REAL-TIME IMPLEMENTATION OF AN OBJECT-BASED CODEC

by

Fergal Connor B.Eng.

A thesis submitted in partial fulfilment of
the requirements for the degree of
Masters in Electronic Engineering

Supervisor: Dr. Thomas Curran

School of Electronic Engineering

Dublin City University

September 1997

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters in Electronic Engineering is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my own work.

Signed [illegible]

ID number 93701187

Date 7/10/97

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

LIST OF TABLES

Abstract

REAL-TIME IMPLEMENTATION OF AN OBJECT-BASED CODEC

Fergal Connor

Modern video coding algorithms are becoming increasingly complex with the result that single general purpose processors are incapable of meeting the computational power required for real time implementation. The coding algorithms are continuously evolving therefore, any multiprocessor solution must not only possess the necessary computational power but must also be flexible enough to adapt to any modifications in the algorithms.

This report presents a possible multiprocessor solution with specific reference to the DCU object-based analysis-synthesis coder. Firstly, an abstract model of the multiprocessor system is defined. The model is based on the dual requirements of computational power and flexibility. An analysis of the DCU coding algorithm is performed in order to refine the basic model by identifying potential realisation options that optimise coder performance. A reciprocal relationship exists whereby hardware constraints require modification of the algorithm. Any modifications are outlined and their effect on overall coder performance is investigated. Computational power costs are given for an implementation based on TMS320C30 DSPs.

From experimental results it is shown that, despite the complexity of the coding algorithm, real time operation is possible. A decoder based on a single TMS320C30 has been developed that is capable of operating at up to 8 Hz.

ACKNOWLEDGMENTS

Firstly, I would like to thank my supervisor, Dr. Thomas Curran, for giving me the opportunity to work in a very interesting area. I would also like to thank the guys in the Video Coding Group at Teltec DCU who always had an answer. My thanks to Cathriona for her encouragement and word skills. Finally, to my parents for their patience and support throughout my academic life.

Chapter 1

INTRODUCTION

1.1 Background

Recent years have seen the ratification of a number of compression standards for the storage and transmission of video, notably MPEG-1 & 2 [1] [2] from the ISO and H.261, H.263 [3][4] from the ITU-T. MPEG-1 is optimised for the storage of audio-visual data at 1.5Mbit/s - basically the data transfer rate available from a double speed CD-ROM - while MPEG-2 is aimed at higher quality, higher bitrate (4-9Mbit/s) for use in digital transmission of broadcast TV. H.261 is the video coding section of the H.320 standard that targets real-time audio-visual communication over ISDN lines. The H.320 standard is designed to work at bitrates available from a single ISDN channel (64kbit/s) to the entire capacity of 2Mbit/s. H.263 is based on H.261 but was developed for video coding over narrowband communication channels (<64kbit/s). All these standards use the same fundamental techniques to achieve video compression. Block-based motion compensation prediction exploits temporal redundancy and DCT is used to remove spatial redundancy.

These standards have found use in many applications that have facilitated the breakdown of traditional boundaries between the telecommunications, computer, and TV/film industries. Audio-visual and communication capabilities are being incorporated into computers, interactive television is being developed, and video and interactivity are being added to telecommunications. The Moving Pictures Expert Group (MPEG) have identified this convergence of industries along with three other major trends - the trend towards wireless communication, the trend towards interactive computer applications, and the increased use of audio-visual data in applications. To address these new expectations and requirements, MPEG have proposed the development of a new standard, known as MPEG-4, that supports new methods for communications, access and manipulation of digital audio-visual data. The focus, the functionalities to be supported, the structure and some of the potential applications of this new standard are described in the MPEG-4 Proposal Package Description (PPD) [5].

MPEG-4 specifies several functionalities that support the envisioned audio-visual applications. In particular, eight key functionalities are distinguished as not being well supported by existing or emerging standards, one of which is to make substantial improvements in coding efficiency, in order to provide subjectively better audio-visual quality at comparable bitrates. Many of the functionalities are content-based, i.e. they require the ability to extract information from an audio-visual scene. These content-based functionalities would, for example, allow interactive or automatic selection of the decoded quality of any particular object(s) in a scene.

The implementation of these functionalities requires fundamentally new algorithmic techniques. In the area of video coding, many new algorithms, commonly referred to as second generation video coding techniques, that address the requirements of MPEG-4 are currently under development. These algorithms are typically more computationally intensive than any existing standard. MPEG-4 is not due for ratification until 1998 and it is expected that technology will have progressed enough to be capable of providing the necessary computational power. However, there remains a shorter term need for hardware systems that can assist in the development and facilitate the trial of the new algorithms. Since the algorithms are continuously evolving, they also have the added requirement that any hardware implementation must be programmable in order to adapt to any necessary changes. Hardware systems for the real-time implementation of existing standards achieve the necessary computational power through the use of specialised hardware and VLSI. However, such solutions are not very programmable and restrict the modification of the algorithms and their parameters. Therefore, these specialised architectures are unsuitable for testing, evaluating or comparing second generation algorithms.

1.2 Objectives of Research

The main objective of the research was to develop a possible hardware system capable of real-time implementation of the DCU Object-Based Analysis-Synthesis Codec. The DCU coder is an example of a second generation video coding algorithm and is aimed at very low bitrate videophone applications. The DCU algorithm is fairly indicative of many second generation video coding techniques in terms of its computational complexity and the basic tools that comprise the algorithm. Execution of the DCU algorithm is asymmetric - coding requires more processing power than decoding - therefore most of the research effort was spent on the coder. For videophone applications, a frame rate of 8-10 Hz is considered to provide acceptable quality and as such this was taken as the target for real-time operation. The proposed solution achieves this through the use of a multiprocessor system and the modification of the DCU algorithm. The construction of the system was considered beyond the scope of the research. However, a demonstration decoder based on a single processor was produced.

Although the system was directed towards the DCU object-based analysis-synthesis codec, it is anticipated that the structure would be suitable for the implementation of other video coding algorithms.

1.3 Structure of Thesis

Chapter 2 begins with a general introduction to second generation video coding methods and then leads on to an explanation of the theory behind object-based analysis-coding. The remainder of the chapter describes the DCU coding algorithm and includes an implementation-independent profiling of the algorithm.

In Chapter 3 the iterative design methodology used for the development of the real-time implementation of the DCU coder is described. This is followed by an analysis of the DCU coding scheme to detect any exploitable parallelism, which leads to a proposed parallel implementation of the algorithm. The hardware structure necessary to support this parallel algorithm is also briefly discussed.

In Chapter 4 the software development process is described. This begins with an introduction to the target processor (the TMS320C30 DSP), the software development tools and testing methods employed. Also included is a number of programming guidelines for producing efficient software on the TMS320C30. The remainder of Chapter 4 details the implementation of major elements of the DCU algorithm.

The performance of this software is analysed in Chapter 5, firstly for a sequential execution on a single DSP and, secondly, for the parallel algorithm proposed in Chapter 3.

The concluding chapter, Chapter 6, includes recommendations for further enhancements of the software, based on experience in developing the software to its current state and the analysis of Chapter 5.Finally, a number of appendices are included which contain, in more detail, results summarised in the main body of the thesis.

Chapter 2

VERY LOW BITRATE VIDEO CODING

2.1 Introduction

Most video coding techniques exploit the fact that in successive frames of a video sequence there will exist some temporal redundancy. Existing video coding standards such as H.261 use a block-based scheme where images are subdivided into square blocks of NxN pels. The basic scheme is to use motion compensation to make a prediction of the current frame from the previous frame, based on the estimated motion of each block. Motion compensation is aimed at removing temporal redundancy. A DCT is applied to the prediction error of each block to exploit any redundancy in the spatial domain. The DCT coefficients are then subject to quantisation, the objective being to set many of them to zero. The results of the whole process - the DCT coefficients, the motion vectors from motion estimation and the quantisation parameters - are processed using entropy coding before transmission.

The H.263 standard is similar in operation to H.261 with some variations, notably in motion compensation, to adapt it for low bitrates (<64kbit/s). However, at very low bitrates, such as those required for videophone applications over PSTN networks, visible distortions, known as blocking and mosquito effects, become accentuated. The motivation for many second generation video coding techniques is to improve image quality at very low bitrates and possibly allow higher compression.

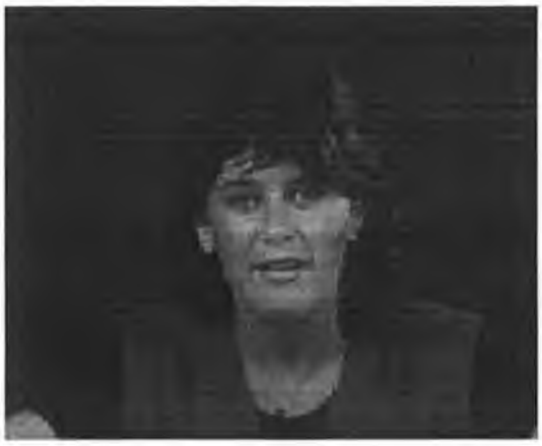


Figure 2.1 A typical videophone scene.

Model-based coding schemes have been developed to exploit certain characteristics of common scenes in videophone applications. A typical videophone scene (Figure 2.1) consists of a person's head and shoulders against a static background. As this information is known *a priori*, e.g., the 3-D shape of the face, it can be incorporated into the coding scheme [6]. Intra-frame motion is limited and mainly due to the global movement of the person's head and shoulders, and the local motion produced by changes in facial expressions. The camera is generally fixed, but even for the cases where this is not valid, e.g., pan, zoom and vibration, motion descriptions can be produced. Videophone communication does not require the same degree of resolution that is provided by, for example, MPEG-1 video. It is sufficient to use the QCIF[1] format with a reduced frame rate of 8 to 10 Hz. The combination of knowledge of the scene, the limited motion and the lower resolution makes it possible to extract more redundancy from a sequence of images, and therefore achieve a higher compression ratio, than traditional block-based schemes.

In model-based coding an input image is considered to be a 2-D projection of a 3-D scene. Coding is performed by modelling the 3-D scene, based on advance

[1]QCIF:	Luminance (Y)	176x144 pels
	Chrominance (U & V)	88x72 pels

knowledge of the scene content, and producing model parameters that are transmitted to the decoder where synthesis is used to reconstruct the image. Model-based coding schemes can be divided into two categories [7] - those schemes that use an explicit model and those that do not. In the former category, common schemes use a 3-D wireframe model that is adapted to match the dimensions of a person's head and shoulders. At the initation of communication the adaptation parameters and a number of images of the person are transmitted to the decoder where a model of the person is constructed by texture mapping the images onto the adjusted wireframe. For subsequent images the encoder only transmits motion parameters produced by a recognition algorithm that detects and tracks the movement of facial features and global head and shoulder motion. The decoder displays a sequence of synthetic images produced by using the motion parameters to animate the model.

Musmann et al. [8] introduced an object-based analysis-synthesis coding scheme in which no explicit model is used. This scheme is not restricted to any one special object and can be applied to a more general class of scene. In this scheme, each image in a sequence is segmented into moving objects and each object is defined in terms of motion, shape and colour (colour denotes luminance and chrominance). These parameters depend on the source model being used. At an abstract level, a source model is a means of describing the type of objects (e.g. rigid, flexible) and their motion (e.g. static, moving in 2-D or 3-D). Therefore, an assumption is made about the contents of the scene to be coded. The parameters are encoded by predictive or transform coding techniques. As in block-based schemes, motion compensated prediction is applied to exploit any temporal redundancy. The receiver uses image synthesis to reconstruct the image from decoded parameters. The same source model used in image analysis applies to image synthesis. Object-based analysis-synthesis coding introduces geometrical distortions in the synthesised image that are less annoying to the human eye than the quantisation error distortions associated with block-based coding schemes.

The following section explains the concept of object-based analysis-synthesis coding. Section 2.3 describes the DCU object-based analysis-synthesis codec.

2.2 Object-Based Analysis-Synthesis Coding

A block diagram of an object-based analysis-synthesis coder is shown in Figure 2.2. Image analysis segments each image of the input sequence into arbitrarily shaped moving objects and describes each object in terms of motion, shape and colour. These three parameter sets are encoded and transmitted. The parameter memory in both the coder and decoder stores the decoded parameter sets. The transmitted image is reconstructed by image synthesis using the decoded parameters. The reconstructed image is displayed at the decoder and is used by the coder for image analysis of the next image in the sequence. Parameter decoding and image synthesis are identical for the coder and decoder. The stored parameters can be used in the predictive coding of the parameters of the next image in the input sequence.

Image analysis consists of three levels - the estimation of object parameters, internal image synthesis and a verification test. The estimation of object parameters depends on the source model being used. Image analysis is based on the assumption that any temporal changes in the input sequence of images can be described by the source model. This assumption is tested using the verification algorithm which compares the original image and the image reconstructed by synthesis from the estimated object parameters. Areas where the assumption holds are known as model compliance (MC) objects. Image analysis fails in areas of the image that cannot be described by the source model.

These model failure (MF) objects typically require more bits to encode. The size of the model failure area depends on the source model. Additionally, different source models will produce parameter sets with different information content and hence different bitrates. Therefore, it is important to use source models and corresponding image analysis algorithms that minimise model failure areas and produce parameter sets that can be efficiently encoded.

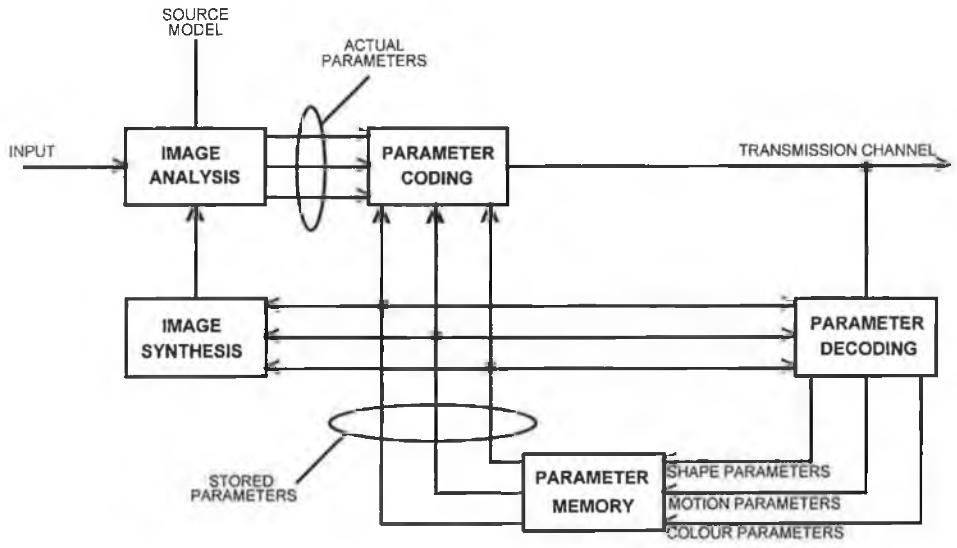


Figure 2.2 Block diagram of an object-based analysis-synthesis coder [8].

The three parameter sets are available to parameter coding in uncompressed form. To further improve the compression ratio, parameter coding techniques are applied individually to the parameter sets. In block-based coding, only two parameters (motion and colour) are transmitted for each block. For object-based coding to be equally or more efficient than block-based coding, there has to be a reduction in the bitrate required to transmit the motion and colour parameters to compensate for the additional bits needed to transmit the shape information. Several properties of object-based coding ensure that this is achieved.

If the shape coding algorithm is efficient, the bitrate required for shape information will be low. For example, a source model that produces smoothly varying shape parameters can be efficiently coded using predictive coding. Also, the transmission of shape information of an object is suppressed if no significant change in the object shape has occurred. For model compliance objects, where the assumption holds that any temporal change of an object can be sufficiently described by the source model, there is no need to transmit the colour information. Similarly, for model failure objects, since the assumption fails, the motion information is not transmitted as it cannot be used to describe the object. Only one motion parameter set is transmitted per model compliance object that can typically cover the area of several blocks which in block-based coding require a motion parameter per block. Image analysis identifies model failure areas that are caused by small position and shape errors. As these geometrical distortions are not annoying to the human eye the update information can be suppressed. The distortions are not annoying because a human observer pays more attention to how natural the image looks rather than its exact position.

Since the transmission of colour information can be suppressed for model compliance objects a good source model is one that maximises the area covered by MC objects (and hence minimise model failure area). The use of a good source model typically results in a decrease over block-based coding in the image area to be updated by colour coding. Through the use of shape information the prediction of colour can be significantly improved at object boundaries. In block-based coding, blocks that contain an object boundary require a higher bitrate to code as motion compensation prediction often results in a large prediction error.

Parameter coding can be controlled from the object information produced by image analysis. Image analysis labels an object according to whether it is model compliance or model failure. For objects marked as model compliance, only motion parameter and shape parameter coding needs to be performed. In the case of model failure objects the motion parameter is omitted and shape and colour parameter coding is performed. Objects can be assigned a priority depending on factors such as size and the amount of update information contained in the parameter sets. Transmission commences with the object of highest priority and continues in decreasing order. The transmission process is halted when the bitrate becomes exhausted (or all objects have been transmitted). Priority control ensures that the most important objects are always transmitted.

To achieve the high compression ratio needed for low bitrate transmission irreversible coding techniques have to be used. Both object-based and block-based coding are irreversible. The irreversible process in block-based coding is the quantisation of the DCT-transformed motion compensation prediction error. Any transformed prediction errors below a threshold are set to zero and those above are set to the nearest multiple of the threshold. At lower bitrates the quantiser threshold is very coarse leading to an overall degradation in the subjective quality of the image. Annoying coding errors known as mosquito and blocking artefacts become accentuated. Mosquito artefacts occur at object boundaries and are due to the fact that in block-based coding only one motion vector is calculated per block. This produces large motion compensation prediction errors if a block contains several objects moving in different directions. These errors require a high bit count to code, which at low bitrates is not available.

Object-based analysis-synthesis coding is able to produce images of higher subjective quality, i.e. more natural looking, than block-based coding at the same bitrate. It also allows higher compression ratios with acceptable image quality to be achieved. Mosquito effects at object boundaries are eliminated by the use of shape information. Small position and shape errors caused by the irreversible process result in geometrical distortions that are less annoying to the human eye than the quantisation errors introduced in block-based coding. Using the object information produced by image analysis to prioritise objects allows the coding of important objects to be improved. For example, it is possible to code small objects, such as the eyes (normally model failure areas) which are important to the subjective quality of the image, with greater accuracy than objects which contribute little to image quality such as the background. Through efficient coding of the motion and shape parameters a higher proportion of the bitrate is available for the colour parameters.

2.3 The DCU Object-Based Codec

The DCU codec was developed in conjunction with work carried out in SIMOC (Simulation model for object based coding), a subgroup of COST 211ter - the video coding section of the pan-European COST (Scientific and Technical Co-operation) organisation. The aim of the SIMOC group is to provide a framework within which a reference model for an object-based analysis-synthesis codec can be developed [9]. The object-based algorithm developed by this group is applicable to videophone sequences containing a static background. The source model used is that of "planar flexible objects that move translationally in the image plane". This section gives an overview of the operation of the DCU codec. A more detailed examination of the algorithm is given in Section 2.4.

The image analysis algorithm is specific to this source model. According to the source model assumption any temporal change in luminance will be due to 2D-object deformation (flexible) and 2D-motion (translational). Image analysis requires the current image and the previous synthesised image. The first step of image analysis is to segment the current image into temporally unchanged (static) and changed areas. This is achieved by comparing the two input images for differences in luminance.

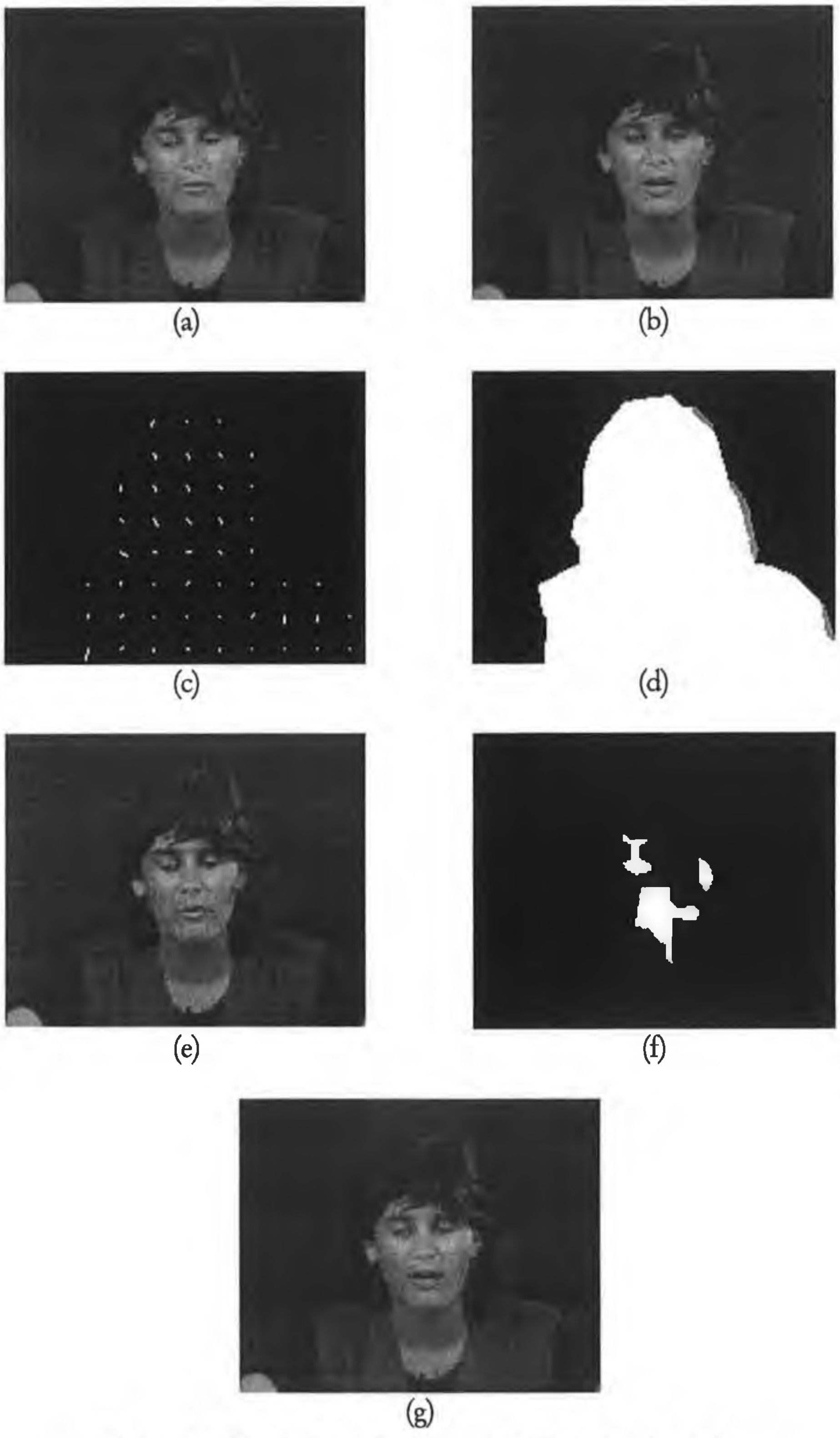


Figure 2.3 Output of various stages of SIMOC coding scheme.

Thresholding is used to distinguish between nonzero differences that are due to noise and those that are due to a scene change. Median filtering and morphological filtering are applied to the binary segmentation mask (change detection mask) in order to eliminate small regions. Each disjunct area is interpreted as a separate object and the binary mask defines the shape of each object. SIMOC refers to this first step as 'Change Detection'.

The various stages of the SIMOC coding scheme shown in Figure 2.3 are

a) previous reconstructed frame
b) current frame
c) grid position motion vectors
d) SMU mask
e) motion synthesised current frame
f) model failure mask
g) decoded frame.

In the second step of image analysis (motion analysis), a set of motion parameters are calculated for each disjunct region. As the source model assumes that any object motion will be in the 2D plane, the motion parameters consist of a displacement vector field. A three-level hierarchical block-matching algorithm based on Bierling's [10] work is used to estimate the motion vectors. Bierling's algorithm estimates true motion rather than calculating the motion vector that minimises the mean absolute displaced frame difference, as is the case in block-based algorithms. Efficient differential coding of the motion parameters can be achieved as the displacement vector field is also homogeneous. Motion vectors are estimated at predetermined grid positions within the changed areas by recursively using the result from a level in the hierarchy as an initial guess in the next lower level. In the first level of the hierarchy a large search window is applied to low-pass filtered versions of the current image and the reconstructed previous image. This provides a rough, but reliable, estimate with respect to the true motion, by reducing the number of false estimates that can be caused by high frequency image components. The second and third levels of the hierarchy locally refine the estimate by using a smaller search window and a maximum update displacement. A summary of the three levels is contained in Table 2.1. The maximum displacement is ±4.5 pels, which is sufficient for the limited motion in videophone applications [11].

Hierarchy Level	1	2	3
Max. update displacement	±3	±1	±0.5
Measurement window size	32x32	16x16	16x16

Table 2.1 Summary of three-level hierarchical block-matching algorithm.

Once the grid motion vectors have been evaluated interpolation is used to generate a full resolution motion vector field with half-pel accuracy. It may happen that one or more of the grid motion vectors required for the interpolation falls outside the changed region. In this case grid motion vectors have to be temporally extrapolated to outside the changed region. When the motion vectors have been interpolated for all pels within the changed region, all vectors in the unchanged region are set to zero.

The motion vector field is used to divide the changed region into a moving region and uncovered background. All pels in the current change detection mask are traced backwards, using their motion vectors. If the inverse motion vector points to a pel that is within the changed region of the previous change detection mask then the current pel is classified as moving, otherwise it is classified as uncovered background.

Shape analysis uses the model compliance (moving area) mask produced by the first steps in image analysis. The shape of an object is considered to be the contour of that object in the binary mask. Object shape is approximated by a polygon representation whereby the object contour is described by a number of vertices connected by straight line segments. The general principle of shape approximation is shown in Figure 2.4. The number of vertices depends on the desired quality of the shape approximation. The quality is controlled by the absolute difference between the actual object contour and the approximated polygon representation. The shape approximation becomes coarser with an increase in the maximum allowable absolute difference and fewer vertices (therefore less bitrate) are required. However, low accuracy leads to synthesis errors near object boundaries and therefore increases the bitrate for the colour parameters. In the DCU coder the maximum allowable absolute distance has been fixed at a value that achieves a balance between the number of vertices produced and the quality of the image synthesis.

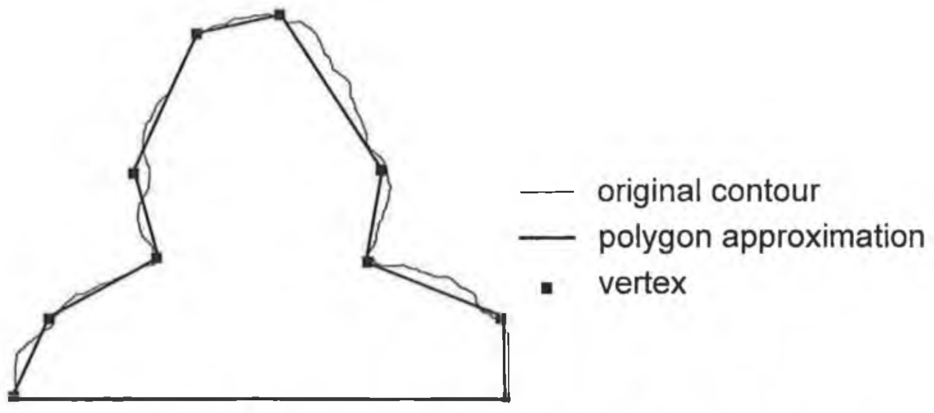


Figure 2.4 An example of coarse polygon approximation.

For shape approximation, objects are split into two categories - those objects with a reference in the previous frame and those without. To determine if an object existed in the previous frame, vertices from the previous frame are motion compensated by applying the vertices' motion vectors stored from the previous frame (the source model assumes that the motion is linear, i.e., if the current frame occurs at k+1, then the motion vectors from k-1 to k are applied to the vertices from k). This motion compensated object shape is used as an initial estimate of the object shape in the current frame. The validity of the estimate is then tested by a verification algorithm. The distance of each displaced vertex to the actual contour is calculated. If the distance is within the maximum allowable absolute distance then the vertex is marked as 'maintained', otherwise it is marked as 'rejected'. Only maintained vertices are used in the shape approximation. Additional vertices may have to be inserted to ensure that the quality of the shape approximation obeys the maximum allowable absolute distance criterion.

If the source model holds, then any changes in object shape will be due to motion alone. Where this is the case, the motion compensated vertices are identical to the actual object and no extra update information, such as inserted vertices, is needed thus reducing the shape parameter bitrate. In the case where vertices are rejected and inserted, i.e., where the source model only partly holds, the indices of the maintained, rejected and inserted vertices as well as the position of each inserted vertices needs to be transmitted.

For objects with no reference in the previous frame, the vertices must be calculated from the actual object contour. First, an initial polygon with four vertices is calculated. Then, the approximation is progressively refined by inserting vertices until the quality criterion is matched everywhere.

The success of image analysis is checked using a verification test (model failure detection). The verification test compares the current image to an image synthesised from the shape, motion and colour parameters produced by image analysis in order to determine if there are any areas where the temporal change cannot be sufficiently described by the source model. Model failure areas can be caused by motion that is more complex than translational motion, or changes that cannot be described by motion at all, e.g., by the introduction of a new object into the scene. Additionally, using the DCU source model it is not possible to detect moving objects in front of moving objects, e.g., eyes/mouth in front of a face.

The synthesised image is reconstructed using a special colour memory, the motion vector field and the SMU mask (Static, Moving, Uncovered). To ensure the encoder synthesis process matches that of the decoder the shape approximated moving area mask is used. The uncovered area and the motion vector field are updated to reflect and changes from the actual mask. The colour memory is used instead of the previous reconstructed frame as it eliminates the effects of repeated filtering due to half-pel motion compensation. The memory is maintained at twice the spatial resolution of the input image, and is initialised by bilinear interpolation of the first image in the input sequence. For image synthesis, a new colour memory is generated by mapping into it pels from the old colour memory according to their classification in the SMU mask. A priority based is used to determine the classification of half-pel positions. If a pel belongs to the static area then it retains its value from the old colour memory. Model Compliance pels (pels belonging to the moving area) are synthesised through motion compensation prediction - each pel is assigned the value of the pel in the old colour memory pointed to by its motion vector.

For each pel in the uncovered area, a prediction can be made in two ways - either by a spatial prediction from neighbouring static pels or by temporal prediction from a special background memory. If it is ascertained, through the use of a special 'already seen' mask, that a part of the background appears in a sequence for the first time, spatial prediction is performed. The background's decoded colour parameters are then added to the background memory to be used for prediction in any subsequent appearance of the same part. The background is also marked as 'already seen' in the mask. Both the coder and decoder maintain identical background memories and 'already seen' masks. The quality of the prediction is evaluated and the prediction error is coded, as per model failure objects, and transmitted.

All full-pel positions within the new colour memory are used to form the image employed in model failure detection.

The shape parameters that are coded consist of the co-ordinates of the vertices from the shape approximation. For MC-objects with a reference in the previous frame a list of maintained, rejected and inserted vertices for is run-length coded, and only the co-ordinates of the inserted vertices must be coded. The x and y components of an inserted vertex are coded relative to its nearest counter-clockwise neighbour. In the case of MC-objects with no reference in the previous frame and MF-objects all vertices are classified as inserted vertices and coded accordingly. Adaptive arithmetic coding is applied to the shape parameters before transmission. No shape or motion information is required for uncovered background as it is determined from the shape and motion parameters of MC-objects.

Motion parameters (the grid motion vectors) are coded using predictive coding in order from top left to bottom right. The left neighbour is used as the prediction vector, if available, otherwise (0,0) is taken. The prediction errors are merged into a single stream and undergo adaptive arithmetic coding before being transmitted.

