PARALLEL

IMPLEMENTATION

OF

FRACTAL IMAGE COMPRESSION

By Ryan F. UYS

University of Natal, Durban

2000

Submitted in fulfillment of the academic requirements for the degree of Masters of Science in the Department of Electronic Engineering, University Of Natal

Abstract

Fractal image compression exploits the piecewise self-similarity present in real images as a form of information redundancy that can be eliminated to achieve compression. This theory based on Partitioned Iterated Function Systems is presented. As an alternative to the established JPEG, it provides a similar compression-ratio to fidelity trade-off. Fractal techniques promise faster decoding and potentially higher fidelity, but the computationally intensive compression process has prevented commercial acceptance.

This thesis presents an algorithm mapping the problem onto a parallel processor architecture, with the goal of reducing the encoding time. The experimental work involved implementation of this approach on the Texas Instruments TMS320C80 parallel processor system. Results indicate that the fractal compression process is unusually well suited to parallelism with speed gains approximately linearly related to the number of processors used. Parallel processing issues such as coherency, management and interfacing are discussed. The code designed incorporates pipelining and parallelism on all conceptual and practical levels ensuring that all resources are fully utilised, achieving close to optimal efficiency.

The computational intensity was reduced by several means, including conventional classification of image sub-blocks by content with comparisons across class boundaries prohibited. A faster approach adopted was to perform estimate comparisons between blocks based on pixel value variance, identifying candidates for more time-consuming, accurate RMS inter-block comparisons. These techniques, combined with the parallelism, allow compression of 512x512 pixel x 8 bit images in under 20 seconds, while maintaining a 30dB PSNR. This is up to an order of magnitude faster than reported for conventional sequential processor implementations. Fractal based compression of colour images and video sequences is also considered.

The work confirms the potential of fractal compression techniques, and demonstrates that a parallel implementation is appropriate for addressing the compression time problem.

The processor system used in these investigations is faster than currently available PC platforms, but the relevance lies in the anticipation that future generations of affordable processors will exceed its performance. The advantages of fractal image compression may then be accessible to the average computer user, leading to commercial acceptance.

Preface

The research work discussed in this thesis was performed by Mr. Ryan F. Uys, initially under the supervision of Mr. J. Prentice, formerly of the Department of Electronic Engineering, University of Natal, and later under the supervision of Prof. A. Broadhurst, also of the Department of Electronic Engineering, University of Natal. The work was financially supported by the Foundation for Research and Development (FRD) under their prestige scholarship programme. Parts of this work were published by the student author and Mr. J. Prentice in the South African Journal *Electron* in July 1997. The findings of the research were presented by the student author at the Comsig '98 conference in Cape Town, South Africa in September 1998.

This whole thesis, unless specifically indicated to the contrary in the text, is the student author's work, and has not been submitted in part, or in whole, to any other University.

Acknowledgements

Thanks go to Mr. J. Prentice for his technical guidance and encouragement during the first year of the work, when I wanted to give up. Thanks go to Prof. A. Broadhurst for his encouragement on the third year when I needed motivation to complete the document, and for his proofreading.

Thanks must be given to Texas Instruments who provided the DSP used for the parallel processor research free of charge under their university development programme. Thanks are also owed to FRD, SA for the financial support which allowed the purchase of the computer equipment and covered the expenses of the student author.

Finally thanks are due to Mr. T. Oellerman, a fellow postgraduate in the Department of Electronics, University of Natal, whose extensive knowledge of computer hardware and the Windows NT operating system was indispensable in rescuing the computer after yet another effort in this research crashed the NT installation or the hard-drive.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AC	alternating current
ALU	arithmetic logic unit
ANSI	American National Standards Institute
AVI	audio-visual interleave
BDFP	block data flow paradigms
BDPA	block data parallel algorithm
CAD	computer aided design
CLIP	cellular logic image processing
COFF	Common-Object File Format
CPU	central processing unit
CRT	cathode ray tube
1-D	one dimensional
2-D	two dimensional
3-D	three dimensional
DC	direct current
DCT	discrete cosine transformation
DMA	direct memory access
DOS	direct operating system
DRAM	dynamic random access memory
DSF	domain skip factor
DSP	digital signal processor
FFT	fast Fourier transform
FLOP	floating point operation
GIF	Graphic Interchange Format
HV	horizontal - vertical (partitioning)
IFS	iterated function system
I/O	input/output
JPEG	Joint Photographic Experts Group
LZW	Lempel-Ziv-Welch
MIMD	multiple instruction, multiple data
MP	master processor
MPEG	moving picture expert's group
MPP	massively parallel processor
MVP	multimedia video processor
PE	processing element

PIFS	partitioned iterated function system
PP	parallel processor
PSNR	peak signal-to-noise ratio
PT	packet transfer
QT	quick time
RGB	Red-Green-Blue
RTSC	reduced instruction set computer
RLE	Run length encoding
RMS	root mean square
SDB	software development board
SNR	signal to noise ratio
SRAM	static random access memory
TC	tran sfer contro 11 er
TI	Texas Instruments
TIFF	Tag Image File Format
VC	video controller
VQ	vector quantisation
VRAM	V-grove random access memory

List of Figures and Tables

1. INTRODUCTION

1.1. Introduction to Digital Image Representation

Data compression is often referred to as coding, particularly in theoretical circles. Information theory is the study of efficient coding and its consequences in terms of speed of transmission and probability of error. Data compression can be viewed as that branch of information theory in which the primary objective is to minimise the amount of data to be transmitted or stored [I ch 13]. A simple characterisation of this generic data compression is that it involves transforming a string of codewords into a new string of bits that contains the same information but whose length is as small as possible. From this generic file compression has evolved a specialised branch, image compression, targeted at files containing image information. This interest in image compression has arisen from the importance of visual information in communicating ideas, information and entertainment. The development of the internet and its ability as a broadcast medium has prompted further development in the field of compression, chiefly due to the implications of file size on the required bandwidth. Bandwidth is at present a valuable commodity due to costs of the physical infrastructure, considerable software overhead and upkeep. Although this may be less of an issue in the future as technology and volumes reduce costs, the demands placed by web broadcasters and expansions into real-time video will ensure that compression will remain an unavoidable component of future developments.

Image compression may also be viewed as the convergence of data compression and image processing disciplines, linage processing in general involves examining an image (usually digitised) and manipulating it to achieve some specific goal. In either case, the required input is the digital representation of the image under consideration.

Typical analogue images consist of a continuum of values within a three dimensional space. The axes of this space consist of two orthogonal positional coordinates and a mutually orthogonal intensity or brightness dimension. Digitising such an image [2] typically involves quantising the information in each of the three dimensions. The spatial dimensions are most conveniently represented as a rectangular grid of points. This convenience stems from the similar arrangement of pixels in a CRT (cathode ray tube) display. Each point may take on one of a set of values to represent the image brightness or intensity at that point. The range of the set of values chosen depends on the required resolution. Common choices include 2 values (1-bit resolution) for binary images, or 256 values (8-bit resolution) for grey-scale images. Other systems are used, and the situation becomes more complicated when moving to colour images (see Section 7.2). In the early days of image processing this type of image representation was referred to as a raster format, from the way the points were scanned row by row, left to right beginning with the top row.

Some calculation reveals the need for compression. A reasonably good quality grey scale image will be quantised into 512 points in each spatial dimension, each at an 8-bit resolution. This will require 512 x 512 = 262 144 bytes, (and triple that, or 786 432 bytes for a colour version in RGB format). In practical terms only 4 or possibly 5 such grey-scale images can fit on a 3.5" stiffy disk. This is less of a concern since the advent of writable data CD's. Perhaps of more concern today is the download time over the internet. Typical shared network connections such as found at universities and other large institutions [1 ch 16] will allow a practical transmission speed of at best lkb/sec per connection. This translates to 256kb/lkb/sec = 256 seconds for the grey scale image (768 seconds for the colour image). Considering that even these transmission speeds may not be achieved due to network

congestion and connection timeouts, these files are clearly unpractically large, and video sequences consisting of hundreds of such frames would be unthinkable.

1.2. Principle of Fractal Compression

Fractal image compression is a relatively new technique for still image compression, which appears to be a promising alternative to the established JPEG standard [3]. Fractal image compression exploits the piecewise self-similarity that is present in real images as a form of information redundancy that can be eliminated to achieve compression. The technique uses partitioned iterated function systems to describe an approximation to a still image using a set of contractive transformations in a complete metric space [4]. This allows a high degree of compression to be achieved. Fractal image compression provides a compression-ratio to fidelity trade-off that is broadly speaking similar to JPEG.

1.2.1. Introduction to Generalised Fractals

The word "fractal" was first used by Mandelbrot to mean a fractured structure possessing similar looking forms at many different scales. Traditionally these are delicate structures produced by an apparently simple algorithm. A key feature of the structure is self- similarity. Consider the representation of the mountain range shown below. Plausibly if one zooms in on the contents of the two smaller circles sufficiently, they will resemble the contents of the largest circle.

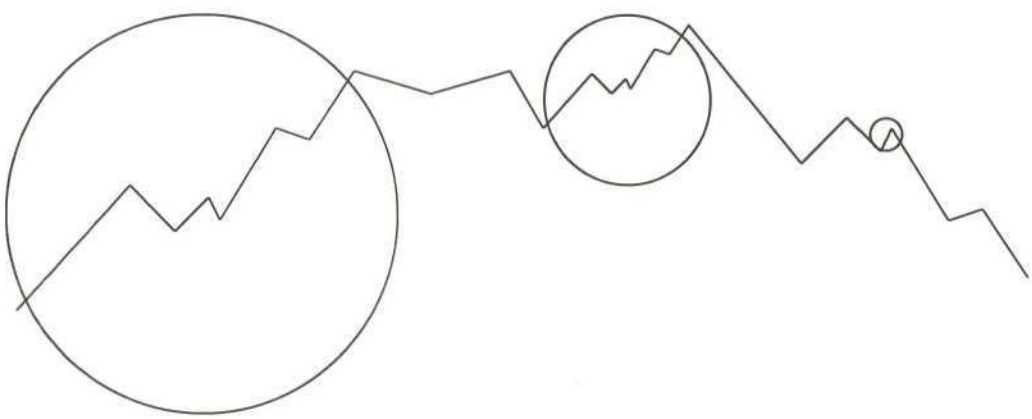


Figure l.l : Self-similarity in naturally occurring structures.

Each portion of the image is actually a reduced copy of the whole. The final image is produced by iteratively applying the algorithm, or transformation, beginning with any initial image. The final image that the process (called an iterated function system or IFS) converges to is called the attractor [4, 5]. Central then to fractals is the concept of these algorithms, or transformations called *affine transformations.* An affine transformation is a function of *n*-dimensional space. It produces some ^-dimensional output which is a combination of a rotation, a scaling, a skew or a translation of the M-dimensional input. A simple two-dimensional affine transformation is

$$W\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} \qquad (1.1)$$

where *a, b, c* and *d* determine the rotation, skew and scaling, while *e* and / determine any translation. These values are the affine coefficients of the transformation. The process illustrated in Figure 1.2 is described by 3 such affine transformations, *w*\, w_2 and w_3. Starting with image (a), each transformation reduces the image to a quarter of its original size, and positions the output so that the combined output W of the 3 transformations are non-overlapping, producing image (b).

$$w_1\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}\frac{1}{2}&0\\0&\frac{1}{2}\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}$$

$$w_2\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}\frac{1}{2}&0\\0&\frac{1}{2}\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}0\\\frac{1}{2}\end{pmatrix}$$

$$w_3\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}\frac{1}{2}&0\\0&\frac{1}{2}\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}+\begin{pmatrix}\frac{1}{2}\\0\end{pmatrix}$$

$$W\begin{pmatrix}x\\y\end{pmatrix}=\bigcup_{i=1}^{3}w_i\begin{pmatrix}x\\y\end{pmatrix} \qquad (1.2)$$

This process is repeated, with each iteration using the previous output as its input (i.e. an *Iterated Function System,* or IFS). This particular example is the Sierpinski triangle.

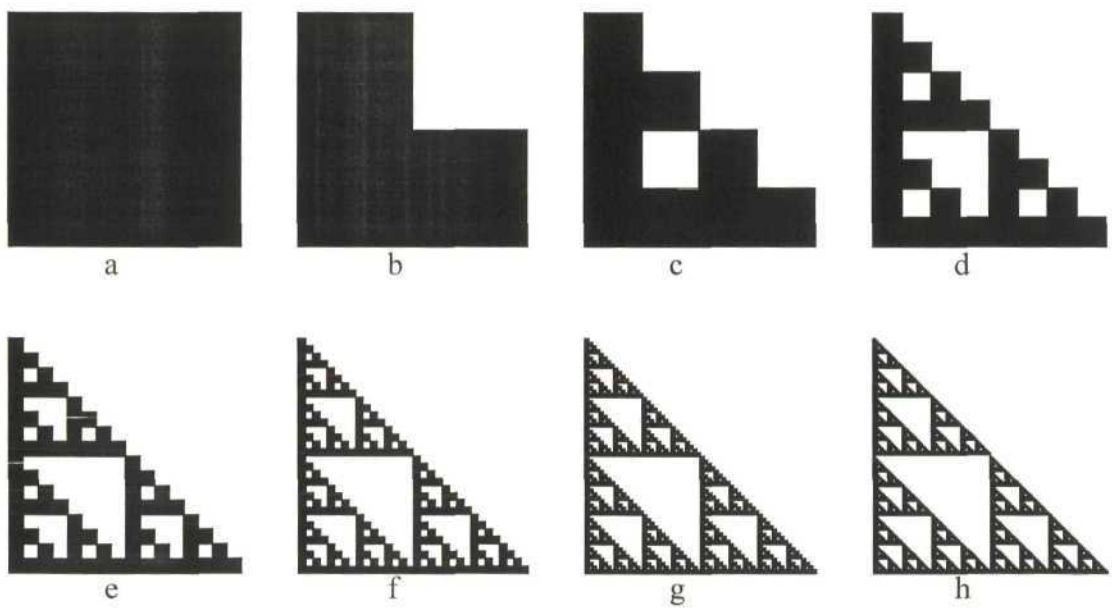


Figure 1.2 : Convergence of Iterated Function System to an attractor

The output converges to some image (h) known as the *attractor* of the IFS. Since the initial image gets reduced in size at each iteration, it will appear only as a point in the attractor, with the result that the attractor is independent of the starting image. This also implies that in order to store the attractor, it is sufficient to store the transformations that describe the IFS. Clearly, if the IFS is to converge to an attractor, each of the transformations must be contractive (i.e. reduce spatial distances). Furthermore, since the attractor essentially consists of reduced copies of itself, it must possess detail at every scale, that is they are fractals. A seemingly complex image with detail at every scale can thus be represented by the coefficients of a simple algorithm. It is therefore tempting to wonder if any arbitrary real image can be

represented as the attractor of a small number of affine transformations, to reduce the storage requirements of the image. If so, it will possess detail at every scale like any other fractal. Hence the image will be decodable at any size simply by scaling the affine transformations.

1.2.2. Self-similarity Applied to Irregular Images

Clearly a real image is not a simple attractor, but all images normally contain some form of self-similarity. To see that this can be so, consider Figure 1.3, the portions of the image inside the two blocks are similar to each other, although not necessarily of the same size or aspect ratio. The modified goal is thus to describe each *part* of an image as a transformation of some other *part* of the same image. This is known as a *Partitioned Iterated Function System* (P1FS). While the previous IFS description assumed a binary image, a grey-scale image may be viewed as a three dimensional object, where the grey level is the third dimension, and brightness and contrast adjustments are incorporated into the transformation.

Figure 1.3 : Examining a real image for self-similarity (From site www.dip.ee.uct.ac.za)

$$\hat{W}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & s \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e \\ f \\ o \end{pmatrix} \qquad (1.3)$$

where s sets the contrast, and o the brightness.

The transformation *Whas* to be contractive in all three directions: *x, y* andz. That implies that |s|<l. If this is not the case, clipping of the image intensity will occur, (some experiments show that if |s|<1.2 for some mappings, the PIFS will still be eventually contractive - see Section 2.4.2).

1.2.3. Image Decoding

Once the image has been completely described by affine transformations, it is said to be encoded. Decoding is straightforward. Start with any initial image of the dimensions required, and iteratively apply the transformations until the attractor appears. Since the initial image gets reduced in size at each iteration, it will appear only as a point in the attractor, with the result that the attractor is independent of the starting image. Convergence to a reasonable resolution is usually achieved after about 10 iterations. This fast decoding time is one of the advantages of fractal compression methods over the more common JPEG technique. However, as will be shown, the encoding process is very slow. This currently tends to limit fractals to one-to-many broadcast applications like Internet web pages, or CD-ROMs.

1.2.4. Image Encoding Technique

Unfortunately natural images are not exactly self-similar, and so no simple algorithm exists for directly calculating the affine coefficients. The only alternative is a tedious iterative approach.

The input image is divided into a collection of range blocks which completely cover the image, and another set of domain blocks. The domain blocks are usually larger than the range blocks to ensure spatial contractivity. The aim is then to find for each range block, an affine mapping from one of the domain blocks. Once all of the range blocks have been covered by suitable mappings, a PIFS will have been formed.

In a simple system, finding the domain block that best matches a particular range block will consist of comparing that range block to each domain block for each possible relative orientation. For square blocks, *eight* comparisons are thus required. (There are initially four possible relative orientations. Flipping one of the blocks will yield another four possible comparisons.) Once a "close" match is found, the affine coefficients for that mapping may be computed (see Chapter 3, Section 3.5). When referring to two blocks "matching" each other, **clearly** an exact match is unlikely, rather the goal is to find matches **that** are "close" in the context of some metric. This is described in Section 3.5.

1.3. Contribution of this Work

Previous work in the field of fractal image compression [6, 7, see Chapter 2] has largely concentrated on theoretical approaches to achieving higher compression ratios and improved fidelity. The practical benefit of an image compression technique, however, depends on its usability. In the past this has been seriously compromised by the prohibitively long compression times. The fractal process is, however, inherently asymmetrical. That is, the decompression process is relatively undemanding computationally, **while the** compression process consists of massive systematic searches [8]. The nature of the searches and the characteristics of images would suggest that algorithms for parallel searches could be implemented. Rather than duplicate the existing body of work, it was decided that the main effort of this research would be directed at such a novel application of a parallel processing

system as an alternative to conventional solutions to the demands of the compression problem. As such, most of the investigations performed revolved around a "standard" algorithm adapted to the requirements of parallel processing. Some of the refinements and adaptive algorithms reported in these other published works were examined, and their findings distilled and implemented.

Most of the research previously done in the field of fractal image compression has concentrated on grey-scale images (See Chapter 2). The system developed during this work has also evolved into a compressor capable of dealing with colour images, and a video sequence compressor.

The primary outputs of the research are as follows :

- Application of a parallel processing system to fractal image compression (1997)
- An appropriate segmentation of the algorithm over these resources (1997)
- Extension of this fractal image compressor handle colour images (1998)
- The development of a video sequence compressor (1998)

At this start of the project, the current benchmark desktop computer was the Pentium 1 type, usually with a processor speed of 133MHz to 200MHz.

1.4. Alternatives

It is important to note that a great many general file compression algorithms exist. During the course of this research the fractal scheme is compared principally to the Graphic Interchange Format (GIF) and Joint Photographic Experts Group (JPEG) formats because they are in the widest use, and must therefore be seen as the competition faced by attempts to commercialise fractal image compression. A discussion of these appears in Section 2.8, and information on other compression formats appears in Appendix C.

The most basic image format is the raw type (file extension .RAW) (and related bitmap formats). These is the raster formats described above, and consist of a sequential listing of pixel values, described row by row, beginning with the top row, working left to right. The file size depends only on the image dimensions and grey-level resolution (number of bytes used to describe each pixel). File sizes tend to be large, and are particularly inefficient when considering simple images. The raw file version of an image is usually used as the reference against which compressed forms are compared to determine the compression ratios of these formats, and are still the preferred route if maximum resolution is required, for example desk-top publishing.

The GIF format performs bitstream compression (so named since the file content is generally sequentially examined) by examining the raster format for runs of identical pixels. This produces a lossless compression, and the degree of compression depends on the amount of local redundancy found within the bitstream. However, redundancy is more likely to be found in a two-dimensional group of contiguous pixels than a long sequence of pixels in one row of an image. This local redundancy may only be exploited if the image is examined "patch-wise". JPEG works by performing examinations of 8 x 8 pixel blocks of the image. A two dimensional discrete cosine transformation is performed over the block and only those frequency components containing a significant amount of energy are retained. Fractal image compression also works "patch-wise" over the image and like JPEG reduces the redundancy in a manner sympathetic to the image content. The move to considering areas within an image that are not necessarily exactly identical, but may be considered similar, implies the transition from lossless compression lossy compression. This implies the compromise that the compressed and decompressed image will not be exactly identical to the original

uncompressed image. Instead, it will approximate the original image with some degree of error. This is the crucial area of interest in lossy compression - what forms of information may be regarded as redundant in the context of the human visual system. The aim is obviously to selectively economise the bitstream and still achieve a satisfactory compression ratio. The successfulness of the compression scheme will depend largely on the sophistication of the algorithm which arbitrates which forms of redundancy may be eliminated.

1.5. Conclusion

Digital image representation tends to be inefficient and requires a large amount of storage space or transmission bandwidth for high resolution representation. However, real images contain much redundancy which can be exploited in a variety of means to improve the efficiency of representation. This has become particularly important in view of the volume of images transmitted over the internet, and the limited bandwidth resources available. Many compressed image formats have evolved to address these needs, with GIF and JPEG gaining particular prominence. GIF is a high resolution format, but suffers from moderate compression ratios. JPEG at the other end of the complexity spectrum offers good resolution and compression ratios by eliminating low energy high frequency information to which the human visual system is relatively insensitive. Fractal image compression represents an attractive alternative to the established formats. It exploits the local partial self-similarity that may be identified within images as a form of redundancy that can be eliminated. This process is based on identifying affine mappings between these areas exhibiting similarity and storing only the parameters of these mappings in the compressed file. The successfulness of this approach hinges on identifying the best possible affine mappings. Thoroughness requires an exhaustive, computationally intensive search process. The repetitiveness of this search process dictates extremely long compression times, and hence hampers the commercial viability of this format. A parallel processing approach is proposed in this research with the aim of addressing the traditionally high compression time. In balance though, fractal schemes potentially allow faster decompression than JPEG, and so present exciting prospects for broadcast applications such as the internet.

It may be stated then that fractal image compression holds a unique position in the complexity - compression ratio tradeoff. It offers compression ratios, comparable to JPEG at a given SNR point for suitable images. It labours under a severely intensive encoding algorithm, orders of magnitude slower than JPEG, but it has a fast, simple decompression rivaling that of GIF for speed, and even surpassing JPEG in this regard.

This then gives the thrust of this research - if the encoding speed which is fractal image compression's Achilles heel can be addressed, the scheme has many positive features that may then find practical application.

2. LITERATURE SURVEY

2.1. Historical Background

Fractal techniques have been applied to image processing only relatively recently. They are based on the mathematical theory *of Iterated Function Systems* (IFS), the aim of which is to represent an image by means of a set of contractive transforms possessing a fixed point close to that image. The so-called fixed point theorem (see Section 3.2, Equation 3.5), often attributed to Banach [9] guarantees that in the context of a complete metric space, the fixed point of such a set of transformations may be produced by applying the transformations iteratively to an arbitrary element of that space. (A mathematical justification appears in Section 3.6). This theory was initially developed in 1981 by John Hutchinson [10], Michael Barnsley, researcher from Georgia Institute of Technology, then observed the possibility of using this IFS theory to encode images, as many fractals describable by an IFS have a highly detailed, "natural" appearance. These results appeared in the *bookFractals Everywhere* [10]. The collage theorem (see Section 3.2, Equation 3.9) which stipulates conditions on these transformations was first developed in this work.

This presented the intriguing possibility that since fractal mathematics could be used to generate "natural" images, the converse may also be true, providing a means of compressing "real" images. This reverse engineering of a given image to find a describing IFS that can generate original (more or less)[4, 9], is known as *the inverse problem.* Having seen the potential, Barnsley then left the Institute and founded Iterated Systems Incorporated with Alan Sloan. They were granted a patent for the commercialisation of the IFS theory. In January 1988, he announced through an article in BYTE Magazine[l I] some calculations of high compression ratio, however the article did not offer the solution of inverse problem.

Ironically, it was one of Bamsley's PhD students, Arnaud Jacquin, who developed the first practical algorithm for fractal image compression of real images. In March 1988, he published a modified scheme for representing images called *Partitioned Iterated Function Systems* (PIFS) [8]. This work represented the foundation of subsequent research on fractal compression using an automated process to calculate the mappings. A PIFS differs from an IFS in that each of the individual mappings take only a subset of the image as an input, rather than the whole image. This may be seen as exploiting a piece-wise self-similarity, rather than an explicit complete self-similarity. Clearly the description of a real image by a PIFS is much more achievable than with an IFS. All contemporary fractal image compression schemes have been based on Jacquin's breakthrough. These schemes have been designated as contemporary schemes [12] as opposed to the classical schemes based on Barnsley's approach. Nevertheless, until recently, a deterministic solution to the inverse problem has remained elusive. This is discussed below in Section 2.4.3.

Within the basis provided by Jacquin's scheme, there are clearly numerous design choices that need to be made in implementing a strategy. They may be grouped as follows [9]:

a) The way in which the image is partitioned into range blocks
b) The way in which the image is partitioned into domain blocks, and in fact which of these possible blocks are selected to form part of the domain pool
c) The nature of the transforms used to map possible domain blocks onto the range blocks
d) The search strategy used in finding a suitable domain block for any particular mapping
e) The space in which this search is performed - this implies a choice of metric for determining the suitability of a possible mapping.
f) The representation and quantisation of the parameters describing the transformations.

