrotechnical Commission and International Telecommunication Union Rec. ISO/IEC 23 001-4:2011.

[38] G. Yu, T. Vladimirova, and M. N. Sweeting, "Image compression systems on board satellites," *Acta Astronautica*, vol. 64, no. 9–10, pp. 988–1005, Feb. 2009.

[39] E. J. Delp and O. R. Mitchell, "Image compression using block truncation coding," vol. 27, no. 9, pp. 1335–1342, Sep. 1979.

[40] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE Transactions on Computers*, vol. C-23, no. 1, pp. 90–93, Jan. 1974.

[41] S. G. Mallat, "A theory for multiresolution signal decomposition: The wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674–693, Jul. 1989.

[42] B. E. Usevich, "A tutorial on modern lossy wavelet image compression: Foundations of JPEG 2000," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 22–35, Sep. 2001.

[43] *Image Data Compression*, Blue Book, Issue 1, The Consultative Committee for Space Data Systems (CCSDS) Recommended Standard 122.0-B-1, Nov. 2005. [Online]. Available: http://www.ccsds.org/

[44] W. Sweldens, "The lifting scheme: A new philosophy in biorthogonal wavelet constructions," in *Mathematical Imaging: Wavelet Applications in Signal and Image Processing III*, A. F. Laine and M. A. Unser, Eds., vol. 2569. SPIE, Sep. 1995, pp. 68–79.

[45] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Transactions on Image Processing*, vol. 9, no. 3, pp. 378–389, Mar. 2000.

[46] Y.-H. Seo and D.-W. Kim, "VLSI architecture of line-based lifting wavelet transform for motion JPEG2000," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 2, pp. 431–440, Feb. 2007.

[47] M. J. T. Smith and S. L. Eddins, "Subband coding of images with octave band tree structures," in *International Conference on Acoustics, Speech, and Signal Processing*, vol. 12, Apr. 1987, pp. 1382–1385.

[48] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG 2000 still image compression standard," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 36–58, Sep. 2001.

[49] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Transactions on Image Processing*, vol. 1, no. 2, pp. 205–220, Apr. 1992.

[50] D. Le Gall and A. Tabatabai, "Sub-band coding of digital images using symmetric short kernel filters and arithmetic coding techniques," in *International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, Apr. 1988, pp. 761–764.

[51] A. Petukhov, "Best wavelet bases for image compression generated by rational symbols," *HAIT Journal of Science and Engineeringn C*, vol. 4, no. 1–2, pp. 263–271, 2007.

[52] A. F. Abdelnour and I. W. Selesnick, "Symmetric nearly shift-invariant tight frame wavelets," *IEEE Transactions on Signal Processing*, vol. 53, no. 1, pp. 231–239, Jan. 2005.

[53] A. F. Abdelnour, "Design of 6-band tight frame wavelets with limited redundancy," in *IEEE International Symposium on Signal Processing and Information Technology*, Aug. 2006, pp. 133–137.

[54] P. G. Howard, "Interleaving entropy codes," in *Proceedings Compression and Complexity of Sequences*, Jun. 1997, pp. 45–55.

[55] S. Saponara, K. Denolf, G. Lafruit, C. Blanch, and J. Bormans, "Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications," *EURASIP Journal on Applied Signal Processing*, vol. 2004, no. 1, pp. 220–235, Jan. 2004.

[56] S.-C. Cheng and H.-M. Hang, "A comparison of block-matching algorithms mapped to systolic-array implementation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 5, pp. 741–757, Oct. 1997.

[57] A. Hamosfakidis and Y. Paker, "A novel hexagonal search algorithm for fast block matching motion estimation," *Journal on Applied Signal Processing*, vol. 2002, no. 6, pp. 595–600, Jun. 2002.

[58] X. Yi, J. Zhang, N. Ling, and W. Shang, "Improved and simplified fast motion estimation for JM," *Joint Video Team of ISO/IEC MPEG & ITU-T VCEG, 16th Meeting, Poznan, Poland*, Jul. 2005, JVT-P021.doc.

[59] J. Zhang, X. Yi, N. Ling, and W. Shang, "Bit rate distribution analysis for motion estimation in H.264," *Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on*, pp. 483–484, Jan. 2006.

[60] T. Dias, S. Momcilovic, N. Roma, and L. Sousa, "Adaptive motion estimation processor for autonomous video devices," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, pp. 41–41, Jan. 2007.

[61] K. Babionitakis, G. A. Doumenis, G. Georgakarakos, G. Lentaris, K. Nakos, D. Reisis, I. Sifnaios, and N. Vlassopoulos, "A real-time motion estimation FPGA architecture," *Journal of Real-Time Image Processing*, vol. 3, no. 1–2, pp. 3–20, Mar. 2008.

[62] H.264 motion estimation engine (DO-DI-H264-ME). [Online]. Available: http://www.xilinx.com/products/ipcenter/DO-DI-H264-ME.htm

[63] T. Spiteri, G. Vafiadis, and J. L. Nunez-Yanez, "A toolset for the analysis and optimization of motion estimation algorithms and processors," in *International Conference on Field Programmable Logic and Applications*, Aug.–Sep. 2009, pp. 423–428.

[64] Tensilica's processor technology. [Online]. Available: http://www.tensilica.com/products/xtensa/index.htm

[65] D. Taubman, "Successive refinement of video: Fundamental issues, past efforts and new directions," in *Visual Communications and Image Processing*, T. Ebrahimi and T. Sikora, Eds., vol. 5150. SPIE, Jun. 2003, pp. 649–663.

[66] Y. Andreopoulos, A. Munteanu, G. Van der Auwera, J. P. H. Cornelis, and P. Schelkens, "Complete-to-overcomplete discrete wavelet transforms: Theory and applications," *IEEE Transactions on Signal Processing*, vol. 53, no. 4, pp. 1398–1412, Apr. 2005.