The colour parameters of model failure regions are coded using spatial vector quantisation on the motion compensated prediction error. Each model failure object is coded separately using one of a set of six vector codebooks which increase in size from 32 to 1024 vector entries. The model failure object is divided into 2x2 blocks of luminance pels and their two corresponding chrominance pels, giving a total of six components per vector. The codebook vectors are used as an approximation (quantisation) of the actual prediction error. Each 2x2 block is assigned the vector that yields the lowest average square difference calculated over the six prediction errors. Vectors are chosen from the codebook that gives the best approximation over the whole object. A representative vector is allocated to each vector to be coded from the chosen codebook applicable to the object. The indices of the representative vectors are further processed using adaptive arithmetic coding.

To date, no bit-stream syntax and control has been defined and no buffer regulation is included. The coder works in open loop mode with the generated bitrate controlled by a quality criterion for model failure regions (31 dB PSNR). Table 2.1 and Table 2.3 give some performance statistics for the DCU coder from simulations on the standard test sequences 'Miss America' and 'Claire' [12]. For QCIF resolution at 8 frames per second, the bitrates achieved represent a compression ratio of 116:1 for the 'Miss America' sequence and 82:1 for the 'Claire' sequence. It is generally accepted that this object-based analysis-synthesis coding algorithm has many limitations, mainly due to the underlying source model. The coder performance is dependent on the source scene content, hence the variation in the compression ratios for the two sequences. However, the DCU algorithm is fairly indicative of many second generation video coding techniques in terms of computational complexity and the basic tools that comprise the algorithm.

SEQUENCE	SHAPE kbit/s	MOTION kbit/s	COLOUR kbit/s
Miss America	7.5	2.4	12.2
Claire	5.6	4.3	19.1

Table 2.2 Simulation results for DCU coder - average bitrates.
Simulation: QCIF 8 fps - Miss America 50 frames, Claire 50 frames.

SEQUENCE	Y - PSNR dB	U - PSNR dB	V - PSNR dB
Miss America	36.39	38.74	36.97
Claire	33.84	29.89	32.23

Table 2.3 Simulation results for DCU coder - average PSNR (Peak SNR).
Simulation: QCIF 8 fps - Miss America 50 frames, Claire 50 frames.

2.4 Profiling of the DCU Object-Based Analysis-Synthesis Coder[2]

2.4.1 Execution Time Estimation.

Given the complete description of the DCU algorithm in [9], it is possible to derive some estimate of the processing power required for its implementation, at least for the major functions. Although the estimates are based on a number of reasonable assumptions, at this stage it is not possible to derive any meaningful estimate for the implementation complexity. Such an estimate only has meaning when a target processor has been selected and the algorithm has been optimised in that processor's native language. The estimates assume that an operation (op.) equates to a multiply, arithmetic/logic, or data move that executes in a single clock cycle, and that there exists sufficient input and output facilities to keep the CPU loaded with data. Also, no account is made for control overheads that would invariably be required. Even given these assumptions, it is difficult to arrive at an exact figure since many of the component parts of the algorithm only operate on the detected moving area, and therefore execution time will vary with the amount of motion in the image. The estimate derived in this section is based on the worst case condition where

[2] In the rest of this chapter only the encoder is considered. It is assumed that, as the encoder contains a decoding loop, any realtime encoder solution will automatically result in a realtime decoder.

the entire image is classified as moving. The figure is only intended to give the order of magnitude of the complexity that could be expected for the sequential execution of the algorithm.

The input image is assumed to be available in QCIF format, i.e., no format conversion is necessary. The dimensions of QCIF are:

Luminance (Y):	176x144 pels
Chrominance (Cr & Cb):	88x72 pels.

2.4.1.1 Change Detection

Only luminance data is used to calculate the change detection mask. There are seven steps involved in the calculation.

a) Compute the absolute difference between the current frame and the previous reconstructed frame.
   This involves 3 operations per pel - subtract, absolute and store, giving a total of 76,032 ops. per frame.
b) For each pel, sum its value and that of its neighbours in a 3x3 mask. Any pel whose neighbourhood sum is greater that 18 should be mapped to a single pel value representing CHANGED regions; all other pels should be set to a pel value representing UNCHANGED regions.
   This requires 10 ops. per pel - 8 additions, 1 compare, 1 store giving 253,440 ops. per frame.
c) Apply a 5x5 binary median filter to the resulting CHANGED and UNCHANGED regions.
   In the binary median filter a pel takes on the binary level that is dominant in a 5x5 mask centred on that pel. This can be done by summing all the pels and if the result is > 12 then the CHANGED

level is dominant (assuming CHANGED=1, and UNCHANGED=0). This requires 26 ops. per pel - 24 additions, 1 compare, 1 store giving a total of 658,944 ops. per frame.

d) Add all MOVING areas from the previous SMU mask to the mask resulting from step c).
An addition and store will be required for this step, totalling 50,688 ops. per frame.

e) Blow (dilate) the CHANGED region three times, using a 3x3 structuring element.
Dilation is a morphological operation that involves the translation of a structuring element throughout an image, setting a pel to '1', in this case CHANGED, where the structuring element has a non-empty intersection with the pels in the neighbourhood. In this case, a pel is set to CHANGED if it or any of its neighbouring pels in a 3x3 mask are CHANGED. This has the effect of expanding (blowing) objects. Dilation can be achieved by ORing a pel with its 3x3 neighbours, requiring 8 ORs and 1 store giving a total of 228,096 ops. per dilation per frame.

f) Shrink (erode) the CHANGED area three times, using a 3x3 structuring element.
Erosion is another morphological operation and is the opposite to dilation. A pel is set to '0', in this case UNCHANGED, if the structuring element does not match all the data surrounding the pel, i.e., in this case a pel is set to UNCHANGED if it or any of its neighbouring pels in a 3x3 mask are UNCHANGED. This has the effect of expanding shrinking objects. Erosion can be achieved by ANDing a pel with its 3x3 neighbours, requiring 8 ANDs and 1 store giving a total of 228,096 ops. per erosion per frame.

g) Eliminate all CHANGED regions whose size is less than the average size of all CHANGED regions.

h) Eliminate all UNCHANGED regions whose size is less than the average size of all UNCHANGED regions.

   The processes involved in these two steps are complex and are not readily translatable into simple operations such as add, subtract, etc. Therefore, a figure cannot be calculated for these steps.

The estimates for change detection are summarised in Table 2.4. The total of approximately 2.5MOPs is independent of the size of the CHANGED area, as it is these steps that actually produce the regions.

STEP	**# of Operations**
a)	76,032
b)	253,440
c)	658,944
d)	50,688
e) (3x228,096)	684,288
f) (3x228,096)	684,288
g)	N/A
h)	N/A
Total	2,407,688

Table 2.4 Summary of processing requirements for change detection.

2.4.1.2 Motion Analysis

A 3-level hierarchical block-matching algorithm is used for motion estimation.

Level 1

A two-step search is performed on 3x3 mean-value filtered versions of the current frame and the previous reconstructed frame. The measurement window is 32x32 pels. The search is performed at every 16th pel in every 16th line in all directions, with a step-size of ±2 pels in the first step and ±1 pel in the second step. The first grid position is positioned at (8,8). The displacement that leads to the lowest mean absolute displaced frame difference (MAD) is taken. Only pels within the CHANGED region are used for MAD calculation.

Firstly, the mean-value filtering must be performed. This requires 10 ops. per pel - 8 additions, 1 divide, 1 store - giving a total of 506,880 ops. for both frames.

Step 1:
# of searches:99
# of search positions:9
window size:32x32
# of ops. per pel: 4 (1 subtract, 1 absolute, 1 multiply[3], 1 add to MAD total)
= 3,649,536 ops. per frame.

Step 2:
same as step 1.

Level 2

A one-step search is performed on non-filtered versions of the current frame and the previous reconstructed frame. The measurement window is 16x16 pels. The

[3]Assumes CHANGED = 1 and UNCHANGED = 0 - by multiplying the absolute difference by a pel's mask value, it will either remain the same or be set to 0 and thus only CHANGED pels contribute to the SAD.

search is performed at every 16th pel in every 16th line in all directions with a step size of ±1 pel.

# of searches: 99

# of search positions: 9

window size: 16x16

# of ops. per pel: 4

= 912,384 ops. per frame.

Level 3

For this level the current frame and the previous reconstructed frame are bilinearly interpolated so that images are obtained with twice the number of pels and lines. A one step search is performed on these images at every 32nd pel in every 32nd line in all directions with a step size of ±1 pel (i.e., ±0.5 pel in the original resolution). The measurement window is 32x32 pels.

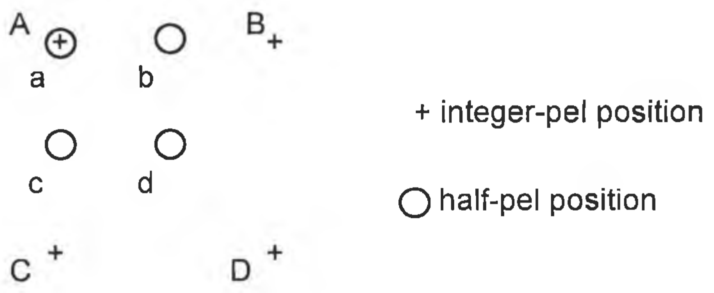


Figure 2.5 Bilinear interpolation scheme

a = A	**Equation 2.1**
b = (A+B)//2	**Equation 2.2**
c = (A+C)//2	**Equation 2.3**
d = (A+B+C+D)//4	**Equation 2.4**

Each pel in the frame to be interpolated has four output pels associated with it (see Figure 2.5). Interpolation requires 12 ops. per pel - 5 adds, 3 divisions and 4 stores. This gives a total of 608,256 ops. for the two frames. Additionally, the change detection mask must be upsampled to the same dimensions as the bilinearly interpolated images. This is achieved by using a biased median value filter that has the following rules (positions are the same as per Figure 2.5):

a is set if A is set (requires 1 store);
b is set if both A and B are set, i.e., b = A.B (requires 1 AND, 1 store);
c is set if both A and C are set, i.e., c = A.C (requires 1 AND, 1 store);
d is set if any three of A,B,C,D are set (requires 3 adds, 1 compare, 1 store).

The upsampling requires 10 ops. per pel, i.e., 253,440 ops. per frame.

The search requires the same number of operations as step 1 of level 1.

LEVEL	# of Operations
Level 1: Filtering	506,880
Step 1	3,649,536
Step 2	3,649,536
Level 2	912,384
Level 3: Search	3,649,536
Interpolation	608,256
Upsampling	253,440
Total	13,229,568

Table 2.5 Summary of processing requirements for motion analysis.

2.4.1.3 Model Failure Detection

Determining the model failure regions requires, firstly, the calculation of the MF threshold, TMF, as follows:

a) Set TMF = 1.

b) Calculate the synthesis error variance, MSEsyn, for all CHANGED pels whose synthesis error is less than TMF.

c) If MSEsyn < 6 set TMF = TMF + 1 and goto step b), otherwise set TMF = TMF -1 and finish.

The conditional statement in c) means that the processing power required to calculate the TMF will vary from image to image. Once the TMF has been calculated, a binary mask can be generated that indicates those pels where the synthesis is greater than or equal to the TMF, i.e., model failure pels. This would require 3 ops. per pel - 1 compare to TMF, 1 loading of appropriate binary value and 1 store of value in mask. The remaining five steps of model failure detection are identical to steps c), e), f), g) and h) of change detection. The estimates for model failure detection are summarised in Table 2.6.

STEP	# of Operations
Calculation of TMF	N/A
Generation of Binary Mask	76,032
5x5 median filter	658,944
3 x dilation	684,288
3 x erosion	684,288
Elimination of regions	N/A
Total	2,103,552

Table 2.6 Summary of processing requirements for model failure detection

2.4.1.4 Summary

The estimates derived cover the three encoder functions that require the greatest processing power [13]. The remaining encoder functions are algorithmically complex and are not readily translated into simple operations. Therefore it is difficult to produce estimates for their processing power requirements. The figures derived above indicate that the DCU encoder has a processing power requirement in the order of 140 MOPs (million of operations per second) even for a modest frame rate of 8Hz. Benchmarks for the processing power of uniprocessors vary by manufacturer. Some uniprocessors are capable, under certain conditions, of completing multiple operations in a single clock cycle, e.g., a multiply, an add and a data move. Thus a device with an instruction cycle of 100ns is termed a 30MOPS device. Even taking this into account, the processing power required for the encoder is above that which is achievable by uniprocessors, typically a few tens of MOPs. Therefore it is necessary to seek a parallel implementation if real-time operation of the DCU object-based algorithm is to be achieved.

2.4.2 Other Properties

In many block-based codec implementations real-time operation is achieved through geometrical image subdivision and distribution of the resulting image segments between parallel processors. This method relies upon each image segment having fixed and (nearly) identical execution times. This condition exists as most block-based coding algorithms are image content independent, i.e., they have a fixed, and predictable execution time regardless of the amount of motion in an image. In contrast, it is a property of the DCU algorithm that execution time depends significantly upon image content. This is due to the fact that much of the analysis and synthesis is concentrated upon areas of the image where motion is detected, therefore the more motion that occurs between frames the greater the execution time required to process the frames. This variation in execution time occurs not only between different image sequences, but also between different frames of the same sequence.

As is evident from the estimates derived in Section 2.4.1, the component of the DCU algorithm that dominates the execution time is motion estimation and its associated functions. This is also the case in many block-based schemes where the full search block-matching algorithm is used for motion estimation. The most common technique used to save processing power in block-based implementations is to reduce the amount of search positions evaluated in the search window, thus decreasing the amount of computationally expensive matching operations. While this method works well for block-based schemes, resulting in only a small degradation in picture quality, it is unsuitable for the motion estimation used in the DCU algorithm. The hierarchical search already has a reduced number of search positions (36 per grid position, compared to 256 per macroblock in H.261). The motion parameters produced by motion estimation are intrinsic to the operation of the DCU algorithm and, as such, the motion estimation scheme outlined in Section 2.4.1 has been optimised with respect to coding efficiency and image quality. Therefore, the use of any other scheme may have adverse effects on the operation of the DCU algorithm. Nevertheless, a more efficient implementation of the motion algorithm may have to be sought if real-time implementation is to be achieved, even if this means sacrificing image quality and/or larger bitrates.

The data memory required by the DCU algorithm can separated into two categories - memory size requirements that are fixed from frame to frame, and memory that is allocated and subsequently de-allocated during the processing of a frame. The first type of data must be available throughout all the stages in the processing of a frame, and are updated during the process for use in the analysis and synthesis of subsequent frames. Examples of this type of data are the current frame, the previous reconstructed frame and the colour memory. This type of data requires over 250kB of memory. The second data group consists of temporary results generated by a component part of the DCU algorithm that is required in subsequent processing of the same frame, e.g., motion estimation requires bilinearly interpolated versions of the current frame and previous reconstructed frame. These temporary results can be discarded when they are no longer required. The memory space required for temporary storage varies during the processing of a frame, but reaches a peak for motion estimation, where over 300kB is required.

Chapter 3

REAL TIME IMPLEMENTATION

3.1 Introduction

In the development of application-specific systems, such as for second generation video coding, there are two main factors that must be taken into account when choosing an architecture - the algorithms that must be supported and the hardware that will be used to construct the system. Algorithms are a decisive factor as they determine the minimum processing power requirements and the cost of software development, while the hardware not only imposes constraints on the cost, size and complexity of the technology, it also limits it in terms of maximum clock rates, memory access times, etc.

In order to define an algorithm fully, a knowledge of the target system is required, e.g., its processor type, number of processors and input/output capabilities. An algorithm that performs a task on an uniprocessor system will be somewhat different to an algorithm that performs the same task on a massively parallel system. However, for application-specific systems, the algorithms must be defined first in order to develop a specialised architecture. This creates a "vicious circle" situation - a knowledge of the target system is required to define the algorithm, but the algorithm must first be defined to allow the architecture to be developed. One way to deal with this problem is to decide on a hardware architecture first and then adapt the algorithm to suit. This approach has the disadvantage of creating either unoptimised systems that are wasteful of resources or systems that are incapable of supporting the algorithm.

The solution suggested here to the vicious circle problem is an iterative approach that begins with an implementation-independent statement of the algorithms, from which the basic hardware requirements are identified and a primary model is defined. Any general hardware constraints are considered in this primary model. The next step involves translating the algorithms for implementation on the basic hardware. In the final step, the model is refined based on any algorithmic needs identified during the translation process. These last two steps can be repeated a number of times to progressively refine both the algorithm and the hardware and produce a final design that is optimised for both the software requirements and hardware constraints.

The implementation-independent profiling of the DCU algorithm in Section 2.4.1 suggests that the computational requirements of the algorithm are greater than that which is achievable by a uniprocessor, and so a parallel system is required. This is the first architectural decision and is the foundation for the basic hardware model. Another requirement of the system is the ability to adapt to any changes that may be necessary as the DCU algorithm evolves, i.e., the system must be programmable. Using programmable elements in the system has other important advantages. Firstly, the complexity of the system lies in the software rather than the hardware and, if the cost of software development is low, then the cost of modifying the algorithm will also be low. Secondly, if a sufficient degree of programmability exists, then not only can the algorithm be modified but the system could also be used to implement other algorithms and finally, the use of standard programmable devices rather than application-specific devices can reduce system cost.

The basic hardware model can be simply defined as a system that is parallel and programmable. The next step in the iterative design process is to adapt the sequentially organised DCU algorithm for implementation on a parallel system. Parallel systems achieve greater processing power over uniprocessor systems by executing concurrent (parallel) tasks simultaneously. A prerequisite of an algorithm for parallel implementation is that the algorithm contains several concurrent elements. Therefore, any concurrent elements of the DCU algorithm must be identified in order to assess its suitability for parallel implementation.

3.2 Detection of Parallelism

Parallelism can exist at various hierarchical levels within an application. At the highest level in the hierarchy, the application can be separated into concurrent jobs or programs. Any functions called by these programs form the second level. If these functions then call further functions, then the latter functions would form the third level, etc. The lower levels of the hierarchy occur after the fundamental functions (functions that do not call other functions). Concurrency can exist between individual instructions of a function, and at the most basic level, it may be possible to have concurrent operations within an instruction. Generally, the higher levels of parallelism are performed algorithmically, while the lower levels are achieved in hardware. However, there is no distinct boundary between the two.

A typical videophone application would consist of video, audio, control and multiplex components. The first level of parallelism would be to separate the application into these parts. In the DCU algorithm, there exists only one program at the highest level and, therefore, detection of parallelism must begin at the second level.

3.2.1 Inter-Function Parallelism

In order for sequentially organised processes in an algorithm to be executed in parallel, the processes must satisfy Bernstein's condition. This basically states that two processes can be executed as two independent and concurrent processes if the input to one process is not dependent on the output of the other, and vice versa, i.e., if I_i represents the input to a process, and O_i represents the output, then Bernstein's condition can be expressed as:

$$I_1 \cap O_2 = \phi \text{ and } I_2 \cap O_1 = \phi$$

This data dependency is the key to the detection of concurrency within an algorithm. To determine the data dependency within the DCU algorithm, it is useful to list all the functions involved together with their required inputs and outputs. This is done in Table 3.1.

The dependency of functions on the output of others, as can be seen in Table 3.1, illustrates the highly sequential in nature of the DCU algorithm. This sequential nature is inherent in analysis-synthesis coding, i.e., analysis must be completed before synthesis can be performed. Within analysis, the motion vector field is integral to the production of the parameter sets, with the result that most analysis functions require that motion estimation be performed first. Motion estimation in turn requires that the change detection mask has been calculated. Only when analysis and, subsequently, synthesis are completed can the verification test and, finally, parameter coding be performed. The order in which the functions must be executed has been reflected in the construction of Table 3.1.

Function	Input	Output
Change Detection	current frame, previous reconstructed frame	change detection mask
Motion Analysis	current frame, previous reconstructed frame, interpolated and filtered versions of same, change detection mask, upsampled change detection mask	grid vectors
Moving area detection	change detection mask, grid vectors	SMU mask, interpolated motion vector field
Shape Approximation	SMU mask, interpolated motion vector field	shape approximated mask, shape parameters
Uncovered background prediction	shape approximated mask, background memory and mask, current frame	updated background memory and mask, colour parameters
Motion Synthesis	current frame, colour memory, interpolated motion vector field, background memory	motion synthesised frame, updated colour memory
Model Failure Detection	current frame, motion synthesised frame	model failure mask
MF Shape Approx.	model failure mask	approx. mask, shape parameters
MF Colour Synthesis	shape approximated model failure mask, colour memory	updated colour memory, colour parameters
Previous Reconstructed frame generation	colour memory	previous reconstructed frame (for use in processing of next frame)
Parameter Coding	shape parameters, grid vectors, colour parameters	output bit stream

Table 3.1 List of functions with input and output requirements.

The first major constraint on potential parallelism is the required input of the previous reconstructed frame into some functions, starting with change detection, during the processing of the current frame. The generation of the previous reconstructed frame is the second last step in the processing of a frame. Therefore, full temporal parallelism (the concurrent processing of several frames independently) is not possible with the DCU algorithm. The generation of the previous reconstructed frame at a late stage in the processing of a frame also precludes any practical partial temporal (pipeline) parallelism. In a pipeline structure, the output of a stage (function) is the input to the next stage of the pipeline (the input to the first stage and the output of the last stage are the input and output of the algorithm). Once a stage has passed its results onto the next stage, it can begin processing the next set of data at its input. Speedup is achieved when the pipeline is full, all the stages are operating concurrently on different parts of the algorithm. The successful operation of the pipeline is dependent upon no stage in the pipeline requiring input from a stage further down the pipeline. Clearly, the feedback of the previous reconstructed frame from the second last stage to the first (change detection) violates this condition and therefore, at this level, the DCU algorithm is not suited to pipeline parallelism.

Another possible parallel method is to have several processors concurrently executing the same algorithm on a subset of the total data. The sub-results are then combined to form the complete solution. This is known as geometric parallelism. Whereas pipeline parallelism is a form of temporal parallelism, geometric parallelism is a form of spatial parallelism. Bernstein's condition is satisfied when a function can calculate results for the subdivided image area based on data contained in that image area and is independent of any results that are generated by another function that is executed on a separate processor. Block-based coding schemes are well suited to geometric parallelism as an image is described by a number of independently moving blocks. In the DCU algorithm, an image is described by independently moving objects that may cover several geometrically divided areas. There are several instances where the complete set of object data is required as input to a function. In shape approximation, for example, an entire object, is needed to produce its contour description. There are other problems with using geometric parallelism at this level. One that cannot be avoided is the image content dependent execution time outlined in Section 2.4.2. Any geometric subdivision of the image will inevitably produce areas that contain different amounts of motion, resulting in varying execution times between areas. This would leave some processors idle while others are still processing their associated areas, which is an inefficient use of system resources. A situation could conceivably occur where all the motion within an image is contained within one geometrically subdivided area, with the result that once the changed area has been detected, only one processor is operating and the others are idle - in effect a uniprocessor system.

There are some functions in the DCU algorithm that are truly independent. For example, the mean-value filtered and interpolated versions of the current frame and the previous reconstructed frame required for motion analysis can be calculated in parallel with change detection. Also, once the moving objects have been identified, each object can be processed independently, and hence concurrently. The degree of speedup that is achievable is dependent on the number of detected objects, but as this is a dynamically varying parameter that is impossible to pre-determine, no theoretical estimate of the speedup is obtainable. In any case, the bulk of the processing has been used to identify the objects and therefore any gain in overall performance would be minimal.

The conclusion that can be drawn from the detection of parallelism at this level is that pipeline parallelism is not an option, and that there is no practical benefit to be gained from geometric parallelism. Also, the true parallelism that does exist at this level forms such a small fraction (relative to the required speedup) of the total execution time, that any exploitation would lead to a minimal speedup of the algorithm. Therefore, it is necessary to examine the next level of parallelism, i.e., within individual functions, and in particular the three functions - change detection, motion analysis and model failure detection - that comprise the bulk of the processing power requirements.

3.2.2 Function-Level Parallelism

The component functions of motion analysis, change detection and model failure detection are listed in Section 2.4.1. Motion estimation, the largest component of motion analysis, does not alter any of the input data - the data is only used to produce the motion vectors for the 99 possible grid positions. These motion vectors are not inter-related in any computational manner, and are calculated from data in a restricted search area surrounding their respective grid positions. This computational independence and limited input data lends itself well to geometric parallelism. The remaining support functions of motion analysis - mean-value filtering, interpolation and upsampling - are all neighbourhood operations. In this class of image processing operations, the output for a given pel is a function of itself and a limited number of pels in its immediate neighbourhood. Neighbourhood operations are well suited to geometric parallelism. Motion estimation execution time is image content dependent, so the same problem of idle processors arises as in the examination of previous level of parallelism. Alternatively, the hierarchical structure of the block-matching algorithm used in motion estimation is suited to pipeline parallelism. Each stage in the pipeline could execute a separate hierarchical step, with each stage in the pipeline calculating an intermediate motion vector that is then passed to the next stage in the pipeline, where it is used as the starting point for the next search level. When the input data from a grid position's search area is available, the pipeline can start calculating the motion vector. Assuming that the support functions can produce this data fast enough, parallel execution with the pipeline should be possible.

The first six steps of change detection are all neighbourhood operations and therefore can be adapted for geometric parallelism. Unlike motion estimation, however, execution for these six steps is fixed regardless of image content. The last two steps, elimination of regions, require the whole image in order to determine the average size of all the regions. Therefore, geometric parallelism cannot be applied. When several neighbourhood operations are cascaded, as is the case here, pipeline parallelism can be applied. When a pipeline stage receives enough input data to perform a neighbourhood operation on a pel, an output is produced that is passed to the next stage in the pipeline. The same sequential order of neighbourhood operations can be maintained, but by reducing the delay of a pel through a pipeline stage, speedup can be achieved through partial temporal parallelism - a step no longer has to wait for the previous step to process the complete frame before it can start. As in the geometric case, the last two steps cause a problem as they do require the previous step to be fully completed before they can be started.

For the large part, model failure detection is the same as change detection, and the same analysis applies. The calculation of the model failure threshold, which is the first step, remains a sequential process due to the conditional loop that is based on the synthesis error variance for the entire frame. It may be possible to exploit some parallelism in the calculation of the synthesis error, but this requires going to the next level of parallelism.

The remaining functions of the DCU algorithm comprise a lesser bulk of the processing power requirements. Any parallelism that may be exploitable would result in a minimal gain in comparison to the three functions analysed above. Although it is possible to examine the algorithm for lower levels of parallelism, it would lead to diminishing gains, since the bulk of the speedup will already have been achieved.

3.3 Proposed Parallel Algorithm

In assessing the quality of a parallel algorithm there are two important measures - its speedup and efficiency. If T_s is the execution time for the best serial algorithm on a single processor, and T_p is the execution time for the parallel algorithm on n processors then speedup can be defined as:

$$S_n = \frac{T_s}{T_p}$$

and efficiency is given by:

$$E = \frac{S_n}{n}$$

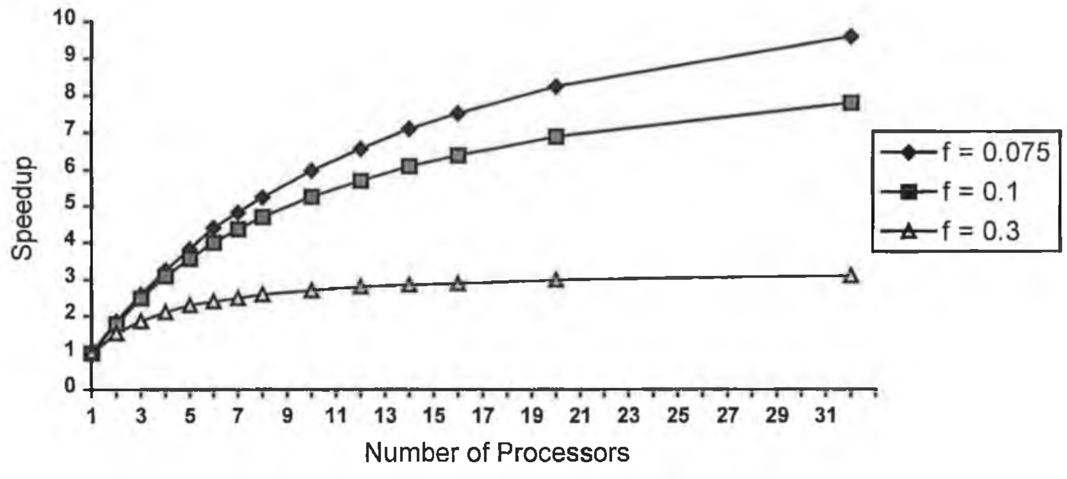
The maximum speedup that can be achieved by a parallel system with n processors is at most n times faster than a single processor (i.e., $T_p = T_s/n$). In practice, the figure for speedup can be much lower due to, for example, communication overheads, inefficient algorithms or idle processors. A much more realistic measure of speedup can be determined from Amdahl's law, which states that the speedup of a parallel algorithm over a corresponding sequential algorithm is limited by the number of operations that cannot be executed concurrently, i.e., its serial (Amdahl) fraction. The parallel processing time can be written as:

$$T_p = fT_s + \frac{(1 - f)T_s}{n}$$

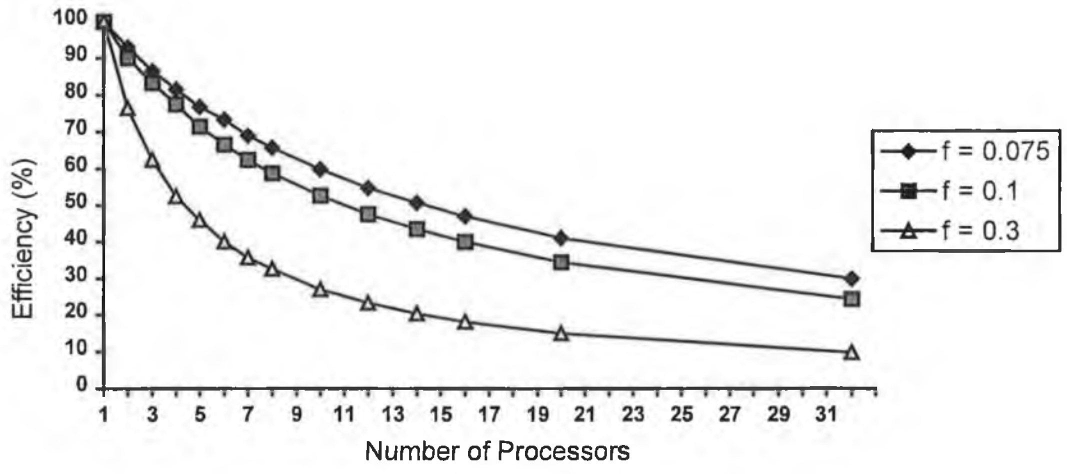
where f is the Amdahl fraction. Therefore speedup becomes:

$$S_n = \frac{1}{f + \frac{(1-f)}{n}} \qquad \text{Amdhal's Law}$$

Thus, for example, if 10% (f=0.1) of an algorithm has to be executed sequentially, then the maximum speedup is 10 regardless of the number of processors. Figure 3.1 (a) and (b) show the predicted speedup and efficiency for increasing numbers of processors and different Amdahl fractions.



(a)



(b)

Figure 3.1(a) speedup vs. number of processors for various values of Amdahl fraction.
(b) efficiency vs. number of processors for various values of Amdahl fraction

From the analysis on the level of parallelism of the DCU algorithm in the previous section, it appears that sufficient parallelism exists to be exploited, especially in the tasks that make up the bulk of the processing, and real-time performance can be achieved. Parts of the algorithm still require sequential execution, and therefore the proposed adaptation of the algorithm will be a combination of sequential and parallel elements. For the parallel tasks, it has to be decided which of the two possible methods - pipeline and geometric - offers the best speedup and efficiency.

For a task such as motion estimation, it is straightforward to make a comparison between the two parallel methods in terms of the speedup in execution times. In Section 2.4.1, the following figures were estimated for the calculation of a motion vector for one grid position at each level in the block-matching hierarchy:

Level 1, Step 1: 36,864 ops.
Level 1, Step 2: 36,864 ops.
Level 2: 9,216 ops.
Level 3: 36,864 ops.

This gives a total of 119,808 ops. per motion vector (11,860,992 per frame) for sequential calculation. For a pipeline implementation, the natural decomposition of motion estimation is into four stages, each performing a separate level or step (see Figure 3.2). The grid positions flow through the pipeline and are operated on by each processor in succession, eventually producing a complete motion vector at the output of the pipeline.