2.2. Image Partitioning Schemes

Conventionally regular partitions of square or rectangle blocks are used because it allows block boundaries to be chosen to coincide with the raster arrangement of pixels on a conventional CRT. The P1FS theory is equally applicable to other block shapes, though these may prove to be less practical.

2.2.1. Fixed Block Size Partitioning

This is the most basic system. The range blocks are of a fixed size, often square, say n x n pixels. The domain blocks are then chosen to have another fixed size. They are chosen to be larger to force the required contractivity conditions (see Section 3.3). A typical choice is to select the domain blocks to be In x 2« pixels. In its most basic form, it is rather non-optimal since equal processing time and byte space in the compressed file is devoted to both featureless areas of the image and areas of high detail. A fixed block size scheme may be made adaptive by concentrating more encoding time (computational effort) and more bits in the compressed file to regions of higher detail. This is not ideal, and is easily surpassed in performance by schemes employing an adaptive block partitioning.

2.2.2. Quadtree Adaptive Partitioning

The classical adaptive scheme, called *quadtree partitioning* [4, 8, 12], starts out trying to use larger range blocks to cover the image (this will result in fewer affine mappings, and hence a greater compression ratio). In areas of higher detail in the image, if a matching domain block cannot be found to cover the range block to a predetermined tolerance, that range block is divided into four smaller blocks. This block size reduction process may be repeated until the entire image is covered to the desired accuracy. This is a form of recursive splitting which results in a partition that may be represented by a tree structure. Exploiting this allows the partitioning details to be stored compactly. This allows smaller blocks to be used in areas of higher detail, promising a better approximation in those areas because contiguous pixels in a real image tend to be highly correlated. This necessarily implies that more bits will be needed to store the encoding in those areas.

Figure 2.1 (a) : Domain blocks of size In x 2«

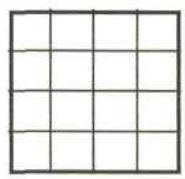


Figure 2.1 (b) : Range blocks initially n x n

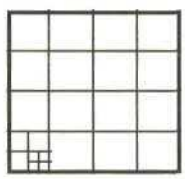


Figure 2.1 (c) : Reduced range blocks

Jacquin's original scheme [8] used a variation of this technique, but restricted to two possible range block sizes.

2.2.3. Horizontal-Vertical Adaptive Partitioning

The quadtree scheme makes no attempt to select the pool of domain blocks to be used in a content-related way. This may be alleviated [4ch 1] by choosing a large domain block pool, but this is counter-productive towards encoding time. Additionally, the boundaries between range blocks bear no relation to edges of structures that may be present within the image information. The horizontal-vertical (HV) partitioning scheme increases the flexibility of the range boundaries. A rectangular image is recursively partitioned by either a horizontal or a vertical partition to form two new rectangles (see Figure 2.2. below)

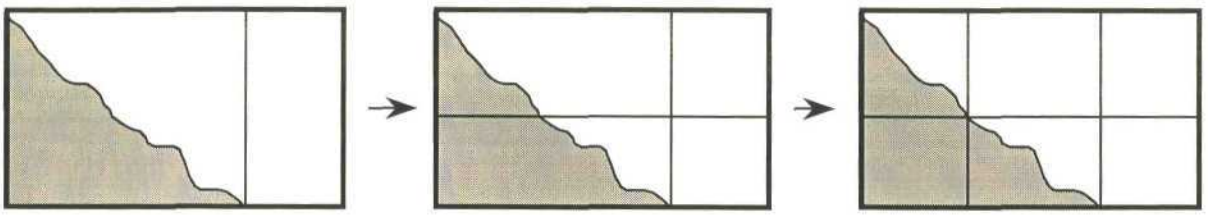


Figure 2.2 : HV partitioning scheme

The partitions are positioned in such a way as to reinforce self-similarity. This may be done by placing boundaries on edge features as in Figure 2.2. The partitioning is repeated until each rectangle can be covered with a domain block to a desired tolerance.

2.2.4. Other Block Partitions

Among many other proposed schemes, [4] suggested a hexagonal partition, and a Delaunay triangulation based partition has been attempted by [13]. In general they do not appear to offer performance gains over the more conventional rectangular partition schemes. Additionally they suffer from difficulties that arise when contraction operations require interpolating between pixels. Thus square block partitioning (principally fixed size, but also quadtree) was selected for this project.

2.3. Compression Process Acceleration

Within the basic constraints defined by the choices in block geometry, there are two (often mutually exclusive) goals frequently pursued in research. Improving compression speed, and optimising the affine transformations produced in the interests of an attractor more closely approximating the original image. The two are obviously interrelated, but may be handled separately for the purposes of this discussion.

2.3.1. Domain pool selection

One of the most obvious means of reducing compression time is to reduce the number of possible domain blocks that are considered whilst searching for the affine mappings. In other words, the size of the domain pool searched is reduced. The optimum set of affine maps will result from an exhaustive search where all possible domain blocks are considered. However, the number of block comparisons that need to be performed, and hence the compression time can be significantly reduced by selectively eliminating possible domain blocks from the

domain block pool. The aim is to do this in such a way that there is a minimal degradation in attractor fidelity. A maximally sized domain block pool results when the blocks are drawn from overlapping areas of the original image positioned so that each block is shifted only by one pixel row relative to the previous block in either the x or y direction. This results in a prohibitively long compression time (see Section 3.8) and it is actually not reasonable to include all these possible blocks since a pair of domain blocks taken from locations shifted only by one pixel will not differ considerably from each other, and on this basis only everyn* block is selected. This is a common method of initial domain pool reduction employed in many works [4 ch3, 12], and referred to as the "domain skip factor", or DSF in this work. It is an example of a method of domain pool reduction where it is also possible to specify the domain block required in a particular mapping with fewer bits. This has a beneficial impact on the size of the compressed file information. Section 6.3.1. contains results for the tradeoff between DSF and attractor PSNR, as employed in this research. (Visual results are given in Figures 6.6 and 6.8.)

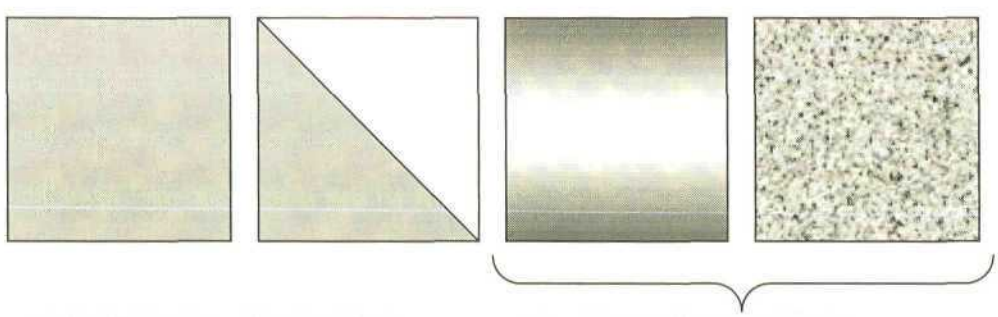
Some [14] have proposed that local domain pool refinement is also feasible. Generally this is done on the premise that for any range block, a suitable (if not optimum) domain block will lie spatially close to it. However there is some disagreement on this point and [4 pp69-71] has demonstrated that there may generally be a low correlation between the spatial positioning within an image of a range block and that domain block which optimally maps to it. (dependent on the image under consideration). (The distribution of domain-range pair distances is shown to be approximately equal to the theoretical distribution between two randomly chosen points in the image.)

2.3.2. Search Patterns

The most basic means of searching for the domain block which best maps to a particular range block is to perform an exhaustive search *within the domain pool* after it is reduced to a manageable size as in Section 6.3.1. This requires performing every possible block comparison, and clearly compression time could be reduced if the search could be more intelligently directed.

2.3.2.1. Block Classification

This was recognised by Jacquin in his original work [8]. He proposed that the domain block pool be divided into groups, called classes, where the members of each group exhibit some common trait. Each range block may then be compared only with the members of that class which it most closely resembles. Jacquin's original scheme defined three classes he called *shade blocks, edge blocks* and *midrange blocks.*



a) shade blocks b) edge blocks c) midrange / texture blocks

Figure 2.3 : Jacquin's block classification

Shade blocks are essentially "smooth" with no significant gradient in grey level. Edge blocks have evidence of a grey-level discontinuity suggesting a piece of an object boundary, whilst midrange blocks have modest grey-level gradient, but no edge discontinuities. This third class may also be interpreted as blocks containing modest texture [4 ch 3, 15].

The idea of classifying blocks according to texture criteria has been developed differently in [16] and by this author in [17]. It was observed that a significant number of blocks within many real images are "smooth" in that they possess little variation in grey level. (Justification appears in Section 6.3.2.) In other words, they contain little "texture" or "edge" information. The second moment of the pixel intensities within the block gives an a priori estimation of which domain blocks are sufficiently featureless that they are unlikely to be used for affine mappings. [17] then suggests that the remaining domain blocks may be further sub-classified according to the degree of variance present in them. Additionally, any range blocks with a low variance must contain pixels with values very near to each-other, and that block may be represented simply by the average grey-level of the block, without visible loss of quality. If a range block may be stored simply as a grey level rather than an affine map, then there are fewer coefficients to store for that block, leading to greater compression ratio for the image. Secondly, if a range block is found to have a low variance, it is not necessary to search for an affine mapping at the encoding stage, so thousands of block comparisons are avoided, leading to improved compression speed. This is the main search-directing strategy employed in this research.

Another popular scheme [18] groups blocks according to the relative brightness of the four quadrants of the block [4]. Figure 2.4 illustrates this idea. Some thought will confirm that by means of rotating and flipping a block, it may be made to fall into one of the three canonical classes, hence it is not necessary to test all 8 symmetry operations.

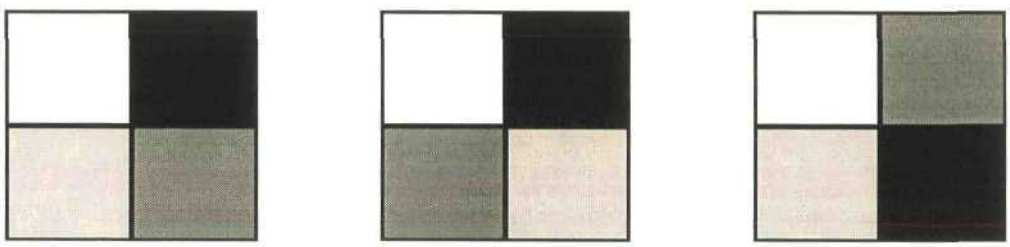


Figure 2.4 : Three classes of image blocks as determined by local brightness

This may be extended [4 ch 3] to treat these three classes as major classes each with up to 24 sub-classes. These sub-classes arise from ordering 4 quadrants with respect to the variance of the pixel intensities. This kind of fine classification approaches the tradeoff limit where the classification calculation overheads negate the benefits of the highly directed search. [4ch 9] shows that more than these 72 sub-classes is not beneficial.

2.3.2.2. Tree Searches

The non-unified search spaces imposed by classification methods have the achieve speed gains by possibly excluding some optimal pairings in favour of sub-optimal, more quickly found, pairings. Other means of directing the search for a suitable domain block have been proposed. [19] describes an approach of organising the domain pool into a highly structured tree. It can be used in conjunction with block classification schemes.

2.3.2.3. Invariant Space Searches

A drawback with the above search strategies is that comparisons must be made between range blocks and a *transformed* version of each domain block produced by an admissible transformation. Thus each of the many required comparisons cannot be made before the affine transformation for every domain block in the pool has been calculated. This is the where the bulk of the computing effort is spent. It may be observed that if for each domain and range block, an appropriate invariant representation can be constructed, the distance between a pair of blocks may be calculated directly. Unsuitable domain blocks may then be eliminated more quickly. Note that the transformation to an invariant representation need only be performed once for each domain block at a preliminary stage.

An n x n block of pixels may be regarded as a position vector in an abstract;?" dimensional position space. Each distinct point in this space represents a different possible block. Fractal image compression then bears resemblance to Vector Quantisation[1] (VQ) but with the important distinction that a codebook is not required because the contractivity conditions allow iterative decoding from an arbitrary image.

There are several ways of translating a block of pixels into a position vector in some space [20]. Computing the discrete cosine transformation (DCT) will yield a vector in frequency space [21, 22, 23], similarly the Karunen-Loeve [24] or Taylor expansion coefficients could be calculated. The simplest approach is to use the pixel intensity values directly [20, 25]. This, and the DCT option have received the most attention in the literature, and are presented below.

2.3.2.4. Position Space Vector Treatment

[20] presents a position space vector treatment where the pixels within each block are normalised to a fixed mean and variance to facilitate comparison of the vectors. Vectors near to each other represent blocks which are *structurally* similar to one another under some affine transformation. A tree structure is used to index the vectors within the space. Quick comparisons are possible because an unsuitable pairing will be apparent after comparing only a few of the vector elements.

[25] presents a related approach using a geometric construction. Each domain block vector generates a plane under a set of admissible transformations. The spaces constructed are of dimensionality 3 or less, allowing solid geometry to be applied. The distance minimisation requirement can then be restated as finding the closest domain "plane" to each range block vector. The calculations revolve around the angles between vectors and planes (or their normals) which principally involve simply computed inner products.

' Vector quantisation treats a block of pixels as a vector with elements given by the pixel intensities. The image is encoded by representing each of the vectors by their closest matching vector from a pre-defined codebook. This conversion requires comparison with all the codebook elements, usually using the Euclidean distance as a similarity measure. The compressed information file contains only references to the codebook elements. The disadvantage of this over fractal techniques is that either the decoder must have a priori knowledge of the codebook, or the codebook must be transmitted with the compressed file, effectively reducing the coding gain. Additionally, use of a standard codebook (usually generated from a set of training images) provides no guarantee that high correlation mappings may be found for a particular image, whilst fractal techniques effectively use an implicit codebook which by the self-affinity assumption must provide for superior correlation mappings.

2.3.2.5. Search in Frequency Domain

[22] proposed a scheme in 1994 where the image is divided into 8 x 8 pixel range blocks and 16x16 pixel domain blocks. These are then transformed by DCT. Range blocks are then classified by their AC coefficients. "Simple" blocks are treated as featureless and coded by their DC value. For other range blocks a search is performed in the transformed space for a suitable domain block. The affine transformations are modified for use in the frequency domain and exploit the invariances of the space. In addition, an error map is Huffman encoded and transmitted with the affine coefficients. The drawback of this method is that decoding requires both iterative decoding in the frequency domain, and then inverse DCT to transform the image back to image space. Thus the decoding process is significantly slower than conventional fractal implementations. The results presented suggest that a higher PSNR may be achieved using this method. In 1996 [23] proposed a modified version where the energy-packing property of the DCT is exploited. The transformed vectors are then normalised to a canonical representation and inserted into a lexicographically ordered array to facilitate efficient binary searching. This search for a suitable contracted domain block is quick, because symmetry operations are quickly performed in the DCT domain, and normalisation of the transformed vectors allows simple checking of the grey-level contractivity. A factor 2 improvement in compression time is reported. [21] adopts a simplified version of this approach, performing the search in the DCT domain. The comparisons consider only the low frequency components of the transformed block (which are the most significant). This allows for even faster encoding. The paper then presents a means of determining the grey-level contrast and brightness affine coefficients in the frequency domain. [26] presents a related approach, performing a Fast Fourier Transform (FFT) over the blocks and using the convolution of these transformations as a measure of similarity.

Common to these schemes involving transformations to other spaces is the benefit that the dimensionality of the search space is reduced, either by construction or by truncation and approximation. Speed gains are achieved because the order of the computational complexity is reduced.

2.4. Attractor Optimisation

2.4.1. Overlapping Partitions

Common to all hard-boundary partitioning schemes is the problem of step discontinuities in grey-level at partition boundaries leading to block artifacts in the decoded image. These artifacts arise because at decode time the range blocks are handled independently of one another, and there is no bound imposed on the grey level difference a the boundary between blocks. One of the most basic improvements to the perceived collage that has been proposed [27, 28, 4] is to allow for the range blocks to be overlapping. A sample range partitioning is illustrated below in Figure 2.5. The principle is also applicable to blocks of other shapes.

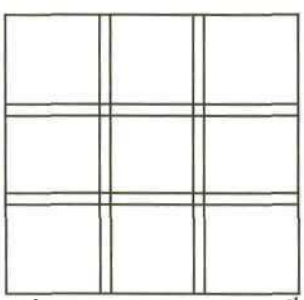


Figure : 2.5 : Representation of a range partitioning al owing for overlapping blocks.

Typically [4] during the decoding process the pixel values in the overlapping areas are obtained from the weighted average of the pixel values produced by the transformations mapping into that area. [27] reports very little quantifiable improvement in PSNR for such a scheme involving square range blocks (overlapping by a number of pixels proportional to the block size), but [28] presents experimental results showing a 0,5 to ldB improvement in PSNR. [28] also develops a modified contractivity theory to support the idea.

A related idea applicable to standard (non-overlapping) partitioning schemes is to perform post-processing [4, 12]. Pixels adjacent to block boundaries may be adjusted by a weighted contributions from neighbours in adjacent blocks to produce a smoothing effect. [4 ch 3] reports a slight PSNR improvement for lower fidelity/bitrate applications, but possibly a degradation for high quality attractors (produced by exhaustive search type compressors).

2.4.2. Eventual Contractivity

A sufficient but not necessary condition for contractivity under the supremum metric can be guaranteed by stipulating that the grey level contractivity and spatial contractivity factors lie strictly between -1 and 1 for every affine mapping (see Section 3.3.). It has been known for some time that allowing some of the affine maps to have grey-level contractivity factors as large as ±2 can be beneficial. Results [18, 4 ch 3] have been promising, yielding improvements of up to ldB in attractor PSNR. In such a case, a particular transformation may not be strictly contractive, but the overall collage can still *beeventually contractive.* The basic problem hampering development in this area was that for many years there existed no explicit bounds on the grey-level contractivity factor to guarantee eventual contractivity. This changed in the early 1990's with the concurrent works of several authors.

[29] published an image space vector treatment where contractivity conditions where derived from the spectral radius[2] of the affine transformation matrix. This lead to the presentation of tighter bounds for allowable contractivity factors. A 1 to 2dB improvement in reconstructed image PSNR results, although the calculation of the eigenvalues imposes an encoding time penalty. [30] then presented a development also based on spectral theory. The improvement offered over [29] is that of a faster method of determining the contractivity factors directly from the norm, rather than the eigenvalues, of the transformation matrix.

The processing overhead imposed by these schemes was deemed contradictory to the aim of this project - namely minimising compression time even at the expense of the loss of some fidelity. Consequently, a ceiling of ± 1 was imposed for contractivity factors.

2.4.3. The Collage Theorem and the Inverse Problem

The problem may be summarised [31] as finding a suitable metric space in which to represent the image, a suitable metric over this space and a suitable contraction mapping within this space. (A mathematical discussion of these issues appears in the next chapter). This main problem has more than one solution, allowing freedom to search for different kinds of optimality. Generally speaking, the attractor is the fixed point of a set of affine transformations, chosen so that the attractor resembles the original image as closely as possible. Although the attractor is recovered by iterative application of the set of contractive

[2] The spectral radius of a linear transform is the maximum absolute value of the eigenvalues of that matrix.

transformations, there is no simple expression [4, 8] for the attractor in terms of the affine coefficients. Thus it is not possible in the general case to optimise those coefficients to make the fixed point as close as possible to the original image. As a result, the distortion in the recovered image cannot be directly optimised. The standard approach is to minimise the collage error[3] [32, 12] since this arises from the individual error contributions of the different affine mappings, for which least-squares optimisation is possible. As the contractive mapping fixed point theorem shows, (see Section 3.2.), the collage error provides (rather loose) bounds for the attractor distortion. Tighter bounds have been presented for less general cases of fractal image compression. For example, [33] imposes restrictions on the allowable domain skip factor and stipulates normalisation of block pixel values.

A pair of papers (1994), [31] and [34], claim a rigorous presentation of a solution to the inverse problem, albeit not the general case. It is applicable to theory developed using a continuous function over two-dimensional space representation of images (as presented in Section 3.4.). Again, restrictions are imposed such as non-overlapping domain blocks, and letting the number of mappings tend to infinity. Nonetheless, by considering further iterates of the mapping (than the single one used by the collage theorem), tighter upper bounds are developed. A sample algorithm for implementing the theory is presented in [34]. The reported compression times (for a self-confessed un-optimised algorithm) are disappointingly slow, ranging from several hundred seconds to thousands of seconds on a 1990's RISC workstation[4]. Nevertheless the work is significant. In 1997 [35] presented another (somewhat artificial) scheme where the main restriction is that the domain block for any particular range block is predetermined by their relative location. This a priori knowledge of mapping pairs leads to a sub-optimal collage, but facilitates conventional multivariate optimisation for the affine coefficients.

The lack of generality, practicality and consensus of works in this area prompted the decision to implement in this research a classical approach based on minimising the collage error on a least-squares basis.

2.5. Hybrid Techniques

In addition to the above schemes which ultimately are all developments of Jacquin's original theory, many hybrid schemes combining elements of fractal compression with discrete cosine transforms, vector quantisation and others have been proposed. [36] proposes a block-based scheme using VQ for blocks of high detail, polynomial approximation for areas of intermediate detail and fractal compression for areas of high self-similarity. This is an interesting implementation, but it suffers from heavy overhead from managing the algorithms. [37] and [38] by the same authors describe block-based encoder in which each domain block is compressed either by fractal techniques or VQ. The blocks are assessed on an individual basis, and grouped into classes centered around clusters. Blocks (again represented by vectors) are encoded with VQ if they resemble one of the clusters sufficiently closely, otherwise they are encoded by affine mapping from the best matching cluster. This also serves to speed the affine map search since the domain search pool is restricted to a small cluster. The algorithm is necessarily complex, but superior fidelity and encoding/decoding times are reported. [39] gives results using fractal compression for blocks containing edges and smooth variation, and DCT for detailed areas. This appears to be one of the most likely commercial applications of fractal compression - complementing the weakness of DCT at

[1] The collage error is essentially the distance between the target image and one iteration of a test mapping (assessed piecewise for each affine mapping). This approach allows least-squares optimisation. Collage issues are discussed in Section 3.2., Equation 3.9.
[4] IBM 355 POWERstation, Fortran code

handling hard edges. (These methods are distinct from using a DCT search space as described above in Section 2.3.2.5.).

In the Jacquin-derived compression schemes, the brightness (grey-level offset) scaling coefficient in the affine mapping is generally a scalar constant (the coefficiento in Equation 1.3.). It has been shown [40] that the use of a linear expression of the form $o = a1+a2i+a3j$ is beneficial to decoded PSNR. [19] gives a modified theory to account for the extra parameters which need optimisation. As with most departures from Jacquin-derived schemes, there is a severe computational complexity penalty.

2.6. Fast Decoding

The other aspect of fractal image representation deserving of attention is obviously the decompression or decoding process. Traditional fractal decoding is an iterative process, and the number of iterates required for adequate results varies with image content, and can be as high as 10. All the previously described refinements of the compression process are compatible with this decoding system, but it is intuitively a sub-optimal approach. There are however several alternative, faster decoding algorithms [9]. Most offer these speed gains independently of the compression process, and do not require any modifications to the compression process. They cannot offer any improvement in the fidelity (either quantitatively or qualitatively) of the decompressed attractor- they simply provide a means of extracting it quicker. The focus of this work is on the compression process, so a discussion of fast decoding techniques has been relegated to Appendix B.

2.7. Parallel Architectures and the TMS320C80

Parallel processing has been a specialised branch of computing since the beginnings of computing

Computer architecture has evolved from the original flat organisation to a more complex organisation based, for example, on memory hierarchy and instruction level parallelism, but the programming model upon which high level programming languages (eg C, Pascal) are based has not evolved accordingly. This has forced programmers to resort frequently to programming directly in assembly language to access and bridge the gap between the architecture and the programming model [41, 42]. At each stage of architecture development, a certain maximum level of hardware performance has been available from a single processor, as dictated by the state of the art in technology. Typically the leading edge processors have always been prohibitively expensive for commercial applications. Since the early days, it has been recognised [43] that combining several sub-state-of-the-art processors into a parallel arrangement could yield a performance exceeding that of cost-no-object single processor solutions.

Parallel computing platforms may be positioned within a spectrum of degrees of parallelism, ranging from the traditional (early) fine grain massively parallel paradigm [43] to the current RISC-based moderately parallel paradigm such as the TMS320C80 [5, 44] under examination in this work. Unfortunately, parallel architectures have generally proven even less suitable for available high-level programming languages.

Much of the development work in parallel architectures has had impetus from image processing applications [42]. The size and regularity of the data structures involved (typically 2 dimensional arrays of pixels) and the nature of the algorithms involved, typically repetitive and acting on blocks of data spatially close to one-another within the image prompted the development of massively parallel (also known as hyperfine) architectures. The blocks of the image have also been called cells [43], hence the term cellular logic image processing (CLIP). Massively parallel processors (MPPs) consist of large numbers of low level processing elements (PEs) associated with each other in a grid structure of data paths. Typically, data sharing is only with nearest neighbour PEs. This can cause data flow bottlenecks, and cellular logic MPPs are generally designed and optimised for specific tasks, and achieve very high throughput on those tasks. There are two main arrangement strategies - processor arrays which provide data flow parallelism, and pipelines providing operational parallelism. Hybrids of the two also exist. Within these broad groups, variations result chiefly from the structure of the data flow paths. Typical structures [43] include pyramid arrangements, linear processor arrays and ring structure. Generally, the capabilities of MPPs are too specific to implement a complete algorithm, so they may be associated with other more general processing elements. The cost, dedicated nature and programming complexity of massively parallel systems disfavoured their diffusion into real-world applications. Additionally, performance ceilings were reached imposed by the need for communication and synchronisation between processors [45],

In the 1980s the technology of general-purpose RISC based microprocessors, driven by the demands of desktop workstations, reached an image processing performance exceeding thai of previous dedicated architectures. [42] presents a thorough quantitative evaluation of current platforms to support this. These developments slowed research into dedicated massively parallel architectures, and left a niche for a new breed of general-purpose moderately parallel architectures. These may be regarded as block data flow paradigms (BDFP) [45]. The parallelism is sufficiently coarse grained as to remain general purpose, and to stipulate the implementation of a block data parallel algorithm (BDPA). In this scenario each processor performs a number of distinct tasks on a block of data, rather than one specific low-level task per processor as in MPPs.