[67] S. Lindner and J. A. Robinson, "Motion palette coding for video compression," in *IEEE Canadian Conference on Electrical and Computer Engineering*, vol. 1, May 1997, pp. 375–378.

[68] M. T. Orchard and C. A. Bouman, "Color quantization of images," *IEEE Transactions on Signal Processing*, vol. 39, no. 12, pp. 2677–2690, Dec. 1991.

[69] P. J. Ausbeck Jr., "Context models for palette images," in *Proceedings Data Compression Conference*, Mar.–Apr. 1998, pp. 309–318.

[70] T.-W. Chen and S.-Y. Chien, "Flexible hardware architecutre of hierarchical K-means clustering for large cluster number," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1336–1345, Aug. 2011.

[71] I. Bravo, P. Jiménez, M. Mazo, J. L. Lázaro, and A. Gardel, "Implementation in FPGAs of Jacobi method to solve the eigenvalue and eigenvector problem," in *International Conference on Field Programmable Logic and Applications*, Aug. 2006, pp. 729–372.

[72] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions Electronic Computurs*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959.

[73] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *International Conference on Field Programmable Logic and Applications*, Aug. 2006, pp. 1–6.

[74] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," in *IEE Proceedings Computer and Digital Techniques*, May 2006, pp. 157–164.

[75] L. Kessal, N. Abel, S. M. Karabernou, and D. Demigny, "Reconfigurable computing: design methodology and hardware tasks scheduling for real-time image processing," *Journal of Real-Time Image Processing*, vol. 3, no. 3, pp. 131–147, 2008.

[76] A. Nabina and J. L. Nuñez-Yañez, "Dynamic reconfiguration optimisation with streaming data decompression," in *International Conference on Field Programmable Logic and Applications*, Aug.–Sep. 2010, pp. 602–607.

[77] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987.

[78] J. L. Nuñez and V. A. Chouliaras, "High-performance arithmetic coding VLSI macro for the H264 video compression standard," *IEEE Transactions on Consumer Electronics*, vol. 51, no. 1, pp. 144–151, Feb. 2005.

[79] L. Bottou, P. G. Howard, and Y. Bengio, "The Z-coder adaptive binary coder," in *Proceedings of the Data Compression Conference*, Mar. 1998, pp. 13–22.

[80] S. W. Golomb, "Run-length encodings (correspondence)," *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, Jul. 1966.

[81] Y. Li and K. Sayood, "Lossless video sequence compression using adaptive prediction," *IEEE Transactions on Image Processing*, vol. 16, no. 4, pp. 997–1007, Apr. 2007.

[82] L. Bottou, P. G. Howard, and Y. Bengio, "The Z-coder adaptive binary coder," in *Proceedings Data Compression Conference*, Mar.–Apr. 1998, pp. 13–22.

[83] Lehrstuhl für Datenverarbeitung, Technische Universität München. Test sequences 1080p. [Online]. Available: ftp://ftp.ldv.e-technik.tu-muenchen.de/dist/test_sequences/1080p/

[84] J. Teuhola, "A compression method for clustered bit-vectors," *Information Processing Letters*, vol. 7, no. 6, pp. 308–311, Oct. 1978.

[85] M. D. Adams and F. Kossentini, "Reversible integer-to-integer wavelet transforms for image compression: Performance evaluation and analysis," *IEEE Transactions on Image Processing*, vol. 9, no. 6, pp. 1010–1024, Jun. 2000.

[86] Kakadu Software. [Online]. Available: http://www.kakadusoftware.com/

[87] GCC. [Online]. Available: http://gcc.gnu.org/

[88] Intel Core i5-760 processor (8M cache, 2.80 GHz). [Online]. Available: http://ark.intel.com/products/48496/Intel-Core-i5-760-Processor-(8M-Cache-2_80-GHz)

[89] Valgrind. [Online]. Available: http://valgrind.org/

[90] x264. [Online]. Available: http://www.videolan.org/developers/x264.html

[91] J. L. Nunez-Yanez, E. Hung, and V. A. Chouliaras, "A configurable and programmable motion estimation processor for the H.264 video codec," in *International Conference on Field Programmable Logic and Applications*, Sep. 2008, pp. 149–154.

[92] Reconfigurable motion estimation engine for H.264. [Online]. Available: http://sharpeye.borelspace.com/

[93] J. W. J. Williams, "Algorithm 232 (HEAPSORT)," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, Jun. 1964.

[94] Xilinx Virtex 5. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-5.htm

Publications

T. Spiteri, G. Vafiadis, and J. L. Núñez-Yáñez, "A toolset for the analysis and optimization of motion estimation algorithms and processors," in *International Conference on Field Programmable Logic and Applications*, Aug.–Sep. 2009, pp. 423–428.

T. Spiteri, G. Vafiadis, and J. L. Núñez-Yáñez, "Flexible motion estimation processors for high definition video coding," in *Fifth UK Embedded Forum*, Sep. 2009, pp. 24–29.

T. Spiteri and J. L. Núñez-Yáñez, "Fine-grained vectorless scalable video coding for space applications," in *Second International Workshop on On-Board Payload Data Compression, ESA*, Oct. 2010.

J. L. Núñez-Yáñez, **T. Spiteri**, and G. Vafiadis, "Multi-standard reconfigurable motion estimation processor for hybrid video codecs," *IET Computer and Digital Techniques*, vol. 5, no. 2, pp. 73–85, Mar. 2011.

T. Spiteri and J. L. Núñez-Yáñez, "Scalable video coding with multi-layer motion vector palettes," *IET Image Processing*, vol. 6, no. 9, pp. 1319–1330, Dec. 2012.