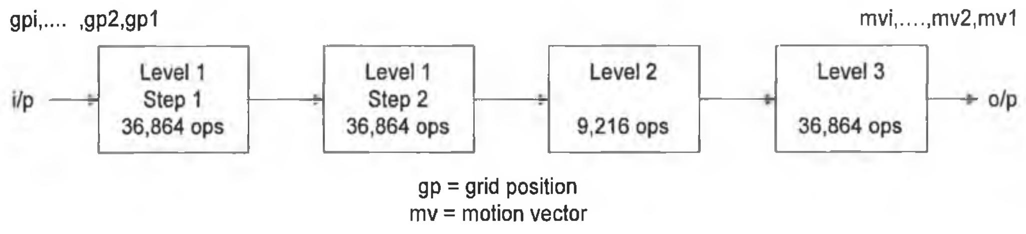


Figure 3.2 Motion estimation pipeline configuration.

Table 3.2 illustrates this flow of grid positions through the pipeline. When the pipeline is full all four processors are executing their relative searches concurrently on different grid positions.

TIME	Stage 1	Stage 2	Stage 3	Stage 4	Output
n	gp_1	-	-	-	-
2n	gp_2	gp_1	-	-	-
3n	gp_3	gp_2	gp_1	-	-
4n	gp_4	gp_3	gp_2	gp_1	-
5n	gp_5	gp_4	gp_3	gp_2	mv_1
:	:	:	:	:	:
:	:	:	:	:	:

n = 36,864 ops

Table 3.2 Flow of grid positions through pipeline.

This pipeline configuration results in the first motion vector being produced in the same time as the sequential execution, but subsequent motion vectors are produced at an interval equal to the stage with the longest processing time. Therefore the processing time for a complete frame (all 99 motion vectors) is:

$$(1x)119{,}808 + (98 \times 36{,}864) = 3{,}732{,}480 \text{ ops.}$$

This figure represents a speedup factor of 3.17 over the sequential implementation.

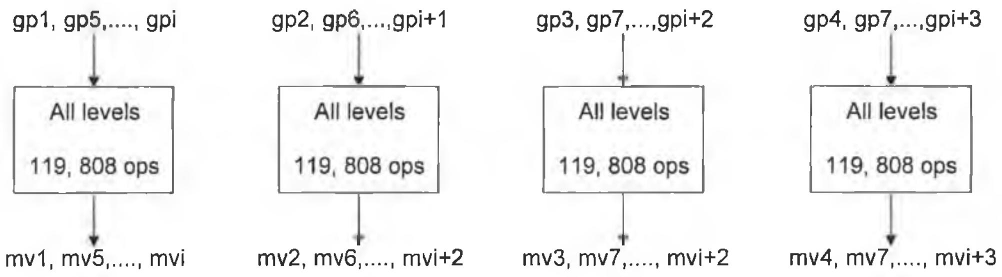


Figure 3.3 Four-processor geometric implementation.

In a geometric system with four processors, a quarter of the grid positions are assigned to each processor for full processing (see Figure 3.3). Now four motion vectors are calculated in the same time as one motion vector in the sequential implementation. Therefore, the speedup factor is 4 (i.e., 2,965,248 ops. per frame) for geometric parallelism with four processors. Clearly, the best improvement in execution time is achieved by the geometric implementation. However, there are other factors that need to be taken into account when making the decision between geometric and pipeline, that both reinforce and weaken the conclusion from this speedup analysis.

In pipeline parallelism, there is generally a limited number of stages into which an algorithm can be practically divided, e.g., four stages in the case of motion estimation. In contrast, with geometric parallelism, the number of processors can be much greater if the data set is large enough and there exists a high degree of data independence. For example, in motion estimation, it could be possible to have 99 processors, each calculating the motion vector for a grid position resulting in 99 motion vectors being calculated in the equivalent time to one motion vector in a sequential implementation. However, there are many parts of the DCU algorithm that have to be executed sequentially. This means that it is not desirable to have a large number of processors as only one processor would be active for the sequential elements while the others were idle, i.e., there would be a high degree of inefficiency.

In geometric parallelism, each processor executes the entire algorithm, as in a sequential system, but only on a subset of the data, and therefore does not require a large restructuring of the algorithm. In contrast, implementing pipeline parallelism can result in an algorithmic structure that is quite different from the original sequential version, and depending on software development costs, can be more expensive to implement. For hardware implementation, a copy of the entire algorithm must be stored at each processor in geometric parallelism, whereas in pipeline parallelism, only a section of the code has to be stored at each processor. This may be a problem depending on the type of processor used. For example, a processor may have a maximum executable program size, or could require expensive program memory. Many processors have an on-chip cache that is used to store executed program code. When an instruction is to be fetched, the cache is first checked for the instruction, and if it is not stored there, the external program memory must be accessed. This can cause bus contention with data accesses, resulting in slower processor performance. It may be possible to store small programs (as in the pipeline case) completely within the cache, thus avoiding any bus contention. With larger programs (the geometric case) this is less likely to occur.

In pipeline parallelism, the algorithm is distributed among the processors, and in geometric parallelism, the data is distributed among the processors. This means that for a pipeline implementation, the program code can be statically divided at startup, whereas for the geometric case, a larger amount of data memory has to be divided dynamically. Also, although in geometric parallelism the results are calculated independently, the calculations may share some common input data. Therefore, many processors could require simultaneous access to the same piece of data, leading to memory conflicts that reduce system performance. In pipeline parallelism, several computations are at different stages of execution and are less likely to require the same input data. These problems can be solved in either software or hardware, but in both cases there is added complexity for geometric parallelism.

Overall, the greater speedup achievable through geometric parallelism and the option to add more processors if further speedup is required is a realistic trade-off against the added complexity that has been identified. These implementation issues can be incorporated into the next step of hardware model refinement as a further hardware-software trade-off. Although geometric parallelism offers the best speedup here, the problem of idle processors caused by the execution time variations for motion estimation has to be solved if geometric parallelism is to be efficient. For algorithm execution on a uniprocessor system, the processor is in use constantly and is therefore 100% efficient. When algorithm execution is on a parallel system, the possibility of processor idleness occurs that reduces overall system efficiency. The solution adopted here (based on [13]) to improve efficiency and retain the same speedup, is to ensure that there are more work-packets (geometrical subdivisions) than processors. A control processor dynamically allocates the work-packets to each processor. When a processor has completed the calculations for a work-packet, it reports the result(s) back to the control processor and is issued with another work-packet. Thus every processor operates continuously irrespective of the work-packet execution time, only becoming idle when the supply of work-packets is exhausted. In the case of motion estimation, the work-packets can be the grid positions.

There are other means of achieving algorithm execution speedup that are not necessarily related to parallelism. The DCU algorithm software was written for simulation purposes, to facilitate algorithm development and evaluate its performance. Realtime execution was not required and, as such, may not have been considered during software development. Therefore, there may be parts of the algorithm for which faster implementations exist. It may also be possible to modify the algorithm and sacrifice image quality for speedup. However, due to the inter-relationships between the elements of the algorithm, any modification of one part may have undesirable consequences for other parts. For example, a faster shape approximation function that produces a less accurate approximation could lead to a larger model failure area and hence larger execution time for the model failure detection process. Whether any overall gain is achieved can only be determined through experimentation. Any such modifications can only be properly evaluated after the algorithm as it stands is implemented.

To determine the number of processors required for real-time execution, it is first necessary to calculate more exact figures for the execution time of the algorithm. This can only be achieved when the algorithm has been optimised for a particular hardware structure. The definition of the hardware structure is Step 3 of the iterative design process.

3.4 Hardware Structure

In the algorithm proposed in Section 3.3, processors are required to co-operate in the execution of a single task, e.g., motion estimation, or operate on separate tasks in parallel, e.g., filtering frames for motion analysis concurrently with change detection. In both cases the execution time is unknown for each processor, and this means that the processors must, preferably, operate asynchronously. At any instance in time there are multiple instructions being executed on multiple pieces of data. The only parallel architecture suitable to operating in this manner are multiprocessors.

A basic multiprocessor system consists of a number of more or less conventional processing elements, each with access to a shared memory and input-output devices (see Figure 3.4). The entire system is controlled by a single operating system that handles the interaction between processors. Access to the common memory is via an interconnection system and each processor can have access to its own private local memory. Inter-processor communication can be performed through the shared memory or through a separate interrupt network.

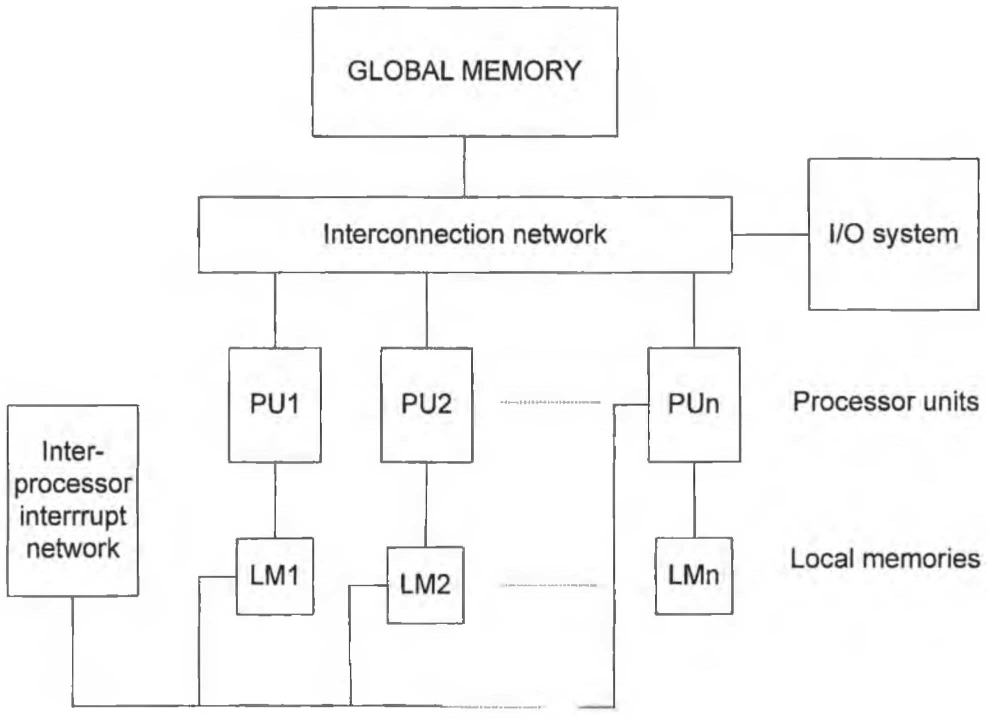


Figure 3.4 Basic structure of multiprocessor system.

In multiprocessor systems inter-processor synchronisation and communication is essential for efficient and correct execution of tasks. When separate processes of a task can be allocated to different processors, the ordering of these processes is a constraint on their execution. The ordering must be followed to ensure the correct outcome of tasks. Also, if the processors have access to the same process queue, the selection of the same process by two or more processors or the omission of a process must be avoided through proper synchronisation. Another situation that requires synchronisation occurs at the inter-process level. Often, processors co-operating in the execution of a task have to exchange data. Since processes may execute with unpredictable speed, this interaction between the processors can result in some processors waiting to be synchronised with the slowest executing process before they can resume execution.

The interconnection network connects the processors to the shared memory (and the I/O devices, if required) and can facilitate the synchronisation process. There are several possible configurations for the interconnection network, most of which are derived from four basic structures - time shared or common bus, crossbar switch, multi-port memory and multistage network. In the time shared arrangement all of the processors, memory modules and I/O devices are connected to a single, passive bus. This is the easiest and least costly arrangement to implement but, since only one processor can use the bus at any time, it is also the most inefficient. The other three structures are multi-bus systems that allow concurrent accesses to memory and I/O devices. Their complexity varies according to the positioning of the control and switching circuitry, and the amount of physical interconnection lines needed.

Chapter 4

SOFTWARE DEVELOPMENT

4.1 Introduction

The task of developing software for the real-time object-based analysis-synthesis coder can be divided into two phases. In the first phase, efficient algorithms are found for the component parts of the overall coding scheme. In doing this, effort spent in the second phase, the actual writing of software, can be concentrated on tweaking the software for optimum performance rather than attempting to enhance the performance of an inferior program. The definition of an 'efficient' algorithm is target platform specific, i.e., it is the algorithm that best exploits the architectural features of the target. The target processor for the real-time implementation is the Texas Instruments' TMS320C30 DSP. The main architectural features of the TMS320C30 are outlined in Section 4.2.

Texas Instruments provide a software development tool-kit that is used to produce executable machine code for the TMS320C30. In addition to an assembler and a linker, these tools include support for high level languages. The ANSI C compiler can produce native code that will run in real-time, with the exception of time-critical applications when the load on the CPU is likely to be high [14]. Where required, speed of execution can be improved by implementing these time-critical sections in assembly language. Typically, the image processing techniques used in object-based analysis-synthesis coding fall into this category - most time is spent on a small section of code that is repeatedly executed. The C compiler provides for a number of methods of including assembly language in a program. Firstly, inline assembly language instructions are supported through the use of the "**asm**" directive. The second method is based on the fact that the output of the compiler is an assembly language program that is passed to the assembler. This program can be edited to introduce optimisations. Alternatively, separate assembly language modules can be linked with compiled C code. If these assembly modules conform to a well defined C function interface, they can call, or be called by, the C modules.

To best exploit the processing capabilities of the TMS320C30 when using assembly language, a knowledge of the instruction set and operation of the device is required. These are also outlined in Section 4.2. The remainder of the chapter details the actual implementation of the software.

4.2 The TMS320C30

The Texas Instruments' TMS320C30 is a 32-bit digital signal processor, capable of performing floating-point, integer and logical operations. The TMS320C30-40 has a 50ns clock cycle time, and with most instructions only requiring one cycle, this gives a total of 20 million instructions per second (MIPS). Furthermore, the TMS320C30 allows two instructions to be executed in parallel, such as a load with a store, or a multiply with an ALU operations, thus giving a peak performance of 40 MOPS. The instruction set also supports block repeats with zero-overhead looping and single cycle branching.

The block diagram in Figure 4.1 illustrates the key architectural features of the TMS320C30. Of much importance to the programming of the device are the memory interface and the pipeline which comprises the CPU and DMA. Maximising the processing throughput depends on the efficient utilisation of these units.

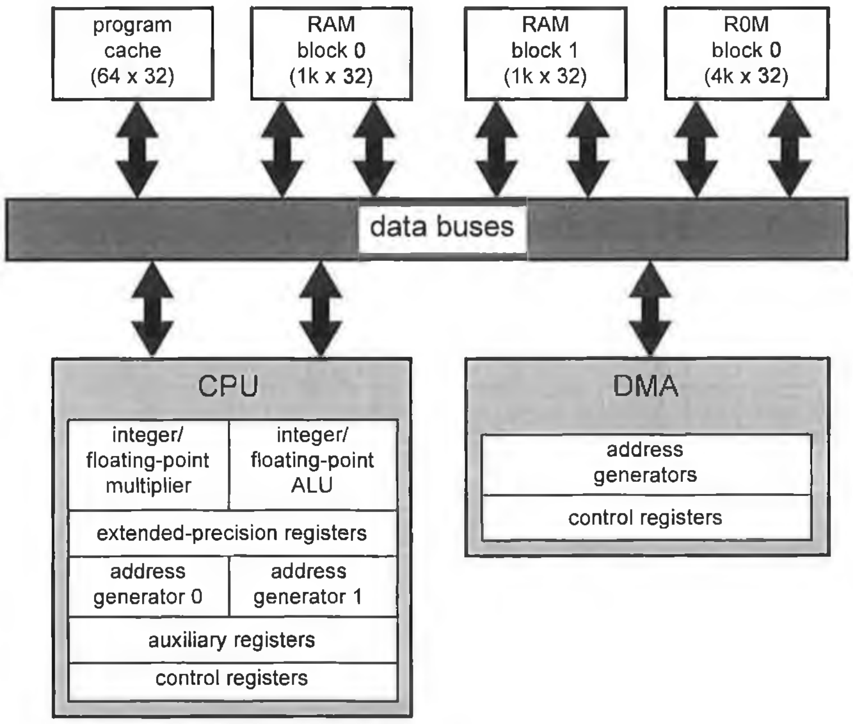


Figure 4.1 Block Diagram of TMS320C30 Architecture.

4.2.1 Main Architectural Features

4.2.1.1 Memory Interface

The TMS320C30 possesses an on-chip Direct Memory Access (DMA) controller that can read from or write to any location in the memory map without interfering with the operation of the CPU. Therefore, it is possible to interface the TMS320C30 to slow external memories and peripherals without reducing the computational throughput of the CPU. The DMA controller contains its own address generator, source and destination registers, and transfer counter. Dedicated DMA address and data buses minimise conflicts between the CPU and the DMA controller. A DMA operation consists of a block or single-word transfer to or from memory. For zero wait-state external memory, a read can be completed in 3 cycles resulting in a maximum transfer rate of 5.56Mwords/s. A write requires 2 cycles (8.33Mwords/s).

On-chip, the TMS320C30 has 2k x 32-bit RAM, equally divided into two blocks, RAM0 and RAM1. In addition, there is a ROM block of 4k x 32-bit. Each RAM block and the ROM block is capable of supporting two data accesses in a single instruction cycle. The separate program, data and DMA buses of the TMS320C30 can provide parallel program fetches, data reads and writes, and DMA operations. For example, a program fetch from the on-chip instruction cache, two data accesses from a RAM block in parallel with the DMA loading the other RAM block can be performed in a single cycle with no effect on the throughput of the CPU.

A 64 word (32-bit) on-chip instruction cache is available to store frequently-repeated sections of code. This code may then be re-fetched from the cache when required, thus reducing the number of necessary off-chip accesses. This frees the external buses for use by the DMA, external memory fetches, or other devices in the system.

4.2.1.2 Pipeline

The TMS320C30 pipeline includes five functional units. Of these, only the DMA does not operate on the instruction. The other units, all of which are in the CPU, form an instruction pipeline that provide the TMS320C30 with the bulk of its processing power. The instruction pipeline allows an overlap in the execution of instructions by dividing each instruction into a number of distinct stages and allocating a separate processing unit to each stage. The execution of an instruction on the TMS320C30 involves four stages (units). These are:

Fetch unit - fetches the instruction from memory and updates the program counter

Decode unit - decodes the instruction word and performs address generation. This unit also controls any changes to the TMS320C30's auxiliary registers and the stack pointer

Read unit - performs any necessary reads of operands from memory

Execution unit - if required, reads operands from registers, performs the necessary operation and writes the generated result to either a register or to memory

In a non-pipeline computer, each of these four steps must be completed fully before the next instruction is started. By cascading the four stages in a pipeline, successive instructions can be executed in an overlapped manner. Once the pipeline is full, an instruction can be completed every instruction cycle whereas a non-pipeline computer would take the equivalent of four cycles. The operation of the pipeline is internally managed on the TMS320C30 with the result that the pipeline is transparent to the user.

4.2.2 Programming the TMS320C30

4.2.2.1 Instruction Set

The TMS320C30 assembly language instruction set supports numeric intensive, digital signal processing and general purpose applications. The instruction set can be organised into the following functional groups:

Load and store instructions
Two-operand arithmetic/logical instructions
Three-operand arithmetic/logical instructions
Program control instructions
Parallel instructions
Interlocked instructions

The load and store instructions perform the movement of a single word to and from the registers and memory. Additionally, the conditional load of a register is supported. Instructions that can manipulate data on the system stack are included in this group.

The two-operand instructions consist of 35 arithmetic/logical instructions. The two operands are the source and destination. The source operand may come from memory, a register or be a part of the instruction word, and the destination is always a register. This group of instructions includes integer, floating-point, and logical operations, and also supports 32-bit or multiprecision arithmetic and logical shifts.

The three-operand arithmetic/logical instructions are a subset of the two-operand group. These 17 instructions allow the reading of two operands from memory and/or the register file in a single cycle and stores the result in a register.

All instructions that affect program flow are included in the program control function group, and can be divided into two main types - repeat modes and branching. Several of the program control instructions are capable of conditional operation based on the contents of the status register after the previous instruction has been completed. The repeat modes can implement zero-overhead looping. RPTS (repeat a single instruction) reduces bus accesses as it requires only one fetch of the instruction that is to be repeated, while similar efficiency can be achieved for several instructions using RPTB (repeat a block of instructions) and the instruction cache. Through the use of stack, block repeats may be nested. The TMS320C30 has two types of branching - standard and delayed. Standard branches empty the pipeline before performing the branch and thus require four instruction clock cycles. Included in this type are calls, returns and traps. Delayed branches do not empty the pipeline and allow the subsequent three instructions to be fetched before the program counter is modified, effectively resulting in a single cycle branch.

The parallel-operations instructions provide the TMS320C30-40 with the capability of achieving 40 MOPs. They include parallel floating-point and integer multiplication with an addition, arithmetic/logical operations in parallel with a store instruction. Parallel loads and stores are also possible.

The interlocked instructions support multiprocessor communication. They use two external flag pins, XF0 and XF1. XF0 signals an interlocked instruction request and XF1 acts as an acknowledge signal for the requested interlocked instruction. Through the use of these external signals, the interlocked instructions can be used to implement busy-waiting loops, to manipulate a multiprocessor counter, to implement a simple semaphore mechanism, or to perform synchronisation between two TMS320C30s.

4.2.2.2 Addressing

The TMS320C30 supports a number of addressing modes which allow access of data from memory, registers, and the instruction word. However, not all instructions support all modes of addressing. In *register* addressing, a CPU register contains the operand. *Direct* addressing allows a data value at a specific memory location to be accessed by explicitly stating that memory address in the instruction. An operand can be explicitly included in the instruction word using *immediate* addressing. The value can be 16-bit (short-immediate) or 24-bit (long-immediate). *Indirect* addressing is used to specify the address of an operand in memory through the contents of an auxiliary register, optional displacements and index registers. Two auxiliary register arithmetic units (ARAUs) can generate two addresses in a single cycle, working in parallel with the CPU. With an ARAU it is possible to pre- or post-increment/decrement the contents of an auxiliary register with optional modification based on the contents of the index registers or a specified displacement. There are two special types of indirect addressing - circular and bit-reversed, which are particularly useful for efficient implementations of convolutions and Fast Fourier Transforms. *PC-relative* addressing is used for branching. The assembler takes a specified label or address and generates a displacement value relative to the program counter (PC). Upon execution of the instruction this displacement is added to the program counter if the branch condition is true.

4.2.2.3 Efficient Assembly Code

In order to obtain the best possible performance for a program, the assembly code must reflect the architectural features of the TMS320C30 that provide it with its processing power. Each program will have particular requirements and it may not be possible to use each feature in every case.

The use of delayed branches instead of standard branches can save instruction cycles. The following three instructions after a delayed branch instruction are executed whether the branch is taken or not. If fewer than three instructions can be placed after the branch instruction due to program flow constraint, a delayed branch can still be used by filling the empty instruction slots with null instructions (NOP). For many algorithms, such as convolutions, there is an inner kernel of code where most of the execution time is spent. Using repeat modes allows these sections of code to be executed in the shortest possible time. For optimum efficiency, the instruction cache should be enabled to free the external busses for operand fetching.

Parallel instructions increase the number of operations executed in a single cycle. By combining a multiplication with and addition/subtraction, arithmetic/logic with a store, or load/store operations, the throughput of the TMS320C30 will be increased. It may be necessary to reorder the program flow so as to achieve these combinations. There is a restricted number of addressing modes that can be used with parallel instructions, so it is important that they are taken into account when assigning registers to operands etc. The TMS320C30 has 28 registers in the CPU register file. These registers have some special functions for which they are particularly appropriate. Additionally, all of these registers can be operated upon by the multiplier and ALU, and can be used as general-purpose 32-bit registers. Extensive use of the register file as scratch-pad memory can avoid potential memory access conflicts that reduce processor efficiency. The on-chip memory is considerably faster to access than external memory and can support two accesses per instruction cycle. By using the DMA to transfer operands to/from the internal memory, system performance can be increased.

The main architectural feature of the TMS320C30 that provides it with its processing power is the instruction pipeline. For time-critical programs, it is essential that instruction cycles are not missed due to blockage of the pipeline. The net result of such pipeline conflicts is a reduction in the effective instruction throughput of the TMS320C30 to less than one instruction per cycle. Under worse case conditions the throughput could be a little as one instruction every four clock cycles. The simulator tools available from Texas Instruments allow the pipeline state to be monitored in order to identify any conflicts. These can usually be removed by a re-ordering of the instructions, or can require a re-writing of the code. The following section describes in more detail the operation of the pipeline, the identification and removal of pipeline conflicts. Throughout the software development process particular attention was paid to the elimination or minimisation of pipeline conflicts, which is reflected in the next section.

4.2.3 Pipeline Operation

4.2.3.1 Pipeline Conflicts

Maximum computational throughput is achieved on the TMS320C30 when there is a perfect overlap in the operation of all four pipeline units involved in the processing of an instruction, producing an effective rate of one instruction execution per cycle. However, there are conditions under which a stall in the pipeline can occur, resulting in a reduction in its efficiency, i.e., the effective throughput of the CPU is no longer one instruction per cycle. These stall conditions are known as pipeline conflicts.

Each pipeline unit has been assigned a priority as follows:

Execute (highest)
Read
Decode
Fetch
DMA

When a pipeline unit has completed its operation on an instruction, it passes the instruction onto the next highest pipeline level. If that level is not ready to accept a new instruction a pipeline conflict occurs. When this happens, the lower priority unit waits until the higher priority unit completes its current operation. Figure 4.2 (a) shows correct operation of the pipeline (perfect overlap) while (b) illustrates a stall in the pipeline.

CYCLE	FETCH	DECODE	READ	EXECUTE
n	instr. 1	-	-	-
n+1	instr. 2	instr. 1	-	-
n+2	instr. 3	instr. 2	instr. 1	-
n+3	instr. 4	instr. 3	instr. 2	instr. 1
n+4	instr. 5	instr 4	instr. 3	instr 2
n+5	instr. 6	instr. 5	instr. 4	instr. 3
n+6	instr. 7	instr. 6	instr 5	instr 4

perfect overlap

(a)

CYCLE	FETCH	DECODE	READ	EXECUTE
n	instr. 1	-	-	-
n+1	instr. 2	instr. 1	-	-
n+2	instr. 3	instr. 2	instr. 1	-
n+3	instr. 4	instr. 3	instr. 2	instr. 1
n+4	instr. 4	instr. 3	(nop)	instr 2
n+5	instr. 5	instr. 4	instr. 3	(nop)
n+6	instr. 6	instr. 5	instr. 4	instr. 3

decode unit not ready to accept new input - conflict

(nop) - no operation performed

(b)

Figure 4.2 (a) perfect operation of pipeline. (b) pipeline conflict resulting in loss of throughput.

Pipeline conflicts can be organised into three main groups:

a) branch conflicts

b) register conflicts

c) memory conflicts

a) Branch Conflicts

This group of conflicts involves program flow instructions that read and/or modify the program counter. A program flow instruction causes a conflict as the pipeline is only used for the execution of the instruction and any instructions that enter the pipeline after the program flow instruction are discarded. This is known as flushing the pipeline, and is essential if proper program flow is to be maintained. Flushing the pipeline guarantees that the instructions succeeding the program flow instruction are not incorrectly (with respect to the program flow) partially executed. The following section of code shows a branch instruction (BR):

```
     BR     SUB_LOOP            ; unconditional branch
     ADDI3  *AR0++,R3,R1        ; not executed
     MPYI   R1,R2               ; not executed
     FLOAT  R2                  ; not executed
            ...
            ...

 SUB_LOOP:
     STI    R1,*AR3             ; fetched after BR is
                                ; executed (a delay of
                                ; 3 cycles)
```

Delayed branches avoid this type of conflict as they do not flush the pipeline and ensure that the subsequent three instructions are fetched before the program counter is modified.

b) Register Conflicts

Register conflicts can occur when reading or writing any of the registers associated with address generation.

These registers are divided into three groups:

Group 1 -	auxiliary registers (AR0-AR7), index registers (IR0,IR1) and the block size register (BK);

Group 2 - data-page pointer (DP);
Group 3 - stack pointer (SP)

In the case of a write to a register in these groups, the decode unit of the pipeline cannot use any register within the same group until the write operation is complete (this occurs at the end of the execute operation). In the following lines of code, a register conflict occurs between the first and second instruction. A register from group 1 (IR0) is written to in the first instruction. The decode unit has to wait two cycles until this instruction is completed before it can read the register from the same group (AR0) specified in the second instruction.

```
ADDI    R1,IR0
MPYF3   *AR0,R3,R4
LDF     R3,R5
STF     R4,*AR2++
```

The result is a two cycle delay in the execution of the second instruction. This is the equivalent of inserting two NOP (no operation) instructions into the program, as follows:

```
ADDI    R1,IR0
NOP
NOP
MPYF3   *AR0,R3,R4
```

If an instruction reads a register from any group, then the decode unit cannot operate on an instruction that uses a register from the same group until the read is complete. The execute unit reads registers at the beginning of a cycle, and therefore a delay of one cycle is incurred. In the case of four registers (IR0, IR1, BK and DP), register-read conflicts do not occur.

c) Memory Conflicts

The TMS320C30's internal RAM and ROM blocks can support two accesses per clock cycle. In addition, the external interface can provide one access per cycle. If instruction fetches and data accesses are specified in a way that exceeds this bandwidth, memory conflicts occur. There are four types of memory conflicts:

Program Wait	an instruction fetch is prevented from beginning;
Program Fetch Incomplete	occurs when an instruction fetch takes more than one cycle due to memory wait states;
Execute Only	occurs when a sequence of instructions requires three CPU data accesses in a single cycle, or during an interlocked load;
Hold Everything	occurs when an access is made to either the primary or expansion bus and the bus is already in use.

If slower external memory is used, then the bandwidth is reduced and memory conflicts are more likely to occur. Therefore, for time critical applications, it is essential that the data is stored in single-cycle access memory.

4.2.3.2 Removing Conflicts

Once a pipeline conflict has been identified, it is normally possible to remove it by rearranging the instructions in the program. However, there will inevitably be cases where the program flow is not flexible enough and conflicts are unavoidable. Branch conflicts can only be removed by replacing the standard branch with the delayed equivalent. Again, this requires a re-ordering of instructions so that the delay slots following a delayed branch instruction can be filled. One method of avoiding register conflicts is to ensure that the relevant registers are not used for any purpose other than address generation. In cases where this is not possible, the instructions that cause the conflict should be separated by instructions that do not use any registers in the appropriate group. Memory conflicts are more difficult to avoid. Using the DMA to transfer data into the higher bandwidth on-chip memory and extensive use of the register file can help minimise memory conflicts.

4.3 Software Implementation

A C implementation of the SIMOC coding algorithm has been developed by the Video Coding Group at DCU for the purpose of simulation and evaluation of the coding scheme (this will be referred to hereafter as the *VCG C implementation*). This implementation served as both a starting point and reference codec for the real-time software implementation.

As a starting point, the VCG C implementation provides two benefits - it establishes program architecture and flow, and each DSP software module can be integrated into a complete codec upon implementation. Any components of the algorithm that do not require optimisation can be compiled using the TMS320C30 ANSI C compiler. At any stage, the operation of the real-time implementation can be verified against the VCG C reference codec.

Along with the TMS320C30 code generation tools (ANSI C compiler, assembler and linker) TI provide a C source debugger/simulator. This a software package that simulates the non-real time operation of the entire instruction set and key peripheral features such as the DMA. Debugging of both C and assembly language is supported, as is the monitoring of the entire register set and memory space. The simulator has two features that are useful in the development of time-critical code. Firstly, the number of instruction cycles consumed by a section of code can be determined through the benchmarking facility. Secondly, each unit in the instruction pipeline can be monitored in order to identify any conflicts.

Loughborough Sound Images' TMS320C30 System Board also provides program debugging capabilities. However, unlike the simulator, program execution is performed in real-time on an actual TMS320C30. This board is a PC-AT ISA card that incorporates a single DSP, with up to 256Kwords of zero-wait state memory, 64K of which is dual-ported RAM and accessible to the host PC. A host interface library is provided that allows the loading of TMS320C30 object code (from a disk file) to the board, to start execution of that code, and to pass data back and forth between the program running on the TMS320C30 and on the PC. This allows a program on the host to off-load processing intensive tasks onto the DSP.