At the inception of the project (January 1997) the Texas Instruments TMS320C80 represented such an implementation [44] of parallelisation of (marginally) sub-state-of-the-art processors. It was supplied in what was essentially a beta version in an evaluation board. It represented the state of the art in parallel processors developed for commercial applications, and was available at a cost of some US$ 3500 (although the evaluation versions were supplied free of charge). It was one of the first implementations where a design effort had been made to simultaneously address the software problems [44] that had traditionally made parallel architectures, especially RISC based, a programmer's nightmare. Whether this was successful is open to some debate. The accompanying software environment requires low-level programming experience to use (see Section 4.2), but at least it does facilitate the sort of low-level manipulation that allowed optimal use of the considerable resources (see Section 5.2.). As such the environment represents a hybrid of high level-programming convenience, and low-level manipulation flexibility. This has spread to other platforms [41]. It is a testament to the progress in processor development that by mid-1998 TI had released *asingle* processor DSP capable of higher throughput (assessed in operations per microsecond) than the TMS320C80 (featuring parallelism at every conceptual level).

[45] presents general theory applicable to the partitioning of image processing problems over such parallel processor architectures. The focus is on minimising interprocessor communications and data transfers by means of BDPA, with specific applicability to BDFP. This has particular relevance to this project, and provides support for the decisions made in Section 5.2.1. The concept may be summarised as treating an image data array blockwise so

that an algorithm may be developed which is sympathetic to the strengths and weaknesses of a BDFP type hardware, such as the TMS320C80.

As with other parallel architectures, the processors within the TMS320C80 may be arranged for data processing parallelism or operational parallelism (pipelining). Since the processor's introduction, there have been several implementations of each type reported. [46] describes an MPEG2 video decoder implementation on the TMS320C80. The algorithm is divided into high level phases which are pipelined. Each stage is then parallelised using the processors of the TMS320C80. The implementation reportedly achieves real-time decoding of 6Mbit/s data streams. [47] describes a true parallel BDPA implementation of scan conversion as needed by 3-dimensional rendering[5] algorithms. [48] reports a TMS320C80 implementation of a H.263 compressed video format[6] encoder. The coarse granularity of the architecture is recognised and the implementation is parallelisation of tasks. This is not pipe-lining, but rather contemporaneous execution of different tasks requiring quasi-simultaneous completion. As in this work, most of the code is written in C, while time-critical aspects are optimised in assembly language. Real-time operation was not achieved, but aspects still requiring optimisation are identified. [51] discusses issues of prototyping code on a modern moderately parallel processor.

2.8. Alternative Compressed Image Formats

It is important to note that a great many general file compression algorithms exist. This discussion is not intended to be exhaustive. Only those systems which are in common use, and may therefore be seen as the competition faced by fractal image compression are considered. A more complete discussion of common formats is available in Appendix C.

The following discussions refer to grey-scale images. With extensions, this applies equally to colour images. The most basic image format is the raw type (file extension .RAW). This consists of a sequential listing of information about successive pixels. Usually the pixels are described row by row, beginning with the top row, working left to right. Each pixel may be represented by a single byte (256 grey level resolution), or more. This is the raster format alluded to in Chapter 1. The file contains no image formatting information. Thus reading and displaying the file requires prior knowledge of the following information not contained in the file:

- Image dimensions
- Grey level resolution
- Pixel scan order
- Byte arrangement (little or big endian format)

The file size depends only on the image dimensions and grey-level resolution. File sizes tend to be large, and are particularly inefficient when considering simple images.

---

[5] Rendering generally consists of the following steps :

- Examination of database containing
- Coordinate transformation
- Clipping & projection transformation
- Scan conversion
- Hidden surface removal and shading

[6] A low bitrate (<28.8kbit/s) video format for videoconferencing and surveillance over telephone lines. It exploits time domain redundancy and motion estimation similarly to other compressed video formats as described in Section 7.3.

Clearly this is a primitive format with limited usefulness for the average user. The user-friendliness is addressed by the bitmap file type. There are several variations in use, but perhaps the most common is the Microsoft Windows bitmap format (file extension .BMP). This is essentially the same as the RAW file type, but has appended a header containing the formatting information listed above. As such, the image may be automatically decoded and displayed by a suitable viewer, with no prior knowledge of the formatting parameters.

It is worth noting that although these uncompressed file formats are inefficient in respect to storage space, they are still the preferred route if maximum resolution is required, for example desk-top publishing. Many commercially available flatbed scanners and their associated software produce files of one of the bitmap types by default. If more efficient representation is then required, the user may compress this output file to one of the formats described below.

2.8.1. Compression Based on Bit Stream Examination

A little thought reveals that raw and bitmap type files may contain a lot of redundancy. Consider an image with a central subject and a plain white background. The majority of the pixels are identical, so the raw file will consist of long sequences of identical bytes or byte groups. Obviously this type of redundancy may be exploited by examining the file content, or what will be referred to as the bitstream (since the file content is generally sequentially examined). This type of algorithm may be termed semantic-dependent

The most obvious choice for compression will be the generalised file compression utilities, which generally employ one or more of a group of established algorithms and yield compressed files in one of the established formats. In general though, these files will suffer from the lack of formatting header information as with the raw file type. The most popular compressors in this category are the zip type (eg PKZIP) (file extension .ZIP) and arj (file extension .ARJ). These are both based on static Huffman coding[7] developed by Huffman in the 1940's. This method assigns the shortest codeword to the most frequently occurring symbol, and progressively longer codewords to the less probable symbols. This means that the summation of the probability weighted codeword lengths will be minimal.

The generalised file compression utilities led to the development of several bitstream compression systems targeted at and optimised for image files. The best known of these are GIF, TIF, and PCX. Due to the predominance of the GIF format it is discussed here. The others are covered in Appendix C.

The Graphic Interchange Format (GIF) was developed by CompuServe as a method of sending compressed files over a network. It is however, rapidly gaining acceptance as a file format standard with many software packages now supporting it. GIF utilises LZW (Lempel-Ziv-Welch) compression as used in many general file compression utilities such as PKZIP.

---

[7] An Huffman code is built as follows :

- Rank all symbols in order of probability of occurrence
- Successively combine the two symbols of the lowest probability to form a new composite symbol, eventually building a binary tree in which the probability of each node is the summation of the probabilities of all the nodes beneath it. This defines a canonical Huffman tree.
- Trace a path to each leaf, noting the direction at each node. The direction at each node determines the bit value at that decision point

This method assigns the shortest codeword to the most frequently occurring symbol, and progressively longer codewords to the less probable symbols. This means that the summation of the probability weighted codeword lengths will be minimal.

Advantages of this format are that it has a good compression ratio and is widely supported across a variety of platforms. Disadvantages include that it can be slower to load than most bitmap formats and it does not support true colour 24-bit images.

2.8.2. Compression Based on Image Content

Considering the sample image of a central subject on a plain white background (such as the CAR test image - see Appendix A), it is apparent that it is a rather artificial image, and it clearly suits bitstream type compressors. However, most natural images do not contain this kind of obvious redundancy. Redundancy is more likely to be found in a two-dimensional group of contiguous pixels than a long sequence of pixels in one row of an image. As a result of the scan pattern of bitmap type images, this two-dimensional group of similar pixels will be dispersed as short runs of pixels separated from each other in the overall file bitstream. Consequently most of the redundancy will not be identified and exploited. This implies that effective compression will demand examining images in a two-dimensional plane in addition to sequential examination of the bitstream. If this were implemented, it can be extended to the next level of sophistication which is to examine an image "patch-wise", not just for identical pixels, but for groups which contain a degree of similarity within some tolerance in the context of an intelligent interpretation.

This leads to the next group of compression formats. They work on differing principles, but they all reduce the redundancy in a manner sympathetic to the image content. The move to considering areas within an image that are not necessarily exactly identical, but may be considered similar, implies the transition from the above lossless compression schemes to lossy compression schemes. Inherent in these schemes is the compromise that the compressed and decompressed image will not be exactly identical to the original uncompressed image. Instead, it will approximate the original image with some degree of error. This is the crucial area of interest in lossy compression - what forms of information may be regarded as redundant in the context of the human visual system. The aim is obviously to selectively economise the bitstream and still achieve a satisfactory compression ratio. The successfulness of the compression scheme will depend largely on the sophistication of the algorithm which arbitrates which forms of redundancy may be eliminated.

The best known such format is JPEG, a mnemonic standing for Joint Photographic Experts Group. These files have the extension JPG, and have become synonymous with images on the internet. The search for redundancy is performed after transforming the image into the frequency domain. Since most of the signal energy resides in the DC and low frequency AC coefficients, significant compression can be achieved by discarding upwards of 50% of the coefficients, beginning with the least significant, highest, frequency components. Additional coding gains are achieved by conversion to luminance-chrominance space, and using a coarser quantisation for the chrominance AC coefficients which are retained. A more detailed discussion of the JPEG and GIF formats appears in Appendix C. Fractal image compression can also be placed in this class of algorithms that examine a 2-dimensional image blockwise and intelligently reduce redundancy.

Note that the image formats discussed in this thesis are applicable to continuous tone images, as opposed to vector or line drawings. Vector formats are generally used where 2 bit images are represented by a collection of lines. These lines represent step discontinuities and generally result in severe compression distortion with the lossy schemes aimed at continuous tone images. They are more successfully stored as a mapping of the lines by their vector

coordinates. These formats are frequently used in specialised applications such as engineering CAD packages, topographical and medical arenas.

2.8.3. Comparison Between Fractal Image Compression and Established Alternatives

A framework for comparison must be defined. The two primary measures are algorithm complexity, and compression ratio. In general there will be a balance between the two, and the approach taken depends on the requirements of the particular application. Broadly speaking bitstream techniques like GIF provide only modest compression, but the algorithm is simple and hence encoding and decoding times are fast. JPEG is at the other extreme, offering compression ratios of up to 20, depending on image content [7], but with the penalty of severe algorithm complexity. Fractal Image compression also falls into this category, but it features the peculiarity of being inherently asymmetrical. That is, the decompression process is relatively undemanding computationally, while the compression process consists of massive systematic searches. This characteristic suggests that fractal image compression is uniquely suited to broadcast applications such as CD ROMs and internet web pages. These applications involve compressing the image once, justifying the higher compression times. This is more than offset by the time gains where the image is decoded by many different viewers, each of whom will benefit from the fast decompression time.

It has been stated that the attractor of an IFS will possess detail at every scale. Leading on from this it is possible to decode the image at any desired size, simply by scaling the transformations. The extra detail needed for decoding the image at larger sizes is generated automatically by the encoding transforms. This extra detail is not "real" in the sense that it is not related to the actual image before compression, it is just a product of the encoding transforms, and appears simply by executing additional decoding iterations. The result is smooth tone graduation, superior to the block artifacts that are introduced when enlarging images conventionally by local pixel replication. This feature of fractal decoding has been termed superresolution. It makes fractal compressed images flexible for useove rthe internet, allowing decoding at user-defined sizes.

An aspect of the robustness of the compression technique that has gained importance with the advent of the internet is that of file behavior in response to a portion of the file being lost or corrupted. Partial file loss is frequently encountered when downloading image files off the internet due to the hostile environment imposed by the ethernet protocol, limited bandwidths, timeouts and overloaded cache servers. Typically the problem manifests itself in the last portion of a file being truncated. Both GIF and JPG files can still be viewed under these circumstances with the effect usually being loss of the lower rows of pixels. The upper rows are still decoded correctly. Fractal compressed files consist of affine maps. If some of these are lost, the corresponding blocks of the image cannot be decoded. At first this may not seem serious, but unfortunately, it is highly likely that other range blocks map from those blocks that could not be decoded. As a result, the loss of information will permeate through the decoded image at each decoding iteration. This will lead at best to a degradation of image quality and at worst some "attractor" unrelated to the image, often clipping at a pure white or pure black image. The results of qualitative and quantitative comparisons between JPEG, GIF and fractal compression appears in Section 6.4.3 (Including sample images).

2.9. Conclusion

It may be stated that fractal image compression holds a unique position in the complexity - compression ratio tradeoff. It offers high compression ratios, comparable to JPEG at a given SNR point. It labours under a severely intensive encoding algorithm, orders of magnitude slower than JPEG, but it has super-fast, simple decompression rivaling that of GIF for speed, and promises to surpass JPEG in this regard.

This chapter has discussed the development of this relatively new compression technique. Initially, the theoretical developments were the premise of researchers approaching the problem with a mathematical background. It presented tantalising opportunities, but they were limited by the current state of the art in computing platforms. Thus any practical implementation required optimisation and algorithm acceleration. A number of (now standard) approaches were developed, including non-unified search spaces implemented by classifying domain and range blocks on local brightness, edge content or grey-level texture criteria. Adaptive partitioning schemes were implemented, such as the quadtree schemes which allows the economies of covering featureless areas with larger blocks, and HV partitioning which takes cognisance of edge features. Attempts were made to speed up the affine map search procedure reducing the size of the domain block pool to be searched, and by transforming the data into other search spaces, preferably that could be truncated to reduce the dimensionality of the search. These included searching in a DCT transformed space, various invariant spaces, and a vector treatment which allowed a tree-structure directed search.

The second chief area of research effort has been in improving the quality of the attractor produced by a P1FS. Alternative partitioning strategies have been suggested, and periodic attempts have been made to produce a theory giving tight bounds on contractivity conditions, and solve the inverse problem. Currently, many of the norms governing design decisions still rest on empirical evidence. However, computing technology advances in the last decade have given the encouragement required to pursue these issues again.

Recent approaches have focused on practical implementations. The state of the art has advanced to the point were still images can now be compressed in just several seconds, and real-time video sequence compression is gaining interest. High-end computing architectures have evolved enormously, with move away from fine-grained massively parallel application-specific processors towards reasonably generic RISC processors. The current state of the art in imbedded processors are coarse-grained modestly parallel RISC processors, such as the Texas Instruments TMS320C80 used in this research. This project develops an algorithm for implementing some of the findings of fractal compression research over this modestly parallel architecture. As evidenced by history, it can be expected that this level of processing power will in the very near future be available from commercial desk-top personal computers. This will then allow the technique of fractal image compression to gain acceptance.

3. THEORETICAL CONSIDERATIONS IN FRACTAL IMAGE COMPRESSION

3.1. Introduction

As suggested in the previous chapter, if an image can be described piecewise by means of a deterministic algorithm for each piece, compression can be achieved by storing only those algorithms, and no explicit information of the image content itself. The algorithms used are transformations which map onto each of the image pieces from some library orcodebook of codewords. Conveniently, rather than generate a separate library, the source codewords chosen are from a second partitioning of the image - the domain pool. This chapter presents a mathematical justification of what has been described in intuitive terms in the introduction. What follows is based on a survey of the existing literature. This theory has its roots in the work of Jaqcuin [8]. Different analyses and notations may be found in other works such as [4,29,31,52,53].

3.2. Preliminary Results

The space used is a metric space[1] (X, *d)* for a set X and a metric d that maps the Cartesian product X x X into the non-negative real-line, ie *d:XxX* —>5R^{+} [4]. Conceptually, X may be seen as a set of points in *W* , or in the case at hand, a set of images. Typically, one of several metrics may be used. Common choices include the following :

1. Euclidean metric in the plane :

$$d((x_1, y_1),(x_2, y_2)) = \sqrt{(x_1 - *i)^2 + (* - y_2)^2} \qquad (3\text{-i})$$

2. Supremum metric:

$$rf_{sup}(/,g) = \sup_{.veil!^2} |/(*) - g(*)| \qquad (3.2)$$

where /,g:^{s}J?^{2} —» Ware real measurable functions. For a function *k(x), sup(k(x))* is the maximum value that *k* achieves overx.

3. *IF* metrics:

$$11/-*l=(l\backslash f-z\backslash''^{dxd}yT' \qquad w$$

where /,g:^{s}J?^{2} —»<Rare real measurable functions. Clearly for/>=2, the RMS metric results.

---

[1] A metric space is a set X on which the real valued distance function *d* : Xx X —> $^{S}J\backslash^{+}$ is defined, with the following properties :

d(a,b)> _0 for a\\a,beX
d(a,b) = 0 iff *a* = *b,* for all *a,be* X
d(a, b) = d(b, a) for all *a. b* e X
d(a,c) ≤ *d(b,a)* + *d(b,c)* for all *a,b,c* e X (triangle inequality)

The choice of metric obviously depends on the nature of the elements of X, and influence the contractivity of the transformation (see below). Both the supremum and the *V* metrics will be used in the following discussions.

Definition 1 *Contractive transformation*
If X is a metric space with metric*d,* then a mapping [52, 53] (or transformation) *w:* X -> X is Lipschitz with Lipschitz factor .v [4] if there exists a positive real value ,v such that

$$d(w(x),w(y)) \leq sd(x,y) \quad \forall \quad x,y \in X \tag{3.4}$$

Further, if and only if s e [0,l) then w is said to be contractive with contractivity s.

From this follows one of the main theorems of the subject.

Theorem 1 *Contractive Mapping Fixed Point Theorem*
If X is a complete metric space with *w:* X —» X a contractive mapping. Then there exists a unique point x_f *e* X such that for any x e X

$$JC, = *<*_{,}) = \lim w^{n}(je) \tag{3.5}$$

where $°n$ denotes the «-fold composition of w with itself.

Then x_f e X is the fixed point of w. [8, 52]
Proof (following [4]):

Take x e X. Then for $n > m$,

$$d\{ w^{°'''}(x), w^{°''}(x)) < sd(w^{'''-1}(x), »•''•'(x)) < s'''d(x, w^{°''\sim m}(x)) \tag{3.6}$$

Using the triangle inequality repeatedly:

$$\begin{aligned} d(x, w^{ok}(x)) &\leq d(x, w^{jkA}(x)) + d(w^{ok\sim l}(x), w^{ok}(x)) \\ &\leq d(x,w(x)) + d(w(x),w(w(x))) + \ldots + d(w^{ok\sim l}(x),w^{ok}(x)) \\ &\leq (1 + ,y + \ldots + s^{k\sim 2} + s^{k,1})d(x, w(x)) \\ &\leq \frac{1}{1-s} d(x, w(x)) \end{aligned} \tag{3.7}$$

(3.6) may now be written as

$$d(x^{°'''}, w^{°''}(x)) \overset{s^m}{<} \underset{1 - ,v}{} —d(x, Mix)) \tag{3.8}$$

Since $s < 1$, as n and m become large, the left hand side will tend to zero. Thus the sequence $x, w(x), w(w(x)), \ldots$ is convergent and since X is complete, the limit point $x_f = \lim w^{°''}(x)$ is in X. contractivity of w implies continuity, so the result holds.

The uniqueness of the fixed point will now be proved (after [4])
Proof: Supppose there exist two fixed points x_i, x_2- Then

$$d(w(x_f),w(x_2)) = d(x_t,x_2).$$

But from definition 1

$$d(w(x_1),\ w(x_2)) \leq sd(x_1\ ,x_2)$$

which would be a contradiction. Hence the fixed point is unique.

Theorem 2 *Collage Theorem*
With the hypothesis of theorem 1, the contractive mapping fixed point theorem,

$$d(x,x_f) \leq \frac{1}{1-s} d(x,w(x)) \qquad (3.9)$$

[8, 12,53]
Proof: In equation (3.8) let *k* tend to co .

As the previous chapter suggested, there is no simple expression [9, 32] for the attractor in terms of the affine coefficients (the inverse problem). Thus it is not possible in the general case to optimise those coefficients (attractor optimisation) to minimise the distortion in the recovered attractor (Represented in Equation 3.9 by $d(x,x_f)$). The alternative, collage optimisation, is practicable. It is non-ideal in that it essentially optimises each mapping independently of the others, but nonetheless the collage error (Represented by $d(x,w(x))$ in Equation 3.9) fixes an upper bound on the real, attractor distortion.

3.3. Iterated Function Systems

Definition *2 Iterated Function System (IFS)*

An Iterated Function System [32], consists of a complete metric space *(X,d)* together **with** a finite set of contraction mappings $w_n : X \to X$ with respective Lipschitz factors s_n for $n = 1,2,...,N$
The IFS may be denoted by $\{(X;w_n): n = 1,2,...,N\}$. The worst case contractivity factor will be used, ie $s = \max\{s_n : n = 1,2,...,N\}$.

Thus an IFS is simply a finite set of contractive mappings.

Consider the IFS defined above. Let *H(X)* be the set of all nonempty, compact subsets of X. Define a transformation $W : H(X) \to H(X)$ by

$$W(B) = \bigcup_{i=1}^{N} w_i(B) \qquad (3.10)$$

for any $B \in H(X)$.
Then *H(X)* is also a complete metric space under the Hausdorff metric defined by:

$$h_d(A,B) = \max\{\max\{d(x,y) : y \in B\} : x \in A\} \qquad (3.11)$$

where *d* is the metric chosen for the metric space in which the IFS above resides. For a proof, see [10]
Conceptually, the Hausdorff distance h_d between two elements $A,B \in H(X)$ is the greater of the two minimal distances which are measured firstly from the elements *of A* to the elements of *B*, and secondly from the elements of *B* to the elements *of A* (using the metric defined for the space X).

It follows from the contractive mapping fixed point theorem that *W* has a unique fixed points in *H(X),* such that

$$A = W(A) = \bigcup_{i=1}^{N} w_i(A) \qquad (3.12)$$

in which case this unique fixed point is the attractor of *W.*

This result may be proved by induction on / and was first demonstrated by Hutchinson. (see [4 JO])

Theorem 3 *Collage theorem for an IFS*

[52, 53] Using the same metric space (X, *d)* as above, choose $Be //(X)$. Set $//\geq 0$. If $\{(X;w_{,()} : n = 1,2,..., TV\}$ is an IFS with contractivity $s\ e\ [0,l)$, and an attractor $A\ e\ H(X)$.
If

$$h_{,,}(B,W(B)) < ,n$$

then

$$h_d\{B,A) <- \frac{£}{(1-s)} \qquad (3.13)$$

Proof: The result follows from substituting the above conditions into the Collage theorem (theorem 2)

Conceptually, the fixed point equation (equation (3.5))

$$A = W(A) = w_l(A)\{Jw_2(A)U. \ldots .\backslash Jw_N(A)$$

implies that the attractor may be formed by assembling transformed copies of itself. The uniqueness of the attractor is of key importance as it means that it is completely specified by the mapping *W.* Thus if *W* can be found for a given *B* such that $B = W\{B)$, then it will be known that the attractor $A = B$. Further, the collage equations (3.13) above imply that even if an IFS producing an exact attractor can't be found, at least the attractor will resemble the original image with some small error dependent on */j* and *s,* bounded by the collage error given by Equation 3.9. What this means in practice is that if an IFS can be found for an image[2] that when applied to that original image produces a sufficiently close approximation to that original image, then the fixed point, or attractor, of that IFS will be a close approximation to the original image.

The image encoding problem then, or the inverse problem as it is generally called [4, 31] is to find an IFS whose attractor is close to some arbitrary given set *B.*

3.4. Partitioned Iterated Function Systems

As Chapter 1 suggested, it is somewhat unrealistic to expect to find an IFS descrbing an image where each component transformation takes as input the entire image. Rather the revised goal is to find a PIFS where each transformation maps from a portion of the image.

Definition 3 *Partitioned Iterated Function System*

[8] If X is a complete metric space, and /), c X for $i = !,....,«$, then a PIFS may be defined by a set of contractive transformations $\{w, : i = \backslash,2,...,n\}$ where $w_i : D,\bullet \text{—> X}$ for $/ = !,....,«$.

[2] because the collage error is essentially the distance between the target image and one iteration of a test mapping (assessed piecewise for each affine mapping).

$$W = \bigcup_{i=1}^{k} Jw,$$

Moving away from the generality of the above discussion, a model of an image may now be presented. Among the literature, there are two basic approaches. Many [31,53] begin with discretised (raster type) images where the pixel intensities within an *nxn* pixel block define a point in $\mathbb{R}^A$ where $N = nxn$. Others [4 ch 2] use a continuous function treatment as is presented here.

An image of infinite resolution may be regarded as function $/ : 9 ?^2 \longrightarrow *R$. To limit the function, restrict the domain to the unit square $1x1 = \{(x,y)\backslash x,y e [0,\backslash]\}$ and the range to $/ = \{x \mid x \text{ e } [0,1]\}$ representing the allowable grey levels.

The PIFS sought will then be in the space $r = \{/ : I^2 \rightarrow \bullet\ y\backslash\}$ of images defined by functions on the unit square.

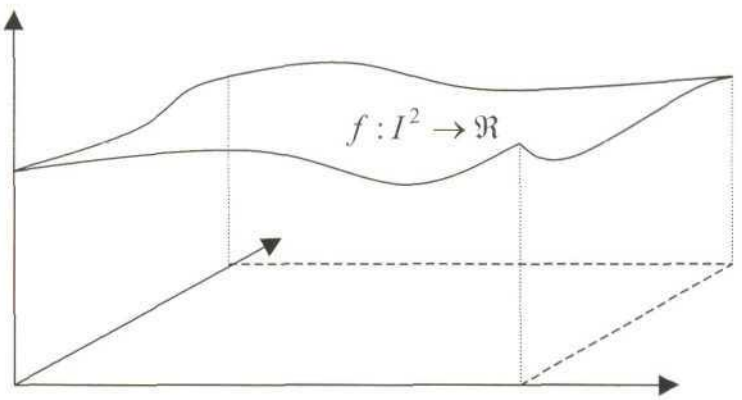


Figure 3.1 : An image function/on the unit square.

The graph of one of these images/will be some grey level intensity map over $/"$. In order to find a PIFS, the following is performed:

Partition the domain of the image function (ie I^2) into a set of n domain blocks $D = \{Dj \text{ c } I^2 \mid / = 1,...,«\}$, and a second partition into a set of m range blocks $R = \{R_/ \text{ c } I^2 \mid j = \backslash,...,m\}$. Note that the domain of/should not be confused with the set of domain blocks D, and similarly for the range of/and the set of range blocks R.

Now the D_l define areas in the unit square under/ The cartesian product Z), x/ may be used to project this area onto the graph of the image function. The intersection of this projection with / produces an area on the graph of/'. This area may then be denoted $f\{^\backslash(D_l \text{ x } /)$ (see Figure 3.2) and represents one of the possible the domains of the transformations {vv, |/ = !,...,&}

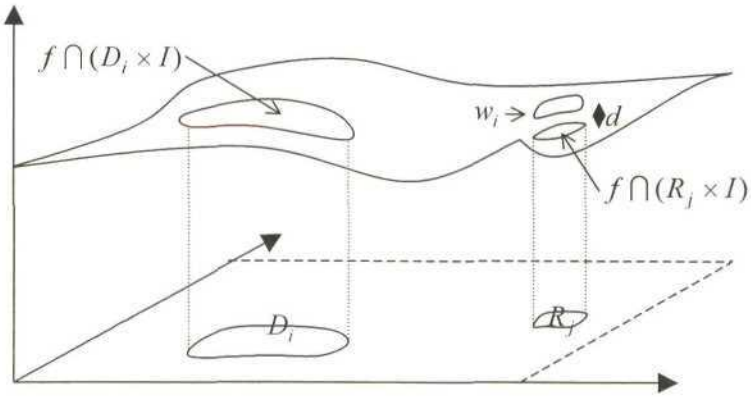


Figure 3.2 : Determining the domain blocks and range blocks for the contractive transformations.

Similarly for the R_j may be defined a projection / f l (•/?,- x 7) onto the graph of the function/ to form the range blocks for the transformations. These transformations $w_l : 7^3 \rightarrow 7^3$ on $f(x,y)$ can now be defined in 7' by

$$w_i(f) = w,(x,yj(x,y)) \tag{3.14}$$

The result of this computation will be a graph of a function over 7".

Thus the aim is to find for each R_f, a $D,$• c 7^2 and w, : 7), x 7 —> 7^3 such that

$$d(f \cap (R_j \times I), w_i(f)) \tag{3.15}$$

is minimised. If this can be achieved, the PIFS will be described by $W = (\mathrm{J}^{k}_{i=1} w_i$

Conceptually it must be pointed out that each of the w,take/over the entire I^2 as input, but the affine transformation used incorporates a spatial selectivity component (see Section 3.7) that ensures only *a portion* of/is mapped into *Wto* form that particular $w^f) \sim R_,$.
Equation (3.15) implies that some metric is used, and in fact to satisfy the theory of metric spaces from above, metrics must be defined.

In most of the literature [4, 53] the supremum metric

$$^SUP(/'£) = \sup_{(Jt,.v)e I^2} |f(x,y)-g(x,y)| \tag{3.16}$$

is used.

As the intuitive discussion in the introduction pointed out, in addition to the required spatial contractivity, it is necessary to enforce contractivity in the grey level.

Definition 4 : z *Contractivity*

A map $w:iR^3 \rightarrow 5K^3$ with behavior $'\{x ,y',z\) = w(x ,y ,z,)$ and $(x',y',z'_2) = w\{x ,y ,z_2)$ is called z contractive if:

$$3 \ s \ e \ [0,l) \text{ such that } |z\text{-}z| \leq |z_1 - z_2| \tag{3.17}$$

Clearly the goal must then be to show that if the ve„...,w_w are z contractive, then the PIFS $W = [JM)$ is contractive in the chosen metric. Typical texts such as [4, 53] show this for the supremum metric. The supremum metric is not really the best choice as it is a kind of "worst case" distortion measure, and will be dominated by any singularities or point errors in a mapping which might otherwise give a good visual approximation to the range block under consideration. A more appropriate measure is the RMS metric. A proof of the contractivity of the PIFS under the RMS metric (defined in equation (3.19)) is now presented :

The RMS metric is one of the family of *IF* metrics :

$$d_p(f,g) = \|f-g\|_p = \left(\int_{I^2} |f-g|^{w)^{u}}\right) \tag{3.18}$$

and appears for/?=2 :

$$d_{RMS}(f,g) = \sqrt{\left(\int_{I^2} |f-g|^2\right)} \tag{3.19}$$

Theorem 4 *Contractivity of Image PIFS under RMS metric*

If the w_1 ,...,»>„ are z contractive, then the PIFS $W = (\sum_{i=1}^{n} w_j$ is contractive in the RMS metric.

Proof:

Let .v = max^v, | /-- !,...,«} where the s,- are the z contractivities of the w,.

$$d_{RMS}(W(f),W(g))$$

$$= \sqrt{\int_{I^2} |W(f)(x,y) - W(g)(x,y)|^2}$$

$$\text{-}\sqrt{\int_{I^2} \left|\sum_{i=1}^{n} \{z-component[w'i\,(x,y,f(x,y))\,\text{-}W)\{x,y,g\{x,y))for\{x,y)eD_i\}\}\right|^2}$$

$$\leq \sqrt{\int_{I^2} \left|\sum_{i=1}^{n} \pounds\{s,[(f(x,y)\text{-}g\hat{}y))for(x,y)\ e\ Z\}\}\right|^2}$$

$$\leq \sqrt{\int_{I^2} \left|\sum_{i=1}^{n} 2\{\bullet [(I(AT.\ y)\text{-}g\{*.\ y))M(x,y)\ \text{e}\ D,\ ]\}\right|^2}$$

$$= \sqrt{\int_{I^2} s^2 \left|\sum_{i=1}^{n} Z\{\{f(x,y)\text{-}g(x,y))for(x,y)eD_i\}\right|^2}$$

$$= \sqrt{j\,I\,s^2\backslash\{f(x,y)\,\text{-}\,g(x,y))for(x,y)\&I^2\Big|^2}$$

$$\text{''}^{s}i\backslash> \sqrt{\left|\ (f(^{x}\text{-}y)\text{~}s(^{x}\hat{}y))for(x,y)^{*}\equiv I^2\right|^2}$$

$$= sd_{RMS}(f,g) \tag{3.20}$$

Thus from this result (3.20) it is justified using the RMS metric. It is used in all the following practical implementations. In practice some manageable measure is needed to assess the difference between two images / and g or image blocks. In the following work the peak signal-to-noise ratio (PSNR) defined by

Definition 5 *Peak Signal to Noise Ratio Measure*

$$PSNR = 20\log_{10}\left(\frac{\sup\{m}{d_{IMS}(f,g)} \quad J\right) \tag{3.21}$$

is used.

sup{max(/),max(g)} is simply the greater of the maximum values achieved by the two images. In practice in a discretised system with 8 bit resolution this will be the ceiling value of255.

3.5. Fractal Image Encoding Algorithm

The results of this chapter may be summarised into an algorithm for fractal image encoding :

Algorithm 1 : *Fractal Image Encoding*

set $d_{mm} = \infty$

while there are uncovered (i.e. unmapped) ranges R_i ***e*** R do {

for all possible domains D_j *e* D (the set of possible domains) do {

for all possible *k* relative flips between R_i and D_j do {

compute W_{jitj} to map D_j to an approximation of R_j

$d_{ikj} = d_{RMS}(R_j , W_{jki}(D_i))$;

if $d_{ikj} < d_{mill}$ {
then $d_{min} = d_{ikj}$
and $w_i = w_{ik/}$
}

}

}

store w_i

}

3.6. Decoding Process

Once the image has been completely described by affine transformations, it is said to be encoded. The description will be of the form of a PIFS and the attractor of the PIFS will be the decoded approximation to the image. Equation (3.10) provides a description :

$$W = \bigcup^{k} w_i \; ; \quad i = 1,\ldots,n \; .$$

Equation (3.5) the fixed point equation shows how the decoding process may be attempted : For the attractor $x_f \in X$

$$x_f = w(x_f) = \lim w^{\circ n}(x)$$

where $\circ n$ denotes the «-fold composition of w with itself.
In other words, starting with any initial image of the dimensions required ((3.5) places no constraints on the starting point), iteratively apply the PIFS until the attractor appears.

(Intuitively the independence on the initial image occurs as the initial image gets reduced in size at each iteration, and thus it will appear only as a point in the attractor. Convergence to a reasonable resolution is usually achieved after about 10 iterations.

This process may be developed into an implementable algorithm :

Algorithm 2 : *PIFS Decoding Process*

for each iteration *q* do {
    for each affine map $w_i \in W$ do {
        identify the domain $D_i \in D$ required by the affine coefficients
        decimate D, to produce a contracted block C,
        rotate/flip C, to match the R_i
        apply $g_i(z) = S_i \cdot z + O_i$ to each pixel value z in C, (contrast/brightness adjust)
        paste the block into the decoded image
    }
}

3.7. Practical Implementation of the Theory

Traditionally the kind of contractive transformations used are affine mappings of the form

$$w_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix} \qquad (3.22)$$

as discussed in the introduction. Other forms have been proposed [40] but the affine system is still the best understood and it is both well behaved and suitable for automated computation in that the algorithm that results is repetitive in nature.

It is convenient to disaggregate this into spatial and grey-level transformations :

Conceptually the spatial portion $T_i\bar{x}$ may be identified as

$$w_i \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix} = T_i\bar{x} + \bar{b} \qquad (3.23)$$

and decomposed as follows [4 ch 7] :

$$T_i\bar{x} = P_i K(D\text{¥}_i \bar{x}) \qquad (3.24)$$

where

F/: a "fetch" operator selecting the domain block D_i from the complete image

D : a "decimation" operator performing the contraction to reduce the chosen D_i to match the dimensions of the R_r This step is essential to satisfy the hypotheses of the theorems in Sections 3.1, 3.2, 3.3 and 3.4.

K: the kernel of the mapping which performs the non-trivial stretch/skew/rotation operations

P/: a "put" operator pasting the output over the Rj.

Clearly then K as the non-trivial operator requires the most concentration. Unfortunately there is no direct means of computing K, and the alternative is a systematic search. Typically, for each Rj a mapping must be sought from each possible D_i. The D_i that provided the closest mapping is then selected, and the affine parameters stored.

The F/ and P./ spatial operations are taken care of by the search system implemented to find this best match.

The D operation is performed for all possible domain blocks at the outset and is stored for later reference. In practice, as in the system implemented for this research, this is predetermined calculation is facilitated by selecting predefined domain and range block dimensions, which hence leads to a (limited) selection of spatial contractivity factors. This is discussed more fully in Section 5.1.

In terms of the previous notation, a specific w, operates only in an area Z), x / above $D_($ and maps to a specific area Z?, x Z.

The grey level transformation was analysed as follows:

During the search for the w,, the comparison between a Z), and a /?• necessitates computation of the grey level affine coefficients s_t which sets the contrast and o, for the brightness offset. A grey-level transformation g,- for a mapping w,may be defined as

$$g_i(z) = s_r z + o_i \qquad (3.25)$$

(extracting the grey level coefficients from equation (3.23))

To assess the match the RMS difference between the Z), and 7?; must be computed.

Consider a system where the range blocks R_f and the domain block Z), *after contraction* by the spatial operator D each contain n pixels (ie square blocks of size $\sqrt{n} \times \sqrt{n}$ pixels). Let the pixels of the range block have intensities of $r_1,\ldots,r_n$ and the pixels of the *contracted* domain block have intensities $d_1,\ldots,d_{ll}$

Then some thought reveals that in this discretised model the RMS difference between the blocks is represented by (substituting equation (3.25) into equation (3.19))

$$(d_{RMS})^2 = R = \sum_{i-1}^{n} (s \cdot d_i + o - r_i)^2 \qquad (3.26)$$

Expanding :

$$R = \sum_{/-\mathbf{I}}^{n} \left(s^2 d_i^2 + 2sd_i o - 2sd_i r_i - 2or_i + o^2 + r_i^2 \right) \quad (3.27)$$

Clearly the aim is to find the .v and o to minimise d_{KMS} or equivalently R.
Taking the partial derivative of R (3.27) with respect to o and setting it equal to zero :

$$\frac{\partial R}{\partial o} = 0 = Y2_S d_i - Y2r_i + Y2o$$

solving for o :

$$o = \frac{1}{n}\left[\sum_{(=1}^{n} r_i - s \sum_{/=!}^{n} d_i \right] \quad (3.28)$$

similarly taking the partial derivative of R with respect to s and setting it equal to zero :

$$\frac{\partial R}{\partial s} = 0 = \sum_{i=1}^{n} 2s^2 d_i^2 + \sum_{i=1}^{n} 2d_i o + \sum_{i=1}^{n} 2r_i d_i \quad (3.29)$$

substituting equation (3.28) into equation (3.29) :

$$0 = \sum_{i=1}^{n} 2s^2 d_i^2 + \frac{1}{n}\left(\sum_{i=1}^{n} 2d_i \left[\sum_{./=!}^{n} r_i - s \sum_{;=1}^{n} d_i \right] \right) + \sum_{i=\backslash}^{n} 2r_i d_i$$

and solving for s :

$$s = \frac{\left[n \sum_{i=\backslash}^{n} d_i r_i - \sum_{i=]}^{n} d_i \sum_{;=\mathbf{I}}^{n} r_i \right]}{\left[n \sum_{i=1}^{n} d_i^2 - \left(\sum_{- /=!}^{n} d_i \right)^2 \right]} \quad (3.30)$$

Once these quantities have been found, R is directly calculated from the pixel intensity values:

$$R = \frac{1}{n}\left[\sum_{i=1}^{n} r_i^2 + s\left(s \sum_{i=1}^{n} d_i^2 - 2 \sum_{i=1}^{n} r_i d_i + 2o \sum_{i=1}^{n} Y, d_i^2 \right) + o\left(no - 2 \sum_{i=1}^{n} r_i^2 \right) \right] \quad (3.31)$$

(substituting (3.28) and (3.30) into (3.27))

3.8. Benchmarking and Predictions

3.8.1. Encoding Complexity

Practically, suppose a 512 x 512 pixel image is considered, with 8 x 8 pixel range blocks so that n = 64. During the calculation of R, s, o the following preliminary results are needed :

Let Mul = Multiplication operation
Add = Addition operation
Ace = Accumulate operation[1]
Div = Division operation
Shf = Barrel shift operation[4]

$\sum_{i=1}^{64}$: requiring [1 x Mul and 1 x Ace] x 64

$\sum_{i=1}^{64}$: requiring [1 x Ace] x 64

$\sum_{i=1}^{64}$: requiring [1 x Ace] x 64

$\sum_{i=1}^{64}$: requiring [1 x Mul and 1 x Ace] x 64

$\sum_{i=1}^{64}$: requiring [1 x Mul and 1 x Ace] x 64

or totally 3 x 64 = 192 multiplication and 5 x 64 = 320 accumulation operations.

Assembling these quantities to determine the *s*, *o* and hence *R* values will require the following :

Q : [1 x Mul and 1 x Add and 1 x Div]
s : [4 x Mul and 2 x Add and 1 x Div]
R : [5 x Mul and 6 x Add and 3 x Shf and 1 x Div]

So then the comparison between two 8 x 8 pixel blocks involves 202 multiplications, 320 accumulations, 9 additions, 3 shift operations and 3 divisions.

This overlooks any overhead processing time for example fetching the data blocks from memory. Many of the multiplications involve the *o* and *s* which would seem to be real numbers necessitating floating point calculations. However the code developed quantises these to positive byte values. This is desirable since integer multiplications are orders of magnitude faster than floating point multiplications [44], and in any case the processors used for the bulk of the processing and integer units not capable of floating point calculations (see Section 4.1.4, 4.1.5). Thus the above analysis is somewhat of a simplification in that it ignores the extra processing required to perform the requantisation and scaling corrections. However, it does give an indication of the large number of calculations that may need to be performed.

For a 512 x 512 pixel image, suppose fixed size 8 x 8 pixel non-overlapping range blocks are used. There will be $(512/8)^2 = 4096$range blocks. Assume the maximum number of possible domain blocks say of size 16x16 pixels where the pivotal coordinate of the domain block is only one pixel shifted from the previous block (ie a value of 1 for what will be called the Domain-block Skip Factor (dsf)). There will be $(512\text{-}15)^2 = 247009$ domain blocks.

Potentially, each range block could be compared to each domain block 8 times (two blocks have 8 possible relative rotations/flips) resulting in $4096 \text{x} 247009 \text{x} 8 = 8{,}1 \text{ xlO}^9$ comparisons. In the context of the TMS320C80 processor used in this research, that could imply as many

[1] An accumulate may be viewed as a kind of running total for repetitive addition operations [54]
[4] A barrel shift is an operation for performing multiplication by a factor of 2 by shifting the bits in the register leftwards by one significant place (or division by 2 by doing a right shift) [54]

packet transfers (see Section 5.2.5). Such a coding scheme would be termed an exhaustive search where for each range block, every possible domain block is considered.

Hence a compression could potentially involve 8,1 x 10^9 x202 = l,6x 10^{12} multiplications and 8,1 xlO^9 x320 = 2,6xlO^{12} accumulate operations. This process will demand an unpractical amount of computing time. Typical preliminary results indicate an encoding time of 120 minutes such an unoptimised exhaustive coding search for a 512x512 pixel grey-scale image when a PI 133MHz processor is used. This is totally unpractical and several of many improvements to the basic scheme may be made to improve the situation. These are examined in Chapter 2, Literature Survey and Chapter 6 which discusses the particular refinements adopted in this research.

3.8.2. Decoding Complexity

Results are presented for a basic system for simplicity's sake. The discussion ignores overhead incurred in file management and manipulating data around memory. To allow some form of comparison with the JPEG compression scheme, a fractal system with fixed range blocks of size 8 x 8 pixels is considered. Further, a fixed domain blocks of size 16 x 16 pixels (ie a constant contractivity factor *ofs* = 2) is used. The image under consideration is 512 x 512 pixels x 8 bit resolution.

Following the decoding algorithm (algorithm 2, Section 3.6) the main computational step of the decoding revolves around equation (3.25)

$$g_i(z) = s_i \cdot z + o_i \qquad (3.25)$$

The pixel values *z* are multiplied by the contrast scaling factor *si* and offset by the mean brightness factor o_i.
In any particular iteration, the calculation of the pixel values for each w,(/) « R_i will involve the following :

Decimation operation :	[4 x Ace and 2 x ShfJ	x 64 pixels
Contrast correction :	[1 x Mul]	x 64 pixels
Brightness offset:	[lxAdd]	x 64 pixels

These calculations will be repeated for each of the 4096 range blocks. The whole procedure is repeated for each of the say 8 iterations. If 8 iterations are assumed (typical for convergence) then a decoding process will involve the following :

4 x 64 x 4096 x 8 = 8,4 x 10^6 Accumulation operations

2 x 64 x 4096 x 8 = 4,2 x 10^6 Shift operations

1 x 64 x 4096 x 8 = 2,1 x 10^6 Multiplication operations

1 x 64 x 4096 x 8 = 2,1 x 10^6 Addition operations

or totally 14,7 x] 0° single cycle operations and 2,1 x 10^6 multiplications. The number of cycles needed for a multiplication operation depends on the processing platform being used, with 3 being a typical value. Note that this is 6 orders of magnitude less than the encoding process. It represents a worst case estimate for the basic iterative decoding algorithm. As discussed in Section 2.6 and Appendix B, there are other more sophisticated decoding algorithms, the best known of which is the matrix inversion approach which is an order of magnitude faster [4 ch 11].

3.8.3. JPEG Complexity

Examining JPEG, the same 512x512 pixel x 8 bit image is considered. The main steps in the algorithm are DCT coding, coefficient quantisation, DC coding and entropy encoding (More detail is contained in Appendix C).

a) DCT coding
For each of the components, the pixels are divided into 8 x 8 pixel blocks and a two-dimensional discrete cosine transformation (2D-DCT) is performed over the block. This yields 64 DCT coefficients. This first one represents the DC value of the block (average brightness) and the others the AC components.

The encoding process is based around equation 3.26, the 1-dimensional discrete cosine transform (DCT) formula [3]. The required 2-D DCT consists simply of performing such a 1-D DCT of each of the 8 columns of the block followed by a 1-D DCT of each of the 8 rows of the result. Thus 64 1-D DCTs are performed for each 8 x 8 pixel block.

$$G_{,l/2} = \frac{K_H}{2} \sum_{A=0} s_x \cos\left(\frac{(2x+1)u\pi}{16}\right) \qquad (3.26)$$

where $K_0 = 42\bar{l}2$ and $K_u = 1$ for $u * 0$

. . . defining c, $= \frac{1}{2}\cos\left(\frac{in}{16}\right)$ (3-27)

the 1-D DCT formula may be written in matrix form :

$$\begin{bmatrix} G_0 \\ G_{1/2} \\ G_1 \\ G_{3/2} \\ G_{\_} \\ G_{5/2} \\ G_3 \\ G_{7/2} \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 \\ c_1 & c_3 & c_5 & c_{,} & -c_7 & -c_5 & -c_3 & -c_1 \\ c_2 & c_6 & -c_6 & -c_2 & -c_{\_} & -c_6 & c_6 & c_2 \\ c_3 & -c_7 & -c_1 & -c_5 & c_5 & c_1 & c_7 & -c_3 \\ c_4 & -c_4 & -c_4 & c_4 & c_{,} & -c_4 & -c_4 & c_4 \\ c_{,} & -c_1 & c_7 & c_3 & -c_3 & -c_7 & c_1 & -c_5 \\ c_6 & -c_{\_} & c_2 & -c_6 & -c_6 & c_{\_} & -c_2 & c_6 \\ c_7 & -c_5 & c_3 & -c_1 & c_1 & -c_3 & c_5 & -c_7 \end{bmatrix} \begin{bmatrix} So \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix} \qquad (3.28)$$

There are 7 distinct c,, and each involves :

[1 x Mul and 1 x Div and 1 x Cosine and 1 x Shfj

Typically, the c, are constants and so may be stored in a look-up table fashion by compress/decompress applications. There is no need to recompute them for each image considered. Thus these calculations will be ignored in this discussion. Note though that thec, range between -0,5 and +0,5. Thus they will typically be scaled to integer values to avoid floating point operations in the following steps.

For each 1-D DCT, calculating each of the 8 G, will require :

[1 x Mul and 1 x Ace] x 8 elements

or totally 8 multiplications and 8 accumulates. This is repeated 64 times for each block, resulting in 512 multiplications and 8 accumulates. Considering that this must be done for each of the 4096 blocks, totally 2,1 x 10^6 multiplications and 2,1 x 10^6 single cycle operations are performed. (Again, a multiplication may be deemed to consume 3 cycles.) This consumes the bulk of the processing time. The following steps vary in intensity depending on the implementation, but together they require typically an order of magnitude fewer calculations than the DCT step.

b) Coefficient Quantisation
Most of the image energy is concentrated in the lower frequencies. The upper frequencies contain progressively less information. Thus they may be quantised at progressively lower resolution without significant impact on the overall information content. This is the lossy stage of JPEG, and is the major influence on the compression ratio.

c) Entropy coding
The reduced coefficients are encoded using Huffman coding. This is a lossless step.

JPEG decompression consists essentially of the same steps in reverse. Instead of the DCT, a 2-D inverse DCT is performed. Again, it may be represented in matrix form, and requires essentially the same number of operations. Considering the extra (unevaluated) steps in the JPEG encode/decode process, and the scaling of the c, factors, the overall JPEG encoding or decoding times will be close to the decompression time for the fractal decompression process. (Again, very much faster than the fractal compression process).

3.9. Conclusion

This chapter presented the theory of transformations defined on functions in metric spaces. Definitions of the metrics commonly used for these spaces are presented. A rigorous discussion of contractivity issues is then presented. This forms the background allowing the concept of an Iterated Function System to be presented. Briefly, and IFS may be seen as a collection of contractive transformations, each taking the entire input set as a domain.

A model of an image as a function over two-dimensional space is discussed. This model is the framework which allows the previously general theory to be applied to the specific case of image functions. The aim is to describe the entire image function as a collection of mappings from itself. The IFS used before attempts to find these mappings where each map takes the entire image as its domain. It is unrealistic to expect real-world images to contain this type of obvious self-similarity. Instead, mappings are sought which take portions of the image function as domains. This type of system is referred to as a Partitioned Iterated Function System. Conceptually then a PIFS is a set of transformations each taking a portion of the image as domain and producing an approximation to another portion of the image as range. It is shown that such a PIFS also forms a continuous function of the metric space consisting of all possible sub-images of the main image function (ie all the possible domain blocks). The metric used in this case is the Hausdorff metric. Having moved to the image model, three-dimensional space is now considered so that in addition to the previous spatial contractivity conditions imposed, grey-level contractivity is also considered. The important requirement is then shown that under the condition of all the constituent mappings being so contractive, the PIFS collection is also contractive in some metric. Traditionally the supremum metric is used. It is explained that then supremum metric is probably not the best choice for image applications. This work then shows that a PIFS can be contractive under the more suitable RMS metric. This justifies the use of the RMS metric, and it was used in most of the practical experimentation.