In the development of the software for the real-time implementation, correct operation of the various optimised modules was verified through testing using both the simulator and the LSI System Board. Due to its superior debugging capabilities, initial testing of modules was performed using the simulator. However, it was found that the simulator could not efficiently handle large amounts of test data, so small sets of randomly generated test data were used in this process. After the elimination of any identified coding errors, the modules were then executed on the LSI System Board using actual test data generated by the VCG C implementation from the 'Miss America' and 'Claire' sequences. The output from individual modules was compared for correctness against the output of the equivalent module in the VCG C implementation.

4.3.1 Change Detection

Change detection is the first step in the image analysis process. The current frame is compared with the previous reconstructed frame in order to segment the current image based on significant changes in intensity. This segmentation is represented by a binary mask, called the change detection mask, indicating object (CHANGED) and static (UNCHANGED) regions. The eight stages of change detection are outlined in Section 2.4.1.

In the first stage of the segmentation process the inter-frame difference is evaluated on a pel-by-pel basis, by calculating the absolute difference between each pel in the current frame and the corresponding pel in the previous reconstructed frame. These pel differences are transformed into a binary mask by thresholding at each pel position, based on the result of the summation of the pel and its neighbours in a 3x3 mask. This is a form of mean-value filtering and eliminates any non-zero differences that are due to camera noise, while retaining the differences of interest, i.e., those that are due to motion. Further noise filtering is achieved by applying a binary median filter to the binary mask. To ensure temporal coherency in the segmentation, the binary mask is combined with the output segmentation mask of the previous processed frame.

The remaining stages of change detection ensure that over-segmentation of the image is avoided.

In dilation, a structuring element is translated through the image, setting a pel to '1' (CHANGED), where at least one of the pel's neighbours matches the corresponding element in the structuring element. Here the structuring element is 3x3 pels in size with all elements set to CHANGED. This has the effect that, in each iteration of the dilation, any pel with an object pel in its 3x3 neighbourhood in the previous iteration is added to the object. Dilation results in an increase in the size of an object.

In erosion, a pel is set to '0' (UNCHANGED) where the neighbourhood pels are not identical to the structuring element. The same structuring element as for dilation is applied, with the result that, in each iteration object pels are removed that were connected to at least one UNCHANGED pel in the previous iteration.

Combining dilation and erosion in this order produces another morphological operation called closing. Closing connects objects that are near to each other, smoothes object contours by filling up narrow channels, and fills in small holes (UNCHANGED regions) within objects. Any remaining small objects and holes are eliminated by applying an average size criterion. The stages in change detection can be divided into three groups. Steps a) and d) are point operations, while binarisation, median filtering, dilation and erosion are neighbourhood operations. The remaining steps are implemented using contour-based techniques.

4.3.1.1 Point Operations

In point operations, the output pel is a function of the pel in the corresponding position in the input image only. The point operations in change detection are relatively straightforward to implement as they involve simple arithmetic/logic instructions that are repeated for every pel. However, a problem arises with the number of memory accesses that are required over a small number of instructions. In the computation of the absolute difference, for example, two operands have to be read from memory and the result stored in three instructions, leading to a high probability of pipeline stalls due to memory conflicts. The minimisation of the number of memory conflicts requires the re-ordering of instructions and the use of DMA. In Listing 4.1, the logical ordering of instructions (subtract, absolute, store) was changed to avoid an *execute only* memory conflict.

```
        SUBI3   *AR0++(1),*AR1++(1),R1      ;setup first result
        ABSI    R1,R2

        LDI     NUM_OF_LOOPS,RC             ;number of loops
        RPTB    END_ABSDIFF                 ;main loop
        SUBI3   *AR0++(1),*AR1++(1),R1      ;subtract pels
        STI     R2,*AR2++(1)                ;store last result
END_ABSDIFF:
        ABSI    R1,R2                       ;absolute result

        STI     R2,*AR2                     ;store final result
```

Listing 4.1 Main loop of absolute difference function.

Over a small number of instructions, as in this case, it is only possible to perform one DMA transfer and therefore only one of the operands or the result can be stored in internal memory. If an operand is located on-chip then the **SUBI** instruction will only require a single cycle (with both operands in external memory this instruction is executed in two cycles). However, requiring the result to be written to external memory leads to a *hold everything* memory conflict. In the instruction pipeline, memory writes are initiated in the execute phase, whereas memory reads are performed in the read phase. In Listing 4.1, the **STI** instruction begins its memory write during the execute phase. Writes to off-chip memory require two external bus cycles (equivalent to two instruction cycles under most conditions). On the next instruction cycle the **SUBI** instruction enters its read phase and attempts to read the operands, but the external bus is busy servicing the write and a pipeline stall occurs. Under these conditions the main loop in Listing 4.1 requires 4 cycles to complete. The same cycle count is achieved if the result is stored in internal memory and transferred off-chip by the DMA. For addressing purposes this was the configuration chosen.

For correct operation a result cannot be written to an address that has not been serviced by the DMA. To avoid this, two buffers of equal length are set up in internal memory. While results are stored to one buffer, the DMA can transfer the contents of the other off-chip. When the CPU fills a buffer, it switches buffers with the DMA.

4.3.1.2 Neighbourhood Operations

The neighbourhood operations used in change detection are 2-D transforms where the output for a given pel is a function of itself and its surrounding pels in either a 3x3 or 5x5 kernel. In common with point operations, the same sequence of instructions can be repeated for each pel. The generation of the binary mask is a straightforward process as it only involves a number of additions followed by a comparison to a given threshold for each pel. It can be expressed mathematically (for a frame size of M_1 x M_2) as:

$$\mathrm{sum}(x,y) = \sum_{j=-N//2}^{N//2} \sum_{k=-N//2}^{N//2} i(x+k,\ y+j) \ \forall\ 0 \le x < M_1\ ,\ 0 \le y < M_2$$

$$o(x,y) = \begin{cases} 1 \text{ if } \mathrm{sum}(x,y) > \text{ threshold} \\ 0 \text{ o/w} \end{cases}$$

Equation 4.1

where:

i(x,y) - input

o(x,y) - output

NxN - size of kernel

// - rounded division

Binary median value filtering can be achieved using the same mechanism - the dominant binary value can be determined by adding all the binary values in the 5x5 neighbourhood, and if the sum is greater than 12, then level '1' is the result, otherwise it is level '0'. Dilation and erosion can be implemented in this manner also. In dilation, if the sum is greater than zero, then at least one pel is an object pel. Similarly, in erosion, if the sum is not 9, then at least one pel is not an object pel. However, the structuring element used has properties that allows for an implementation that uses only logical operations.

A common, and important, characteristic of the neighbourhood operations used in change detection is that they are separable, i.e., the 2-D operation can be implemented as two 1-D operations. Separation can reduce the amount of necessary calculations for processing a frame. For example, calculating the sum of a pel and its neighbours, as expressed in **Error! Reference source not found.**, requires $(M_1 \text{ x } M_2)N^2$-1 additions. The equivalent separated function can be expressed as:

$$\text{sum}'(x,y) = \sum_{k=-N//2}^{N//2} i(x+k,y) \ \forall\ 0 \le x < M_1 \ ,\ 0 \le y < M_2$$

Equation 4.2 (a)

$$\text{sum}(x,y) = \sum_{j=-N//2}^{N//2} \text{sum}'(x,y+j) \ \forall\ 0 \le x < M_1 \ ,\ 0 \le y < M_2$$

Equation 4.2(b)

Equation 4.2(a) represents the horizontal component of the summation and requires $(M_1 \text{ x } M_2)N$-1 additions, as does the vertical component (Equation 4.2(b)). This represents a reduction of 50% and 66% in the number of calculations for a 3x3 kernel and 5x5 kernel, respectively. This gain is achievable due to the data-recurrence inherent in these neighbourhood operations, i.e., some calculations are repeated a number of times. For example, the computation of the 3x3 sum for pel i(x,y) requires, in part, the evaluation of i(x-1,y) + i(x,y) + (x+1,y). This calculation is also required in the computation of the output for pels i(x,y-1) and i(x,y+1). By performing this calculation only once, i.e., as in the separated case, less calculations are required than in the case represented by Equation 4.1.

In the processing of a 1-D summation, N pel values are combined to produce one output. For the calculation centred on pel position i(x) this involves the pels in the window [i(x-N//2), ..., i(x), ..., i(x+N//2)]. At the next pel position, i(x+1), this window is moved one place to the right, i.e., pel i(x-N//2) leaves the window and pel i(x+N//2+1) enters it. Therefore, the output for pel i(x+1) can be calculated from the output for pel i(x) by subtracting the value of the pel that leaves the window and adding the value of the pel that enters it. For N=3 this maintaining of a running sum achieves no overall gain (1 add and 1 subtract as opposed to 2 adds). However, for N=5 there is a saving of 50% in the number of required calculations (1 add and 1 subtract as opposed to 4 adds). The running sum can be implemented for both the horizontal and vertical operations.

With the structuring element used in this algorithm it is also possible to separate the 2-D dilation and erosion. Here the dilation can be implemented as a logical OR of all the binary pels in a 3x3 kernel (separated into three 3x1 1-D components), while the erosion process is equivalent to a logical AND of the pels in the kernel.

It is possible to implement both separated 1-D components as two successive zero-overhead loops. However, for relatively large geometric divided image areas (> 2k pels), the results of the first loop cannot be stored in internal memory. This increases the possibility of encountering memory conflicts in accessing external memory for the second loop. To avoid unnecessary storing and subsequent re-loading of data, operands required for the vertical component should be processed as soon as they are produced by the horizontal component. This involves combining both components in a single loop. Since a horizontal result is required in N vertical calculations, it has to be stored until all these calculations are complete. This storage requirement amounts to (N-1)lines, and is much less than it would be in a two separate loop implementation. This storage area is implemented as a circular buffer, using the TMS322C30's circular addressing mode, where it the oldest data in the buffer is overwritten as new data is added. A circular buffer is also used in the 3x3 sum and binary median value filter implementations, to store the running sum of each column for the vertical component. Both circular buffers are located in internal memory.

In the generation of the binary mask and binary mean value filtering there is sufficient memory bandwidth to avoid any pipeline conflicts. However, in dilation and erosion there are almost as many memory accesses over fewer instruction, so DMA has to be used to transfer the result to external memory.

4.3.1.3 Contour Based Techniques

The final two steps of change detection use shape representation techniques to calculate the area of CHANGED/UNCHANGED objects, and for the elimination of objects that do not meet the size criterion. Central to these techniques is the detection of object boundaries. The boundary, or contour of an object, is the set of all pels that have at least one 4-connected neighbour[4] that is not an object pel. These pels can be found by contour following. This process starts with a pel previously found to be on the boundary, and then adding a neighbourhood pel that is also a boundary pel to the set. This addition of pels to the set is repeated, with each new added pel serving as the starting point in the next iteration. The following process stops when some termination condition is met. In the DCU algorithm a chain code are used to describe the set of contour pels. Instead of storing the co-ordinates of all the contour pels, only the co-ordinates of the starting point and a sequence (chain) of integer codes,

[4] the pels to the left, right , above and below

corresponding to the direction of the next pel on the contour, are stored. There are eight possible directions, one for each of the eight pels in a 3x3 neighbourhood (see Figure 4.3)

3	2	1
4	P	O
5	6	7

Figure 4.3 Contour direction codes.

The chain code is constructed by a contour following algorithm [15] where the starting pel is found by a top-to-bottom, left-to-right scan of the image. This pel also represents the terminating point of the algorithm, which occurs when the pel is encountered for a second time. In addition to the external contour of an object, the contours of any holes within the object have to be determined. This is done using the chain codes of the external contour to determine if the contour point is located on a downward arc. If this is the case then the object is scanned horizontally until either a hole pel is found or the opposite edge of the object is reached. The first hole pel found is the starting point for the contour following of the hole.

Since the chain code is a complete representation of an object's shape, the area of an object (or hole) calculated from this information, in a method similar to numerical integration [16]. This is much faster, for sizeable objects, than counting pels since the contour contains only a small number of the object's pels. For objects with holes, the hole area must be subtracted from the object area. Finally, the area of the UNCHANGED background is calculated by subtracting the total area of all objects and holes from the area of the image.

Those regions that do not meet the average size criterion are marked for elimination.

4.3.2 Motion Analysis

In motion analysis a full resolution half-pel motion vector field is produced that represents the translational motion of the CHANGED objects identified by change detection. Motion vectors are calculated at predetermined grid positions within the CHANGED area using Bierling's motion estimation algorithm. These motion vectors are subsequently interpolated to provide a motion vector for each CHANGED pel. The first level in Bierling's algorithm (see Section 2.4.1.2) requires the low-pass filtering of the input images to reduce the number of erroneous estimates that can be produced by high frequency image components. For half-pel accuracy, the final level of the motion estimation is performed on bilinearly interpolated versions of the input images. The change detection mask must also be upsampled to the same dimensions. Low-pass filtering, bilinear interpolation, and upsampling constitute the motion estimation support functions.

In addition to the motion vector field, motion analysis produces a ternary mask that is derived from the change detection mask and the motion vector field. In this mask, called the SMU mask, each pel is classified as either part of the background (Static), part of a model complaint object (Moving), or a pel that was previously occluded (Uncovered).

4.3.2.1 Motion Estimation Support Functions

The low-pass filter is realised by replacing each pel in the input images by the mean value of itself and its neighbours in a 3x3 kernel. Although a mean value filter is not an accurate approximation of a low-pass filter, it was chosen by Bierling due to its lower computational complexity. The mean value filter has the same separable characteristic as the neighbourhood operations described in Section 4.3.1.2. Figure 4.4 shows the filter structure used in the implementation.

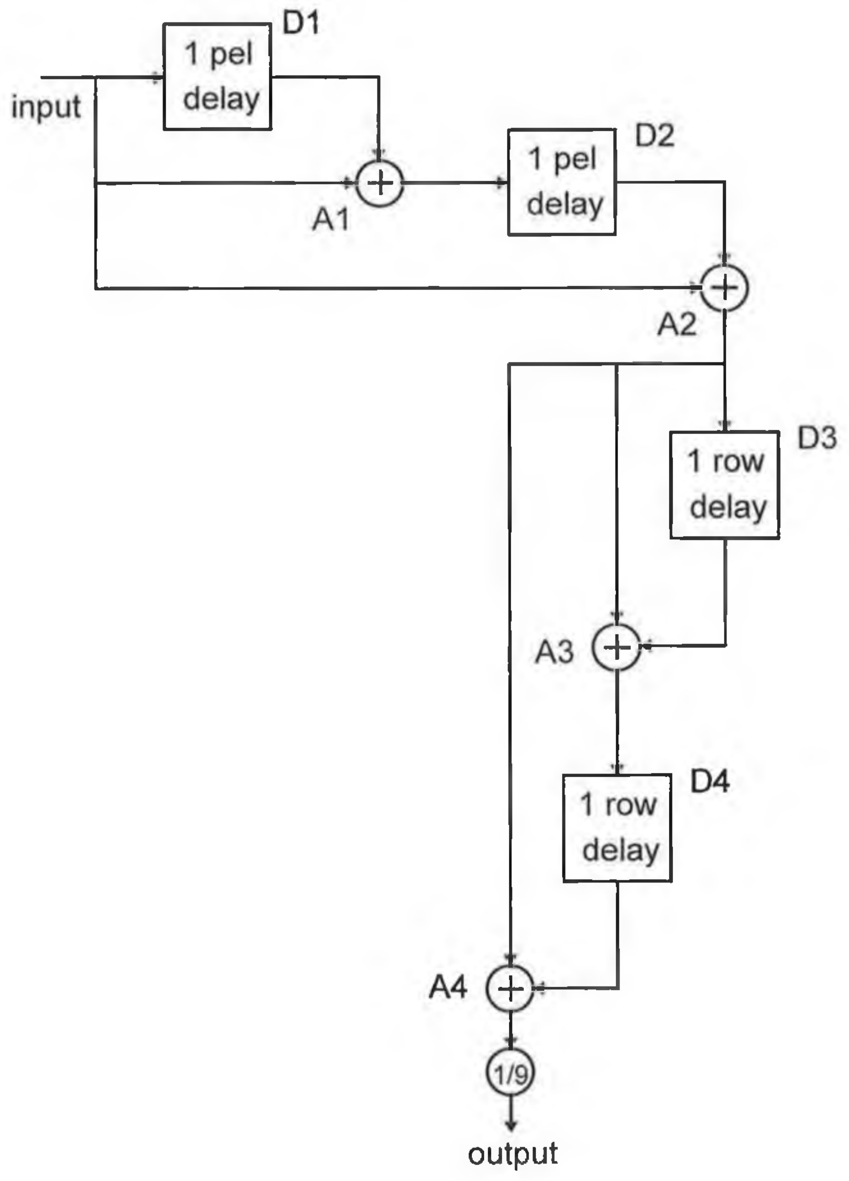


Figure 4.4 Mean-value filter structure.

Delay D1 is implemented using two pointers to the input, one for the current pel and the other for the previous pel. D2 uses a register to store the result of addition A1 for one loop iteration. D3 & D4 are FIFO buffers, equal in length to one row of the input data. Each buffer has two associated pointers, one for either end of the buffer. Circular addressing is used to redirect a pointer to the physical start of the buffer in memory when the pointer has reached the physical end. The division by nine uses the floating point capabilities of the TMS320C30. The result of A4 is converted to floating point representation and then multiplied by 1/9 before being converted back to fixed point.

The general equation for bilinear interpolation is:

$$f(p,q) = A + (C-A)p + (B-A)q + (D+A-C-B)pq$$

Equation 4.3

Equations 2.1 to 2.4 are obtained by substituting the following known values for p and q:

$$a = f(0,0)$$
$$b = f(0,0.5)$$
$$c = f(0.5,0)$$
$$d = f(0.5,0.5)$$

Although the operations used in bilinear interpolation are straightforward, there is a large overhead in the number of memory accesses required to support the calculations. Each pel in the input frame has four associated output pels that require up to four operands in their calculation. The number of stores is fixed, but it is possible to reduce the number of operand reads. Equation 2.3 can be written as (for pel(x,y)):

$$c_{x,y} = \frac{pel(x,y) + pel(x,y+1)}{2}$$

Equation 4.4

Similarly for Equation 2.4:

$$d_{x,y} = \frac{\text{pel}(x,y) + \text{pel}(x,y+1) + \text{pel}(x+1,y) + \text{pel}(x+1,y+1)}{4}$$

Equation 4.5

It can be seen that an addition required in the evaluation of Equation 4.5 has already been performed in the calculation of Equation 4.4. Furthermore, another addition is duplicated in the processing of pel(x+1,y):

$$c_{x+1,y} = \frac{\text{pel}(x+1,y) + \text{pel}(x+1,y+1)}{2}$$

Therefore, the three additions in Equation 4.5 can be replaced by a single addition of two results generated in other calculations. As all divisions required in the bilinear interpolation are by a power of 2, they can be implemented by logical right shifts.

It is possible to logically express the rules for upsampling the change detection mask given in Section 2.4.1.2 as:

$a = A$

$b = A.B$

$c = A.C$

$d = A.C.(B+D) + B.D.(A+C)$

As with bilinear interpolation, there is an overlap in the calculations performed, in particular for half-pel position d, which can be expressed as (for pel(x,y)):

$$d_{x,y} = c_{x,y}.\alpha_{x+1,y} + c_{x+1,y}.\alpha_{x,y}$$

where:

$$\alpha = A+C$$

Due to the high number of data transfers that have to be made, DMA is used in both bilinear interpolation and upsampling to transfer the results from on-chip to off-chip memory.

4.3.2.2 Motion Estimation

In motion estimation there are a number of equally spaced grid positions defined, for which motion vectors may have to be calculated. The first position is located at (8,8) with the remaining positions at separations of (16,16). Bierling's algorithm estimates the motion using different sized measurement windows centred on these grid positions. Motion vectors are only calculated for those grid positions within the CHANGED area. This leads to two possible conditions - the measurement window is either entirely within the CHANGED area or straddles the object boundary. This is illustrated in Figure 4.5.

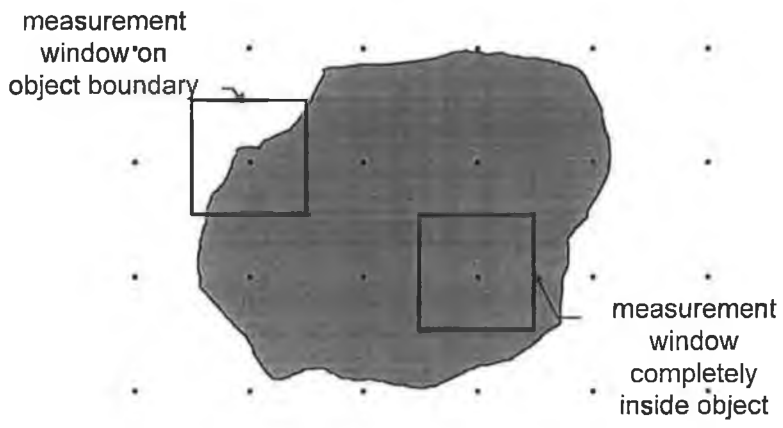

measurement window on object boundary
measurement window completely inside object

Figure 4.5 The two possible measurement window cases.

In the first case, the formula for the block-matching MAD can be written as:

$$\mathrm{MAD}(x,y) = \frac{1}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left|\mathrm{curr}(k+i,l+j) - \mathrm{prev}(k+i+x,l+j+y)\right|$$

Equation 4.6

where:

N^2 is the size of the measurement window

(k,l) is the upper left corner of the measurement window

(x,y) is the motion vector under evaluation

For the second case, Equation 4.5 must be modified to exclude those pels outside the CHANGED area from contributing to the MAD calculation. This can be expressed as:

$$\mathrm{MAD}(x,y) = \frac{1}{N_{chng}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left|\mathrm{curr}(k+i,l+j) - \mathrm{prev}(k+i+x,l+j+y)\right| * \left(!(\mathrm{mask}(k+i,l+j) == 0)\right)$$

Equation 4.7

where:

N_{chng} is the number of CHANGED pels in the measurement window.

Equation 4.7 is equivalent to Equation 4.6 when all pels are CHANGED, i.e., $N_{chng} = N^2$ and $(!(\mathrm{mask}(k+i,l+j)==0)$ is always 1.

In determining the motion vector that yields the lowest MAD, the magnitude of all the calculated MADs are compared. The same result can be achieved by comparing the magnitudes of the sum of absolute difference (SAD) of each motion vector. This removes the need for the division by N_{chng} and consequently the counting of this value. By choosing to represent CHANGED pels as 0xff (255) and UNCHANGED as 0x0, Equation 4.7 can be rewritten as:

$$\mathrm{SAD}(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left|\mathrm{curr}(k+i,l+j) - \mathrm{prev}(k+i+x,l+j+y)\right| . \mathrm{mask}(k+i,l+j)$$

Equation 4.8

The most efficient representation of Equation 4.8 in assembly language is given in Listing 4.2, and requires 4 cycles per pel. From this it is possible to determine the number of cycles required at each level in Bierling's hierarchical algorithm, and can be expressed mathematically as[5]:

$$4StepSP(GP_{chng}N^2)$$

Equation 4.9

where:

Step is the number of steps in the level

SP is the number of search positions (9)

GP_{chng} is the number of grid positions within the CHANGED area

```
        RPTB    END_SAD_LOOP

        SUBI3   *AR0++(1),*AR1++(1),R1      ; subtract pels
        ABSI    R1                          ; absolute result
        AND     *AR2++,R1                   ; apply mask
END_SAD_LOOP:
        ADDI    R1,R7                       ; add result to
                                            ; cumulative total
```

Listing 4.2 Efficient implementation of Equation 4.8.

As an alternative to this implementation it is possible to predetermine, before block-matching is performed, those pels that will contribute to the SAD, and eliminate those that do not from all calculations - effectively removing the testing of the mask in Equation 4.8. Also, in doing this, those grid positions whose measurement window is completely within the CHANGED area can be identified, for which a more efficient implementation than Listing 4.2 can be used. It has to be ascertained whether the extra overhead in eliminating the UNCHANGED pels can be offset by the reduction in both the number of

[5] This assumes that loop overhead is negligible compared to the cost of evaluating Equation 4.8 for each pel.

SAD calculations and the complexity of these calculations, i.e., is this method ('Method B') more efficient than that represented by Equation 4.8 ('Method A'). The number of cycles for Method B can be calculated as follows:

$$
\begin{aligned}
&C_1 N^2 GP_{chng} \quad a) \\
&+ C_2 StepSPN^2 GP_{full} \quad b) \\
&+ C_3 StepSPChng \quad c)
\end{aligned}
$$

Equation 4.10

where:

C_1, C_2, C_3 are the cycle counts for each stage

GP_{full} is the number of grid positions whose search window lies within the CHANGED area

Chng is the number of CHANGED pels in boundary search windows

Listing 4.3 gives the implementation of these three stages, yielding cycle counts of $C_1 = 4$, $C_2 = 3$ and $C_3 = 5$.

a)

```
      RPTB   END_GET_CHANGED_PELS
      AND3   *++AR0(1),R1,IR1     ; mask off LSB: IR1 = 1 if
                                  ; CHANGED and list will be
                                  ; incremented
      SUBI3  R6,AR0,R5            ; get offset relative to
                                  ; search origin
END_GET_CHANGED_PELS:
      STI      R5,*AR1++(IR1)     ; store result
```

b)

```
      RPTB   END_CHANGED_SEARCH

      SUBI3  *AR0++(1),*AR1++(1),R1
      ABSI   R1                   ; calculate abs. difference
END_CHANGED_SEARCH:
      ADDI   R1,R7                ; add abs. difference to
                                  ; cumulative total
```

c)
```
        RPTB    END_SEARCH

        SUBI3   *+AR0(IR0),*+AR2(IR0),R1
        LDI     *++AR3(1),IR0       ; load next CHANGED pel
                                    ; offset
        ABSI    R1
END_SEARCH:
        ADDI    R1,R7           ; add abs difference to
                                ; cumulative total
```
Listing 4.3 Code extracts from implementation of Method B.

In Listing 4.3 a), a list of offsets, relative to the grid position, of all CHANGED pels is produced. If the size of this list is equal to the search window size, then all pels are CHANGED and Listing 4.3 b) can be used to determine the SAD, otherwise Listing 4.3 c) has to be used.

From simulation, using the VCG implementation and test sequences 'Miss America' and 'Claire' it has been found that Method B is the more efficient of the two methods. The results of this simulation are summarised in Table 4.1.

Sequence	**Method A**				**Method B**			
	Level 1	Level 2	Level 3	Total	Level 1	Level 2	Level 3	Total
Miss America	3.78	0.48	1.89	6.15	3.4	0.44	1.75	5.59
Claire	4.94	0.62	2.47	8.03	4.45	0.55	2.22	7.22

All values have been averaged over 50 frames and are rounded to the nearest higher 10000 cycles.

Table 4.1 Cycle count (in millions of cycles) for implementation of methods A & B.

Although Method B represents an improvement in performance over Method A, the processing power requirements are still high - approximately 35% of a single TMS320C30 per frame. To reduce this figure, sub-optimal block-matching techniques have to be used where the reduction is achieved by sacrificing some image quality. Equation 4.10 gives the cycle count for a level, and for it to be decreased the value of one or more parameters must be reduced.

The number of grid positions (GP_{chng}) is a fixed figure determined by the change detection mask. Also the number of search positions (SP) and steps is fixed according to Bierling's algorithm. The only parameter that can be changed is the number of pels over which the SAD is calculated. This can be achieved by two methods:

a) reduce measurement window size
b) pel subsampling within the measurement window

A third option exists that involves the elimination of a level in the hierarchy. Each level successively refines the motion vectors. In Level 3, the refinement is the smallest (± 0.5 of a pel), making it the obvious choice for dropping. This also has the added benefit of removing the need for bilinear interpolation of the motion estimation frames (upsampling of the change detection mask is still required for the moving area detection discussed in Section 4.3.2.4).

Pel subsampling within the measurement window is based on the pixel decimation technique described by Liu and Zaccarin in [17] where a 4:1 subsampling ratio is used. The measurement window is divided into 2x2 blocks, with each pel assigned a label a, b, c, or d (see Figure 4.6). Four subsampling patterns, A, B, C, and D, are defined consisting of all the pels labelled a, b, c, d respectively. The subsampling pattern alternates with each search position. This means that only ¼ of the pels in the measurement window are used in the SAD calculations for a search position.

Finding the motion vector is a two-step process. Firstly, for each of the subsampling patterns a motion vector is obtained that produces the minimum SAD over all the search positions where the pattern was used. This results in four motion vectors for which a SAD is then calculated over all the pels in the measurement window in the second step. The motion vector that then has the smallest SAD is chosen as the motion vector for the grid position.

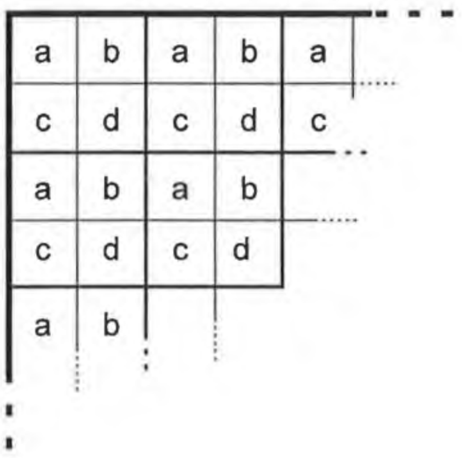


Figure 4.6 Pel-subsampling pattern.

Computationally this requires $K(N_{chng}/4)$ cycles per search position for the first step (K is the number of cycles per SAD calculation, N_{chng} is the number of CHANGED pels in the measurement window). A further $3KN_{chng}$ are needed in the second step to complete the SAD calculation for the four candidate motion vectors (the SAD only has to be calculated over the ¾ pels not included in the original computation). Equation 4.10 can be updated to give the cycle count for this sub-optimal method, which is given by:

$$\begin{gathered} C_1N^2GP_{chng} \\ + C_2StepSP(N^2GP_{full}/4) + 3C_2N^2GP_{full} \\ + C_3StepSP(Chng/4) + 3C_3Chng \end{gathered}$$

Equation 4.11

The search strategy and measurement window size of the modified Bierling algorithm specified by SIMOC were determined through experiment to provide optimal algorithm performance over a number of statistical values [11]. These included PSNR, area of the model failure region and the output bitrate. Accordingly, each sub-optimal option was evaluated against the optimal method under these values. The results of these simulations are shown graphically in Appendix A, and are summarised in Table 4.2 (a) and (b). The corresponding complexities are also given. The quality/bitrate versus complexity trade-off is discussed in Chapter 5. In the simulation for the pel subsampling method the following search position / subsampling pattern was used:

search position	**subsampling pattern**
(0,0)	A
(-step, step)	A
(0,step)	B
(step, step)	C
(-step,0)	D
(step,0)	A
(-step,-step)	B
(0,-step)	C
(step,-step)	D

For the reduced measurement window method, a window size of 20x20 was used for Levels 2 & 3, while a 10x10 window was used for Level 2.

	Optimal	Pel subsampling	Reduced Window Size	No Level 3
Y-PSNR (dB)	36.39	36.44	36.06	36.28
Model Failure Area	4.4%	4.2%	5%	5.3%
Bitrate (kbit/s)	22.1	22.2	24.9	24.9

(a)

	Optimal	Pel subsampling	Reduced Window Size	No Level 3
Y-PSNR (dB)	33.84	34.34	33.29	34.27
Model Failure Area	4.5%	5.2%	5.5%	5.9%
Bitrate (kbit/s)	29	29.6	33.6	31

(b)

Table 4.2 Summary of sub-optimal motion estimation method simulation results, (a) Miss America, (b) Claire

4.3.2.3 Motion Field Interpolation

Once the grid position motion vectors have been calculated, the next step is to perform interpolation to produce a motion vector for each CHANGED pel. A motion vector field of QCIF dimension is constructed, which is divided into 16x16 interpolation blocks with the grid positions at the corners (see Figure 4.7). For any pel with horizontal and vertical indices x and y respectively into its associated interpolation block, its motion vector is given by:

$$\mathrm{mv}(x,y) = \frac{(16-x)(16-y)A + x(16-y)B + (16-x)yC + xyD}{256}$$

Equation 4.12

This expression is evaluated for both the horizontal and vertical components of the motion vector. If any of the grid positions A, B, C, or D lie outside the CHANGED area, their motion vectors must be extrapolated from those within the area. This extrapolation is achieved by setting the external motion vector to the value of that of its nearest grid position inside the area. If there is more than one nearest grid position then an average value is taken.