Some practical considerations are then considered. It is demonstrated that the expected encoding time will be very high unless some rationalisation can be implemented. The encoding and decoding times for JPEG are shown to be roughly symmetric, with the encoding much faster than for fractal image compression. The expected decoding time of fractal systems is then shown to be potentially much faster than JPEG.

4. RESOURCES USED

This chapter describes the TMS320C80 system from the point of view of the hardware architecture and the software development environment. It is important to have an understanding of these issues to appreciate the decisions taken during the design of the code and partitioning of the algorithms. Many of the crucial decisions stem directly from limitations and architectures of the processor used and the software development environment. An overview is presented here, but more detail is available in Appendix B.

4.1. Hardware Architecture

At the inception of the project (January 1997) the TMS320C8x generation was the flagship in the TMS320 family of digital signal processors (DSPs) from Texas Instruments, (see Appendix B.2.1 for more information)

4.1.1. Background to the Multimedia Video Processor

Historically, applications that required image processing, recognition or compression have driven the digital signal processing requirements. Algorithms for these types of applications include convolution and frequency domain transforms that are multiply intensive and the required processing speeds have forced the development DSPs. As suggested in Section 2.7, the traditional method of utilising then-current technology to realise greater processing speeds has been to incorporate multiple processors on an integrated circuit. This approach is found in the TMS320C80 which features five DSPs plus peripherals in one chip. Previously the processors available provided for, in general, only 16-bit (or less) fixed-point multiplies with 32-bit accumulates. The TMS320C80 provides for full 32-bit by 32-bit multiplications. Until recently, DSPs have not been very successful at processing bit-fields and multiple pixel quantities as required for efficient image processing. However, the hardware and instruction set of the MVP's parallel processors (PP) differ greatly from those of traditional DSPs in their ability to manipulate bit fields and to process multiple pixels in parallel through the data path.

Another aspect of realisable image processing speed has been the bottlenecks created in transferring data between processors and memory. The MVP features high bandwidth for internal and external data transfers, ensuring that the processors do not need to wait on data and that interprocessor communication does not bottleneck. This is achieved with a dedicated on-chip transfer controller (See Section 4.1.6, Transfer Controller), which arbitrates over an internal crossbar network for transferring data between on-chip functional units, and interfacing to external memory units.

4.1.2. Multimedia Video Processor Architecture

While the massive 'search' which is required to compress an image using fractal techniques is computationally intensive, is very repetitive in nature. It is thus an ideal candidate for a parallel processing solution. The processor chosen for the task is the Texas Instruments TMS320C80 multimedia video processor. A development board containing this processor was donated to the University of Natal by Texas Instruments, and provided the hardware platform for this project. This board interfaces to the PCI bus of a host PC running Windows NT, as shown in Figure 4.1.

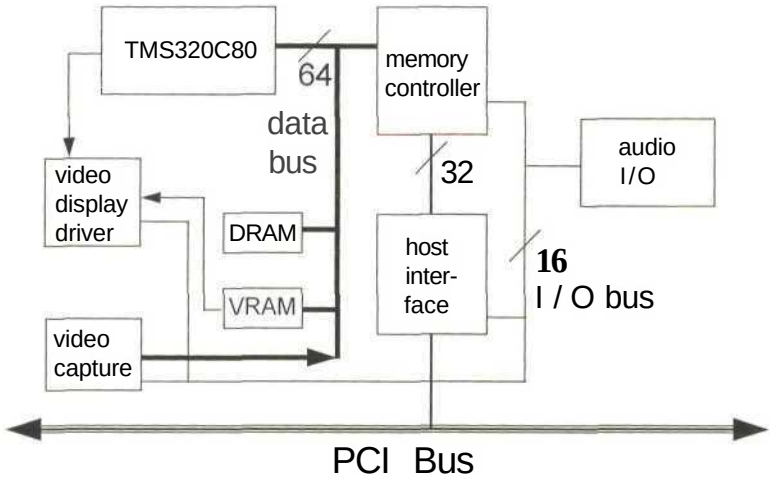


Figure 4.1 : TMS320C80 Software Development Board (SDB) functions.

The host computer used was of the Pentium I type running at 133MHz, supported by 32Mb RAM. (Considered powerful at the inception of the project in early 1997). The MVP board [5, 44] incorporates audio and video capture facilities, together with 2Mb VRAM for video display, and 8Mb DRAM for data storage off the main processor chip. A memory controller arbitrates data transfers on the board, in conjunction with the host interface controller.

4.1.3. TMS320C80 Parallel Processing System

The TMS320C80 processor itself provides a complete parallel processing system geared towards image processing applications on a single integrated circuit, and includes (see Figure 4.2):

- A 32 bit wide RISC type master processor (MP) with IEEE-754 floating-point unit.
- Four 32 bit wide integer DSP processors (PPO to PP3) running as slaves under control of the master processor
- Each of these 5 processors has I Ok bytes of on-chip SRAM associated with it for local data and instruction storage.
- A sophisticated direct memory access (DMA) controller with interfaces to external DRAM, SRAM, and VRAM memory. This transfer controller (TC) also arbitrates the transfer of data among the 5 processors and between the processors and off chip RAM. The TC performs packet transfers (PT) that move data between on- and off-chip memory. The TC also performs instruction- and data-cache servicing for each of the five processors.
- These data transfers take place over a network of crossbars which link the processors to allow efficient data sharing, and simultaneous access to multiple banks of the RAM.
- All instruction and data paths within the chip are 32 bits wide.

The crossbars allow 5 instruction fetches (I per processor) and 10 parallel data accesses per cycle. The device is clocked at 40MHz, allowing execution speeds of up to 2 billion RISC instructions per second to be attained.

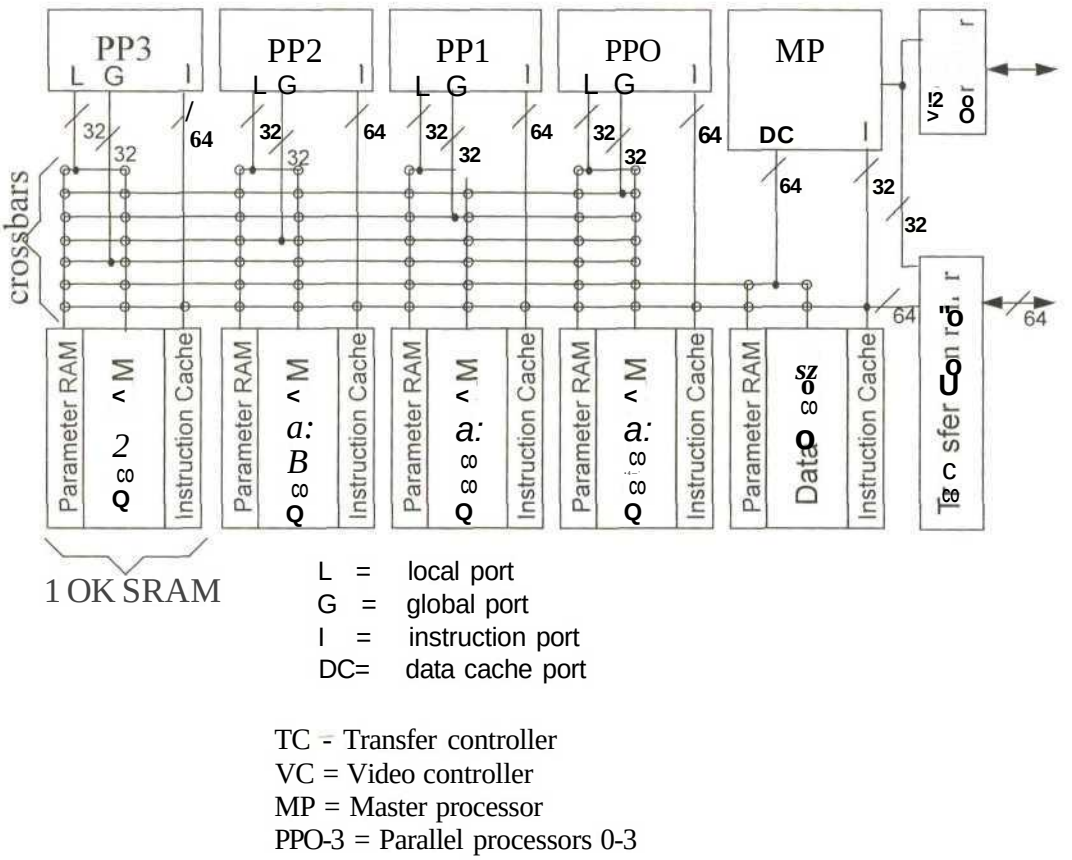


Figure 4.2 : Internal architecture of TMS320C80 showing master processor, parallel processors, SRAM and crossbars.

The processors on the MVP can be configured for a variety of multiple-instruction, multiple-data (MIMD) operations and are connected by a crossbar network to on-chip SRAMs and to the external memory via the transfer controller. This allows the shared memory to be used efficiently and helps to eliminate processing delays resulting from contention.

4.1.4. The Master Processor

The MP is a 32-bit RISC (reduced instruction set computer) processor with an integral IEEE-754 floating-point unit. The MP's primary role is to perform the general-purpose computations necessary to direct the MVP's on-chip processing resources. The MP must be used for all required floating-point calculations, as it is the only processor on the chip so capable. The MP architecture is designed for efficient execution of C code. As with other RISC processors, all accesses to memory are performed with load and store instructions, and most integer and logical operations are performed on registers in a single cycle.

To improve performance, the MP uses several hierarchical levels of pipelining. At the hardware level, for example, floating-point instructions are pipelined so a single-precision multiply or any floating-point add instruction can be started on each clock cycle. This floating point unit runs in parallel with addition, load, and store functions allowing up to 80 MFLOPs in performance at 40 MHz internal clock rate.

On the programmer's level, several instructions can be in successive stages of completion in a given clock cycle as they flow through a pipeline. This is facilitated by the MP having several sub-execution units which can run in parallel. Consider a process that acts on several

items of data in the same way. The first data item may be at the third stage of execution (a barrel shift, say) while the second data item is at the second stage of execution (accumulate, say). A third operation may be started in the same clock cycle provided it does not use the barrel shifter of the accumulator. This pipelining can take place in parallel with another operation (such as a load or store). Thus there is internal parallelism with underlying pipelining.

An example of MP assembly code exploiting pipelining and parallelism is shown below, (taken from the packet transfer code - see appendix D, pg D-33)

```
(1)   d7 = 0x4000100
(2)   d7 = 0xl\\l0
(3)        | |*(a8 + [3] ) = d7
```

(the Ox prefix indicates hexadecimal, and the operand | | indicates operations that occur in parallel with each other.)
In line 1 register d7 is loaded with an immediate hex. value. In line 2 the same register d7 is loaded with another immediate value of 04 00 hexadecimal (l hex shifted 10 places to the left). This implies a pipelining of the barrel shifter (\\ operation) before the transfer of the value. Simultaneously (line 3) with this operation, the memory location 3 bytes above that pointed to by register a8 (register offset addressing mode) is loaded with the value previously loaded into d7 in line 1. It might appear at first that this value was destroyed in line 2, but infact the load operation of line 3 occurs in parallel with the barrel shift phase of line 2. The load phase of line 2 (destroying the value loaded in line 1) occurs fractionally after line 3 is complete, possibly simultaneously with a suitable operation in line 4. Careful thought is needed to exploit this low-level parallelism of execution units, whilst maintaining the correct sequence of operations.

The pipelining and parallelism is taken care of by the C compiler, but there is a limit to what the compiler can achieve, so code must be written with this in mind. For optimised performance in this project, time-critical loops were written in assembly language to ensure low level control over this pipelining and parallelism.

If MP code is written to exploit the parallelism that may be achieved with the multiple execution units, it is the responsibility of the programmer to maintain data-cache coherency when several processors may be acting on the same data.

4.1.5. The Parallel Processors

The MVP contains four of these identical processors which provide the MVP's computational power. For this project, the PP's serve as high-speed pixel coprocessors for the RISC master processor.

The 4 MVP parallel processors (PP) are each a programmable DSP-like 32-bit integer processor with a 64-bit instruction word. The instruction word has fields that independently control the data unit (with its multiplier and ALU data paths) and the two address units.

The PP can execute in parallel a multiply, an arithmetic logic unit (ALU) operation (such as a shift-and-add), and two memory accesses, within one single-cycle instruction. As with the pipelining of the MP code, it is up to the programmer to exploit these capabilities. The internal parallelism allows a single PP to achieve over 500 million operations per second for certain algorithms.

To clarify, it is important to appreciate that the TMS320C80 then has five general-purpose processors (MP + 4 x PPs). By controlling the code flow, these can be used as component processors to set up a high level pipeline, or a parallel processing scheme. In either case, the five processors each incorporate internal parallelism and pipelining within themselves. Exploiting these capabilities while ensuring data coherency depends on the suitability of the algorithm and the way the code is written.

4.1.6. The Transfer Controller

The transfer controller (TC) is the sixth programmable processor in the MVP. It is a combined direct memory access (DMA) machine and memory interface that intelligently queues, prioritizes, and services the data requests and cache misses of the five programmable processors on the MVP (the MP and the four PPs). Through the TC, all of the processors can access off-chip memory. In addition, data-cache or instruction-cache misses are automatically handled by the TC. The TC also supports a host-interface mechanism that allows a host processor to gain access to the MVP system. This facility is used for example when the compression process starts, to allow the host computer to write the image data onto the MVP memory.

The transfer controller is the central component of the implementation of so called packet transfers (PT). This represents the most sophisticated form of content orientated data transfers between on-chip and off-chip memory. These data transfers are specifically requested by the PPs or the MP in the form of linked-list of parameters, which are interpreted by the TC. These requests allow multidimensional blocks of information to be transferred between a source and destination, either of which can be on-chip or off-chip. The TC can transfer data between a source and a destination that have different logical dimensions where the block boundaries may be controlled in a content-related manner. The ability of the TC to make these transformations autonomously greatly improves the efficiency of processing by the PPs or MP.

Packet transfers may be initiated code running on the MP or the PPs. The programmer's code must provide the linked list of parameters. The TC services the requests using the fixed and round-robin prioritizations. Once a processor has submitted a request for a transfer, it can continue program execution. The packet transfer is completed by the TC without the need for additional cycles by the requesting processor.

Packet transfers formed a vital part of the code developed for this project, allowing the large arrays of image data to be stored on the off-chip DRAM and selectively imported to the on-chip caches for processing.

4.2. The MVP Software Development Environment

The process of writing code for the processor begins with separate source code for each processor. This must then be compiled, assembled and linked to runtime- support functions. This results in an assembled module for each processor. These modules are then inter-linked to form a Common-Object File Format (COFF) file which can be downloaded to the MVP as the executable. All stages of assembling and linking are done using DOS based applications and required customising to address the needs of the project. Each application can take a wide variety of command line directives to govern its behavior. This developmental user interface can seem obstructive, but it provides grass roots level control of the build processes. Exploiting these capabilities requires the programmer to have an intimate knowledge of the architecture. Figure 4.3 illustrates the process of code development for the MVP

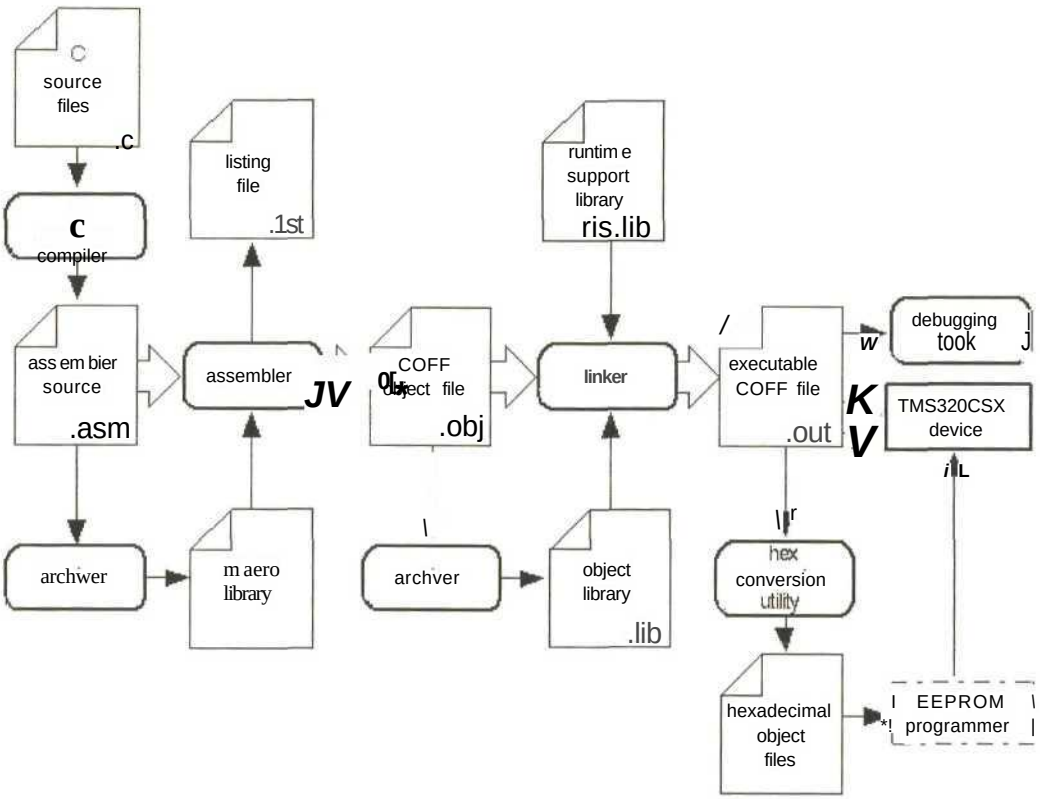


Figure 4.3 : Developing Code for MVP Applications

4.2.1. C Compiler Description

Since the MP and PP's have different instruction sets, different compilers must be used. Each source code module (there may be several separately defined functions called from one processor's code) must be separately compiled. At this stage it is possible to specify options pertaining to type checking on pointers and other syntactical options. The compiler package includes a configurable optimisation program which may be invoked at this stage to improve the execution speed and reduce the size of C programs by simplifying loops, rearranging statements and expressions, and allocating variables into registers [44, 51]. The optimizer runs as a separate pass between the parser and the code generator. Totally then the compilers are of the three pass type. (Parser, Optimiser, Code generator) (See Appendix B.2.6 for more information). As with most aspects of this processor, the optimiser can be confused, and care must be taken when writing code, bearing in mind the peculiarities of the optimisation process, to avoid ambiguities which lead to syntactically correct but logically incorrect optimisations.

4.2.2. Assemblers

The assembler translates assembly language source files into machine language object files in common object file format (COFF). There are two separate assemblers for the MVP (for MP and PP's), and two separate assembly languages, but the format of the assembly language files accepted by both assemblers is similar. During the development one must assemble both modules written directly in assembly, and those produced by the C compilers. Again, the

assemblers can accept many command line options to fine-tune their operation to the user's requirements.

4.2.3. Linking

The next step is combining the object code from the compilation stage into a separate COFF executable stream for each processor. Separate linkers must be used for MP and PP code. At this stage various options for the specific processor concerned must be specified. For example the size of the stacks (for MP and PP's), and heap sizes must be specified.

The memory map for the processor must be specified by the programmer. Directives must be used to locate the sections of the compiled code (eg data segment, code segment, stack segment, segments for malloc'ed memory, scratch pad areas, and other segments pertaining to the multitasking executive). These segments may be allocated to areas of memory of a programmer-defined size, which may be located on the TMS320C80 (for speed - eg time critical repetitive code) or in off-chip memory (for bulk storage). These capabilities are used in the project to optimise the allocation of the limited-size high-speed on-chip memory.

These features require a low-level knowledge of the processor and its memories, but facilitate directing the *run time* allocation of specific items of data [44]. This feature is used in this project to allocate the memory where the image data will be uploaded from the host computer in such away that it takes cognisance of the block boundaries and byte-alignments in the memory structure, so that the data content can be sympathetic to the hardware limitations and preferences. Other scratch-pad areas for temporary data storage are allocated in on-chip high speed SRAM. Optimising these advanced aspects allows performance gains to be had during packet transfers.

The downfall of this system is that the programmer is free to write code that may be "badly-behaved" in the sense that it has free reign to read and write to any memory location. In fact, it is even possible to remap some of the allocations at runtime. There are none of the safeguards enjoyed when writing C code for conventional PC processors with more rigid environments. Thus extreme care must be exercised at the coding stage to ensure well-behaved performance.

Additionally, the use of in-line assembly language, or linking in of bulk assembly modules provides complete freedom from C structures, if needed. This allows hardware level communication with the processors, and direct manipulation of the control registers [44]. These features are facilitated by the runtime-support functions as explained below, and were used during the development of the project.

4.2.4. Runtime-Support Functions

Some of the tasks that a C program may need to perform (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C language itself. However, the ANSI C standard defines a set of runtime-support functions that perform these tasks. The TMS320C8x C compilers include a set of libraries that implement the complete ANSI standard library. In addition to the ANSI-specified functions, the TMS320C8x runtime-support libraries include routines that allow a user to have direct access to specific MVP master processor commands, and direct C language I/O requests to a user-specified device.

The runtime-support libraries can be edited by the user. (This source code must reverse-compiled, edited, and be "built" into libraries using a build utility provided in the development package.) This provides a unique form of low-level control not found in

conventional C development environments. The runtime-support libraries contain macro definitions type definitions, for example the parameters governing floating point precision and format can be accessed and altered for special applications. The memory maps of the transfer controller and other processor control registers are also defined here, as are the formats for packet transfers. This feature was used extensively in the project to customise the behavior of packet transfers.

4.2.5. Common Object File Format

The assemblers and linker create object files that can be executed by a TMS320C8x device-common object file format (COFF) [44]. The COFF format is a modular structure and provides a means of structuring the code and managing the sections to reflect the memory map definitions set up by the programmer during the linking stage. (The COFF format is discussed in Appendix B.2.8). As such, a holistic approach to code development must be adopted where the code structure is sympathetic to the hardware architecture, and the logical appearance of hardware structures and boundaries as defined by the user at link time.

4.2.6. Host Code

The host computer's processor can also run code in parallel with the processes on the MVP. The host can also access the MVP's memory and share data with the TMS320C80 at run time. This feature is not used in the project because it places a demanding load on the TMS320C80's TC leading to degraded performance, and it leads to serious data coherency difficulties. The host code is written in Microsoft Visual C++. In the project the host computer's processor is used solely for accessing the hard-drive, and uploading image data to the MVP at the start of the compression process, and to retrieve the compressed data after the process. This is discussed in Chapter 5. In any case, the processing abilities of the host processor are so far short of that of the TMS320C80, that what little it could add to the overall speed is negated by the delays introduced by increased TC load.

4.2.7. Debugging

The final and most time consuming stage of any code development is debugging. Included in the software development package is a set of DOS based emulators for each of the processors. This allows the contents of individual memory locations and memory registers to be examined during simulated run-time. A large portion of the development time was spent examining the output of various portions of the code by these means. It is important to note that when the code is running on the MVP, it is essentially isolated from the host system and there is no way to perform conventional debugging using such techniques as writing intermediate results to the display screen. The system runs "blind". The process of debugging via the emulators is an inexact task because it is nearly impossible to step through the code of the 5 MVP processors and the host processor in parallel simulating the exact timing relationships that occur at normal run-time. These impediments necessitated a meticulous paper design of the code before implementation could proceed. This is covered in Section 5.2.

4.3. Conclusion

The TMS320C80 represents a modern RISC based interpretation of the block data flow paradigm (BDFP) of moderately parallel architectures that has gained acceptance in recent years. It features coarse grained parallelism at a high level. Each of the constituent processors features internal intermediate and low-level parallelism arising from multiple execution units. The different processors are individually semi-application specific. The master controller is used in this work for all floating point operations, with the bulk of

(integer) calculations distributed among the parallel processors. Data transfers are accelerated by a dedicated transfer controller.
The state of the art crossbar network linking the processors allows them to be configured to form a high level pipeline or parallel structure, as selected for this work. This configurability distinguishes this current generation of parallel architectures from early massively parallel processor arrays.

The hardware is supported by a software environment featuring low-level run-time support functions, and a high degree of programmer configurability. This work makes extensive use of sophisticated data content orientated packet transfers, as facilitated by the transfer controller. The programmer may resort to assembly language to best exploit the low level pipelining and parallelism possible within the code of each processor. This facilitates a code design which may be structured to achieve very high levels of processor utilisation, provided thought is given to structuring the data flow over the crossbar network and external memory ports. Data coherency, timing control, run-time code location and data code location are completely flexible, and the programmer's responsibility.

5. DESIGN OF THE CODE

5.1. Goals For the Solution

As alluded to in Chapter 3, Theoretical Considerations and Chapter 2, Literature Survey, there are a great many refinements to the basic algorithm that have been proposed. Many of these have been demonstrated as successful in [4, 9, 12]. Often, the schemes target mutually exclusive goals, such as reducing compression times or improving perceived quality of the decompressed images. It is not the aim of this research replicate these works by implementing all possible refinements. It was decided that in the time available the primary aim should be to reduce compression times by exploiting the compression algorithm's suitability for partitioning over a parallel processor architecture. In many cases the decisions taken favoured compression speed over absolute fidelity and PSNR. It was decided at the outset that a square block partitioning scheme would be utilised for reasons of practicality listed in Section 2.2. The basic algorithm is presented again below.

Algorithm 5.1 : *Basic compression algorithm*

set $d_{min} = \infty$

while there are uncovered (i.e. unmapped) ranges R_j e R do {

for all possible domains D_i e D (the set of possible domains) do {

for all possible k relative flips between R_j and D_i do {

compute w_{ikj} to map D_i, to an approximation of R_j

$d_{ikj} = d_{ms}(R_j, W_{ikj}D_i))$;

if $d_{ikJ} < d_{mill}$ {
then $d_{min} = d_{ikJ}$
and $W_j = W_{jkj}$
}

}

}

store w_i

}

Initially such a quadtree scheme was implemented using conventional sequential code on the host computer's processor. The results of this were used for benchmarking. It is important to appreciate that in theory an ideal scheme would allow for the domain and range block dimensions to vary over a range of values (within the limits of the contractivity requirements). This would however lead to an almost infinite number of possible block comparisons. Clearly some procedure is needed to define a finite domain block pool. In order to control this and produce a practical solution a set of basic restrictions were imposed as follows :

a) Square domain and range blocks were selected to facilitate comparing blocks conveniently for all 8 possible relative orientations. This also implied that the spatial contractivity factor is the same in both the x and y dimensions.

b) A fixed range block size of 8 x 8 pixels was selected since this gives an acceptable resolution over a 512 x 512 pixel image. Secondly, this is the same block size used by the JPEG system [3], allowing some comparison.

c) A fixed domain block size of 16 x 16 pixels was selected initially, forcing a constant contractivity factor of s = 2, and a manageable domain block pool. The use of contractivity factors that are not powers of 2 was rejected since they would imply comparing a domain and range block where the points of comparison do not coincide with pixel boundaries. The sub-pixel calculations required would be very computationally time consuming and imprecise. Allowing a variety of contractivity factors greatly increases the number of block comparisons that need to be made for a modest PSNR improvement [4],

d) Initially an exhaustive search was implemented to ensure maximum PSNR in the context of the above restrictions.

This system produced a compressor which served as a benchmark for further refinements. Allowing variable domain block sizes would improve PSNR, but massively increases the time to search for affine maps. Conventionally [8, 12, 39J, when implementing a quadtree scheme, a fixed contractivity factor of 2 is imposed. Resolution is improved by allowing the range blocks to assume one of several preselected dimensions, but still maintaining a manageable domain block pool.

Those refinements to the basic scheme that were implemented during this research are a combination of previously published works that were appropriate to the specific aims of this research, and some original refinements that in general arose from the particular environment of the MVP.

5.2. Compression Algorithm

5.2.1. Processor configuration

Overall, then for an optimally efficient algorithm, a theoretical maximum efficiency (100%) and the best compression times will be achieved when all the processors on the MVP are occupied for 100% of the time available [45]. Any idle time by any of the processors reduces the achieved efficiency below this theoretical limit.

This problem may be sub-divided into two aspects:

a) Dividing the algorithm into an "optimally efficient" set of tasks

b) Scheduling these tasks over the available resources in such a manner as to ensure maximum throughput by all processing devices and zero idle time.

Obviously a) must be performed in a manner sympathetic to the architecture so as to maximise the chances of achieving b). Unfortunately, specific hardware and software environment limitations [44, Chapter 4] prevent b) from being perfectly achieved. The design process then becomes one of optimisation and balancing a) and b) against the limitations.

Most of the processing time required (>90%) to compress a typical image is devoted to finding the affine mappings (See Section 3.9). (Experimental evidence is given in Figure 5.5) As such, this aspect of the algorithm deserves the most attention. Clearly, it is where the greatest gains may be realised. Examining the architecture of the MVP, the PPs represent the brute processing power. The problem of partitioning the algorithm over the resources may then be seen as utilising the capabilities of the PPs as efficiently as possible to address the task of finding the affine mappings.

Conceptually, there are two configurations [43, 45] that may be implemented :

a) Pipelining

The four PPs can be used as the constituent processors of a four stage pipeline where each stage performs a phase of the compression. This approach was dismissed for the following reasons:

i) The algorithm does not partition neatly into phases requiring roughly equal processing time. Failure to equalise the processing time for the stages would seriously impact the idle time.

ii) The time taken to find the affine mapping for a particular range block is initially unknown. It can vary by an order of magnitude depending on the block content and the search pattern used to examine possible domain blocks. This uncertainty means that the algorithm does not lend itself to a deterministic, fixed-time scheme. It is more suited to a free-running environment where the processing focus moves autonomously to successive range blocks on completion of preceding blocks. By nature pipeline schemes tend to be more appropriate in fixed-time systems because the input of one stage is critically dependent on the availability of the output of the preceding stage.

iii) The architecture of the MVP dictates that if a particular block of data were operated on sequentially by several processors, there would be a lot of traffic over the crossbar network. Additionally transfer of data between two processors would mean a temporary stall by those two processors while the transfer is completed by the TC.

b) Parallelism

This is the more natural configuration for the MVP generally, and especially in the case of this algorithm. The most obvious partitioning is to allow any particular range block to be processed entirely by one PP. Equivalently, the different PPs process spatially distinct areas of the image. The assignment of spatial areas of the image to specific processors requires some thought[1].

i) This is sympathetic to the fact that the required processing time is biased towards finding the affine mappings and as such does not partition into equal time phases.

[1] The assignment of the range blocks to be encoded amongst the 4 parallel processors is not as arbitrary as it might seem. In this implementation it is arbitrated by the master processor which sets up lists of blocks for each PP to process. The PPs then operate in a free running mode once they have been assigned their lists. The lists are compiled so that the top left block of the image goes to PPO, next block to the right to PP1, and so on until the 5th block again goes to PPO and the cycle repeats. Another perhaps more obvious assignment would be to divide the image into quarters (square quarters or vertical or horizontal strips) and assign them to the PPs. However this falls short when irregular images with areas of localised detail and featureless areas are considered.

ii) It allows a free running scheme where the exact time spent processing a particular range block is unimportant - the other processors independently deal with their own range blocks. This allows greater flexibility to implement refinements that minimise the compression time of a particular block in a content orientated manner while increasing the uncertainty around the expected time of completion of processing. In the system implemented, the PPs autonomously proceed to successive range blocks without interference from the progress of the other PPs. (Certain constraints are imposed by the TC and the crossbar network - see Section 4.1.2, 4.1.6)

iii) In such a parallel implementation, the TC can also run largely independently of the other processors and its throughput is maximised by being able to transfer data to the PPs in a anticipatory manner that pre-empts the requirements of the PPs. PP throughput is maximised by eliminating delays waiting for data. These anticipatory transfers also maximise the TCs throughput and allow transfers to be scheduled efficiently according to the limitations of the TC. (see Section 4.1.6)

The bulk of the processing time is spent with the PPs in such a parallel configuration as they autonomously search for affine mappings.

Thus the configuration of the hardware may be represented as in Figure 5.1.

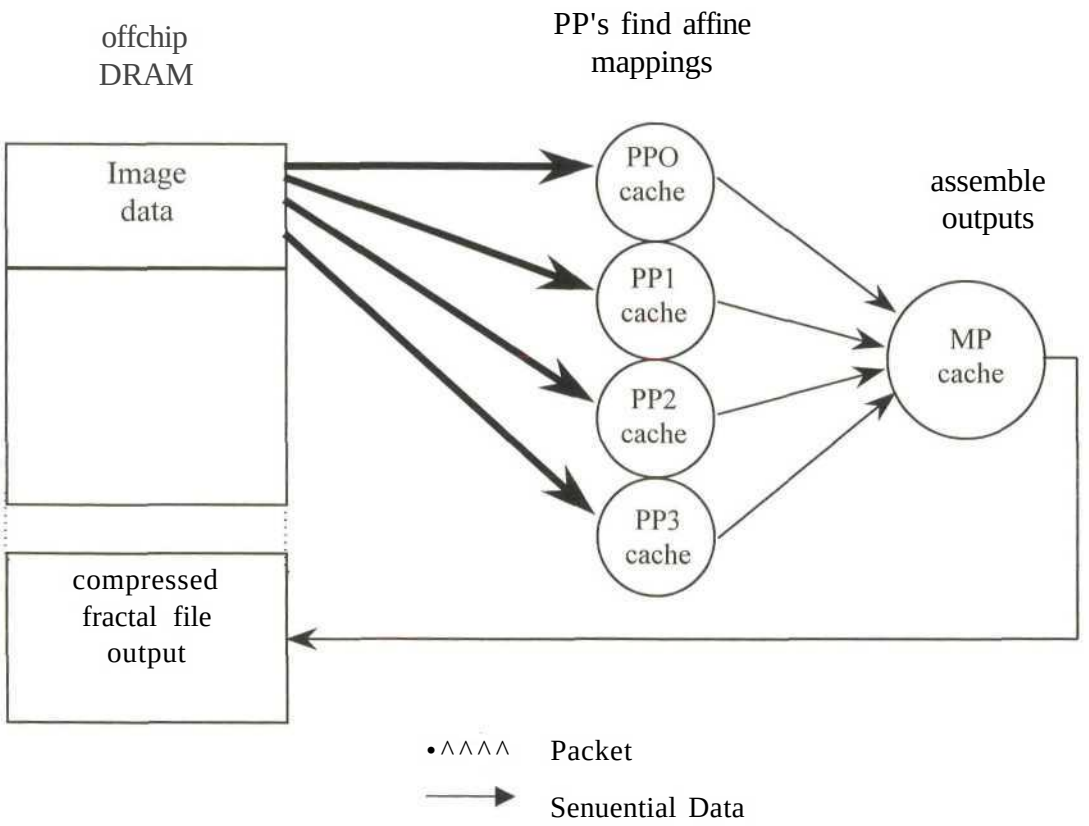


Figure 5.1 : Relationship between processors in parallel scheme

The role of the master processor in this may be seen as supervisory. It is occupied with receiving the affine coefficients calculated by the PPs and compiling them into a suitable

format for the file output file. The MP is also responsible for allocating work among the four PPs and controls synchronisation and communications between the MP, PPs and host.

The transfer controller is occupied in an asynchronous mode principally servicing the packet transfer requests posted by the other five processors. The requests are prioritised on a round-robin basis. This arbitration scheme was at the control of the programmer, and is the best option for giving equal support to the four PPs.

5.2.2. Two Pass Strategy

Examination of the encoding strategy reveals that each domain block will be compared to a great many range blocks. A comparison between two blocks involves computing the metric between them. The metric calculation involves the following preliminary results :

From equations (3.27), (3.38), (3.30) (for *R, s* and *o)* : when comparing domain block A with *n* pixels of intensities *di,...,d,,,* and range blocks, with *n* pixels of intensities r/,...,r,, :

The following quantities are needed:

$$\sum_{i=\backslash}^{n} d_i r_i ; \quad \sum_{/=!}^{n} d_i ; \quad \sum_{/=!}^{n} r_i ; \quad \sum_{i=\backslash}^{n} d_i^2 ; \quad \sum_{i=\backslash}^{n} r_i^2$$

Those quantities pertaining to the domain blocks (containing *d)* will be recalculated each time that domain block is involved in a comparison. Clearly this is grossly inefficient. The approach adopted was to scan through the possible domain blocks and compute these quantities in advance before the affine map search begins. These preliminary quantities are stored in the off-chip DRAM in a record structure. The complete freedom of code design (Section 4.2.1 - 4.2.4) offered by the MVP allowed the record structure to be custom designed and the memory for this storage to be allocated in such a manner that memory block boundaries coincide with the logical content. The records are structured that the quantities for a domain block translate directly from the spatial coordinates of that block in the image. This allows the precompiled parameters to be efficiently accessed.

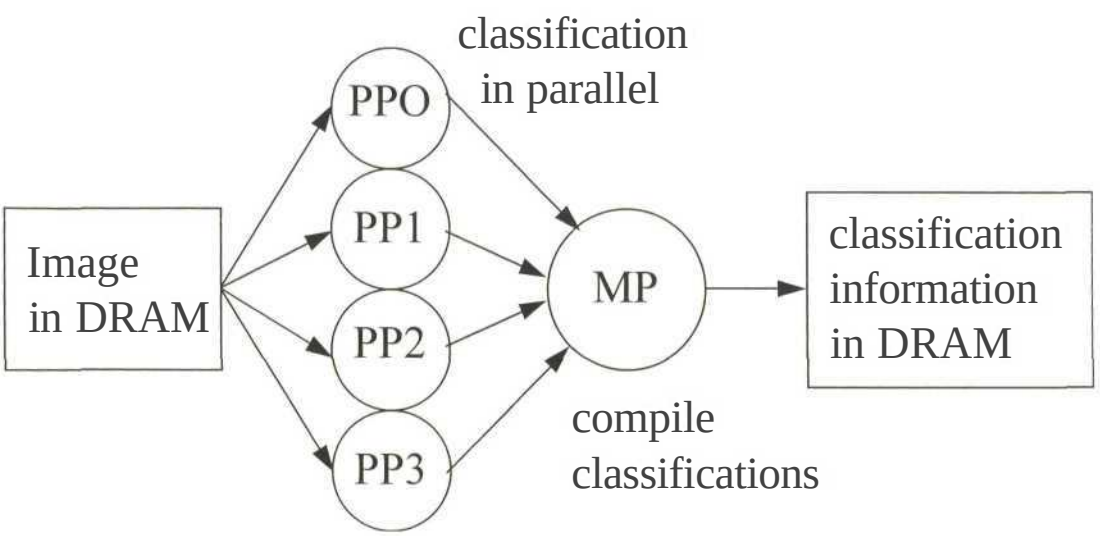


Figure 5.2 : Pass 1

The overall code for the process may be seen as a two pass system, or as a very high level pipeline. The first pass or stage is this precalculation stage, as represented in Figure 5.2 above. Figure 5.3 below represents the second phase, the affine map search previously described. A similar approach was first proposed by this author in [55].

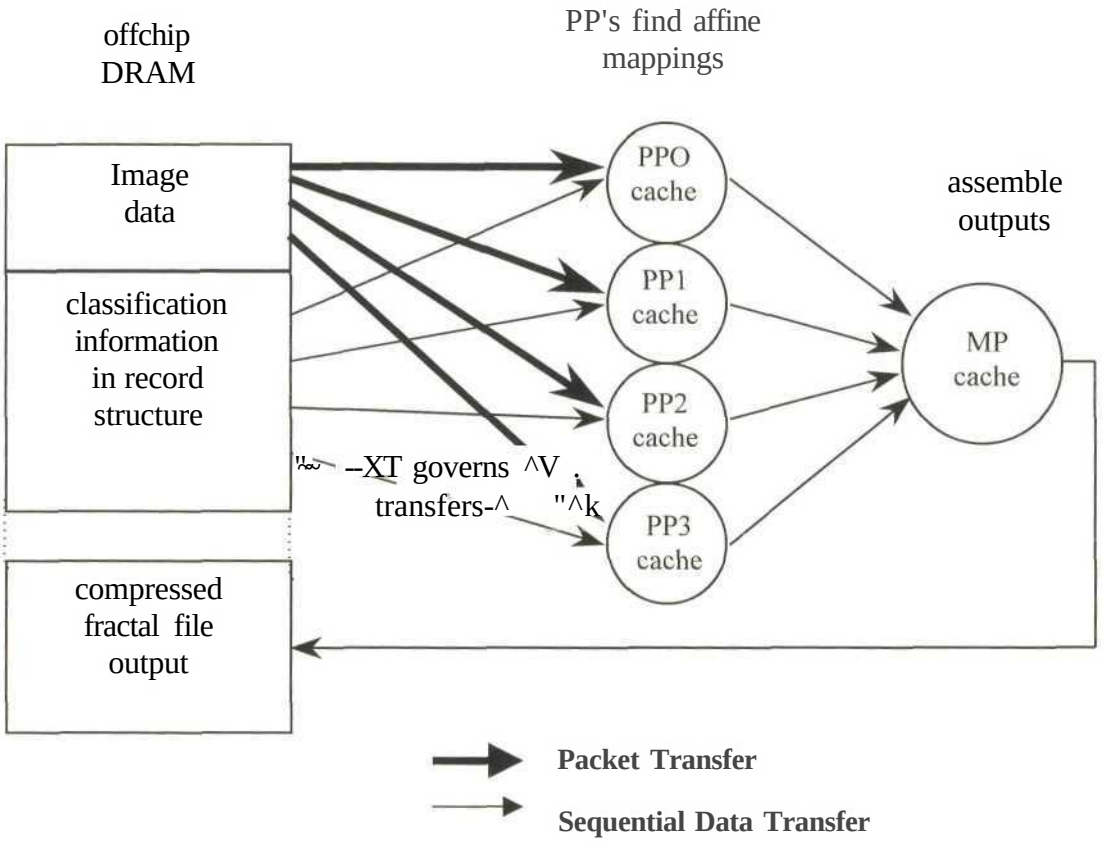


Figure 5.3 : Pass 2

5.2.3. Memory Configuration

Another important consideration is the flow of data with respect to memory resources. As described in Section 4.1.3, there are three main areas of memory available :

a) Bulk memory on the host platform

b) 4Mb of DRAM on the MVP board (also referred to off-chip memory since it is off the TMS320C80 per se)

c) 50kb of SRAM on the TMS320C80 itself. Of this memory, 10kb is associated with each of the PP's and the MP. (see Figure 4.2)

The host memory is used at the start of the compression process to temporarily store the image data. At the same time the MVP initialises itself under control of the master processor code. The MVP can directly access the host memory, but this is a slow process, so the data is transferred into the off-chip DRAM. This is dictated by the size of the data (typically 256kb for a 512 x 512 pixel 8 bit greyscale image) which would not fit in the faster on-chip SRAM. For optimal processing speeds, the data is transferred piecewise to the PP's onchip SRAM as

needed for the calculations. The PPs have faster access to their own SRAM (by an order of magnitude) than to the shared off-chip memory. One may view the on-chip memory as a collection of data caches. These transfers are in the form of packet transfers controlled by the TC.

In the context of the algorithm's demands there is not much room for changing this arrangement mainly because :

a) The host memory must be used as intermediate storage between the hard disk and the MVP off-chip memory because the MVP cannot directly access the hard-disk and vice versa.

b) During the processing the bulk data must be stored in the MVP off-chip memory because
   i) Repeated access to the host memory is very slow
   ii) The data is too large to fit in the on-chip SRAM (fastest memory)

c) During processing the data blocks currently under consideration should be stored in that processor's on-chip SRAM. The alternative of working directly from the off-chip DRAM would create queuing at the memory ports and a vast amount of traffic on the cross-bars. Such cross-bar requests are handled sequentially and would lead to significant idle time as the processors wait for data.

This arrangement is indicated in Figure 5.4

During the design of the code, at link time, the memory map of the processor was specified to locate the sections of the compiled code (eg data segment, code segment, stack segment, segments for malloc'ed memory to reflect the choices above. The TMS320C80 environment allows these segments to be allocated programmer-defined sizes. The memory was mapped in such a way that it takes cognisance of the block boundaries and byte-alignments in the memory structure, so that the data content can be sympathetic to the hardware limitations and preferences. Optimising these advanced aspects allowed performance gains to be had during packet transfers.

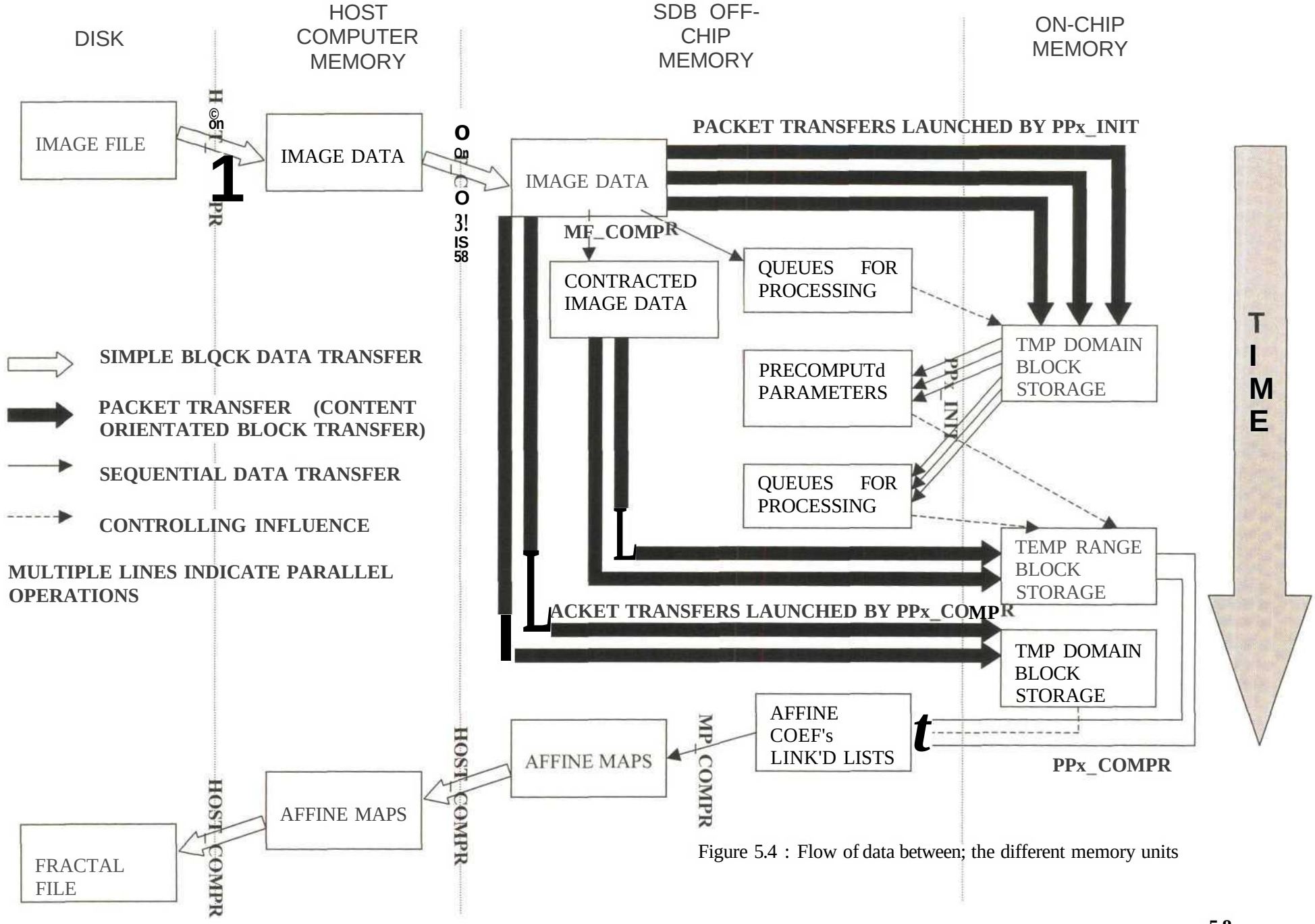


Figure 5.4 : Flow of data between; the different memory units

Referring to Figure 5.4 above showing the data transfers : The vertical axis represents time progress and the vertical columns represent the different areas of memory used by the code for data storage. The labeled blocks represent items of data and the column in which they lie in the diagram shows which area of memory they reside in. Their position down the vertical length of that column indicates at which temporal stage they are created or used. The arrows show the flow of data with the different kinds of transfer being as explained by the key. The name over the arrow indicates the item of code responsible for initiating that transfer where the following can be noted :

HOSTCOMPR : Code running on host processor during compression operation
MPCOMPR : Code running on master processor during compression operation
PPx_INIT : Code running on parallel processor* (for *x = 0,..,3)* during precalculation phase of compression operation
PPxCOMPR : Code running on parallel processor* (for * = 0,..,3) during affine map search phase of compression operation

(Sample code listings appear in Appendix D together with diagrams indicating how the units relate to eachother)

The challenge was then to utilise the on-chip SRAMs as caches as efficiently as possible. This involved several aspects :

a) Minimising the number of transfers between the processor's on-chip SRAM and the bulk off-chip DRAM to reduce the chance of contention induced delays.

b) Judiciously selecting blocks of data to be transferred to the very limited on-chip SRAM.

c) Controlling the data coherency problems that arise when processing data in a cache separate from the primary data records.

This process included intermediate-level pipelining (see Section 5.2.5.)

5.2.4. Interprocessor Communications and Protocols

The decisions outlined above concerning the use of processor configurations and memory utlisation presupposes protocol support for the design. The MVP provides for some specific modes of interprocessor communications [44]. These were considered in conjunction with the restrictions listed above at the design phase. As suggested, with the assortment of memory resources used, data coherency can be a problem. Data coherency must be maintained on two levels :

a) Whilst a processor is performing operations on data in its own cache as described above

b) Between MVP and host memory.

Generally the PPs run asynchronously and while in this mode there is no sharing of data between the different PPs due to the decisions on the parallel configuration. This minimises crossbar traffic and overhead processing time conventionally needed to monitor data coherency in such a parallel cache scheme. The PPs signal the MP at several crucial stages of the compression process to allow the MP to monitor progress. This is done by the use of

interrupts. This approach avoids having the MP waste time in loops polling the PPs and eliminates the communications traffic that would be so induced.

5.2.5. Code Flow

From the considerations above, the code was constructed and the temporal relationships between the processors defined as in Figure 5.5 below (see next page). It is a timing diagram of the compression process depicting the activities of the processors with respect to time. The vertical axis shows a percentage completion. This is an approximation based on the findings of the research, and obviously the exact values depend on the image under consideration. The vertical columns represent the different processors - host processor, master processor, transfer controller and parallel processors. The arrows between the different columns represent an event, with the direction of the arrow signifying the initiating processor at the arrow base and the responding processor at the arrow head. Note that the two bands of the diagram shaded in grey are the main processing phases of the two passes. In these phases the PPs run asynchronously and a great many data transfers into and out of the domain of the PPs occurs under the control of the TC. The volume of transfers is not accurately represented by the diagram - only a few transfers are listed. Section 3.8 explained that up to the order of 10^9 to 10^{10} packet transfers may be required. (This was reduced by several measures as described in the next chapter.