A straightforward implementation of Equation 4.12 would require at least 8 additions/subtractions, 8 multiplications plus a logical shift right for the division by 256. However, a much more efficient implementation can be achieved by exploiting some regularity that occurs in successive evaluations of the expression. Firstly, a horizontal and a vertical difference are defined as:

$$h_{diff}(y) = \mathrm{mv}(x+1,y) - \mathrm{mv}(x,y) = \frac{-(16-y)A + (16-y)B - yC + yD}{256}$$

$$v_{diff}(x) = \mathrm{mv}(x,y+1) - \mathrm{mv}(x,y) = \frac{-(16-y)A - xB + (16-x)C + xD}{256}$$

then any position can be evaluated in either of two ways:

$$mv(x,y) = mv(0,y) + xh_{diff}(y)$$

$$mv(x,y) = mv(x,0) + yv_{diff}(x)$$

now:

$$mv(0,y) = mv(0,0) + yv_{diff}(0)$$

Therefore:

$$mv(x,y) = mv(0,0) + yv_{diff}(0) + xh_{diff}(y)$$

Another difference can be defined as:

$$vh_{diff} = h_{diff}(y+1) - h_{diff}(y) = \frac{A - B - C + D}{256}$$

Equation 4.12 can now be rewritten as:

$$mv(x,y) = mv(0,0) + yv_{diff}(0) + x(h_{diff}(0) + yvh_{diff})$$

Equation 4.13

where:

$$mv(0,0) = A$$

$$v_{diff}(0) = \frac{-16A + 16C}{256}$$

$$h_{diff}(0) = \frac{-16A + 16B}{256}$$

For any horizontal line in the interpolation block Equation 4.13 can be evaluated using a single addition using iterative calculations within a loop as follows:

$$mv(x+1,y) = mv(x,y) + h_{diff}(0) + yvh_{diff}$$

The last two terms represent an additive factor that can be calculated outside of the loop.

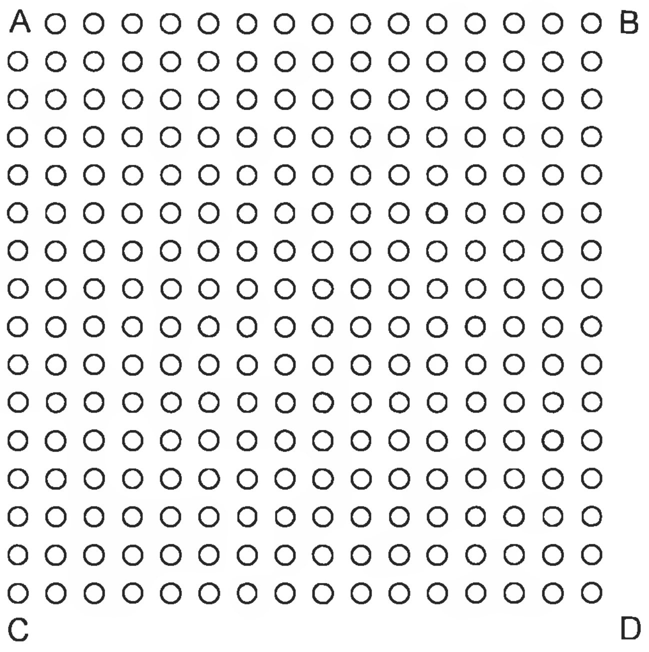


Figure 4.7 Interpolation block showing location of grid positions

4.3.2.4 Moving Area Detection

In this process, a new binary mask, called the moving area mask, is generated using the change detection mask and the motion vector field. For each CHANGED pel, its motion is undone (using its motion vector). If the pel pointed to is also within the CHANGED area, then the current pel is deemed to be MOVING and is represented as such in the new mask.

There are two methods of implementing moving area detection. In the first method, all pels in the change detection mask are tested for their state and only those that are CHANGED are passed to a sub-routine for further processing of their motion vectors. In the second method, the same process is repeated for each pel irrespective of state. This is possible as all UNCHANGED pels have zero motion vectors, and therefore on undoing their motion, they point to themselves, i.e., the become non-MOVING (STATIC). Extracts from the code used to implement each method is given in Listing 4.4. Due to half-pel accurate motion vectors, the testing of the pointed to pel state is done on the upsampled change detection mask.

a)
```
       LDI    *AR3++,R7                ; load mask pel

       RPTB   END_MOVING_DET
       BNZD   SUB_ROUTINE              ; branch if no-zero pel
       LDF    *AR0++(1),R2             ; load horizontal mv offset
       MPYF   2.0,R2                   ; convert to in ½ pel value
       NOP    *AR2++(2)                ; increment position
END_MOVING_DET:
       LDI    *AR3++,R7                ; load next mask pel
....
....
....
SUB_ROUTINE:
       LDF    *AR1++(1),R3             ; load vertical mv offset
       MPYF   R5,R3                    ; convert to ½ pel value
       ADDF   R2,R3                    ; add horiz. value
       BD     END_MOVING_DET           ; delayed return
       FIX    R3,IR1                   ; IR1 = offset in absolute
                                       ; co-ordinates
       LDI    *+AR2(IR1),R4            ; load pointed-to mask pel
       STI    R4,*AR2++                ; if mask pel = CHANGED
                                       ; then result = MOVING
                                       ; else result = STATIC
```

b)
```
       RPTB   END_MOVING_DET
       LDF    *AR0++(1),R2             ; load horizontal mv offset
       LDF    *AR1++(1),R3             ; load vertical mv offset
       FIX    R6,IR1                   ; IR1 = previous offset in
                                       ; absolute co-ordinates
       MPYF   2.0,R2                   ; convert to ½ pel value
       MPYF   R5,R3                    ; convert to ½ pel value
       LDI    *+AR2(IR1),R4            ; load pointed-to mask pel
       STI    R4,*AR3++                ; store result
       ADDF3  R2,R3,R6                 ; combine offsets
END_MOVING_DET:
       NOP    *AR2++(2)                ; increment position
```

Listing 4.4 Moving Area Detection implementations a) sub-routine method, b) processing all pels.

In the first method, the sub-routine is called (and exited) using a delayed branch rather than a **CALL** instruction, as a condition call requires 5 cycles and the delay slots after the instruction cannot be used (the same applies to the return from sub-routine instruction **RETS**). In this implementation, all pels are processed at a cost of 5 cycles per pel, with a further 10 cycles for each CHANGED pel. In the implementation of the second method, 9 cycles per pel are required and all pels are processed. The total cost of the first method, and whether this it is faster than the second method is solely dependent upon the size of the CHANGED area, i.e., the following must hold:

$$5(176x144) + 10xCHANGED < 9(176x144)$$

Therefore, the number of CHANGED pels must be less than 10,138 for the first method to be fastest. From simulation with the test sequences this condition is never satisfied.

4.3.2.5 Uncovered Background Detection

Inter-frame motion of objects can result in regions that were occluded in a previous frame appearing in the current frame. These uncovered background regions are usually long narrow areas at object boundaries. In SIMOC these regions are processed differently to MOVING and STATIC areas. The detection of uncovered background is a four step process involving the motion vector field and the shape-approximated moving area mask.

The first step in uncovered background detection is similar to moving area detection - for each MOVING pel its motion is undone and the pel pointed to by the motion vector is deemed to belong to the UNCOVERED region provided it is not within the MOVING area. If this is the case, then in step 2 the motion vector is traversed in single pel distances setting the nearest full-pel positions to UNCOVERED provided they do not belong to the MOVING area. In the implementation of uncovered background detection, these steps have been combined in order to avoid a duplication in the scanning of the shape-approximated moving area mask. Unlike moving area detection, however, these two steps are sufficiently complex so as to preclude the processing of a complete frame and, therefore, the sub-routine structure is used.

In the third step, a 3x3 binary median filter is applied for noise filtering purposes. Finally, any pels that belonged to a MOVING area in the previous SMU mask, but are part of the STATIC region in the current moving area mask, are also considered to belong to the UNCOVERED region. This end result is combined with the moving area mask to produce the current ternary SMU mask. The following listing, taken from the VCG implementation, performs these last two tasks.

```
/* Step 4 */
for(y=0;y<dimY;y++)
  for(x=0;x<dimX;x++)
       if((prev_object_mask.hY[y][x] == MOVING) &&
              (approx_mask.hY[y][x] == UNCHANGED))
                     bbu_mask.hY[y][x]=
                            UNCOVERED_BACKGROUND;


/* Add detected uncovered background to seg mask */
for(y=0;y<dimY;y++)
  for(x=0;x<dimX;x++)
       if(bbu_mask.hY[y][x]==UNCOVERED_BACKGROUND) {
              approx_mask.hY[y][x]=UNCOVERED_BACKGROUND;
              pel_count++;
       }
```

Listing 4.5 VCG C implementation of final steps in uncovered background detection.

This implementation includes two conditional statements, which are relatively costly to execute on the TMS320C30. However, it is possible to avoid these conditionals and to combine the two loops if logical operations are used, thus leading to a much more efficient implementation, as shown in Listing 4.6.

```
        LDI     @UNCOVERED,R7       ;R7 = UNCOVERED (0x80)
        RPTB    END_LOOP
        AND3    *AR2++(1),R7,R0     ;if previous SMU mask =
                                    ;MOVING (0xff)then R0 =
                                    ;UNCOVERED else R0 = STATIC
||      STI     R0,*AR3++(1)        ;store last result to
                                    ;current SMU
        OR      *AR0++(1),R0        ;if moving area mask =
                                    ;MOVING then R0 = MOVING,
                                    ;else if previous SMU mask
                                    ;= MOVING and moving area
                                    ;mask = STATIC then R0 =
                                    ;UNCOVERED otherwise R0 =
                                    ;STATIC
        CMPI    *AR1++(1),R7        ;if background mask =
                                    ;UNCOVERED
END_LOOP:
        LDIEQ   R7,R0               ;then R0 = UNCOVERED
```

Listing 4.6 Real-time implementation of final steps in uncovered background detection.

4.3.3 Shape Approximation

The contour of a MOVING object in the moving area mask is approximated as a polygon, represented by a number of vertices connected with straight line segments.

SIMOC classifies two types of MOVING objects - those with a reference in the previous frame (called MC1) and those with no reference (MC2). Before shape approximation it has to be ascertained to which class the current MOVING object belongs. To do this, the object vertices from the previous frame are first of all motion compensated. If a displaced vertex is within a threshold distance from the current object contour, then it is maintained as a vertex for the current object. An object is deemed to be MC1 if there is more than four such maintained vertices, otherwise the object is MC2.

For a MC1 object, the maintained vertices are used as the initial vertices for the polygon approximation. The initial vertices for a MC2 object are determined by first finding the two pels on the object contour that have the maximum separation. If a contour has N pels then the number of possible pairs is given by binomial expansion as:

$$\frac{N!}{2.(N-2)!}$$

For a contour with, say 500 pels, this equates to approximately 125,000 pairs. To reduce the number of distance calculations, those pairs whose perimeter distance is less than the current maximum distance are not evaluated (the perimeter distance between two points on a contour is at a minimum equal to the Euclidian distance). The remaining initial vertices are the contour pels that are the maximum distance left and right of the line joining the first two pels.

Starting with the initial vertices, the polygon approximation is completed by inserting vertices where required to ensure the approximation satisfies a minimum distance criterion. A new vertex is inserted between two adjacent vertices if the distance from any contour pel between to the straight line joining the vertices is above a given threshold.

Shape approximation uses co-ordinate geometry that can involve division or the calculation of a square root. Neither of these are readily implemented on a TMS320C30 in a efficient way. They are usually implemented by iteratively evaluating their Taylor expansions. Both the accuracy and the speed of calculation depend upon the number of iterations. Acceptable accuracy can be obtained at an approximate cost of 30 cycles for a division and 50 cycles for a square root. Fortunately, the calculations in shape approximation are mostly used in comparative tests and the need for division or square roots can be avoided. For example, the distance of a point (j,k) from a line $ax+by+c=0$ is given by:

$$\text{distance} = \frac{|aj + bk + c|}{\sqrt{a^2 + b^2}}$$

However, when comparing two points (h,k) & (m,n) to determine which is the furthest from the line, it is sufficient to evaluate the following equality:

$$|aj + bk + c| \leq |am + bn + c|$$

The remaining encoder functions use the shape approximated moving area mask, which is reconstructed from the list of vertices. Bresenham's Line Drawing Algorithm is used to draw the straight line segments between the vertices, thereby reconstructing the object contour. The contour is then filled using the same filling algorithm as in change detection. This algorithm requires the chain code of the contour so Bresenham's algorithm is modified to produce a chain code as well as drawing the contour pels.

4.3.4 Motion Synthesis

Motion synthesis is the process whereby the current frame is reconstructed from the shape, motion and colour parameters produced by the analysis sections. This reconstructed frame is used in model failure detection (Section 4.3.5) to verify the success of the analysis. To eliminate filtering effects that can occur due to the half-pel motion compensation used in motion synthesis, a special colour memory is maintained at twice the spatial resolution of the previous reconstructed frame. All synthesis operations are based on this memory and not on the previous reconstructed frame.

In motion synthesis the colour memory is updated based on the contents of the SMU mask and the motion vector field. As the SMU mask is only available in QCIF resolution a priority based rule is used which states that a half-pel position is given the same classification as that of its full-pel neighbour with the highest priority. The priority is, in descending order, STATIC, UNCOVERED, MOVING.

If a pel in the colour memory is deemed to be STATIC then no updating is necessary. For a pel in the MOVING area, it is updated by motion compensation - it is given the value of the pel pointed to its motion vector. As with the SMU mask the motion vector field is only QCIF resolution. Therefore, for half-pel positions, bilinear interpolation of the motion vectors at the full-pel positions gives the required displacement. Synthesis of pels in the UNCOVERED area is achieved using a special prediction scheme that adaptively selects between spatial prediction from neighbouring STATIC pels and a temporal prediction from a 'background memory' that contains all background pels revealed in previous frames.

The method used for implementing motion synthesis involves separating the process into four steps - one for the full-pel positions, and one each for the three half-pel positions. This has two key benefits. The priority base rule does not have to be applied to full-pel positions which leads to faster processing of these positions. Also, only full-pel positions in the colour memory are used to reconstruct the current frame. By processing the full-pel positions first, model failure detection can be performed in parallel to motion synthesis of the half-pel positions.

For the half-pel positions, the priority rule can be implemented as an ANDing of the full-pel positions, assuming the following representations in the SMU mask:

STATIC = 0x0
UNCOVERED = 0x80
MOVING = 0xff

4.3.5 Model Failure Detection

In model failure detection the motion synthesised image is compared to the original in order to identify any regions where there is significant luminance differences. The output of model failure detection is a binary mask indicating these model failure (MF) regions. The remaining area is considered to be model compliant (MC). Pels in this mask are set by applying a threshold to the synthesis error between the motion synthesised and current frames. This threshold is determined globally from the synthesis error variance before being applied locally to each pel. This process is given in Section 2.4.1.3.

The production of the binary mask is an iterative process in which the threshold is varied until a desired synthesis error variance for the MC pels is attained. Although STATIC regions are deemed to be MC, they do not contribute to the synthesis error variance calculations. The MF region is initialised to be the MOVING area in the SMU mask. At each iteration of the process, pels are eliminates from the MF area if the synthesis error is less than the threshold for that step. Once this process is completed, the mask is then subject to the same noise reduction and elimination of over-segmentation that is applied in the change detection process.

```
        RPTB   ENDLOOP

        SUBI3  *AR0++,*AR1++,R1     ; subtract pels
        STI    R2,*AR2++            ; store previous result
ENDLOOP:
        MPYI3  R1,R1,R2             ; square the difference
```

Listing 4.7 Main loop of squared error function.

The main component in MF detection is the calculation of the synthesis error variance, which in SIMOC is given by the mean-squared error. This involves the calculation of the squared error for each MOVING pel at each iteration. To avoid the repetition of these calculations in each iteration, they can be performed once before the iterative process begins. Listing 4.7 shows the main loop that is used to generate a new frame containing the squared error between each pair of pels.

```
        RPTB   ENDLOOP

        LDI    *AR0++(1),R0 ; load squared error
        CMPI   R7,R0        ; compare to squared TMF
        LDIGT  0,R0         ; greater than => error does not
        LDIGT  0,R2         ; contribute to variance
        LDIGT  0x0,R1       ; also output is MODEL_FAILURE
        LDILE  0xff,R1      ; o/w output is MODEL_COMPLIANCE
        LDILE  1,R2         ; and contributes to variance
        AND3   *AR1++(1),R4,R3     ; test seg. mask
        LDIZ   0,R0         ; if=0 then pel is MODEL_FAILURE
        LDIZ   0,R2         ; and does not contribute to
        LDIZ   0x0,R1 ;     ; variance


        STI    R1,*AR2++(1) ; store output pel
        ADDI   R0,R6        ; add error to cumulative total
ENDLOOP:
        ADDI   R2,R5        ; add contribution to cumulative
                            ; total of MODEL_COMPLIANCE pels
```

Listing 4.8 Main loop of model failure detection function.

In the implementation of the second part of the model failure detection (Listing 4.8), two tests are made on each pel. A pel is MODEL_COMPLIANCE if the SMU mask == MOVING (0FFh) and the squared error <= TMF (or if mask==STATIC), otherwise the pel is MODEL_FAILURE. Only pels that are MODEL_COMPLIANCE in the model failure mask and MOVING in the SMU mask contribute to the variance calculations. For these pels ERROR is added to TOTAL_E and COUNT is incremented by setting R2 to 1. For MODEL_FAILURE and STATIC pels both ERROR and R2 are set to 0.

The MF mask is shape approximated to produce shape parameters for transmission to the decoder. The colour parameters of the MF region are coded using the spatial vector quantisation method outlined in Section 2.3. Finally, the special colour memory used in motion synthesis must be updated with these new colour values.

4.3.6 The Decoder

Apart from parameter decoding, the SIMOC decoder is equivalent to the synthesis elements of the encoder. The moving area and model failure masks are reconstructed from the shape parameters in the same manner as they are reconstructed in the encoder after shape approximation (see Section 4.3.3). The motion vector field is interpolated from the decoded grid vectors as per motion field interpolation in the encoder (Section 4.3.2.3). With the moving area mask and the motion vector field, the decoder has enough information to perform uncovered background detection. Motion synthesis is then performed using the resultant SMU mask and any decoded uncovered background prediction error. Finally, the model failure areas are updated with the decoded colour information.

Chapter 5

PERFORMANCE ANALYSIS

5.1 Introduction

With the TMS320C30 software implementation of the DCU coder, it is now possible to determine the number of processors that are required for the multi-processor structure described in Section 3.4, to attain real-time execution. Using the TMS320C30 Code Debugger, an estimate of the computational cost of each algorithm part can be obtained without the need for an actual hardware system. Test data for the estimation of costs is provided by the standard test sequences 'Miss America' and 'Claire'. The key to attainable speedup, and hence real-time execution, is the Amdhal Fraction, i.e., that time spent executing parts of the algorithm that cannot be parallelised can be derived from these estimated costs.

From a computational requirements perspective, the various algorithm functions described in Section 4.3 can be divided into three categories:

a) those with fixed costs, irrespective of image content
b) those whose costs vary with image content, but can be determined before without simulation from knowledge of the image parameters, e.g., the size of the change detection mask
c) those whose costs also vary with image content but whose behaviour is sufficiently complex as to prevent estimation without simulation

In the first group, the computation cost can be determined from benchmarking the relevant sections of code, and can often be reduced to a simple 'cycles per pel' figure. For the second group, the costs can only be estimated through a combination of benchmarking and simulation of algorithm behaviour. The costs are usually only dependent on a few parameters and it is possible to conclude heuristic 'rules-of-thumb' for their requirements. In the first two groups, benchmarking can be performed without the need for actual input data. However, in the last group, the costs are dependent on several parameters and requires actual data to perform the benchmarking, provided from simulation using the VCG C implementation.

In Section 5.2 the computation cost for a single frame are estimated based on simulation over 50 frames of the test sequences. This is done first for sequential execution, whose figures are then used as a basis for determining the achievable speedup of a parallel implementation using the proposed parallel algorithm described in Section 3.3. In the final section these results are discussed from which conclusions and recommendations are drawn.

5.2 Execution Time Requirements

5.2.1 Sequential Execution

Although real-time sequential execution will not be possible, it is useful to derive the execution time in order to determine the speed-up factor that the parallel implementation must obtain. Also, for the most part, the cycle counts estimated for the sequential execution of algorithm functions will apply to the parallel execution as well. As the parallel execution will be operating on smaller data sets, it is beneficial to express the computation costs independently of the size of the input data set. Therefore, where possible, all costs are given in terms of 'cycles per pel'. The following sections contain tables that summarise the various costs involved. A complete list of all costs can be found in Appendix B.

5.2.1.1 Change Detection

The first six steps of change detection have fixed computation costs. These are summarised in Table 5.1. For an input data set of QCIF dimension this equates to 1,317,888 cycles. The remaining steps of change detection fall into the third category of functions. Table 5.2 gives a summary of the costs based on simulation of 50 frames of the 'Miss America' and 'Claire' test sequences.

Function	Cycles/pel
Absolute Difference	4
Binarisation	7
Median Value Filter	7
Combining with Previous	4
Dilation	5
Erosion	5

Table 5.1 Computation costs for fixed steps of change detection.

Function	Miss America			Claire		
	min	max	avg	min	max	avg
Calculate Chain Codes	170,300	202,500	181,800	174,600	228,400	202,100
Calculate Area	4,700	9,200	5,200	4,000	13,100	8,600
Redraw Objects	71,800	113,500	101,600	36,800	172,600	146,500

Table 5.2 Summary of cycle counts for non-fixed steps of change detection.

From the simulation results it is difficult to determine any useful relationship between the cost of calculating the chain codes and any properties of the change detection mask. Apart from the number of contours, the chain code algorithm varies with the length of the contours and the number of direction changes the contour makes, i.e., the size of the object and the complexity of its shape. Shape complexity is an abstract parameter that is not readily measurable [18] and, therefore, no estimation of computation cost can be drawn from the object. However, there is a high degree of correlation in the computation costs between successive frames. This reflects the temporal coherency in the change detection process achieved by combining the MOVING areas of the previous mask with the current mask. Therefore, at run-time, a good estimate of the computation cost for calculating the chain codes can be obtained from the previous frame. The calculation of area is completely dependent on the length of the contour and is sufficiently small to ignore. There does, however, appear to be a direct relationship between the cost of redrawing the maintained object(s) and the size of the change detection mask. For both sequences the cost of this process was approximately 8 times the size of the mask. An estimate of the order of magnitude can be determined before execution from the results of the area calculations.

5.2.1.2 Motion Analysis

The motion estimation support functions, given in Section 4.3.2.1, have all got fixed computation costs. These are:

Mean-value filtering 7 cycles/pel

Bilinear Interpolation 14 cycles/pel

Upsampling 14 cycles/pel

Applying the filtering and interpolation to the two input frames and the upsampling of the change detection mask requires 1,419,264 cycles.

The relationship between the change detection mask and the cost of motion estimation has already been established in Section 4.3.2.2, and is given by Equation 4.10 for the optimal, small search window, and no level 3 sub-optimal methods, and by Equation 4.11 for the pel subsampling method. The simulation results are summarised in Table 5.3. Before motion estimation proper, the grid positions within the CHANGED area are separated into two groups at a fixed cost given by the first term in both Equation 4.10 and Equation 4.11. The cost of motion estimation for grid positions whose search window is completely inside the CHANGED area is fixed and is given by the second term in those equations. The remaining grid positions have a variable cost based on the number of CHANGED pels in the search window. However, this cost can be calculated before execution from the third term in the same equations.

Method	Miss America			Claire		
	min	max	avg	min	max	avg
Optimal	3.84	6.29	5.58	1.55	9.2	7.23
Pel Subsampling	2.38	3.89	3.45	0.96	5.71	4.48
Small Search Window	1.52	2.4	2.16	0.63	3.46	2.85
No Level 3	2.61	4.35	3.84	1.05	6.36	5.01

All values have been rounded up to the nearest 10000 cycles

Table 5.3 Summary of computation costs (in millions of cycles) for different motion estimation methods.

Motion field interpolation costs depend on the number of interpolation blocks that need to be evaluated. The cost per block is 3072 cycles (12 cycles/pel). The number of blocks, in turn, is dependant upon the shape and size of the CHANGED area. As in change detection, there is no available measure for this, but again, like change detection, there is a degree of similarity of cost between successive frames. A summary of the computation costs are give in Table 5.4.

Function	Miss America			Claire		
	min	max	avg	min	max	avg
Motion Field Interpolation	301,100	362,500	339,800	159,800	491,600	449,900
First steps of Uncovered Background	255,500	335,600	308,000	184,800	444,700	381,000

Table 5.4 Summary of computation costs for some motion analysis functions.

Moving area detection has a fixed cost of 9 cycles/pel, i.e., 228,096 cycles per frame.

Uncovered background detection, is composed of both fixed and variable parts. The first two steps comprise the variable element, for which the computational cost can be expressed as:

$$\begin{gathered}5*\text{all input pels}\\+\ 14*\text{MOVING pels}\\+\ 102*\text{BBU pels}\end{gathered}$$

Equation 5.1

The first term in Equation 5.1 represents the cost of a complete scan of the input data that is required to determine any MOVING pels. This is performed at a cost of 5 cycles/pel. The second term is the cost of processing any MOVING pels (14 cycles each). The final term gives the cost of traversing the motion vector of any MOVING pel that points outside the MOVING area. A summary of the computation costs determined from simulation is given in Table 5.4. The last steps of uncovered background, the median filter and combining with the previous SMU mask, are fixed at a cost of 7 cycles/pel and 4 cycles/pel respectively.

5.2.1.3 Shape Approximation

As with the elimination of regions in change detection, the execution time of shape approximation is completely dependent on the complexity of shape and size of the object(s). A more complex shape requires more vertices to approximate it, and a larger size means more area to fill when reconstructing the approximated object. Again, a temporal coherency exists in the computation costs of successive frames. Table 5.5 summarises the costs from simulation.

Function	Miss America			Claire		
	min	max	avg	min	max	avg
Shape Approx. of Moving Area	173,500	236,200	187,400	110,700	274,400	243,600

Table 5.5 Summary of computation costs for shape approximation.

Function	Miss America			Claire		
	min	max	avg	min	max	avg
Position a	225,200	294,000	276,300	171,000	379,800	336,300
Position b	248,200	317,700	299,900	194,700	403,000	359,900
Position c	248,900	317,000	299,500	195,400	401,000	358,100
Position d	297,300	366,100	348,600	244,500	449,600	406,800

Table 5.6 Summary of computation costs for motion synthesis.

5.2.1.4 Motion Synthesis

The execution time of motion synthesis is dependent on the SMU mask produced by motion analysis. The total cost, summarised in Table 5.6, is comprised of two factors - the cost of a scan of all pels to determine their status, and the extra processing cost for MOVING and UNCOVERED pels. The following are the costs for each pel position (positions are Figure 2.5):

Full pel position a:

5*all input pels
+ 12*MOVING pels
+ 16*UNCOVERED pels

Half pel position b:

6*all input pels
+ 12*MOVING pels
+ 16*UNCOVERED pels

Half pel position c:

6*all input pels
+ 12*MOVING pels
+ 16*UNCOVERED pels

Half pel position d:

8*all input pels
+ 12*MOVING pels
+ 16*UNCOVERED pels

5.2.1.5 Model Failure Detection

For the most part, the steps in model failure detection have fixed costs. As described in Section 4.3.5, the first step of producing the initial model failure mask is split into two stages. The first stage, calculation of the squared error, has a fixed cost of 4 cycles/pel. Each iteration of the second stage has a fixed cost of 13 cycles/pel. However, the number of iterations is unknown at the time of execution. The average number of iterations is 7.06 for 'Miss America' and 7.86 for 'Claire'. A summary of the computation costs for this stage is given in Table 5.7. Also summarised in Table 5.7 are the costs for elimination of regions. This is the same process as used in change detection. However, the temporal coherency present in change detection is absent from model failure detection, leading to a high degree of fluctuation in the computation cost of successive frames. Not shown in Table 5.7 are the costs for the other steps in the process, namely, binary median filtering, dilation and erosion. The cost of these has already been given in Table 5.1.

Function	Miss America			Claire		
	min	max	avg	min	max	avg
Find TMF	1,977,000	2,636,000	2,327,000	1,977,000	2,966,000	2,589,000
Chain Codes	159,700	220,400	184,000	151,100	184,500	176,500
Calculate Area	1800	12,800	6,400	2,500	7,000	5,100
Redraw Objects	3400	44,300	17,300	5,800	31,400	16,500

Table 5.7 Summary of cycle counts for non-fixed steps of model failure detection.

5.2.1.6 Summary

Table 5.8 summarises the various computation costs for the encoding of a single frame over the simulation of 50 frames. Given that the processing power of a TMS320C30 is 20 million cycles, real-time execution is, as expected, beyond the capability of a single device. The effect of image content on execution time can clearly be seen, not only between the to sequences, but also between the maximum and minimum values for the same sequence. The higher costs for the 'Claire' sequence reflect known problems with the source model used in the SIMOC algorithm [12]. In this sequence there exists subtle, but continuous, variations in the scene illumination that are interpreted by change detection as areas of motion. This leads to a change detection mask (and subsequently a moving area mask) that is larger than it should be. As most of the processing, in particular motion estimation, is concentrated on this area, higher than normal computation costs are incurred. It is likely that the overall computation cost for the 'Claire' sequence represents a worst-case scenario, whereas the figure 'Miss America' is closer to what would be expected under normal circumstances.

A summary of the cycle counts for the encoding of a frame using the sub-optimal motion estimation methods is given in Table 5.9. It is worth noting that due to the inter-relationship between the motion vector field and, for example, the size of the moving area mask (and, therefore, the next change detection mask), that the computation costs for functions other than motion estimation will also change. However, this will be to a much less extent and, in the construction of Table 5.9, the values from the optimal motion estimation implementation have been taken as indicative.

From the summary of the computation costs of the decoder in Table 5.10, it can be seen that a single TMS320C30 is capable of decoding the 'Miss America' sequence at 8 frames per second under average conditions.

Function	Miss America			Claire		
	min	max	avg	min	max	avg
Change Detection	1.57	1.65	1.61	1.54	1.74	1.68
Motion Analysis	6.32	8.91	8.15	3.82	12.06	9.98
MC Shape Approx.	0.18	0.24	0.19	0.11	0.28	0.25
Motion Synthesis	1.02	1.3	1.23	0.81	1.64	1.47
MF Detection	3.19	3.96	3.58	3.18	4.23	3.83
MF Shape Approx.	0.1	0.33	0.19	0.1	0.33	0.2
Total	12.38	16.39	14.95	9.56	20.28	17.41

Table 5.8 Summary of cycle counts (in millions of cycles) for the encoding of a single frame.

Method	Miss America			Claire		
	min	max	avg	min	max	avg
Pel Subsampling	10.92	13.99	12.82	8.97	16.78	14.65
Small Search Window	10.05	12.5	11.53	8.64	14.54	13.03
No Level 3	10.43	13.57	12.49	8.35	16.72	14.47

Table 5.9 Summary of cycle counts (in millions of cycles) for the encoding of a single frame using sub-optimal motion estimation method.

Function	Miss America			Claire		
	min	max	avg	min	max	avg
Reconstruct Moving Area Mask	0.12	0.18	0.13	0.04	0.27	0.14
Interpolate Motion	0.3	0.36	0.34	0.16	0.48	0.44
Uncovered Background Detection	0.54	0.62	0.59	0.47	0.73	0.66
Reconstruct MF Mask	0.03	0.27	0.13	0.04	0.27	0.14
Motion Synthesis	1.02	1.3	1.23	0.81	1.64	1.47
Total	2.01	2.73	2.42	1.52	3.39	2.85

Table 5.10 Summary of cycle counts (in millions of cycles) for the decoding of a single frame.

5.2.2 Parallel Execution

Given a TMS320C30 instruction cycle time of 50ns, the average figure for the computation costs of a single frame of the 'Miss America' sequence (14.95 million cycles) translates to about 0.75ms execution time. For a frame rate of 8Hz, a frame can only have, on average, an encoding execution time of 0.125ms. Therefore, a speedup of approximately 6 must be achieved if real-time encoding is to be possible. The maximum speedup attainable by a parallel implementation over a sequential one has been shown in Section 3.3 to be limited according to Amdahl's Law, i.e.,

$$S_n = \frac{1}{f + \frac{(1-f)}{n}}$$

To determine the number of processors (n) required to achieve the desirable speedup (S_n) it is first necessary to calculate the Amdahl Fraction (f), which is given by:

$$f = \frac{T_s}{T_{Total}}$$

Equation 5.2

i.e., the ratio of the execution time of those parts of the algorithm that cannot be parallelised (T_s) to the overall execution time (T_{Total}). Several possibilities for exploiting parallelism in the algorithm have been identified in Section 3.2. From the complete TMS320C30 software implementation it is now possible to refine these possibilities and to determine a figure for the Amdahl Fraction. In Section 3.2 is was shown that most exploitable parallelism exists at the function level, e.g., within change detection, motion analysis, etc., and that some also existed at the inter-function level.

5.2.2.1 Function Level Parallelism

Within change detection, geometric parallelism can be applied to the steps involving point and neighbourhood operations. For the elimination of regions each object can be processed independently. However, the identification of these objects (the calculation of their chain codes) is a single step procedure involving a scan of the complete mask that cannot be parallelised. Also, for the test sequences, only one object remains after the elimination decision. The process of reconstructing this object from its chain code, by the nature of the filling algorithm, can only be performed on a single processor. The only part of elimination of regions that can be parallelised is the calculation of the objects' areas, but this is a sufficiently small overhead that can be neglected. Therefore, for change detection, the elimination of regions remains a sequential process.

In motion analysis, the support functions for motion estimation are all neighbourhood operations, and therefore geometric parallelism can be applied. A parallel solution for motion estimation has been given in Section 3.3. A geometric division of the input data is inherent in the interpolation of the motion vector field. The field is divided into a number of 16x16 blocks, which can be distributed between the parallel processors. The remaining sections of motion analysis, moving area detection and uncovered background detection can be considered to be point operations and, hence, can be subject to geometric parallelism. Therefore, all the component parts of motion analysis can be parallelised.

Shape approximation shares many properties with the elimination of regions process in change detection, the most important of which is the need for a scan of the complete mask to identify the objects for approximation. In the simulation on the test sequences, only one object needed to be approximated in each frame. Thus the only possible parallelism left would be to divide the initial vertices between the processors to determine if any more vertices needed to be inserted. However, as the bulk of the computation cost of shape approximation lies in the reconstruction of the object, this parallelism would have a relatively negligible effect. With no practicable parallelism, shape approximation must be executed as a sequential process.

All the constituent parts of motion synthesis can be considered to be point operations and geometric parallelism can be applied.

As described in Section 4.3.5, the calculation of the initial model failure mask is an iterative process, with the number of iterations depending on a synthesis error variance of the complete frame. Fortunately, this global parameter can be calculated by combining the synthesis error variance of the regions resulting from sub-division of the frame when using geometric parallelism. The remainder of model failure detection is similar to change detection. The median filer, dilation and erosion can use geometric parallelism. Although more objects are maintained after elimination of regions than in change detection, the low cost of redrawing the objects means that there is no significant gain achievable through processing each object on a separate processor. The same is true for the shape approximation of the model failure regions, so this also remains a sequential process.

5.2.2.2 Inter-function Parallelism

In Section 3.2.1 it was shown that the mean-value filtering and bilinear interpolation required for motion estimation could be performed in parallel to change detection. There is no benefit to be gained from performing these operations in parallel to the parallelisable steps of change detection as all processors will be 100% busy. However, during the elimination of regions, only one processor is in use and, therefore, it is possible, and beneficial, to perform the filtering and interpolation on the idle processors. In doing this, the contribution of elimination of regions to the Amdahl Fraction is reduced, to a limit defined by the outstanding processing to be done on the elimination process when the filtering and interpolation is complete. The remaining execution time can be calculated from Equation 5.3. This will depend on the execution time for the elimination of regions (T_{elim}), mean-value filtering (T_{filter}), bilinear interpolation (T_{inter}), and the number of idle processors (n-1).

$$T'_{elim} = T_{elim} - \frac{T_{filter} + T_{inter}}{n - 1}$$

Equation 5.3

If this figure is negative, then elimination of regions does not contribute to the Amdahl Fraction.

The motion synthesis implementation strategy, described in Section 4.3.4, allows for a degree of inter-function parallelism between motion synthesis and model failure detection. Motion synthesis of the half-pel positions can be performed on the idle processors during the elimination of regions step in model failure detection and the subsequent shape approximation of the maintained regions. Again, the reduction in the contribution to the Amdahl Fraction is bounded by the amount of time taken to complete the elimination/shape approximation process after motion synthesis is finished. This figure is given by:

$$T_{mf\_remain} = T_{mf\_elim} + T_{mf\_shape} - \frac{T_{synth}}{n - 1}$$

Equation 5.4

5.2.2.3 Summary

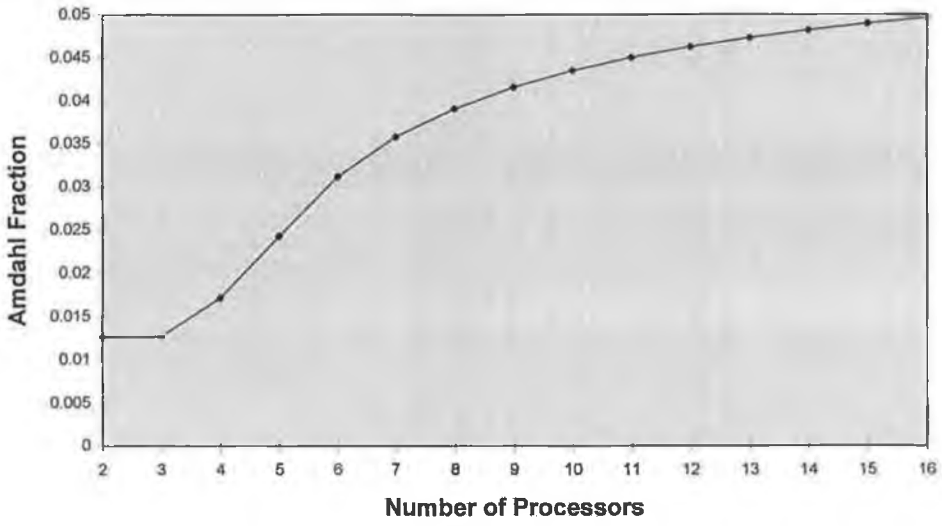
Apart from the two factors represented by Equation 5.3 and Equation 5.4, the serial part of the total execution time also includes the time for the shape approximation of the moving area mask, i.e.:

$$T_s = T'_{elim} + T_{mf\_remain} + T_{shape}$$

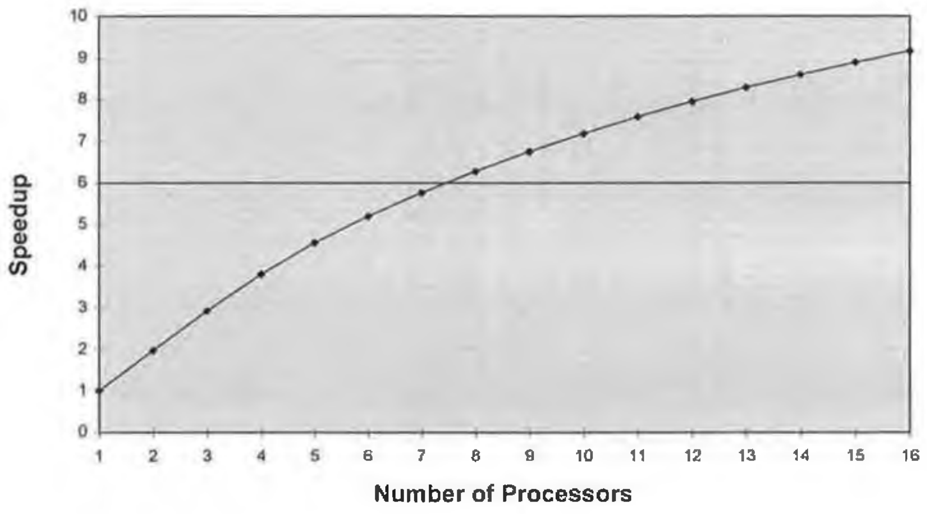
Taking the average figures for the 'Miss America' simulation from Section 5.2.1, the Amdahl Fraction equates to:

$$f = \frac{\max\left(0, \left(0.29 - \frac{1.07}{n-1}\right)\right) + \max\left(0, \left(0.4 - \frac{1}{n-1}\right)\right) + 0.19}{14.95}$$

Figure 5.1(a) shows the variation of the Amdahl Fraction with the number of processors, while Figure 5.1(b) shows the speedup versus the number of processors for the corresponding Amdahl Fraction. The target speedup factor of 6 is achieved with 8 processors. The actual speedup achieved with 8 processors is 6.28, leaving some spare capacity for extra processing.



(a)



(b)

Figure 5.1 (a) Amdahl Fraction vs. number of processors, (b) speedup vs. number of processors

5.3 Discussion

Although real-time execution of the DCU algorithm can be achieved, it is questionable whether the cost of achieving it, i.e., 8 processors for the encoder plus one for the decoder, represents a practical solution. Therefore, there remains a gap in performance to be bridged if the ultimate goal of this research is to be achieved. It is possible that some of the shortfall can be made up through the development of faster processors, such as the planned 25MHz version of the TMS320C30 [19]. An increase in processing power of this order would reduce the processor count by two. Alternatively, the latest device in the TMS320 series - the TMS320C80 multimedia processor [20]- which combines four parallel advanced DSPs with a RISC master processor on a single chip may provide a suitable platform for the proposed parallel algorithm. However, further improvement of the proposed parallel algorithm would, more than likely, be required. From the development of the algorithm and software to date some possibilities for improvements have been identified and are outlined below.

A main factor in all of the synthesis functions and model failure detection is the moving area mask and its derivative, the SMU mask. In the current software implementation of the synthesis functions and model failure detection, each pel in the mask is tested for its status (STATIC, MOVING, UNCOVERED) in order to determine what action needs to be taken. With geometric division of the mask data it is possible that a subdivision will contain only one type of data (the smaller the divisions the more likely this will be the case). If subdivisions that were all STATIC or all MOVING were identified at an early stage, it would obviate the repeated status testing of the pels in the subdivisions. Indeed, for many functions, subdivisions that are all STATIC need not be processed at all. Pel status testing would be retained for those subdivisions that contain a combination of pel types, i.e., those subdivisions that contain the object boundary.

The shape approximation of the moving area mask contributes a value 0.19 to the Amdahl Fraction, of which approximately 65% is spent on reconstruction the mask from the estimated shape parameters. It may be possible to develop a parallel polygon filling algorithm based on the method proposed in [21]. In this method a polygon is divided into a number of trapezoids that can be filled in parallel. Using a parallel filling algorithm like this would reduce the effect of shape approximation on the Amdahl Fraction and, therefore, result in a speedup of the parallel algorithm.

The above two possibilities for improvement do not affect the operation of the DCU algorithm. In addition, it has been shown that, through the use of sub-optimal motion estimation techniques, there exists scope to reduce the complexity of the algorithm by directly altering the operation of the algorithm.

The remaining possibilities for improving the execution time performance of the proposed parallel algorithm all modify the operation of the DCU algorithm.

The DCU algorithm has been optimised to operate at a frame rate of 8Hz, but according to [22] the absolute lowest acceptable frame rate for very low bitrate video coding schemes is 5Hz. Therefore, it is worth investigating if the algorithm can be tuned to optimally work at a lower frame rate than it does at present. Even a small reduction to 7Hz would decrease the required speedup in Section 5.2.2 to 5.25, a figure that is almost achievable with 6 processors.

As already shown in Section 4.3.2.2 and subsequently Section 5.2.1.2, sub-optimal motion estimation methods can provide a considerable reduction in the computational complexity of the DCU algorithm while maintaining an overall algorithm performance that is close to the optimum, especially in the pel subsampling case. The results would indicate that further examination into the use of sub-optimal motion estimation methods is worthwhile. By studying the effects on the remainder of the algorithm of the changes introduced in the motion vector field, it may be possible to alter the other elements in order to achieve better overall performance.

Another possible area in which the processing cost of the algorithm may be reduced is in the open loop control. Currently, the generated bitstream is controlled by a quality criterion that requires the model failure regions to have a PSNR of 31dB. Reducing this value, i.e., by reducing the target synthesis error variance in model failure detection, would decrease the number of iterations in the calculation of the motion failure threshold and, possibly, the size of the model failure area. The overall effect of this would be to lessen the cost of model failure detection and shape approximation of the model failure area.

At this stage two cycles of the iterative design method described in Section 3.1 have been completed. Starting with the basic description that the hardware system must be parallel and programmable, the first cycle refined this to a multiprocessor architecture. This was as a result of an analysis of the DCU algorithm that realised a parallel algorithm structure which could only be supported by such an architecture. In the second cycle software was developed for the target processor, the TMS320C30. The cycle was completed by concluding that 8 processors would be required in the multiprocessor system if real-time encoding is to be achieved.

Chapter 6

CONCLUSION

This thesis presented the results of research into the development of a real-time object-based analysis-synthesis coder. The thesis began with a general introduction to second generation video coding and then, specifically, the DCU algorithm which is an implementation of the COST 211ter SIMOC Reference Model. An implementation-independent complexity analysis of the DCU algorithm indicated that its processing power requirements were beyond the capability of a single processor. Therefore, a parallel implementation of the algorithm was sought.

In Chapter 3 an iterative design model was proposed with the aim of maximising the performance of the parallel algorithm software within any hardware constraints. Having identified that a parallel system was required, the DCU algorithm was examined for any inherent parallelism. The outcome of this was a proposed solution that combined parallel elements with parts of the algorithm that could only be executed sequentially. This exercise also identified a significant problem which was the dependence of the algorithm execution time on scene content, especially the amount of motion. The proposed solution was based on 'excess parallelism' where the input data is divided into a greater number of geometric subdivisions than the number of processors in the system. Another conclusion from this chapter was that the only type of hardware architecture onto which the parallel algorithm could be mapped was a multiprocessor system.

In Chapter 4 the largest part of the research was presented, namely the writing and optimising of the algorithm software for the chosen target processor, the

TMS320C30. Alterations to the DCU algorithm were proposed that reduced the computational complexity. However, these changes resulted in a degradation in the output quality of the algorithm. From the software it was possible, in Chapter 5, to determine relatively accurately the processing power requirements of the algorithm. The conclusion of this was that a multiprocessor system with 8 processors would be required for real-time encoding. This could be reduced by 2 if the planned 25MHz version of the TMS320C30 was used. Real-time decoding at 8Hz can be achieved with a single TMS320C30.

At this stage two cycles of the iterative design model had been completed and no further research work was carried out. The main conclusion drawn from the work was that, although it was shown that real-time operation was possible, it was felt that the number of processors required made the hardware system somewhat impractical. A number of recommendations were given, based on the knowledge of the algorithm gained through developing the software. These recommendations could provide the start to another cycle of the iterative design model. The recommendations can be divided into two categories, those which do not affect the operation of the algorithm, and those that change the parameters of the algorithm and consequently, affect its output .

In the first category the following two recommendations were made:

- The introduction of a classification scheme for the geometric subdivisions of the moving area mask and SMU mask. This would reduce the costly repeated status testing of pels in those masks during synthesis and model failure detection.

- Development of a parallel polygon filling algorithm. This would reduce the contribution of shape approximation to the Amdahl fraction of the proposed parallel algorithm. The Amdahl fraction is the limitation factor on the overall achievable speedup.

Falling into the second category are the following:

- A reduction in the frame rate. The DCU algorithm has been designed to work optimally at 8Hz, and reducing this rate may require re-optimising the algorithm performance.

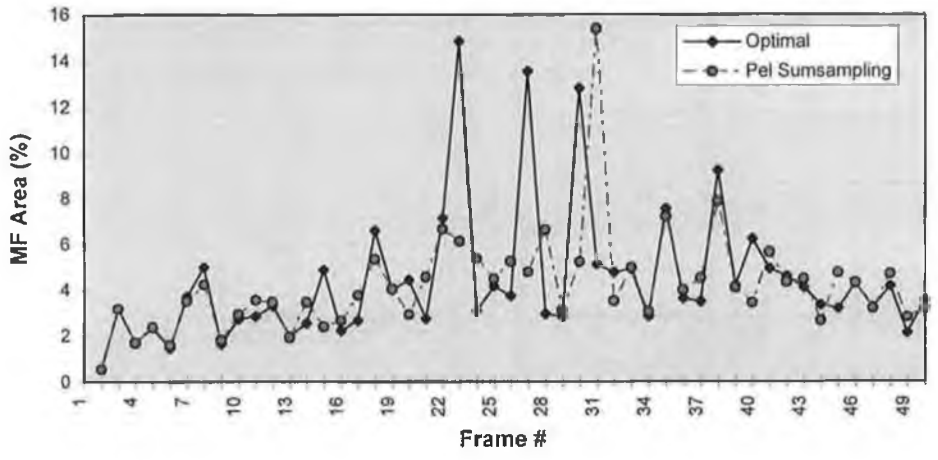
- The use of sub-optimal motion estimation methods. Motion estimation requires the most processing power and large reductions in this requirement can be gained by using a sub-optimal method. The motion vectors produced by motion estimation are intrinsic to the operation of the algorithm. Therefore, the effect on the motion vector field introduced by a sub-optimal method, and the subsequent effects this has on the rest of the algorithm need to be examined.

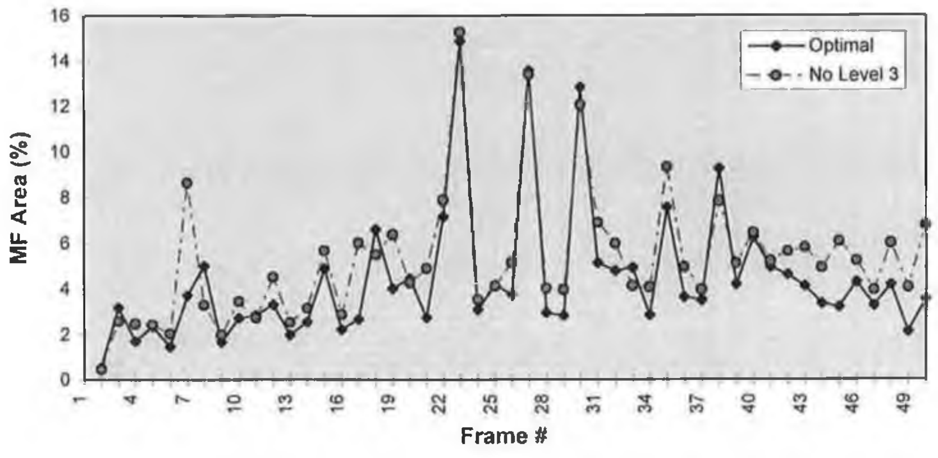
- A reduction of the quality criterion. This would lead to a decrease in the complexity of model failure detection and a reduction in the size of the model failure area.

The ultimate goal of this research was to determine whether the complexity of the coding techniques used in this type of scheme would prevent real-time operation being achieved, and to eventually provide a platform on which the techniques could be developed and tested. The first part of this goal has been accomplished and the foundation has been laid for future work. In the past, the development of hardware systems for video coding has been done as a reaction to the standardisation of algorithms. Only now are systems for MPEG-1 and H.261 commonplace even though the standards are several years old. Similarly, systems that implement MPEG-2 and H.263 are only beginning to reach the market. By developing hardware in parallel, or in this case at an earlier stage, to the standardisation process, not only will systems be available at an early stage after standardisation, but valuable information can also be gained that can influence the process. Although the SIMOC algorithm will not become a standard, several of techniques used are being applied to the development of the MPEG-4 video coding standard. It is possible that the iterative design model used in this research could be applied to the development of a real-time MPEG-4 video coder.

Appendix A

PERFORMANCE COMPARISON OF SUB-OPTIMAL MOTION ESTIMATION METHODS





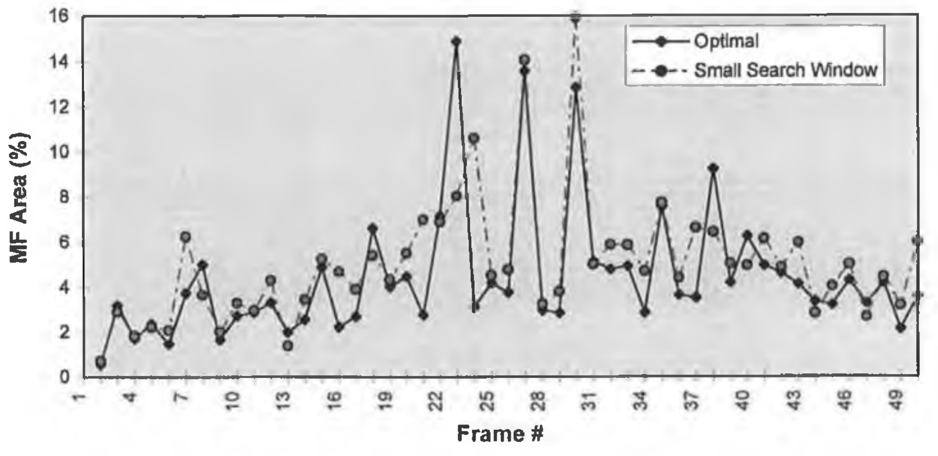
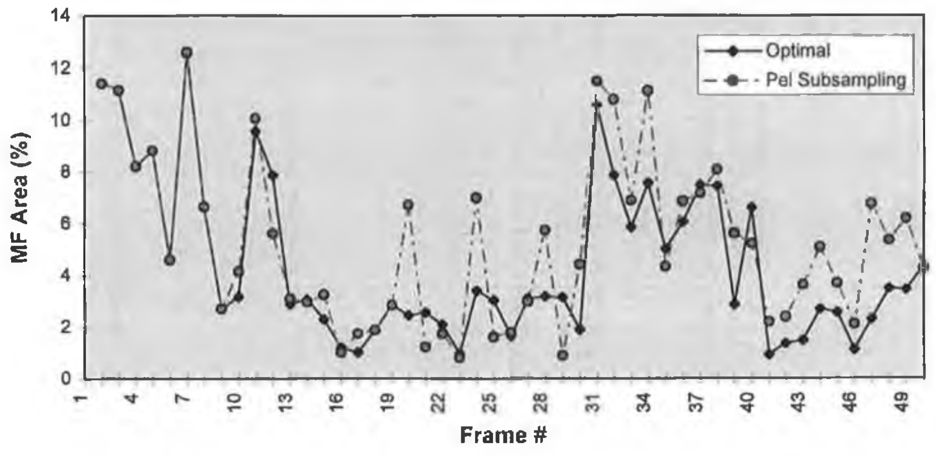
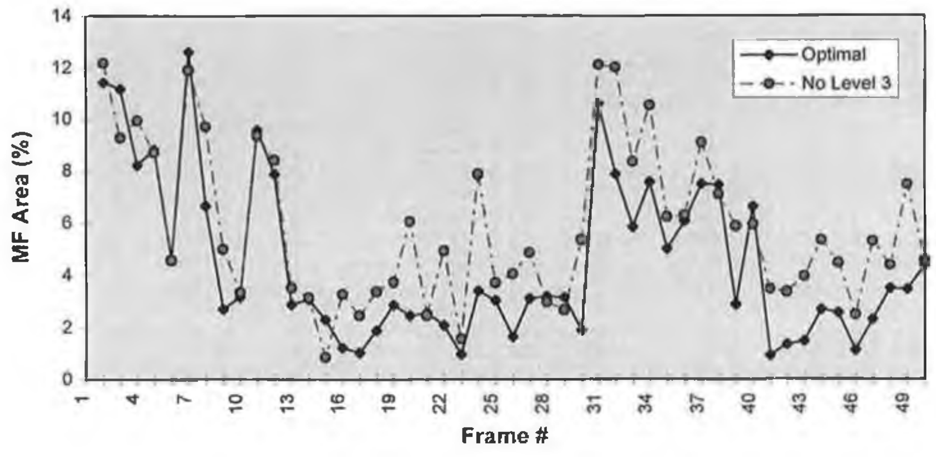


Figure A.1 Comparison of Model Failure Area sizes for 'Miss America' sequence.





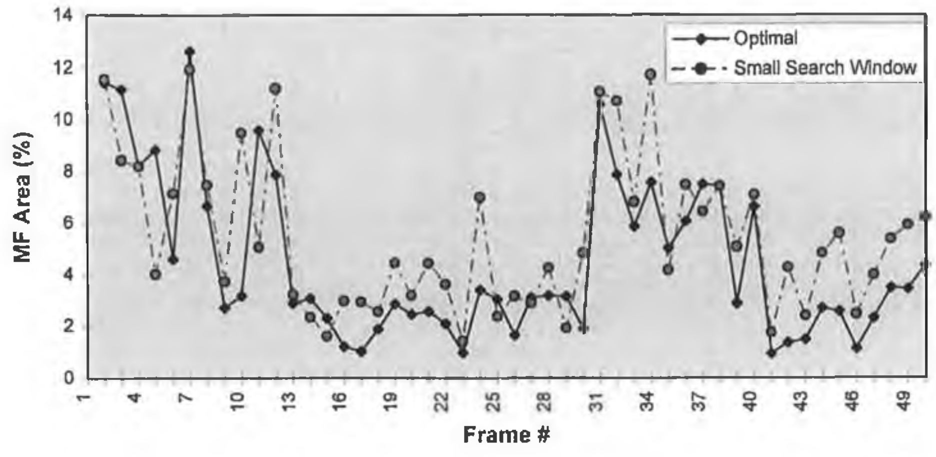
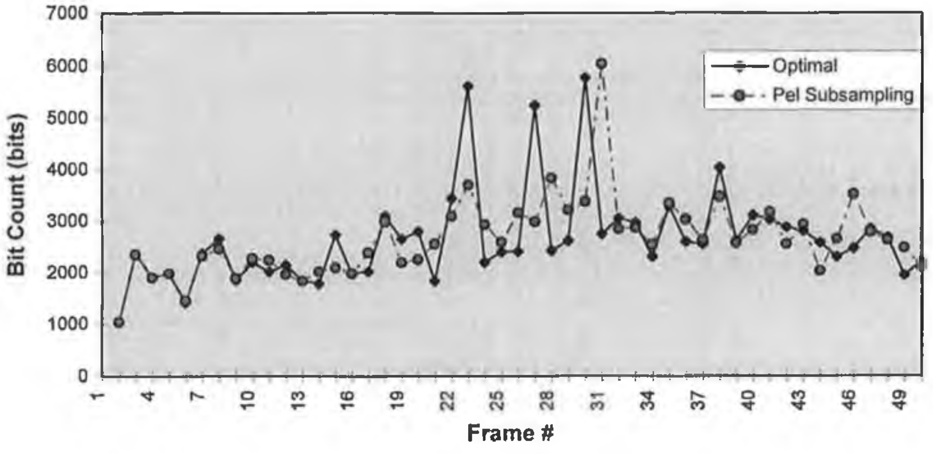
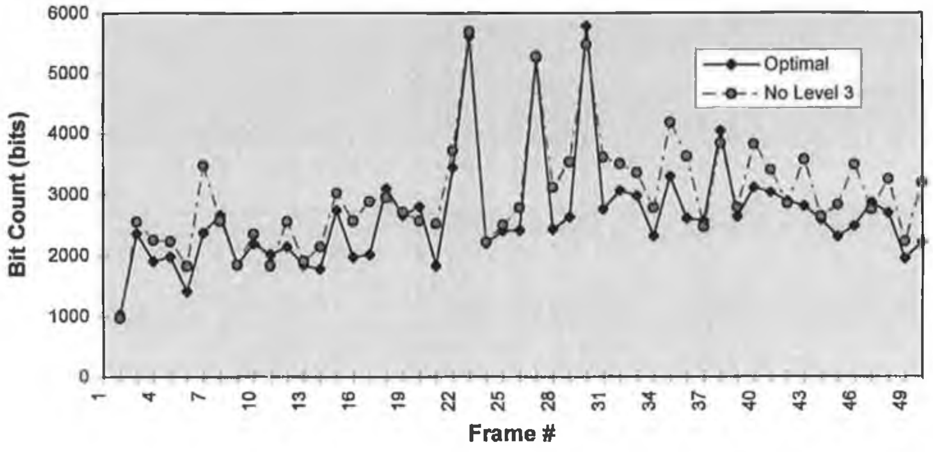


Figure A.2 Comparison of Model Failure Area sizes for 'Claire' sequence.





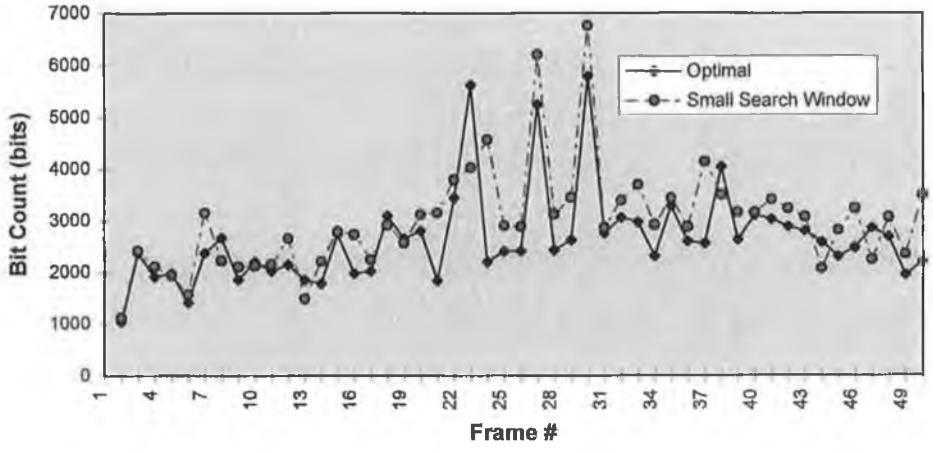
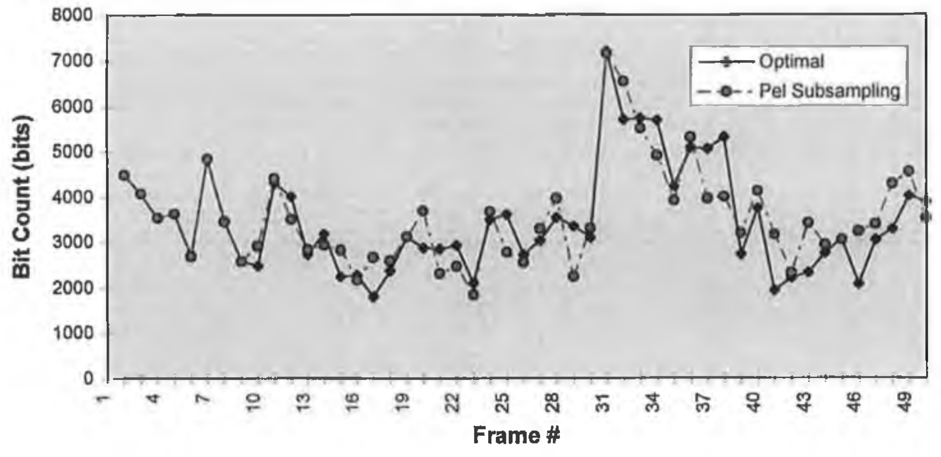
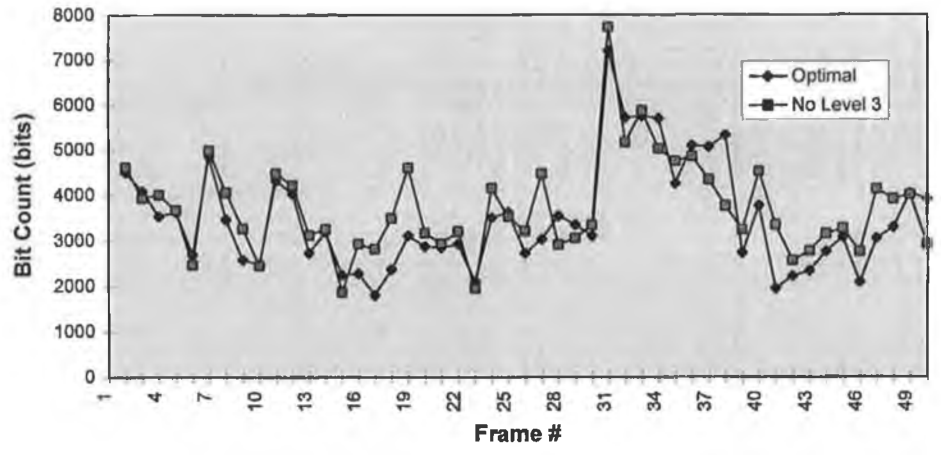


Figure A.3 Comparison of generated bit counts for 'Miss America' sequence.