It was found experimentally by benchmarking the code that some time seemed to be lost waiting for these packet transfers. The solution was internal pipelining within each processor on the conceptual level where each PP operates on a particular block previously transferred to its on-chip SRAM cache whilst the next required block is transferred by the TC to a contiguous area of SRAM, with no wait cycles executed by the PP.

5.2.6. Summary of Compression System

To summarise, both the hardware used and the software developed incorporate pipelining and parallelism at every level :

On the lowest hardware and firmware level the processors of the MVP incorporate pipelining of floating point operations. Such operations may occur in parallel with the other execution units within that particular processor. On the programmer's level, these units are used to set up a pipeline allowing several instructions can be in successive stages of completion. Thus there is internal parallelism with underlying pipelining. The code has been written to exploit this.

On the next level of organisation, the five general-purpose processors (MP + 4 x PPs) have been used as the component processors to set up a high level parallel processing scheme. Within this there is some pipelining of data flow by the packet transfers that were set up.

This construction as a whole is then used as the execution unit of a top-level pipeline consisting of two phases - precomputation of frequently used parameters, and then the affine map search stage.

Sample code listings appear in Appendix D.

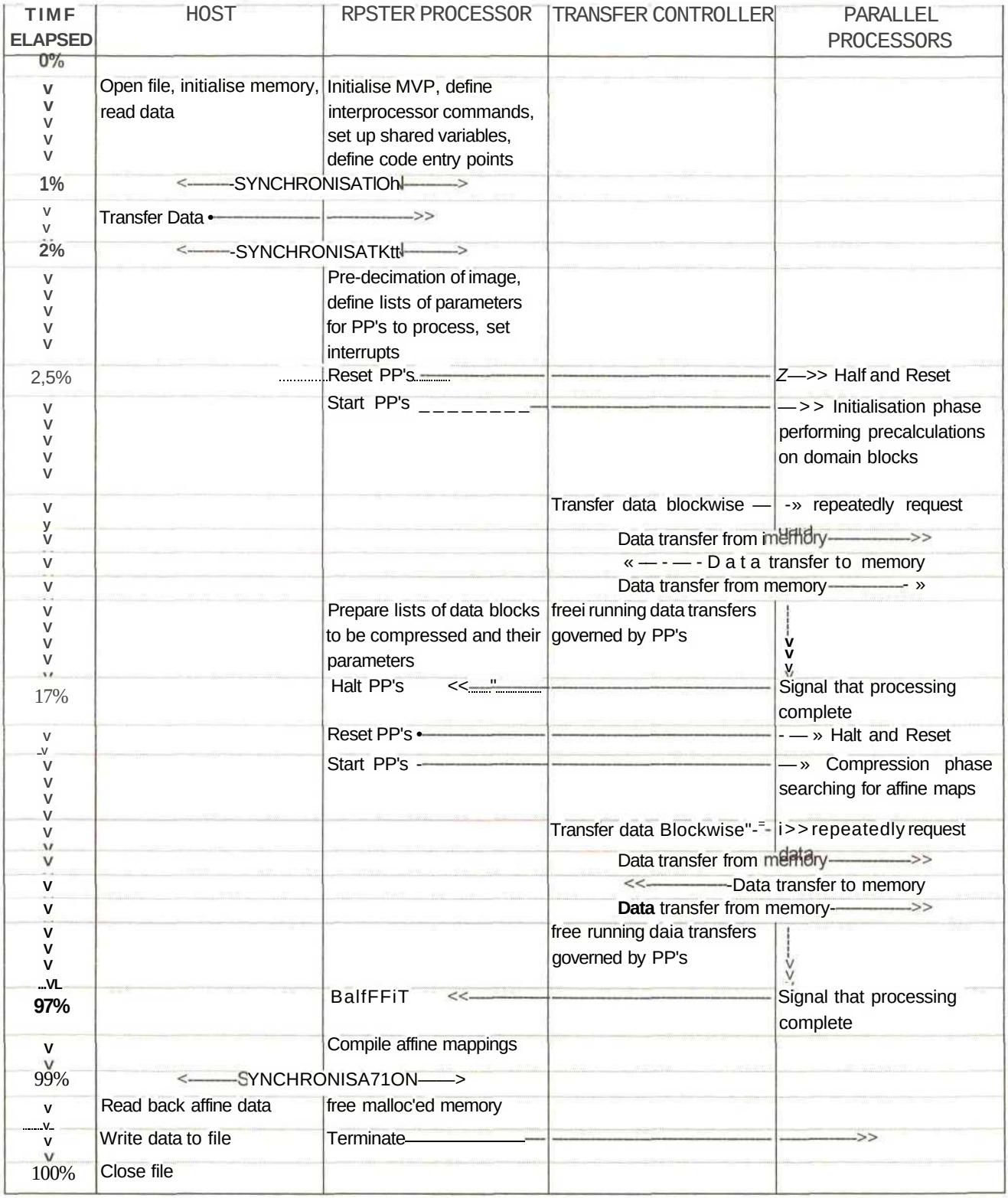


Figure 5.5 : Timing diagram of code flow and data passing

5.3. Structure Of The File Format

In identifying a suitable structure for the files containing the compressed affine mappings the obvious goal was to minimise the number of bytes used. The ratio of the original image data byte count to the number of bytes in the fractal file can be defined as the compression ratio.

5.3.1. Identifying a Basic Structure for the Compressed Files

Examining the problem of storing the affine maps, it would appear that the following information needs to be stored for each map :

(a square block or quadtree scheme using the assumptions of Section 5.1 is assumed)

x_R : x coordinate of that particular range block
y_R : y coordinate of that particular range block
L : dimensions of the range block
x_D : x coordinate of that particular domain block that maps to that range block
y_D : y coordinate of that particular domain block that maps to that range block
s : spatial contractivity factor for that mapping
r : relative rotation/flip required to map that domain block to that range block
s : grey level contrast factor between the domain block and the range block
o : relative brightness offset between the domain block and the range block

5.3.2. Parameter Minimisation and Byte Rationalisation

At first it would seem that this is a large number of parameters to store for each affine map. However, some structuring of the file format allows some parameters to be omitted at compression time and then inferred at decompression time. Further, some of the parameters for the affine mappings span a range of values much smaller than the resolution available from an 8 bit byte. Specifically each of the parameters can be rationalised as follows :

a) The coordinates x_R and y_R of the range block do not need to be explicitly stored. They can be inferred from the position of that mapping in the file sequence as an offset index, and knowledge of the image dimensions.

b) L : dimensions of the range block
In the initial scheme this is not stored since a range block size of 8 x 8 pixels has been preset.

c) The coordinates x_D and y_D of the domain block must obviously be stored, but not necessarily as absolute values. If the coordinates are divided by the domain skip factor, indices result which occupy the minimum number of bits for that domain skip factor. Again, they may be viewed as a pair of offset indices.

d) The spatial contractivity factor s has been predefined at a constant 2 and so need not be stored.

e) The relative rotation/flip r can assume only 8 possible values and so needs 3 bits.

f) The grey level contrast factor s and the relative brightness offset o can potentially have a range of 255 values and so each requires full 8 bit resolution. Results in [1] suggest that they may be requantised to lower bit resolution but there is a definite PSNR penalty.

g) Additionally a single toggle bit is stored to indicate whether a block is a flat block or a conventional affine mapping.

Further rationalisation can be achieved by noticing that flat blocks do not require any of the standard affine coefficients to be stored. It is sufficient to store the toggle bit and then the directly coded grey level of that block. In other words, only 9 bits will be needed to store all information pertaining to that block of 64 pixels.

The successfulness of this byte rationalisation can be determined to a degree by attempting to further compress the fractal file using some of the commercial compression programs such as WinZip. This gives experimental confirmation of the degree of residual redundancy. See Section 6.4.2 for sample results.

5.4. Decompression Algorithm

As indicated before, the most promising possible use for fractal image compression is for one-to-many broadcast applications. Thus there would be limited usefulness for a decompression scheme implemented on a specialised processing architecture. In any case, the decompression algorithm, which was discussed in Section 3.6, is computationally unintensive. It is presented below again.

Algorithm 5.2 : *PIFS Decoding Process*

for each iteration *q* do {
    for each affine map *wj* e *Wdo {*
        identify the domain *D,* e D required by the affine coefficients
        decimate *D,* to produce a contracted block C,
        rotate/flip *C,* to match the *Rj*
        apply $gi(z) = Sj \cdot z + Oj$ to each pixel value z in *C,* (contrast/brightness adjust)
        paste the block into the decoded image
    }
}

For the purposes of this research a basic decompression system was implemented on the host computer as per the algorithm.

The following design decisions may however be taken:

a) For fractal image decompression, any initial image may be used with the same final result. Without a priori knowledge of the image to be decompressed however, slightly faster convergence may be obtained by beginning with a uniform grey initial image (All pixels assigned a value of 127 - central in the 0 to 255 range provided by 8 bit resolution). Intuitively this seems reasonable since most real images would seem to have an average grey level of near 127. This is demonstrated in Section 6.1 for the test images used.

b) The order in which the different kinds of range blocks are decoded is not as arbitrary as it might first appear. They are best processed in the following order :

i) Flat blocks
For each iteration the flat blocks are decoded first since they will directly translate to their final value in the image during the first iteration of the decoding process. This tends to reduce the number of iterations required for convergence during decoding because many of the conventional affine mappings may map from areas including these flat blocks and will benefit from their immediate convergence.

ii) Affine map range blocks
The conventional affine maps are handled second to benefit from the immediate convergence of the flat blocks.

Note that this sequence is executed for each iteration. The exact number of iterations required for convergence depends on the image under consideration, but 10 is sufficient in most cases.

This system of handling the different kind of blocks as groups requires an initial pass over the file to extract the different kinds of blocks into groups. It might seem that it would be more efficient to store the mappings in this format at compression time, as this would eliminate this phase of the decompression, and remove the necessity to store the block type for each map in the file. This was not done because it would only save 1 bit per map, and instead require direct storage of the range block coordinates requiring up to 16 bits per affine map. In any case, this classification phase of the decoding is fast, and is executed simultaneously with scanning the information off disk.

Sample code listings appear in Appendix D.

5.5. Conclusion

The system implemented incorporates the features of the basic algorithm discussed in Chapter 3.5, with reductions to the otherwise unmanageable search process imposed by practical considerations. These reductions were selected to produce a workable algorithm whilst suffering minimal loss of quality in the final decode image. They revolve around reducing the possible domain pool to a finite, practical size by using only those domain blocks satisfying a fixed contractivity factor, using a block aspect ratio of 1, a selection of fixed block sizes and a selectable domain skip factor. These choices also took cognisance of limitations imposed by the processor architecture, memory structures, and concerns about an optimal format for the final file containing the affine mapping information. This Chapter has also presented the rationale behind the resources configuration chosen, and the development of the software. This process was controlled by the above design decisions and the original goal of producing a fast compressor, with some choices made in this favour possibly at the expense of some loss of final PSNR.

The parallel processors have been configured in a true asynchronous parallel style, each with dedicated memory cache. They each perform the search for affine maps on a private set of range blocks drawn from a list compiled by the master processor. As such it represents a true BDPA (block data parallel algorithm). The transfer controller is responsible for transferring all the data blocks in an efficient manner. The packet transfer structure provided for by the MVP environment is exploited fully. There is some pipelining of these transfers implemented to eliminate processor idle time. The master processor compiles the affine maps returned by the parallel processors. It is also responsible for ensuring data coherency and synchronisation issues between itself, the parallel processors and the host machine. A further design refinement was the implementation of a two-pass strategy where the first pass involves the

pre-calculation and storage of many parameters which are needed many times during the second pass search for affine maps.

The sympathetic treatment of the design goal, BDFP (block data flow paradigm) processor structure, memory considerations and peculiarities of the software environment has produced a synergistic working solution. Results and a more detailed discussion of some further refinements are presented in the next chapter.

6. RESULTS AND REFINEMENTS

6.1. Results of a Standard Solution on a Conventional Processor

Initially a basic fixed range block size fractal image compression scheme was implemented on a conventional sequential processor. The processor used was the host computer's Pentium I 133MHz processor. The design constraints and simplifications imposed were as listed in Section 5.1. Figure 6.1 below shows an original uncompressed bitmap image used as input to the compressor. It is of dimensions 512 x 512 pixels and has an 8-bit resolution (256 grey levels). The uncompressed file totalled 262 144 bytes. (Full resolution reproductions of all test images used appear in Appendix A)

Figure 6.1 : Original "GIRL" image (Source for original image : www.dip.ee.uct.ac.za)

The compressor used an exhaustive search scheme with no form of classification performed on the blocks. As might be expected, this led to a very slow compression time, but the resolution achieved from the decompressed image was the optimum in the context of the

imposed design restrictions. The process lasted for 90 minutes and resulted in 4096 affine maps. These maps were compiled into a file structure of the form described in Section 5.3, and yielded a file of size 23kBytes, giving a compression ratio (as defined in Section 5.3) of 11:1. This compressed information was then used as an input to the decompressor developed as per Algorithm 2 in Section 3.6. The initial image used as the starting point was all-grey (pixel values 127) and appears in Figure 6.2. (Shown at a reduced size of 128x128 pixels to save space).

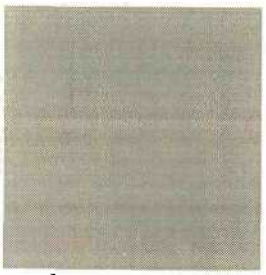


Figure 6.2 : Initial grey image input to decompressor.

Successive images in Figure 6.3 below show the output of the decompressor after 1, 2, 4 and 10 iterations (Figure 6.3 a), b), c) and d))

Figure 6.3 a) Decoder output after 1 iteration (PSNR 18.1dB)

Figure 6.3 b) Decoder output after 2 iterations (PSNR 19.9dB)

Figure 6.3 c) Decoder output after 4 iterations (PSNR 23.7dBJ

Figure 6.3 d) Decoder output after 10 iterations (PSNR 26.4dB)

The final attractor that the PIFS converged to (Figure 6.3 d)) was compared with the original uncompressed image and the PSNR was found to be 26.4dB (Applying equation 3.21 - Definition 5 in Section 3.4). The decoding process lasted 0,7 seconds for ten iterations. This is very fast considering that the basic decoding algorithm used wasunoptimised. Works such as [49, 4 ch 9] (see Appendix B) promise an order of magnitude improvement over this by direct decoding techniques. Nevertheless the decoding speed compares favourably with an optimised JPEG implementation such as incorporated in PaintShop Pro[1].

From the figures the following observations can be made :

a) A good approximation to the attractor appeared after 10 decode iterations
b) Some blockiness on an 8 x 8 pixel grid is apparent due to the fixed block sizes used and their regular arrangement. This is the aspect where adaptive block size schemes show a distinct advantage.
c) Despite the design restrictions made, some of which might appear extreme, the decode image displays a good visual fidelity to the original. Even the "difficult" detail of the hair has been adequately reproduced. This is one of the strengths of fractal techniques - given an appropriate domain pool even step discontinuities in the grey-level can be handled. JPEG on the other hand by design effectively incorporates low-pass filtering (from the

[1] Product of and copyright to : JASC Tnc
P.O.Box 44997
Eden Prarie, MN 55344 USA

truncation of the high frequency coefficients) and so tends to soften hard edges and discontinuities,

d) These decoding speeds suggest the feasibility of real-time video sequence decoding.

The convergence of the system was assessed by measuring the PSNR of the decoded image after successive iterations. Results appear below in Figure 6.4. for 3 of the test images used. The original images appear in Appendix A. There is a substantial difference in measured PSNR for the 3 images due to the content of the images. Inspection of the images in Appendix A shows that the both DRIVER and especially the CAR image incorporate hard edges and areas of dense detail that make them less suitable for the fixed block size scheme used.

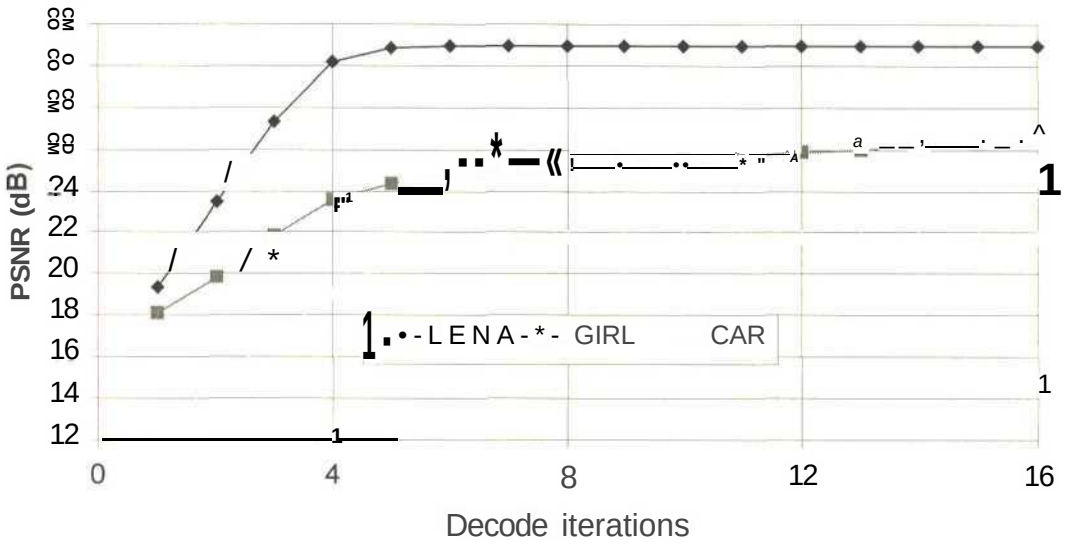


Figure 6.4 : Measured PSNR as a function of decoder iterations for several test images.

6.2. Results of Compression on Parallel System

To examine the performance of the parallel processing implementation, the full design presented in Sections 5.1 to 5.3 was implemented.

Figure 6.5 below shows the results of an exhaustive search compression - decompression performed on the standard "LENA" image with the parallel processing system. The compression process lasted 15 seconds and produced a file of 23 kbytes, or a compression ratio of 11:1. The PSNR of the decoded image is 31.0dB. The measured PSNR and the visual quality show good correlation to previously published results. This is confirmed by Table 6.1 below showing a summary of some previously published results, including this work. All are for 512 x 512 x 8 bit LENA.

Figure 6.5 : Decompressed LENA, PSNR 31 .OdB.

Table 6.1 : Summary comparison of published results.

PSNR	*Comp Ratio*	*Encoding time*	*Hardware*	*Technique*	*Reference*
34.56 dB	8.48:1	56.6 sec	IRIS 4D/35 Workstation	Quadtree, spatial domain	[4 ch 3]
31 to32dB	-14:1	3 to 15 sec	Silicon Graphics R4600	Domain pool reduction	[16]
31.93	12.3:1	593 sec	133MHz Pentium **I**	Hybrid VQ/fractal	[55]
31.0 dB	**11:1**	15 sec	**TMS320C80**	**Fixed, spatial domain**	**This work**
30.77	16.5:1	Not stated	Not stated	Frequency domain	[21]
29.38 dB	25.99:1	17.8 sec	IRIS 4D/35 Workstation	Quadtree, spatial domain	[4 ch 3]

The different results show a spread of PSNR's, compression ratios and encoding times. These three aspects of fractal image compression obviously represent mutually exclusive goals, and any particular implementation represents some trade-off between them, probably weighted towards some particular priority. Also note that the encoding time depends heavily on the hardware platform used. Additionally it can be commented that different works quite probably have used "Lena" images from different sources. Different scans of the same image will not be identical due to the following:

a) Different scanner hardware/software combinations will not yield the exact same grey-level pixel value for any particular point in the image.
b) The original print which appeared in a 1960's issue of a men's magazine will not have identical grey levels in different copies of the issue
c) The "standard" image is actually a crop from the picture in the magazine, and there is no guarantee the same crop is used.

These variations will probably result in a non-identical PIFS for different scans of the same image, even when using the same fractal compressor. Thus one cannot directly compare PSNR's measured in different works exactly, but they can be used as a general guide for comparison. Further, many works use 256 x 256 pixel scans of "Lena".

Nonetheless, the results reported in this work at least show good correlation with the general trend of reported results.

As expected, running the algorithm distributed over the TMS320C80's four parallel processors yielded a speed gain factor close to 4, over a benchmark implementation using only one of them. Obviously there is some overhead incurred in managing the four parallel processors and ensuring data coherency, but this was catered for by the master processor and transfer controller. There is no reason to believe that this approach cannot be extended to greater numbers of processors. When allowing such modifications as direct encoding of flat blocks and quadtree partitioning (and hence variable sized blocks), the range blocks take different lengths of time to encode. It became important to assign the range blocks to the parallel processors in such a way that they are all fully occupied all the time. The system used scans through the image row by row assigning adjacent blocks to different processors. This is to allow for the tendency of real images to have detail concentrated in localised areas (see Figure 6.9)

As anticipated at the design stage, congestion of crossbar traffic did seem to occur when each of the processors requests multiple data blocks. This was alleviated by the pipelining where the processors issue transfer requests to the TC allowing anticipated data to be transferred in the background while the processor works on the previously transferred block.

6.3. Refinements to the Basic Scheme

6.3.1. Domain Pool Selection

Initially a fixed domain block size scheme was considered. When selecting this domain block pool, a maximally sized pool results when the blocks are drawn from overlapping areas of the original image positioned so that each block is shifted only by one pixel row relative to the previous block in either the x or y direction. As commented in Section 3.8, for 16 x 16 blocks on a 512 x 512 pixel image, this yields 247 009 blocks. This large domain pool implies very lengthy affine map search procedures. Further, it is actually not reasonable to include all these possible blocks since a pair of domain blocks taken from locations shifted only by one pixel will not differ considerably from each other, and on this basis only every «'" block is selected. In other words the top left pixels of successively selected blocks were positioned n pixels apart in the original image. This is called the "domain skip factor", or DSF in this work. See also [12].

Simple control over the size of the domain pool searched was had by stipulating the domain skip factor. Common choices of DSF result in different pool sizes as shown in Table 6.2.

Table 6.2 : Relationship between DSF and domain pool size

DSF, 512x512 image	Blocks (16 x 16) in domain pool
1	247 009
2	61 504
4	15 376
8	3 844
16	961
32	225

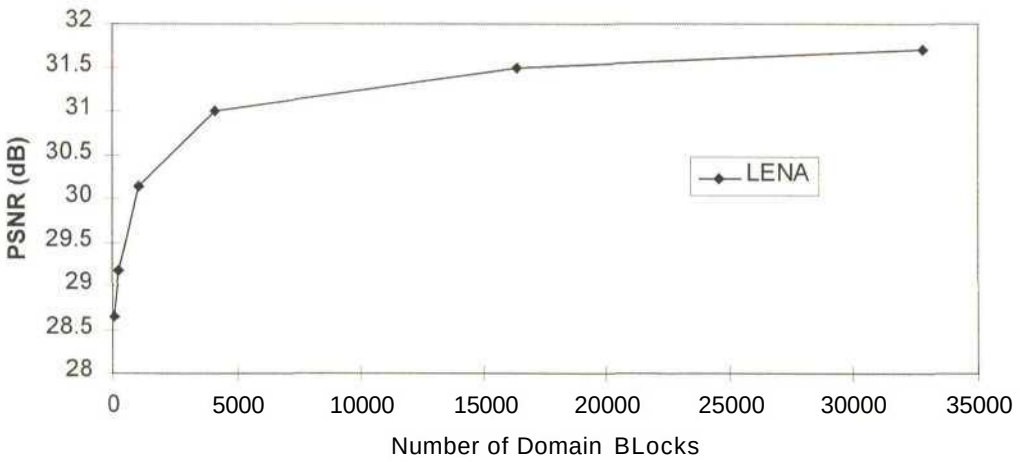


Figure 6.6 : Effect of reducing number of domain blocks searched on PSNR.

Figure 6.6 shows how the PSNR is affected by reducing the domain pool. The expected trade-off between domain pool size and fidelity is apparent in Figure 6.6, but there is justification in reducing the number of domain blocks searched if encoding speed rather than absolute fidelity is the goal. The encoding time gains are confirmed by Figure 6.7 below. As might be expected, the domain block pool size is approximately linearly related to the encoding time.

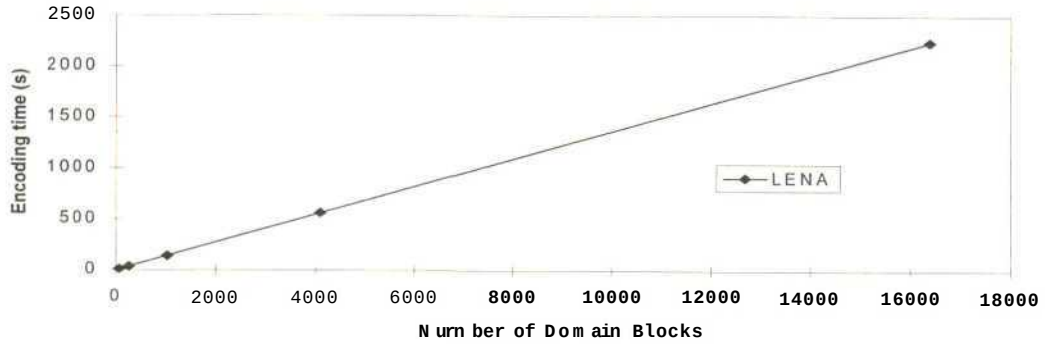


Figure 6.7 : Trade-off between domain block pool size and encoding time

From these results it was established that a DSF of 8 inflicts only a modest loss of **quality** whilst producing a dramatic reduction in compression time, (from table 6.2 : 3844 blocks in domain pool). For the purposes of this research, this was the selected value used in the following experimentation. Figure 6.8 a) to d) below show the results of a compression - decompression for different DSF values. A cropped, magnified portion is displayed in each case. PSNR values in each case are the same as in Figure 6.6. Visually, this affirms that a choice of 8 for the DSF is a reasonable compromise between encoding time and fidelity loss.