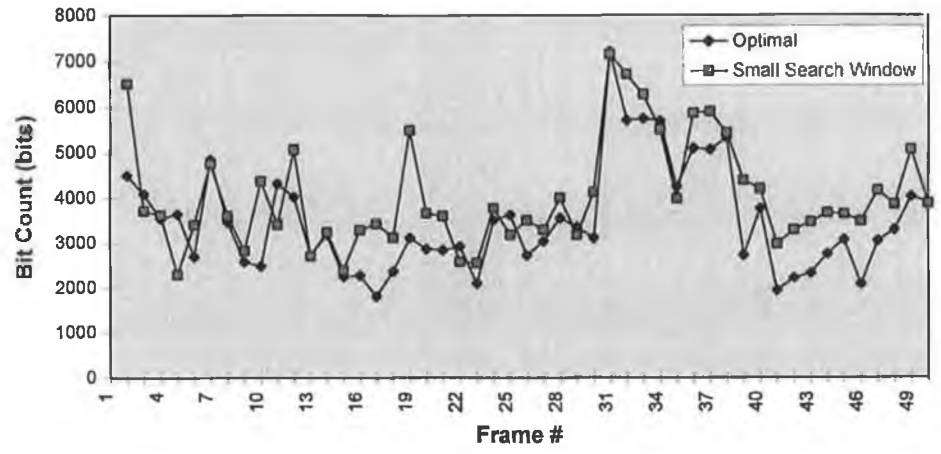
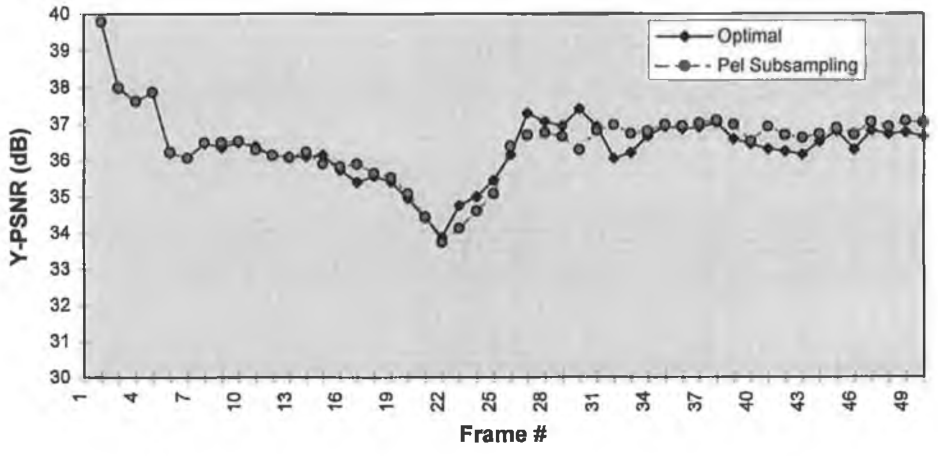
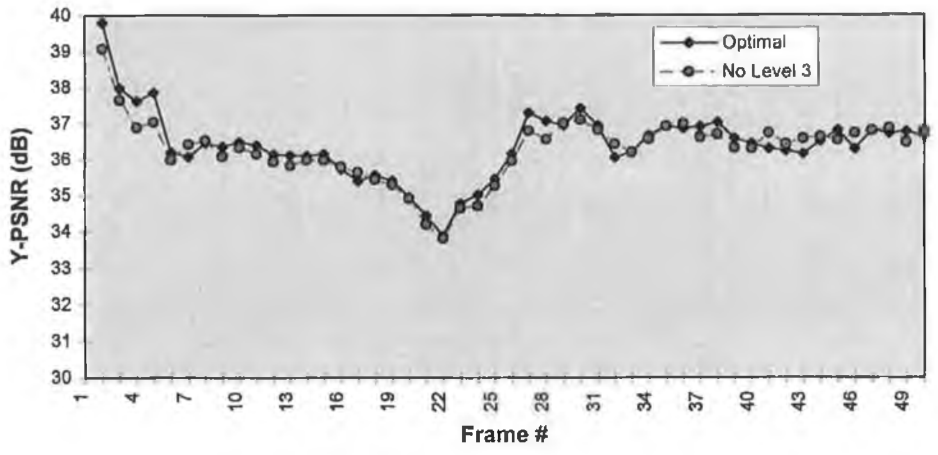


Figure A.4 Comparison of generated bit counts for 'Claire' sequence.





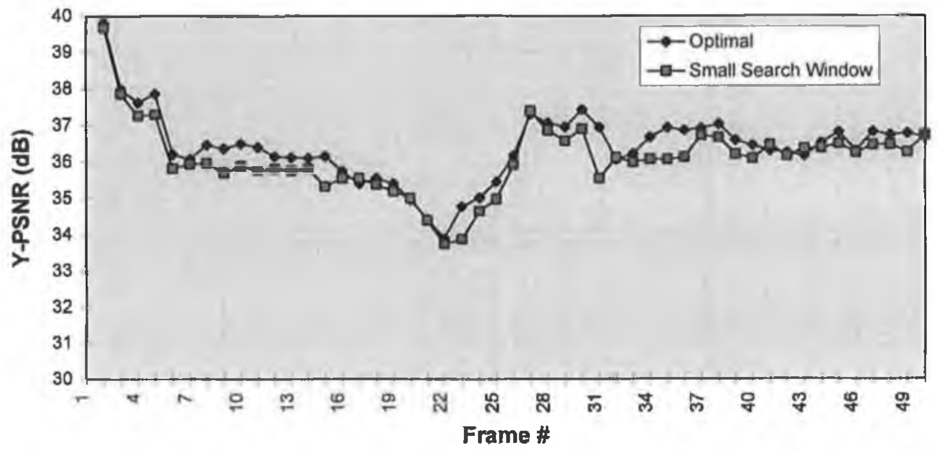
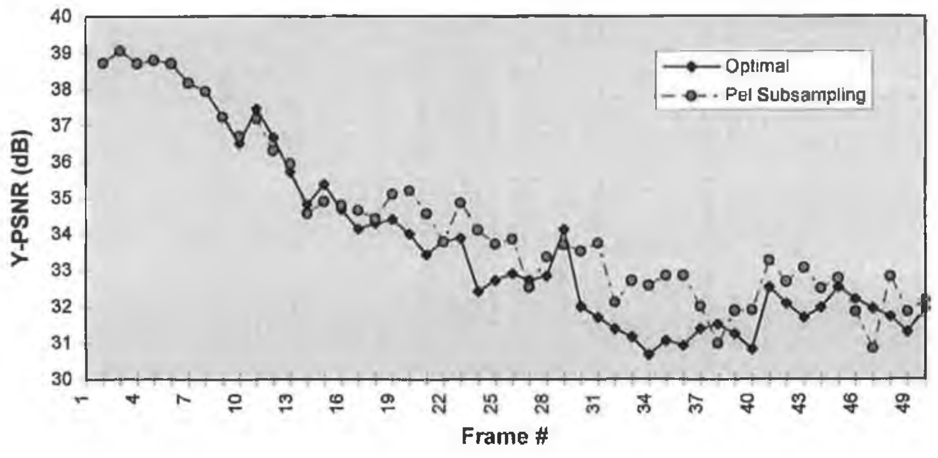
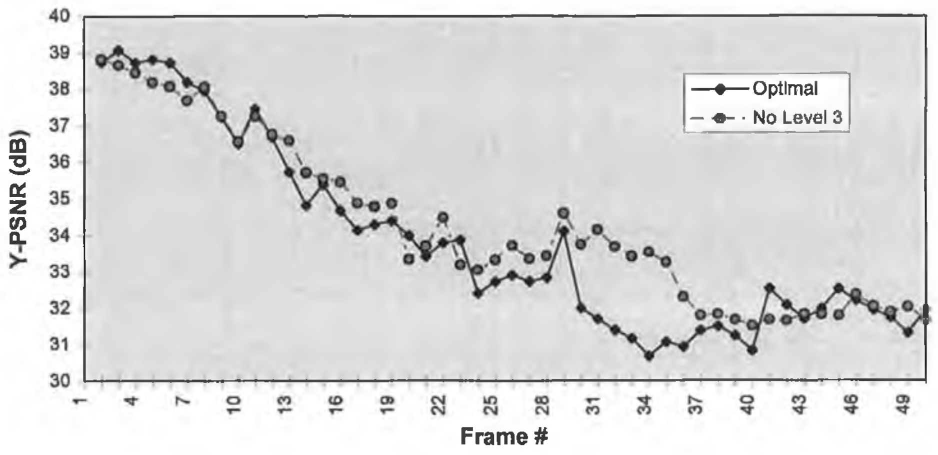


Figure A.5 Comparison of luminance PSNRs for 'Miss America' sequence.





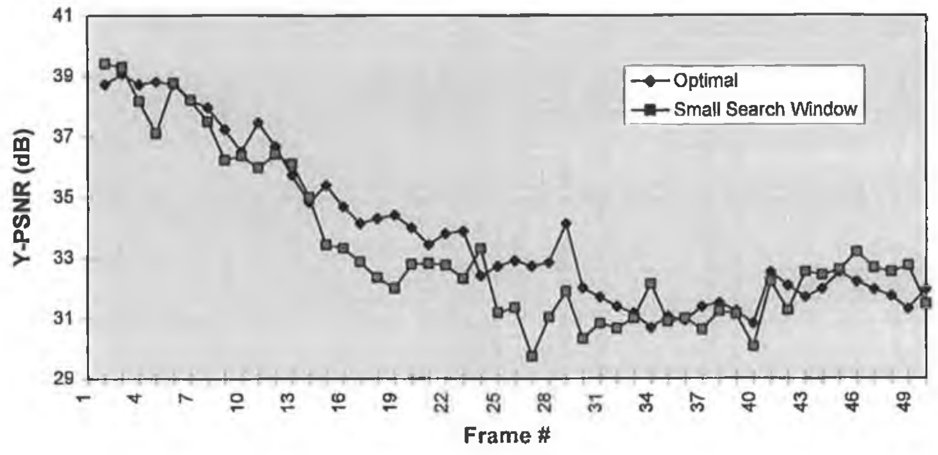


Figure A.6 Comparison of luminance PSNRs for 'Claire' sequence.

Appendix B

CYCLE COUNTS OF INDIVIDUAL FUNCTIONS FROM SIMULATION

Frame	Change D	Motion An	MC Shape	Motion Sy	MF Detect	MF Shape	Total
2	1601296	6320805	236135	1019460	3839557	165056	13182309
3	1598896	6936723	173461	1104228	3856972	172270	13842550
4	1599268	7116013	173823	1116060	3851768	218317	14075249
5	1596745	7328423	177666	1144068	3854591	148569	14250062
6	1601281	7599435	176608	1162492	3854810	237989	14632615
7	1597714	7621594	178107	1176188	3541528	236304	14351435
8	1598280	7626566	178297	1179996	3547454	246423	14377016
9	1598877	7634904	177632	1179396	3851746	204597	14647152
10	1598338	7685174	178087	1176972	3545715	187534	14371820
11	1598555	7711638	189933	1181796	3540097	137129	14359148
12	1598737	7741900	177675	1183208	3540686	215038	14457244
13	1598060	7745776	178267	1183380	3535571	147960	14389014
14	1600988	7773710	177740	1188540	3537544	214655	14493177
15	1602578	7873776	178030	1209148	3549725	149187	14562444
16	1597813	7747293	178231	1207812	3528850	96164	14356163
17	1589081	7827340	178373	1207428	3531308	196164	14529694
18	1598180	7832579	178588	1205916	3556989	254286	14626538
19	1598722	7830361	178797	1204308	3551283	171191	14534662
20	1599840	7926708	178597	1205892	3557353	145430	14613820
21	1600756	7896712	179488	1203500	3540433	228227	14649116
22	1600989	7900043	178392	1217784	3568215	280645	14746068
23	1602344	7974821	179065	1219124	3599233	297486	14872073
24	1601531	7999723	179674	1213764	3215044	231444	14441180
25	1600548	8047640	178876	1211220	3547974	249419	14835677
26	1602676	8201275	180243	1220588	3554929	95495	14855206
27	1607870	8119666	181240	1232260	3277042	164892	14582970
28	1613411	8760100	191223	1277884	3551500	162690	15556808
29	1613657	8749094	193059	1279628	3563221	172698	15571357
30	1616270	8746568	195917	1290064	3293293	134377	15276489
31	1616578	8843637	196752	1287900	3559745	255733	15760345
32	1617030	8901260	197072	1290732	3568417	158017	15732528
33	1615385	8819978	197348	1294492	3565642	145896	15638741
34	1616482	8763479	196098	1289756	3552006	90750	15508571
35	1614175	8753897	195307	1276252	3560230	240384	15640245
36	1613287	8747578	194580	1271932	3556057	207881	15591315
37	1612611	8682978	194238	1270708	3557819	157318	15475672
38	1612524	8631653	193813	1271220	3568105	109079	15386394
39	1611673	8577249	194143	1268028	3558668	320678	15530439
40	1611734	8546974	192531	1260952	3566369	241730	15420290
41	1611957	8493363	193031	1261364	3564473	286808	15410996
42	1611390	8559576	193439	1257756	3569998	192677	15384836
43	1613294	8564581	193737	1263604	3569511	148477	15353204
44	1614219	8567798	194432	1262428	3556899	161735	15357511
45	1614254	8575486	195102	1258164	3556108	183157	15382271
46	1613616	8542527	194457	1259172	3560123	131022	15300917
47	1614377	8608772	195774	1262268	3558460	127889	15367540
48	1616616	8707571	197500	1259604	3555820	98489	15435600
49	1615932	8611277	195086	1257660	3542220	140421	15362596
50	1615732	8601167	194889	1261716	3553652	205215	15432371
Min.	1564520	6320805	173461	1019460	3180613	90750	12349609
Max.	1642980	8917935	236135	1294492	3952237	320678	16364457
Average	1606452	8150554	187358.2	1224241	3573158	187040.7	14928805

Table B.1 Cycle count of main functions for 'Miss America' sequence.

Frame	Abs. Differ	Binarisatio	Median Va	Combining	Dilation	Erosion	Eliminatio	Total
2	101376	177408	177408	101376	380160	380160	283408	1601296
3	101376	177408	177408	101376	380160	380160	281008	1598896
4	101376	177408	177408	101376	380160	380160	281380	1599268
5	101376	177408	177408	101376	380160	380160	278857	1596745
6	101376	177408	177408	101376	380160	380160	283393	1601281
7	101376	177408	177408	101376	380160	380160	279826	1597714
8	101376	177408	177408	101376	380160	380160	280392	1598280
9	101376	177408	177408	101376	380160	380160	280989	1598877
10	101376	177408	177408	101376	380160	380160	280450	1598338
11	101376	177408	177408	101376	380160	380160	280667	1598555
12	101376	177408	177408	101376	380160	380160	280849	1598737
13	101376	177408	177408	101376	380160	380160	280172	1598060
14	101376	177408	177408	101376	380160	380160	283100	1600988
15	101376	177408	177408	101376	380160	380160	284690	1602578
16	101376	177408	177408	101376	380160	380160	279925	1597813
17	101376	177408	177408	101376	380160	380160	271193	1589081
18	101376	177408	177408	101376	380160	380160	280292	1598180
19	101376	177408	177408	101376	380160	380160	280834	1598722
20	101376	177408	177408	101376	380160	380160	281952	1599840
21	101376	177408	177408	101376	380160	380160	282868	1600756
22	101376	177408	177408	101376	380160	380160	283101	1600989
23	101376	177408	177408	101376	380160	380160	284456	1602344
24	101376	177408	177408	101376	380160	380160	283643	1601531
25	101376	177408	177408	101376	380160	380160	282660	1600548
26	101376	177408	177408	101376	380160	380160	284788	1602676
27	101376	177408	177408	101376	380160	380160	289982	1607870
28	101376	177408	177408	101376	380160	380160	295523	1613411
29	101376	177408	177408	101376	380160	380160	295769	1613657
30	101376	177408	177408	101376	380160	380160	298382	1616270
31	101376	177408	177408	101376	380160	380160	298690	1616578
32	101376	177408	177408	101376	380160	380160	299142	1617030
33	101376	177408	177408	101376	380160	380160	297497	1615385
34	101376	177408	177408	101376	380160	380160	298594	1616482
35	101376	177408	177408	101376	380160	380160	296287	1614175
36	101376	177408	177408	101376	380160	380160	295399	1613287
37	101376	177408	177408	101376	380160	380160	294723	1612611
38	101376	177408	177408	101376	380160	380160	294636	1612524
39	101376	177408	177408	101376	380160	380160	293785	1611673
40	101376	177408	177408	101376	380160	380160	293846	1611734
41	101376	177408	177408	101376	380160	380160	294069	1611957
42	101376	177408	177408	101376	380160	380160	293502	1611390
43	101376	177408	177408	101376	380160	380160	295406	1613294
44	101376	177408	177408	101376	380160	380160	296331	1614219
45	101376	177408	177408	101376	380160	380160	296366	1614254
46	101376	177408	177408	101376	380160	380160	295728	1613616
47	101376	177408	177408	101376	380160	380160	296489	1614377
48	101376	177408	177408	101376	380160	380160	298728	1616616
49	101376	177408	177408	101376	380160	380160	298044	1615932
50	101376	177408	177408	101376	380160	380160	297844	1615732
Min.	101376	177408	177408	101376	380160	380160	246632	1564520
Max.	101376	177408	177408	101376	380160	380160	325092	1642980
Average	101376	177408	177408	101376	380160	380160	288564.4	1606452

Table B.2 Cycle counts of change detection functions for 'Miss America' Sequence.

Frame	Chain Cod	Area	Fill	Total
2	202499	9150	71759	283408
3	191669	6872	82467	281008
4	188220	6393	86767	281380
5	185663	5846	87348	278857
6	185454	5837	92102	283393
7	181830	5154	92842	279826
8	181936	5172	93284	280392
9	182271	5270	93448	280989
10	181830	5254	93366	280450
11	181794	5189	93684	280667
12	181688	5171	93990	280849
13	181394	5082	93696	280172
14	182891	5484	94725	283100
15	182327	5302	97061	284690
16	179286	4722	95917	279925
17	170236	4901	96056	271193
18	179339	4731	96222	280292
19	179604	4776	96454	280834
20	180675	4637	96640	281952
21	180739	5009	97120	282868
22	180616	4904	97581	283101
23	180840	5027	98589	284456
24	180469	4964	98210	283643
25	179816	4812	98032	282660
26	180187	4875	99726	284788
27	182059	5234	102689	289982
28	182714	5263	107546	295523
29	179905	4786	111078	295769
30	180741	4930	112711	298382
31	180541	4894	113255	298690
32	180832	4867	113443	299142
33	180170	4831	112496	297497
34	181093	5031	112470	298594
35	180435	4876	110976	296287
36	180329	4858	110212	295399
37	180276	4849	109598	294723
38	180170	4831	109635	294636
39	180117	4822	108846	293785
40	179958	4795	109093	293846
41	180211	4881	108977	294069
42	180273	4840	108389	293502
43	181177	5002	109227	295406
44	181442	5047	109842	296331
45	181495	5056	109815	296366
46	180975	5103	109650	295728
47	181548	5065	109876	296489
48	182755	5272	110701	298728
49	182649	5254	110141	298044
50	182449	5218	110177	297844
Min.	170236	4637	71759	246632
Max.	202499	9150	113443	325092
Average	181787.5	5186.51	101590.4	288564.4

Table B.3 Cycle counts from elimination of regions for 'Miss America' sequence.

Frame	Mean Valu	Bilinear Int	Upsamplin	Motion Est	Motion Fie	Moving Ar	Uncovere	Total
2	354816	709632	354816	3838185	301056	228096	534204	6320805
3	354816	709632	354816	4417065	319488	228096	552810	6936723
4	354816	709632	354816	4605597	313344	228096	549712	7116013
5	354816	709632	354816	4808745	313344	228096	558974	7328423
6	354816	709632	354816	5064633	319488	228096	567954	7599435
7	354816	709632	354816	5076486	325632	228096	572116	7621594
8	354816	709632	354816	5086566	325632	228096	567008	7626566
9	354816	709632	354816	5094486	325632	228096	567426	7634904
10	354816	709632	354816	5144598	325632	228096	567584	7685174
11	354816	709632	354816	5171148	325632	228096	567498	7711638
12	354816	709632	354816	5193378	325632	228096	575530	7741900
13	354816	709632	354816	5202738	325632	228096	570046	7745776
14	354816	709632	354816	5227344	325632	228096	573374	7773710
15	354816	709632	354816	5310576	331776	228096	584064	7873776
16	354816	709632	354816	5198355	325632	228096	575946	7747293
17	354816	709632	354816	5272956	331776	228096	575248	7827340
18	354816	709632	354816	5276601	331776	228096	576842	7832579
19	354816	709632	354816	5281731	325632	228096	575638	7830361
20	354816	709632	354816	5376888	325632	228096	576828	7926708
21	354816	709632	354816	5345118	325632	228096	578602	7896712
22	354816	709632	354816	5329809	331776	228096	591098	7900043
23	354816	709632	354816	5407335	331776	228096	588350	7974821
24	354816	709632	354816	5415777	350208	228096	586378	7999723
25	354816	709632	354816	5466294	344064	228096	589922	8047640
26	354816	709632	354816	5619537	344064	228096	590314	8201275
27	354816	709632	354816	5529978	344064	228096	598264	8119666
28	354816	709632	354816	6146460	356352	228096	609928	8760100
29	354816	709632	354816	6128604	362496	228096	610634	8749094
30	354816	709632	354816	6132096	362496	228096	604616	8746568
31	354816	709632	354816	6229035	362496	228096	604746	8843637
32	354816	709632	354816	6280074	362496	228096	611330	8901260
33	354816	709632	354816	6195816	362496	228096	614306	8819978
34	354816	709632	354816	6146811	356352	228096	612956	8763479
35	354816	709632	354816	6144903	356352	228096	605282	8753897
36	354816	709632	354816	6137208	356352	228096	606658	8747578
37	354816	709632	354816	6080706	350208	228096	604704	8682978
38	354816	709632	354816	6031017	350208	228096	603068	8631653
39	354816	709632	354816	5986053	344064	228096	599772	8577249
40	354816	709632	354816	5949918	344064	228096	605632	8546974
41	354816	709632	354816	5904639	344064	228096	597300	8493363
42	354816	709632	354816	5976252	344064	228096	591900	8559576
43	354816	709632	354816	5964687	356352	228096	596182	8564581
44	354816	709632	354816	5970168	356352	228096	593918	8567798
45	354816	709632	354816	5986350	344064	228096	597712	8575486
46	354816	709632	354816	5957253	344064	228096	593850	8542527
47	354816	709632	354816	6025932	344064	228096	591416	8608772
48	354816	709632	354816	6107085	356352	228096	596774	8707571
49	354816	709632	354816	6015537	356352	228096	592028	8611277
50	354816	709632	354816	5995647	362496	228096	595664	8601167
Min.	354816	709632	354816	3838185	301056	228096	534204	6320805
Max.	354816	709632	354816	6293773	362496	228096	614306	8917935
Average	354816	709632	354816	5576616	339800.8	228096	586777.7	8150554

Table B.4 Cycle counts of motion analysis functions for 'Miss America' sequence.

Frame	Level 1	Level 2	Level 3	Total
2	2292390	311193	1234602	3838185
3	2666120	351584	1399361	4417065
4	2790592	364684	1450321	4605597
5	2936618	371576	1500551	4808745
6	3077326	394486	1592821	5064633
7	3112966	393433	1570087	5076486
8	3121066	393883	1571617	5086566
9	3125296	394603	1574587	5094486
10	3148508	400037	1596053	5144598
11	3162818	402332	1605998	5171148
12	3171998	404942	1616438	5193378
13	3165968	408020	1628750	5202738
14	3174284	411152	1641908	5227344
15	3272196	408144	1630236	5310576
16	3194570	401018	1602767	5198355
17	3251136	404814	1617006	5272956
18	3253836	404994	1617771	5276601
19	3256986	405399	1619346	5281731
20	3267328	422299	1687261	5376888
21	3276328	414163	1654627	5345118
22	3253504	415648	1660657	5329809
23	3284150	425240	1697945	5407335
24	3295130	421037	1699610	5415777
25	3334244	427175	1704875	5466294
26	3432750	434706	1752081	5619537
27	3363378	433968	1732632	5529978
28	3764530	476863	1905067	6146460
29	3749924	476123	1902557	6128604
30	3757844	471695	1902557	6132096
31	3798598	482905	1947532	6229035
32	3862434	483807	1933833	6280074
33	3786186	482124	1927506	6195816
34	3764946	476346	1905519	6146811
35	3734436	478659	1931808	6144903
36	3716436	480882	1939890	6137208
37	3680084	476591	1924031	6080706
38	3663712	473785	1893520	6031017
39	3623698	472840	1889515	5986053
40	3630178	464029	1855711	5949918
41	3575944	465892	1862803	5904639
42	3651472	465217	1859563	5976252
43	3620450	465503	1878734	5964687
44	3597086	471281	1901801	5970168
45	3633860	470831	1881659	5986350
46	3592136	469706	1895411	5957253
47	3640512	477615	1907805	6025932
48	3682506	478155	1946424	6107085
49	3630448	469960	1915129	6015537
50	3631070	469571	1895006	5995647
Min.	2292390	311193	1234602	3838185
Max.	3862434	483807	1947532	6293773
Average	3397346	435120.6	1744149	5576616

Table B.5 Cycle counts of each level in motion estimation for 'Miss America' sequence.

Frame	First Steps	Median Fil	Combine	Total
2	255420	177408	101376	534204
3	274026	177408	101376	552810
4	270928	177408	101376	549712
5	280190	177408	101376	558974
6	289170	177408	101376	567954
7	293332	177408	101376	572116
8	288224	177408	101376	567008
9	288642	177408	101376	567426
10	288800	177408	101376	567584
11	288714	177408	101376	567498
12	296746	177408	101376	575530
13	291262	177408	101376	570046
14	294590	177408	101376	573374
15	305280	177408	101376	584064
16	297162	177408	101376	575946
17	296464	177408	101376	575248
18	298058	177408	101376	576842
19	296854	177408	101376	575638
20	298044	177408	101376	576828
21	299818	177408	101376	578602
22	312314	177408	101376	591098
23	309566	177408	101376	588350
24	307594	177408	101376	586378
25	311138	177408	101376	589922
26	311530	177408	101376	590314
27	319480	177408	101376	598264
28	331144	177408	101376	609928
29	331850	177408	101376	610634
30	325832	177408	101376	604616
31	325962	177408	101376	604746
32	332546	177408	101376	611330
33	335522	177408	101376	614306
34	334172	177408	101376	612956
35	326498	177408	101376	605282
36	327874	177408	101376	606658
37	325920	177408	101376	604704
38	324284	177408	101376	603068
39	320988	177408	101376	599772
40	326848	177408	101376	605632
41	318516	177408	101376	597300
42	313116	177408	101376	591900
43	317398	177408	101376	596182
44	315134	177408	101376	593918
45	318928	177408	101376	597712
46	315066	177408	101376	593850
47	312632	177408	101376	591416
48	317990	177408	101376	596774
49	313244	177408	101376	592028
50	316880	177408	101376	595664
Min.	255420	177408	101376	534204
Max.	335522	177408	101376	614306
Average	307993.7	177408	101376	586777.7

Table B.6 Cycle counts of uncovered background functions for 'Miss America' sequence.

Frame	Position A	Position B	Position C	Position D	Total
2	225120	248220	248832	297288	1019460
3	246468	269700	269736	318324	1104228
4	249384	272712	272640	321324	1116060
5	256380	279768	279588	328332	1144068
6	260836	284484	284076	333096	1162492
7	264272	287928	287496	336492	1176188
8	265152	288864	288456	337524	1179996
9	264996	288708	288312	337380	1179396
10	264392	288124	287684	336772	1176972
11	265572	289332	288888	338004	1181796
12	265940	289688	289232	338348	1183208
13	265956	289704	289308	338412	1183380
14	267180	290952	290640	339768	1188540
15	272296	296056	295840	344956	1209148
16	271956	295716	295512	344628	1207812
17	271872	295632	295404	344520	1207428
18	271524	295284	294996	344112	1205916
19	271092	294852	294624	343740	1204308
20	271512	295260	295008	344112	1205892
21	270948	294696	294376	343480	1203500
22	274556	298288	297928	347012	1217784
23	274836	298580	298304	347404	1219124
24	273508	297236	296976	346044	1213764
25	272860	296604	296328	345428	1211220
26	275276	299000	298624	347688	1220588
27	278192	301912	301548	350608	1232260
28	289888	313500	312780	361716	1277884
29	290252	313956	313196	362224	1279628
30	292864	316564	315808	364828	1290064
31	292364	316056	315224	364256	1287900
32	293060	316680	316008	364984	1290732
33	293916	317660	316908	366008	1294492
34	292832	316412	315784	364728	1289756
35	289412	313036	312420	361384	1276252
36	288308	311968	311312	360344	1271932
37	287912	311696	310980	360120	1270708
38	288096	311864	311068	360192	1271220
39	287352	311048	310288	359340	1268028
40	285528	309252	308540	357632	1260952
41	285592	309360	308644	357768	1261364
42	284688	308460	307740	356868	1257756
43	286236	309968	309156	358244	1263604
44	285892	309652	308884	358000	1262428
45	284784	308568	307836	356976	1258164
46	285100	308828	308080	357164	1259172
47	285816	309588	308868	357996	1262268
48	285180	308964	308160	357300	1259604
49	284712	308460	307692	356796	1257660
50	285848	309572	308624	357672	1261716
Min.	225120	248220	248832	297288	1019460
Max.	293916	317660	316908	366008	1294492
Average	276279.8	299967.6	299476.7	348517.1	1224241

Table B.7 Cycle counts of motion synthesis functions for 'Miss America' sequence.

Frame	Square Dif	Find TMF	Median Va	Dilation	Erosion	Eliminatio	Total
2	101376	2635776	177408	380160	380160	164677	3839557
3	101376	2635776	177408	380160	380160	182092	3856972
4	101376	2635776	177408	380160	380160	176888	3851768
5	101376	2635776	177408	380160	380160	179711	3854591
6	101376	2635776	177408	380160	380160	179930	3854810
7	101376	2306304	177408	380160	380160	196120	3541528
8	101376	2306304	177408	380160	380160	202046	3547454
9	101376	2635776	177408	380160	380160	176866	3851746
10	101376	2306304	177408	380160	380160	200307	3545715
11	101376	2306304	177408	380160	380160	194689	3540097
12	101376	2306304	177408	380160	380160	195278	3540686
13	101376	2306304	177408	380160	380160	190163	3535571
14	101376	2306304	177408	380160	380160	192136	3537544
15	101376	2306304	177408	380160	380160	204317	3549725
16	101376	2306304	177408	380160	380160	183442	3528850
17	101376	2306304	177408	380160	380160	185900	3531308
18	101376	2306304	177408	380160	380160	211581	3556989
19	101376	2306304	177408	380160	380160	205875	3551283
20	101376	2306304	177408	380160	380160	211945	3557353
21	101376	2306304	177408	380160	380160	195025	3540433
22	101376	2306304	177408	380160	380160	222807	3568215
23	101376	2306304	177408	380160	380160	253825	3599233
24	101376	1976832	177408	380160	380160	199108	3215044
25	101376	2306304	177408	380160	380160	202566	3547974
26	101376	2306304	177408	380160	380160	209521	3554929
27	101376	1976832	177408	380160	380160	261106	3277042
28	101376	2306304	177408	380160	380160	206092	3551500
29	101376	2306304	177408	380160	380160	217813	3563221
30	101376	1976832	177408	380160	380160	277357	3293293
31	101376	2306304	177408	380160	380160	214337	3559745
32	101376	2306304	177408	380160	380160	223009	3568417
33	101376	2306304	177408	380160	380160	220234	3565642
34	101376	2306304	177408	380160	380160	206598	3552006
35	101376	2306304	177408	380160	380160	214822	3560230
36	101376	2306304	177408	380160	380160	210649	3556057
37	101376	2306304	177408	380160	380160	212411	3557819
38	101376	2306304	177408	380160	380160	222697	3568105
39	101376	2306304	177408	380160	380160	213260	3558668
40	101376	2306304	177408	380160	380160	220961	3566369
41	101376	2306304	177408	380160	380160	219065	3564473
42	101376	2306304	177408	380160	380160	224590	3569998
43	101376	2306304	177408	380160	380160	224103	3569511
44	101376	2306304	177408	380160	380160	211491	3556899
45	101376	2306304	177408	380160	380160	210700	3556108
46	101376	2306304	177408	380160	380160	214715	3560123
47	101376	2306304	177408	380160	380160	213052	3558460
48	101376	2306304	177408	380160	380160	210412	3555820
49	101376	2306304	177408	380160	380160	196812	3542220
50	101376	2306304	177408	380160	380160	208244	3553652
Min.	101376	1976832	177408	380160	380160	164677	3180613
Max.	101376	2635776	177408	380160	380160	277357	3952237
Average	101376	2326476	177408	380160	380160	207578.5	3573158

Table B.8 Cycle count of MF detection functions for 'Miss America' sequence.

Frame	Chain Cod	Area	Fill	Total
2	159633	1708	3336	164677
3	166562	2773	12757	182092
4	166281	3058	7549	176888
5	167836	3208	8667	179711
6	168964	3521	7445	179930
7	175657	4488	15975	196120
8	179461	5233	17352	202046
9	166677	3184	7005	176866
10	180124	5495	14688	200307
11	176728	5096	12865	194689
12	177004	4924	13350	195278
13	173575	4648	11940	190163
14	175952	4867	11317	192136
15	179401	5294	19622	204317
16	169812	3569	10061	183442
17	171063	3917	10920	185900
18	184358	6296	20927	211581
19	182270	6019	17586	205875
20	185073	6508	20364	211945
21	178300	5423	11302	195025
22	190706	7737	24364	222807
23	202986	9482	41357	253825
24	179146	5368	14594	199108
25	180957	5578	16031	202566
26	186265	6835	16421	209521
27	211183	11202	38721	261106
28	183637	6459	15996	206092
29	190962	8879	17972	217813
30	220361	12729	44267	277357
31	187793	7214	19330	214337
32	193441	8072	21496	223009
33	194010	8514	17710	220234
34	184953	6862	14783	206598
35	186617	7173	21032	214822
36	186244	6904	17501	210649
37	187194	7410	17807	212411
38	187988	7185	27524	222697
39	188091	7176	17993	213260
40	193542	8464	18955	220961
41	195622	7982	15461	219065
42	195804	8665	20121	224590
43	196212	8864	19027	224103
44	186970	7158	17363	211491
45	187155	7005	16540	210700
46	190831	7617	16267	214715
47	187614	7452	17986	213052
48	184744	6877	18791	210412
49	179716	6062	11034	196812
50	186921	6922	14401	208244
Min.	159633	1708	3336	164677
Max.	220361	12729	44267	277357
Average	183926.4	6389.306	17262.71	207578.5

Table B.9 Cycle counts of MF elimination of regions functions for 'Miss America' sequence.

Frame	Change D	Motion An	MC Shape	Motion Sy	MF Detect	MF Shape	Total
2	1537980	3814801	110696	805332	3216376	252447	9737632
3	1549342	4261880	114007	843420	3224037	181911	10174597
4	1553307	5157987	129569	925980	3549313	269125	11585281
5	1550189	5174782	130805	929820	3557251	271032	11613879
6	1552067	5195951	131842	935364	3548501	217343	11581068
7	1588580	5414610	156976	951396	3563982	255009	11930553
8	1608189	5954401	163921	1035588	3552619	177976	12492694
9	1664858	6509073	215171	1114312	3541047	227810	13272271
10	1673564	7043672	228858	1173348	3215627	179364	13514433
11	1671153	7203261	222258	1175552	3555616	94077	13921917
12	1672013	7446237	223308	1201468	3550727	209153	14302906
13	1675086	7828857	203766	1265104	3518246	150242	14641301
14	1685002	8364778	237291	1338740	3875122	157850	15658783
15	1684315	10862249	240343	1568356	4203788	148280	18707331
16	1694563	10875104	261824	1576476	4188175	116229	18712371
17	1694461	10806000	262148	1576020	4175522	146333	18660484
18	1700882	10893793	263611	1579380	4180536	211811	18830013
19	1701681	11153418	267235	1607020	4191491	203887	19124732
20	1700353	11147045	264403	1606616	4201143	211017	19130577
21	1701140	11155656	265777	1610052	3873959	105991	18712575
22	1697282	11149019	265305	1613604	3872724	181779	18779713
23	1700870	11153202	264972	1622632	3867926	94990	18704592
24	1700725	11154869	266751	1627468	3853089	212191	18815093
25	1701737	11145612	268241	1624420	4210181	238390	19188581
26	1698876	11127150	268621	1627684	3842312	149187	18713830
27	1700164	11164056	266889	1627088	4205638	101792	19065627
28	1699389	11151738	271332	1632212	3864516	170270	18789457
29	1699849	11171271	269600	1633152	3881695	210571	18866138
30	1700594	11168553	267233	1620764	3845081	112870	18715095
31	1700212	11187289	267354	1618040	3882622	205417	18860934
32	1700277	11221043	268713	1626540	3882346	276870	18975789
33	1700098	11282946	269177	1615912	3868191	240170	18976494
34	1699805	11271363	268514	1617068	3898507	156750	18912007
35	1699701	11354058	270955	1605396	3890193	182493	19002796
36	1699723	11326079	269879	1600572	3884226	290103	19070582
37	1700044	11513174	272180	1609140	3894684	259499	19248721
38	1700678	11536836	274337	1606804	3888721	248025	19255401
39	1700206	11533088	273139	1604036	3895176	186461	19192106
40	1701014	11527934	273913	1605240	3884723	259752	19252576
41	1700786	11863373	270983	1606924	3880123	171344	19493533
42	1689091	11882996	271463	1607100	3870035	198695	19519380
43	1689347	12035629	273973	1605504	3884664	94355	19583472
44	1689389	11903495	268916	1610836	3849874	205361	19527871
45	1688767	11854487	273632	1603380	3857729	206474	19484469
46	1690276	11930834	272287	1606948	3860074	99475	19459894
47	1689778	11889237	272677	1600196	3870719	329472	19652079
48	1692424	11807001	273514	1605972	3875891	222545	19477347
49	1692620	11805338	273765	1601132	3856277	239651	19468783
50	1692511	11699587	274309	1597092	3861803	155162	19280464
Min.	1533171	3814801	110696	805332	3183368	94077	9541445
Max.	1731869	12059055	274337	1633152	4227979	329472	20255864
Average	1674999	9981119	243600.7	1461269	3825772	193612.3	17380372

Table B.10 Cycle counts of main functions for 'Claire' sequence.

Frame	Abs. Differ	Binarisatio	Median Va	Combining	Dilation	Erosion	Eliminatio	Total
2	101376	177408	177408	101376	380160	380160	220092	1537980
3	101376	177408	177408	101376	380160	380160	231454	1549342
4	101376	177408	177408	101376	380160	380160	235419	1553307
5	101376	177408	177408	101376	380160	380160	232301	1550189
6	101376	177408	177408	101376	380160	380160	234179	1552067
7	101376	177408	177408	101376	380160	380160	270692	1588580
8	101376	177408	177408	101376	380160	380160	290301	1608189
9	101376	177408	177408	101376	380160	380160	346970	1664858
10	101376	177408	177408	101376	380160	380160	355676	1673564
11	101376	177408	177408	101376	380160	380160	353265	1671153
12	101376	177408	177408	101376	380160	380160	354125	1672013
13	101376	177408	177408	101376	380160	380160	357198	1675086
14	101376	177408	177408	101376	380160	380160	367114	1685002
15	101376	177408	177408	101376	380160	380160	366427	1684315
16	101376	177408	177408	101376	380160	380160	376675	1694563
17	101376	177408	177408	101376	380160	380160	376573	1694461
18	101376	177408	177408	101376	380160	380160	382994	1700882
19	101376	177408	177408	101376	380160	380160	383793	1701681
20	101376	177408	177408	101376	380160	380160	382465	1700353
21	101376	177408	177408	101376	380160	380160	383252	1701140
22	101376	177408	177408	101376	380160	380160	379394	1697282
23	101376	177408	177408	101376	380160	380160	382982	1700870
24	101376	177408	177408	101376	380160	380160	382837	1700725
25	101376	177408	177408	101376	380160	380160	383849	1701737
26	101376	177408	177408	101376	380160	380160	380988	1698876
27	101376	177408	177408	101376	380160	380160	382276	1700164
28	101376	177408	177408	101376	380160	380160	381501	1699389
29	101376	177408	177408	101376	380160	380160	381961	1699849
30	101376	177408	177408	101376	380160	380160	382706	1700594
31	101376	177408	177408	101376	380160	380160	382324	1700212
32	101376	177408	177408	101376	380160	380160	382389	1700277
33	101376	177408	177408	101376	380160	380160	382210	1700098
34	101376	177408	177408	101376	380160	380160	381917	1699805
35	101376	177408	177408	101376	380160	380160	381813	1699701
36	101376	177408	177408	101376	380160	380160	381835	1699723
37	101376	177408	177408	101376	380160	380160	382156	1700044
38	101376	177408	177408	101376	380160	380160	382790	1700678
39	101376	177408	177408	101376	380160	380160	382318	1700206
40	101376	177408	177408	101376	380160	380160	383126	1701014
41	101376	177408	177408	101376	380160	380160	382898	1700786
42	101376	177408	177408	101376	380160	380160	371203	1689091
43	101376	177408	177408	101376	380160	380160	371459	1689347
44	101376	177408	177408	101376	380160	380160	371501	1689389
45	101376	177408	177408	101376	380160	380160	370879	1688767
46	101376	177408	177408	101376	380160	380160	372388	1690276
47	101376	177408	177408	101376	380160	380160	371890	1689778
48	101376	177408	177408	101376	380160	380160	374536	1692424
49	101376	177408	177408	101376	380160	380160	374732	1692620
50	101376	177408	177408	101376	380160	380160	374623	1692511
Min.	101376	177408	177408	101376	380160	380160	215283	1533171
Max.	101376	177408	177408	101376	380160	380160	413981	1731869
Average	101376	177408	177408	101376	380160	380160	357111.1	1674999

Table B.11 Cycle counts of change detection functions for 'Claire' sequence.

Frame	Chain Cod	Area	Fill	Total
2	178561	4799	36732	220092
3	185218	5995	40241	231454
4	181228	5134	49057	235419
5	174586	3965	53750	232301
6	175410	4109	54660	234179
7	195254	7686	67752	270692
8	205362	9573	75366	290301
9	225362	12998	108610	346970
10	228320	13091	114265	355676
11	225081	12665	115519	353265
12	223058	12350	118717	354125
13	221542	12062	123594	357198
14	224867	12612	129635	367114
15	220759	11892	133776	366427
16	204759	8979	162937	376675
17	205316	9185	162072	376573
18	206996	9456	166542	382994
19	207078	9509	167206	383793
20	205398	9115	167952	382465
21	205122	9119	169011	383252
22	202341	8537	168516	379394
23	203808	8845	170329	382982
24	203596	8809	170432	382837
25	204567	9015	170267	383849
26	202430	8611	169947	380988
27	203030	8751	170495	382276
28	202583	8516	170402	381501
29	202536	8629	170796	381961
30	202748	8665	171293	382706
31	202734	8629	170961	382324
32	202801	8674	170914	382389
33	203066	8719	170425	382210
34	202960	8701	170256	381917
35	202589	8638	170586	381813
36	202900	8736	170199	381835
37	202801	8674	170681	382156
38	203278	8755	170757	382790
39	203013	8710	170595	382318
40	203437	8783	170906	383126
41	204219	7113	171566	382898
42	191883	6820	172500	371203
43	192042	6847	172570	371459
44	192253	6883	172365	371501
45	192360	6901	171618	370879
46	193208	7045	172135	372388
47	193155	7036	171699	371890
48	194798	7315	172423	374536
49	195169	7378	172185	374732
50	195116	7309	172198	374623
Min.	174586	3965	36732	215283
Max.	228320	13091	172570	413981
Average	202055.1	8578.327	146477.8	357111.1

Table B.12 Cycle counts of elimination of regions for 'Claire' sequence.

Frame	Mean Valu	Bilinear Int	Upsamplin	Motion Est	Motion Fie	Moving Ar	Uncovere	Total
2	354816	709632	354816	1544211	159744	228096	463486	3814801
3	354816	709632	354816	1977876	165888	228096	470756	4261880
4	354816	709632	354816	2813229	202752	228096	494646	5157987
5	354816	709632	354816	2830104	202752	228096	494566	5174782
6	354816	709632	354816	2843649	208896	228096	496046	5195951
7	354816	709632	354816	2964870	294912	228096	507468	5414610
8	354816	709632	354816	3480183	301056	228096	525802	5954401
9	354816	709632	354816	3823839	473088	228096	564786	6509073
10	354816	709632	354816	4343814	479232	228096	573266	7043672
11	354816	709632	354816	4494123	479232	228096	582546	7203261
12	354816	709632	354816	4700907	491520	228096	606450	7446237
13	354816	709632	354816	5070825	491520	228096	619152	7828857
14	354816	709632	354816	5593086	491520	228096	632812	8364778
15	354816	709632	354816	8033139	491520	228096	690230	10862249
16	354816	709632	354816	8050086	491520	228096	686138	10875104
17	354816	709632	354816	7977510	491520	228096	689610	10806000
18	354816	709632	354816	8062281	491520	228096	692632	10893793
19	354816	709632	354816	8301222	491520	228096	713316	11153418
20	354816	709632	354816	8297055	491520	228096	711110	11147045
21	354816	709632	354816	8315100	491520	228096	701676	11155656
22	354816	709632	354816	8300943	491520	228096	709196	11149019
23	354816	709632	354816	8305056	491520	228096	709266	11153202
24	354816	709632	354816	8311041	491520	228096	704948	11154869
25	354816	709632	354816	8298126	485376	228096	714750	11145612
26	354816	709632	354816	8287776	485376	228096	706638	11127150
27	354816	709632	354816	8310816	485376	228096	720504	11164056
28	354816	709632	354816	8315586	485376	228096	703416	11151738
29	354816	709632	354816	8315091	485376	228096	723444	11171271
30	354816	709632	354816	8332443	485376	228096	703374	11168553
31	354816	709632	354816	8344413	485376	228096	710140	11187289
32	354816	709632	354816	8378865	485376	228096	709442	11221043
33	354816	709632	354816	8440326	485376	228096	709884	11282946
34	354816	709632	354816	8435061	485376	228096	703566	11271363
35	354816	709632	354816	8521632	485376	228096	699690	11354058
36	354816	709632	354816	8499447	485376	228096	693896	11326079
37	354816	709632	354816	8685954	485376	228096	694484	11513174
38	354816	709632	354816	8705412	485376	228096	698688	11536836
39	354816	709632	354816	8697402	491520	228096	696806	11533088
40	354816	709632	354816	8690022	491520	228096	699032	11527934
41	354816	709632	354816	9024471	491520	228096	700022	11863373
42	354816	709632	354816	9041094	491520	228096	703022	11882996
43	354816	709632	354816	9196731	491520	228096	700018	12035629
44	354816	709632	354816	9065205	491520	228096	699410	11903495
45	354816	709632	354816	9016911	491520	228096	698696	11854487
46	354816	709632	354816	9088182	491520	228096	703772	11930834
47	354816	709632	354816	9055989	491520	228096	694368	11889237
48	354816	709632	354816	8973189	491520	228096	694932	11807001
49	354816	709632	354816	8961642	491520	228096	704816	11805338
50	354816	709632	354816	8867277	485376	228096	699574	11699587
Min.	354816	709632	354816	1544211	159744	228096	463486	3814801
Max.	354816	709632	354816	9196731	491520	228096	723444	12059055
Average	354816	709632	354816	7224147	449891.3	228096	659720.2	9981119

Table B.13 Cycle counts of motion analysis functions for 'Claire' sequence.

Frame	Level 1	Level 2	Level 3	Total
2	917456	126908	499847	1544211
3	1179044	157799	641033	1977876
4	1725072	215574	872583	2813229
5	1739472	216069	874563	2830104
6	1748652	216924	878073	2843649
7	1803770	234092	927008	2964870
8	2135802	258897	1085484	3480183
9	2389650	280635	1153554	3823839
10	2694332	327731	1321751	4343814
11	2779384	337348	1377391	4494123
12	2933060	351656	1416191	4700907
13	3133764	378963	1558098	5070825
14	3438958	419194	1734934	5593086
15	4911780	619104	2502255	8033139
16	4927980	615486	2506620	8050086
17	4893734	607487	2476289	7977510
18	4935972	619536	2506773	8062281
19	5080784	631403	2589035	8301222
20	5077184	635021	2584850	8297055
21	5108414	631988	2574698	8315100
22	5141174	625922	2533847	8300943
23	5131904	632285	2540867	8305056
24	5136224	632645	2542172	8311041
25	5128484	631610	2538032	8298126
26	5138474	627137	2522165	8287776
27	5151074	629252	2530490	8310816
28	5152874	629837	2532875	8315586
29	5155214	629297	2530580	8315091
30	5157014	628892	2546537	8332443
31	5153774	632195	2558444	8344413
32	5149814	636263	2592788	8378865
33	5181360	642507	2616459	8440326
34	5181360	641472	2612229	8435061
35	5246926	645376	2629330	8521632
36	5226766	645421	2627260	8499447
37	5345658	658944	2681352	8685954
38	5342778	667260	2695374	8705412
39	5338548	666540	2692314	8697402
40	5337468	665325	2687229	8690022
41	5544194	693773	2786504	9024471
42	5549594	699686	2791814	9041094
43	5640676	713299	2842756	9196731
44	5556210	703500	2805495	9065205
45	5565326	691991	2759594	9016911
46	5605512	698595	2784075	9088182
47	5610974	691181	2753834	9055989
48	5561842	680707	2730640	8973189
49	5549872	684595	2727175	8961642
50	5487590	673817	2705870	8867277
Min.	917456	126908	499847	1544211
Max.	5640676	713299	2842756	9196731
Average	4449448	550635.5	2224064	7224147

Table B.14 Cycle count of each level in motion estimation for 'Claire' sequence.

Frame	First Steps	Median Fil	Combine	Total
2	184702	177408	101376	463486
3	191972	177408	101376	470756
4	215862	177408	101376	494646
5	215782	177408	101376	494566
6	217262	177408	101376	496046
7	228684	177408	101376	507468
8	247018	177408	101376	525802
9	286002	177408	101376	564786
10	294482	177408	101376	573266
11	303762	177408	101376	582546
12	327666	177408	101376	606450
13	340368	177408	101376	619152
14	354028	177408	101376	632812
15	411446	177408	101376	690230
16	407354	177408	101376	686138
17	410826	177408	101376	689610
18	413848	177408	101376	692632
19	434532	177408	101376	713316
20	432326	177408	101376	711110
21	422892	177408	101376	701676
22	430412	177408	101376	709196
23	430482	177408	101376	709266
24	426164	177408	101376	704948
25	435966	177408	101376	714750
26	427854	177408	101376	706638
27	441720	177408	101376	720504
28	424632	177408	101376	703416
29	444660	177408	101376	723444
30	424590	177408	101376	703374
31	431356	177408	101376	710140
32	430658	177408	101376	709442
33	431100	177408	101376	709884
34	424782	177408	101376	703566
35	420906	177408	101376	699690
36	415112	177408	101376	693896
37	415700	177408	101376	694484
38	419904	177408	101376	698688
39	418022	177408	101376	696806
40	420248	177408	101376	699032
41	421238	177408	101376	700022
42	424238	177408	101376	703022
43	421234	177408	101376	700018
44	420626	177408	101376	699410
45	419912	177408	101376	698696
46	424988	177408	101376	703772
47	415584	177408	101376	694368
48	416148	177408	101376	694932
49	426032	177408	101376	704816
50	420790	177408	101376	699574
Min.	184702	177408	101376	463486
Max.	444660	177408	101376	723444
Average	380936.2	177408	101376	659720.2

Table B.15 Cycle counts of uncovered background detection functions for 'Claire' sequence.

Frame	Position A	Position B	Position C	Position D	Total
2	170916	194632	195356	244428	805332
3	180336	204180	204852	254052	843420
4	201156	224988	225324	274512	925980
5	202116	225936	226296	275472	929820
6	203556	227388	227616	276804	935364
7	208632	231972	231048	279744	951396
8	229404	253152	251964	301068	1035588
9	250876	274036	270428	318972	1114312
10	265628	289120	284872	333728	1173348
11	266276	289580	285516	334180	1175552
12	272548	295760	292288	340872	1201468
13	288728	311876	307996	356504	1265104
14	306800	330116	326568	375256	1338740
15	363404	386864	384632	433456	1568356
16	365292	388848	386712	435624	1576476
17	365244	388724	386608	435444	1576020
18	366288	389792	387220	436080	1579380
19	373296	396616	394212	442896	1607020
20	373212	396572	394068	442764	1606616
21	373896	397320	395028	443808	1610052
22	374816	398176	395948	444664	1613604
23	377172	400496	398144	446820	1622632
24	378280	401656	399400	448132	1627468
25	377540	400724	398808	447348	1624420
26	378388	401584	399580	448132	1627684
27	378144	401444	399416	448084	1627088
28	379504	402748	400680	449280	1632212
29	379728	402960	400932	449532	1633152
30	376308	399616	398096	446744	1620764
31	375736	398944	397408	445952	1618040
32	378132	401360	399240	447808	1626540
33	375304	398588	396692	445328	1615912
34	375564	398840	397016	445648	1617068
35	372620	395956	394064	442756	1605396
36	371256	394596	393012	441708	1600572
37	373560	396888	395004	443688	1609140
38	372808	396136	394588	443272	1606804
39	372100	395460	393880	442596	1604036
40	372424	395800	394136	442880	1605240
41	372924	396200	394608	443192	1606924
42	372940	396224	394648	443288	1607100
43	372588	395740	394316	442860	1605504
44	373932	397060	395672	444172	1610836
45	372032	395224	393788	442336	1603380
46	373056	396148	394628	443116	1606948
47	371296	394488	392932	441480	1600196
48	372744	395880	394428	442920	1605972
49	371736	394616	393284	441496	1601132
50	370580	393632	392236	440644	1597092
Min.	170916	194632	195356	244428	805332
Max.	379728	402960	400932	449532	1633152
Average	336547.3	359890.9	358065.1	406766.1	1461269

Table B.16 Cycle counts of motion synthesis functions for 'Claire' sequence.

Frame	Square Dif	Find TMF	Median Va	Dilation	Erosion	Eliminatio	Total
2	101376	1976832	177408	380160	380160	200440	3216376
3	101376	1976832	177408	380160	380160	208101	3224037
4	101376	2306304	177408	380160	380160	203905	3549313
5	101376	2306304	177408	380160	380160	211843	3557251
6	101376	2306304	177408	380160	380160	203093	3548501
7	101376	2306304	177408	380160	380160	218574	3563982
8	101376	2306304	177408	380160	380160	207211	3552619
9	101376	2306304	177408	380160	380160	195639	3541047
10	101376	1976832	177408	380160	380160	199691	3215627
11	101376	2306304	177408	380160	380160	210208	3555616
12	101376	2306304	177408	380160	380160	205319	3550727
13	101376	2306304	177408	380160	380160	172838	3518246
14	101376	2635776	177408	380160	380160	200242	3875122
15	101376	2965248	177408	380160	380160	199436	4203788
16	101376	2965248	177408	380160	380160	183823	4188175
17	101376	2965248	177408	380160	380160	171170	4175522
18	101376	2965248	177408	380160	380160	176184	4180536
19	101376	2965248	177408	380160	380160	187139	4191491
20	101376	2965248	177408	380160	380160	196791	4201143
21	101376	2635776	177408	380160	380160	199079	3873959
22	101376	2635776	177408	380160	380160	197844	3872724
23	101376	2635776	177408	380160	380160	193046	3867926
24	101376	2635776	177408	380160	380160	178209	3853089
25	101376	2965248	177408	380160	380160	205829	4210181
26	101376	2635776	177408	380160	380160	167432	3842312
27	101376	2965248	177408	380160	380160	201286	4205638
28	101376	2635776	177408	380160	380160	189636	3864516
29	101376	2635776	177408	380160	380160	206815	3881695
30	101376	2635776	177408	380160	380160	170201	3845081
31	101376	2635776	177408	380160	380160	207742	3882622
32	101376	2635776	177408	380160	380160	207466	3882346
33	101376	2635776	177408	380160	380160	193311	3868191
34	101376	2635776	177408	380160	380160	223627	3898507
35	101376	2635776	177408	380160	380160	215313	3890193
36	101376	2635776	177408	380160	380160	209346	3884226
37	101376	2635776	177408	380160	380160	219804	3894684
38	101376	2635776	177408	380160	380160	213841	3888721
39	101376	2635776	177408	380160	380160	220296	3895176
40	101376	2635776	177408	380160	380160	209843	3884723
41	101376	2635776	177408	380160	380160	205243	3880123
42	101376	2635776	177408	380160	380160	195155	3870035
43	101376	2635776	177408	380160	380160	209784	3884664
44	101376	2635776	177408	380160	380160	174994	3849874
45	101376	2635776	177408	380160	380160	182849	3857729
46	101376	2635776	177408	380160	380160	185194	3860074
47	101376	2635776	177408	380160	380160	195839	3870719
48	101376	2635776	177408	380160	380160	201011	3875891
49	101376	2635776	177408	380160	380160	181397	3856277
50	101376	2635776	177408	380160	380160	186923	3861803
Min.	101376	1976832	177408	380160	380160	167432	3183368
Max.	101376	2965248	177408	380160	380160	223627	4227979
Average	101376	2588709	177408	380160	380160	197959.2	3825772

Table B.17 Cycle counts of MF detection functions for 'Claire' sequence.

Frame	Chain Cod	Area	Fill	Total
2	169769	3200	27471	200440
3	175826	4258	28017	208101
4	174618	4330	24957	203905
5	181651	5304	24888	211843
6	177969	4708	20416	203093
7	181654	5614	31306	218574
8	179343	5027	22841	207211
9	176625	4611	14403	195639
10	180062	5055	14574	199691
11	179151	4943	26114	210208
12	182420	5597	17302	205319
13	152063	5098	15677	172838
14	178950	5086	16206	200242
15	178585	4975	15876	199436
16	170137	3579	10107	183823
17	162891	2478	5801	171170
18	166232	3168	6784	176184
19	173157	4316	9666	187139
20	177320	4943	14528	196791
21	180960	5639	12480	199079
22	178370	5356	14118	197844
23	176037	5185	11824	193046
24	168264	3682	6263	178209
25	183854	6182	15793	205829
26	151063	4557	11812	167432
27	181262	5831	14193	201286
28	173177	4562	11897	189636
29	186287	5514	15014	206815
30	152063	5094	13044	170201
31	183803	6372	17567	207742
32	182987	6163	18316	207466
33	176183	4789	12339	193311
34	186488	6259	30880	223627
35	184566	5939	24808	215313
36	183190	5757	20399	209346
37	187956	6505	25343	219804
38	186513	6584	20744	213841
39	189442	6991	23863	220296
40	182443	5975	21425	209843
41	184134	5476	15633	205243
42	175941	4740	14474	195155
43	182443	5808	21533	209784
44	165920	3083	5991	174994
45	170144	3930	8775	182849
46	171976	4140	9078	185194
47	177424	5084	13331	195839
48	182336	5961	12714	201011
49	169197	3773	8427	181397
50	172760	5376	8787	186923
Min.	151063	2478	5801	167432
Max.	189442	6991	31306	223627
Average	176440.9	5032.592	16485.69	197959.2

Table B.18 Cycle count of MF elimination of regions functions for 'Claire' sequence.

REFERENCES

[1] ISO/IEC JTC1/SC29/WG11, "Coding of Moving Pictures and Associated Audio for Digital Storage Media up to about 1.5 Mbit/s - Part 2: Video", ISO/IEC 11172-2, 1993

[2] ISO/IEC JTC1/SC29/WG11, "Generic Coding of Moving Pictures and Associated Audio Information: Video", ISO/IEC 13818-2, 1995

[3] CCITT Recommendation H.261, "Video Codec for Audio-Visual Services at p*64 kbits/s", CDM XV-R37-E

[4] ITU-T Draft Recommendation H.263, "Video Coding for Low Bitrate Communications", November 1995

[5] ISO/IEC JTC1/SC29/WG11 MPEG4 AOE Group, "Proposal Package Description (PPD) - Revision 3", Tokyo, July 1995

[6] B. Welsh, "Model-Based Coding of Images', Ph.D. Thesis Essex University, January 1991

[7] K. Li, A. Lundmark, and R. Forchheimer, "Image Sequence at Very Low Bitrates: A review", *IEEE Trans. Image Processing*, vol.3, no.5, pp 589 - 606, Sept 1994

[8] H. C. Musmann, M. Hötter, J. Osterman, "Object-Oriented Analysis - Synthesis Coing of Moving Images", *Signal Processing: Image Communication*, vol. 1, no. 4, pp. 117 - 138, Oct.1989

[9] "SIMOC 1", Cost 211-ter, Simulation Subgroup Document No. SIM(94)61

[10] M. Bierling, "Displacement Estimation by Hierarchical Block Matching", Vis. Commun. Image Proc. '88, *Proc. SPIE 1001*, Cambridge, MA, pp. 942 - 951, Nov. 1988

[11] P. Gerken, "Object-Based Analysis-Synthesis Coding of Image Sequencies at Very Low Bit Rates", *IEEE Trans. Circuits Systems for Video Technology*; vol.4, no. 3, pp. 228 - 235, June 1994

[12] L. Ward, N. Brady, N. O'Connor, "Desire 3.0 Implementation Test Results", Video Coding Group, Dublin City University.

[13] A. C. Downton, "Performance Profiling of the Dublin City University Object-Oriented Coder", AICR Project Document 5825-00/TECH/END/005

[14] Texas Instruments, "Digital Signal Processing Applications with the TMS320 Family: Theory, Algorithms, and Implementations - Vol. 3", pp. 33-48, Literature No. SPRA017, March 1990

[15] T. Pavildis, "Algorithms for Graphics and Image Processing", pp. 108-110, Computer Science Press, 1982

[16] B. Jahne, "Digital Image Processing - Concepts, Algorithms, and Scientific Applications", pp.212-213, Springer-Verlag, 1991

[17] B. Liu and A. Zacharrin, "New Fasr Algorithms for the Estimation of Block Motion Vectors", *IEEE Trans. Circuits Systems for Video Technology*, vol. 3, no. 2, pp. 148-157, April 1993

[18] D. Vernon, "Machine Vision - Automated Visual Inspection and Robot Vision", pp. 85-86, Prentice Hall, 1991

[19] Texas Instruments, "TMS320C8x System-Level Synopsis", Literature No.SPRU113B, Sept. 1995

[20] http://www.ti.com/

[21] Texas Instruments, "TMS320C80 Multimedia Video Processor Technical Brief", Literature No.SPRU106A, June 1994

[22] F. Seytter, "Delay in Very Low Bit-Rate Coding Algorithms", Proceedings VLBV94, April 1994