Figure 6.8 a) : DSF-4

Figure 6.8 b) : DSF - **8**

Figure 6.8 c) : DSF = 16

Figure 6.8 d) : DSF = **32**

6.3.2. Content Related Domain Block Selection

It is important to note that when using only the DSF to limit the domain pool, selection of the members of the domain search pool occurs in a manner unrelated to the actual block content by selecting possible domain blocks at regular grid co-ordinates from the image. Some trade-off between compression time and fidelity is inevitable, but clearly an improvement can be expected if domain search pool reductions are performed in a manner less randomly correlated with the information content of the image. If this can be achieved, the fidelity loss for a given domain pool size will be less than the situation depicted in Figure 6.6.

Traditionally this may be achieved by integrating some selection criteria into a block classification scheme [8, 9], where some blocks may be entirely rejected from the domain pool at classification time. Some possible classification schemes were discussed in Section 2.3.2.1. Probably the best known is the classification by relative brightness of the 4 quadrants of a square block [4]. Unfortunately this requires a lot of computations and was deemed inappropriate for this work in view of the goals discussed in Section 5.1.

It was observed that a significant number of blocks within many real images are "smooth" in that they possess little variation in grey level. In other words, they contain little "texture" or "edge" information. Conceptually, if a two-dimension cosine transform were performed over such blocks, most of the energy would be concentrated in the DC term, with high frequency coefficients close to zero. Schemes for block classification according to these Fourier coefficients have been proposed by [9, 22] (See Section 2.3.2.5.), but they suffer from significantly increased encoding complexity. A less computationally intensive calculable measure of the texture within a block is given by the second moment of the pixel intensities within the block. This was presented by this author in[17], and a similar approach is reported in [16]. The square of this second moment gives the variance of the pixel intensities. The formula used is

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \qquad (6.1)$$

where there are n pixels in the block with grey levels x_i. $\bar{x}$ is the mean grey level of the block. Expanding equation (6.1), it is seen that it requires many quantities that are also precalculated for the calculations in an RMS comparison between blocks during the first pass of the compression phase, (see Section 3.7 , equations (3.26, 3.28, 3.29),). Thus a pre-examination of blocks by variance incurs a minimal increase in encoding time.

Considering first of all the range blocks, the assumption is that if the variance of the block is very low, then it is essentially flat, that is the values of all the pixels in that block are very near to each other. If this threshold is chosen carefully, then that block may be represented simply by the average grey-level of the block, without visible loss of quality. The benefits of this are three-fold. Firstly if a range block may be stored simply as a grey level rather than an affine map, then there are fewer coefficients to store for that block, leading to greater compression ratio for the image. Secondly, if a range block is found to have a low variance, it is not necessary to search for an affine mapping at the encoding stage, so thousands of block comparisons (and packet transfers) are avoided, leading to improved compression speed. The last benefit is at decoding time. Blocks directly coded with their grey-level decode exactly on the first iteration. Since these parts of the image converge faster, other conventional affine maps that might map from areas including these directly calculated blocks also converge faster.

To justify this, the 512 x 512 Lena image was divided into 8x8 blocks (yielding 4096 blocks over the whole image). The variance over each of the blocks was computed. Figure 6.9

below shows the spatial distribution of the variance level in the Lena image. White blocks have a very high variance, and dark blocks a low variance. Clearly, a high proportion of the blocks have very low variance. This is confirmed by Figure 6.10, the frequency distribution of the variance values in the 512x512 pixel LENA image.

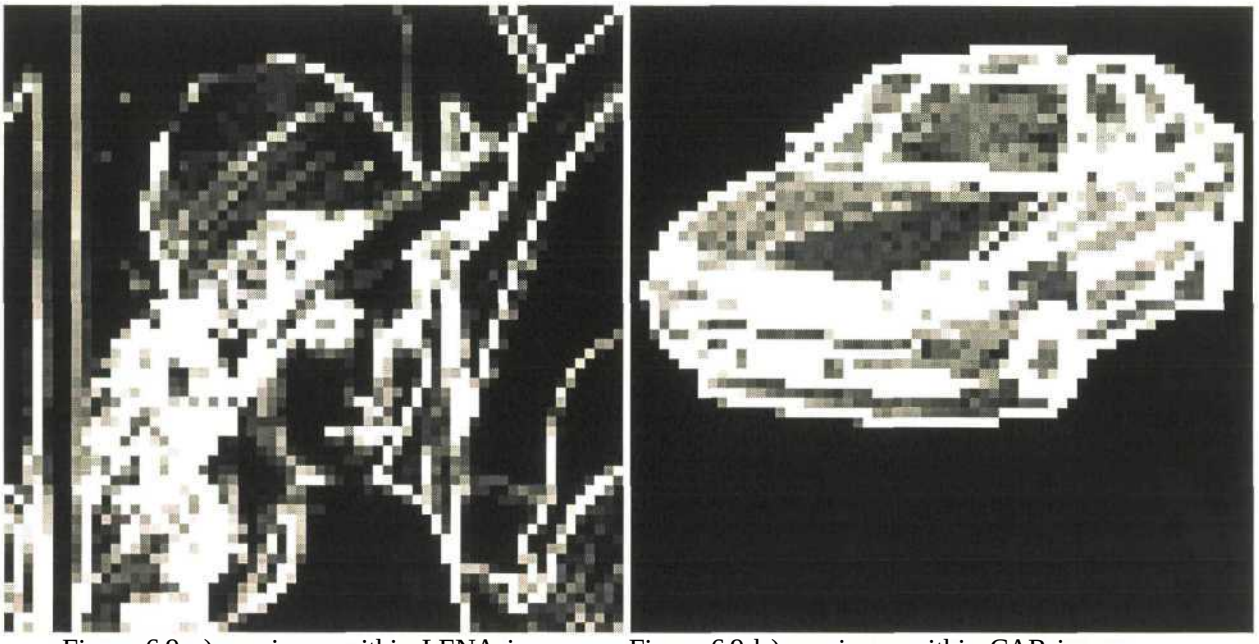


Figure 6.9 a): variance within LENA image

Figure 6.9 b): variance within CAR image

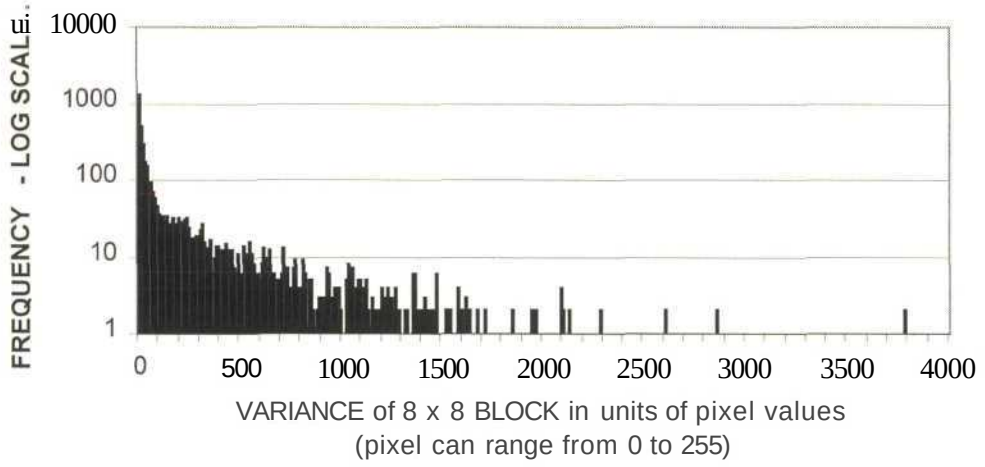


Figure 6.10: Frequency distribution of block variances for 512x512 Lena image.

The results indicated that 1271 of the 4096 blocks (31%) had variances of 10 or less. That is to say, on average the pixel values in each of those blocks differed by not more than $\sqrt{10}$ grey-levels (or $\sqrt{10}/255 \times 100 = 1{,}2\%$) from the mean for the block. When these blocks are directly encoded, visually, results are nearly indistinguishable to those obtained for complete encodings. Figure 6.11 below shows the PSNR penalty inflicted by higher variance cut-offs.

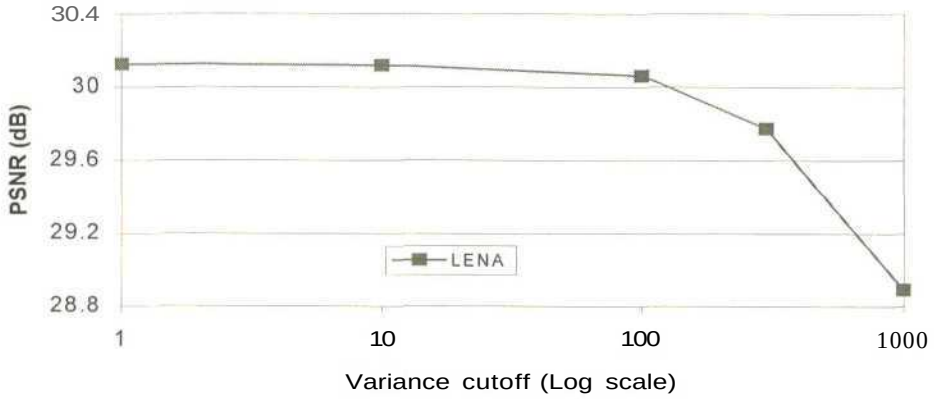


Figure 6.11 : Selecting optimum value of variance cut-off.

When treating low variance range blocks as flat, the encoding speed gains measured were closely related to the percentage reduction in the number of range blocks encoded by full affine mappings.

For demonstration purposes, the results presented here are for images of 512 x 512 pixels (8 bit resolution). Encodings were performed using fixed range block sizes of 8 x 8 pixels. See Appendix A for the original images used. Only the range blocks were considered for elimination on the grounds of low variance. That is, all domain blocks were considered in the comparison.

From the data used for Figure 6.12, encoding time could be reduced by more than 50% by setting the variance cut-off to 10. From Figure 6.13, the measured PSNR suffers only mildly from this process, decreasing from 30.14 dB to 29.68 dB for the LENA image.

Typical images have block variances that range between close to zero and just over 4000 units. In practice a variance cut-off of 100 (2,5% of the range of values) seems to be a reasonable ceiling (see Figure 6.12), giving a marked reduction in encoding time, with a decoded image that visually differs little from a version coded fully by affine transformations.

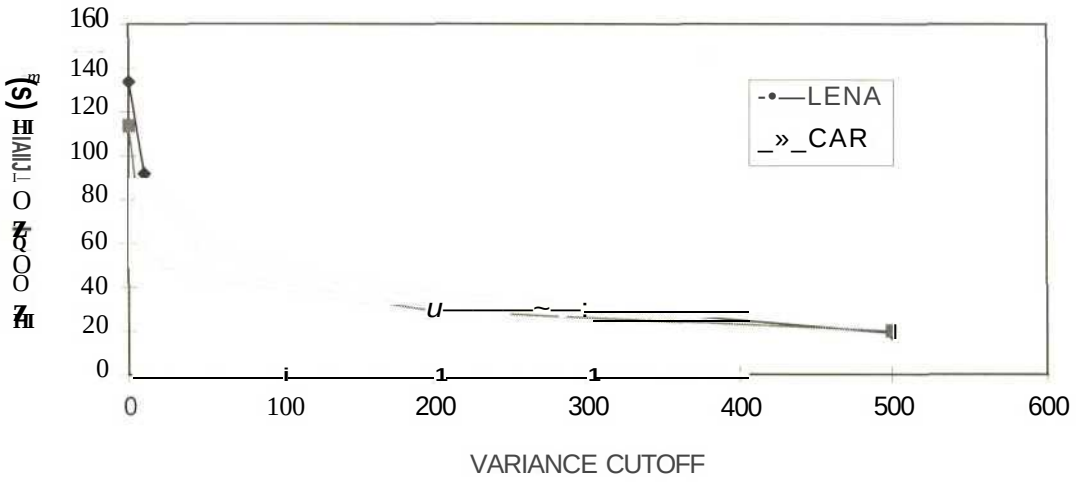


Figure 6.12 : Effect of treating low variance blocks as flat on the encoding time.

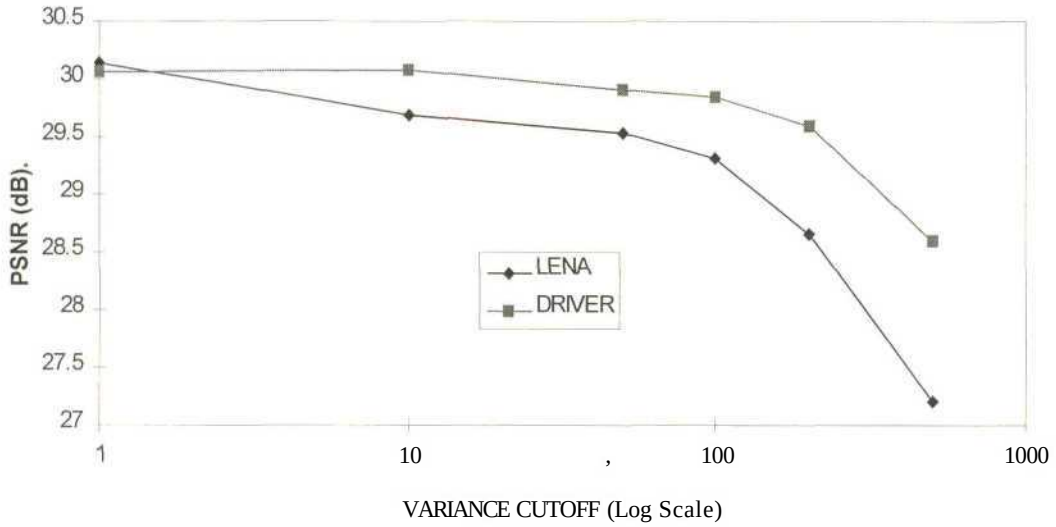


Figure 6.13: Effect of treating low variance blocks as flat on the PSNR of the decoded image

Higher variance cut-offs lead to noticeable blockiness in decoded images, with relatively modest incremental encoding speed gains. Additionally, larger proportions of the mappings are no-longer true affine maps, so some of the advantages of fractal compression schemes such as superresolution and decode size independence are lost. Thus rather than apply such unreasonable cut-offs to the range blocks, a more modest value such as 25 was applied to both the domain and the range blocks.

The concept was applied to domain blocks during the first pass of the compression algorithm. While the blocks are being examined to compute such preliminary results as the mean grey-level $\bar{x}$ and the summation of the square pixel values $\sum^{n} I\hat{}*,)^2$ (to be used during block comparisons), these results are used to calculate the variance of the block. Blocks with low variance are eliminated from the record structure of classified domains. With domain blocks, overlapping is allowed, so there are very many more domain than range blocks, so this is actually where the more significant gains may be made. This approach allows most encoding time and effort to be devoted to the more detailed areas. Thus domain pool reduction may be achieved such that those blocks discarded from the domain pool are unlikely to produce many affine mappings anyway. When setting equal variance cut-offs for both the domain and range blocks, encoding speed improves with *the square* of the cut-off value, since both the domain pool and range lists are similarly reduced. As an extension, both domain and range blocks may be classified according to the range in which their variance lies. As is conventional, block comparisons across class boundaries are prohibited.

The LENA image below in Figure 6.14 was compressed using fixed block sizes, and variance cut-offs of 100 for both domain and range blocks. A reduced set of 1024 domain blocks was used. The compression time was 19 seconds, and yielded a compression ratio of 27.8 : 1, or a file of 9396 bytes. When decoded, the PSNR was measured at 29.3 dB.

Figure 6.14 : LENA image decompressed from 9kb file

Larger domain pools reduced the blockiness slightly, **but** at great encoding time expense. The image in Figure 6.14 is most poor in areas of high detail like the eyes. This remains little improved by increasing the size of the domain pool. It would seem that an adaptive scheme allowing for variable range block sizes would better address this problem, (see [9, 12])

6.3.3. Close-enough Encoding

An obvious way of increasing compression speed (and one which is widely implemented) is to avoid an exhaustive search through the chosen domain pool by searching only until a domain block *D-*, producing a sufficiently close affine mapping to the range block *Rj* under consideration is found. In other words, using Equation 3.15, *D,* is sought so that $d(fC\backslash(R_lxI\backslash w_i(f))<$ some tolerance. Once this is achieved, the algorithm moves to the next range block. This presents some interesting possibilities though, because clearly those domain blocks positioned near the beginning of the domain pool listing may be used more frequently for affine mappings than those occurring later. This was exploited by avoiding the obvious sequential row-by-row scanning of the image when generating the domain pool. To see why this is not a sensible choice, consider the DRIVER or the CAR test images in Figure 6.15.

Figure 6.15a): original 512 x 512 x 8 bit DRIVER image (shown at reduced size)

Figure 6.15 b) : original 512 x 512 x 8 bit CAR image (shown at reduced size)

The first few rows of the image consist only of uniform grey level blocks, and will produce few useful domain blocks. Occupying the first portion of the domain pool with such blocks would slow the compression appreciably because they will be scanned through for every range block, before getting to the more detailed blocks from the centre of the image. The arrangement used specified a domain block search sequence that pulled possible domain blocks from different rows of the image in a cyclical pattern. This yielded a 15% reduction in encoding time for the DRIVER image. More sophisticated schemes have been proposed to address this problem, such as localised domain pools [14], or tree structure search paths [19] for the domain pool. However this comes at the cost of a considerably more time consuming first pass in the compression process.

Figure 6.16 : Visual results of compression/decompression on CAR image
(Variance cut-offs = 25, encoding time = 6 sec, C/R =18:1, PSNR = 21.8dB)

Figure 6.16 above demonstrates the cumulative benefits of domain pool reduction on variance criteria, close-enough encoding and careful domain pool ordering. As with the DRIVER image, the CAR image also contains a lot of featureless blocks and so benefits significantly from these techniques. The 512x512 pixel x 8 bit image was encoded in only 6 seconds.

6.4. Analysis of Solution Implemented

6.4.1. Successfulness of Resource Exploitation

Resource exploitation was examined from several aspects :

a) Processor idle time
Utilisation of the parallel processors was examined initially. The full compression scheme was implemented using only one of the PPs running the compression code under control of the MP. The other three were idle. The code was streamlined to remove any of the overheads associated with the parallel environment. The performance that was then measured could be regarded as a benchmark to which a multiprocessor system could be compared. The process was then distributed over the 4 PPs using all the refinements discussed in Chapter 5.2. The resulting compression times were 26 - 29% of those measured when using only the single PP depending on the test image used. It might be expected that the times would reduce to 25%, but the 1 - 4% overhead measured can be accounted for as follows. Considering Figure 5.4 (Section 5.2.5.), 80% of the processing time is spent in the second pass (affine map search) of the compression process. This is where the unexpected time increase occurred (established from pausing the process at key points both when using 1 and 4 processors).

i) There is no PP performance penalty from parallel environment overheads because the system designed dedicates the MP to these tasks.

ii) The refinements made to the basic compression algorithm generally result in unknown, variable processing times for the different range blocks. The asynchronous approach described in Chapter 5.2.1. alleviates this, but it is inevitable that at the end of the second pass, the 4 PPs will not finish at exactly the same time, resulting in some of the processors having a small amount of idle time.

iii) Having multiple PPs continually requesting data asynchronously may lead to queuing delays at the TC and crossbar congestion (see below).

The transfer controller seems to have been well utilised, but because of the asynchronicity of requests placed on it, it is inevitable that it has periods of idle time. However, this was not considered a problem, since it is a task-dedicated support resource. Its value lies not in achieving a zero idle time, but rather in meeting service-on-demand requests timeously so as to avoid idle time in the other processors.

The master processor has a supervisory role, and additionally has a service-on-demand function. Thus some idle time is unavoidable. Referring to Figure 5.4 (Section 5.2.5.), there is room for improved use of this resource. This processing time was reserved for specific use performing intra-frame calculations when the system was extended to video compression (see Section 7.2.).

b) Crossbar congestion
This was minimised using the pipelining of transfer requests as discussed in section 5.2.5. In balance, the 1 - 4% overhead of the parallel scheme may be considered sufficiently

low to conclude that this approach was successful. (Before pipelining, up to 15% overhead was measured).

c) Memory usage
By design, maximum exploitation of the fastest memory resource (on-chip SRAMs) was approached. The careful design decisions allowed this to be achieved without a detrimental increase in cross-bar traffic. Also at design time, the memory divisions were defined in a data content related manner, so as to maximise the potential benefits of packet transfers. This was discussed in Section 5.2.3.

6.4.2. Efficiency of the File Structure

As described in the previous chapter, efforts were made to establish the minimum number of bits that could possibly be stored to represent the required parameters. Assuming the accuracy of this, the file structure implemented must be regarded as minimal in this respect. Other economies were had from careful ordering of the different affine maps within the file. Consideration must be given to the decoding process when selecting the file ordering. The order in which the different kinds of mappings are decoded is not as arbitrary as it might first appear. They are best processed in the following order:

i) Flat blocks
For each iteration the flat blocks are decoded first since they will directly translate to their final value in the image during the first iteration of the decoding process. This tends to reduce the number of iterations required for convergence during decoding because many of the conventional affine mappings may map from areas including these flat blocks and will benefit from their immediate convergence.

ii) Larger affine map range blocks
The conventional affine maps are handled second to benefit from the immediate convergence of the flat blocks. , again because they have a greater influence on the final image than the smaller blocks, and so processing them earlier encourages faster convergence to an attractor.

iii) Smaller affine map range blocks
Successively smaller blocks are decoded and are responsible for bringing out the detail of the image.

Note that this sequence is executed for each iteration. The exact number of iterations required for convergence depends on the image under consideration, but 10 is sufficient in most cases.

This system of decoding the different kind of blocks as groups requires an initial pass over the file to extract the different kinds of blocks into groups. It might seem that it would be more efficient to store the mappings in this format at compression time, as this would eliminate this phase of the decompression, and remove the necessity to store a parameter indicating the block type for each map in the file. This was not done because it would only save 2 bits per map, and instead require direct storage of the range block co-ordinates requiring up to 16 bits per affine map. Thus in fact the obvious file ordering of placing the affine maps in sequence according to their position within the image is in fact the optimal. The chief savings are then in the compressed file size, because positional co-ordinates need not be explicitly stored, and can be inferred by treating the position in the file as an offset index within the image. In any case, this classification phase of the decoding is fast, and is executed simultaneously with scanning the information off disk.

Some indication of the residual redundancy contained in the compressed files may be seen from the results of compressing that compressed file using one of the commercially available file archiving utilities such as WinZip, or PKZip. This was done using WinZip[2] (based on LZW compression). In each case the file could be further compressed by at most 5% indicating that the byte usage was efficient. Note that this reflects only on the efficiency of the file structure developed, *not* on the thoroughness of the fractal compression process at reducing the data to approach the information entropy.

6.4.3. Comparison with JPEG and GIF

The two primary criteria in assessing compression formats are algorithm complexity, and compression ratio. In general there will be a balance between the two, and the approach taken depends on the requirements of the particular application. Broadly speaking bitstream techniques like GIF provide only modest, but lossless, compression, using an algorithm that is simple, implying fast encoding and decoding times. JPEG is at the other extreme, offering compression ratios of up to 20 and more, depending on image content, but with the penalty of severe algorithm complexity. Fractal Image compression also falls into this category, but it features the peculiarity of being inherently asymmetrical. That is, the decompression process is relatively undemanding computationally, while the compression process consists of massive systematic searches. Sample images appear in Figures 6.17 to 6.19.

Table 6.3 : Qualitative comparison of compression formats

	GIF	JPG	**Fractal**
Loss	Lossless	Lossy	Lossy
Compression ratio	~ 1:1 to 1:4	~ 1:5 to 1:20	-1:5 to 1:20
Encoding speed	high	medium	very slow
Decoding speed	high	medium	high to very high
Decode quality	lossless	medium to high	medium to high
Decode size	pixel replication	pixel replication	superresolution
Effect of partial file loss	proportional image loss	proportional image loss	catastrophic

[2] WinZip® Copyright © 1991-1998 by Nico Mak Computing, Inc.
http://www.winzip.com

Figure 6.17

LENA image 256 x 256 pixels x 8 bit compressed using GIF

File size :	63kB
Compression Ratio :	0,98:1
PSNR:	Lossless
Compression time :	lsec
Decompression time :	1 sec

Figure 6.18

LENA image 256 x 256 pixels x 8 bit compressed using fractal algorithm developed in this research

File size :	12kB
Compression Ratio :	5,33:1
PSNR:	30.2dB
Compression time :	1 lsec
Decompression time :	0.8sec
Variance Cutoff:	10

Figure 6.19

LENA image 256 x 256 pixels x 8 bit compressed using JPEG algorithm of Paintshop Pro

File size :	8kB
Compression Ratio :	1:8
PSNR:	32.3dB
Compression time :	0.7sec
Decompression time :	0.7sec

The relevant parameters are listed next to each image. They are generally self-explanatory, but the following points should be noted :

a) The GIF image is indistinguishable from the original since it is a lossless compression scheme.
b) The PSNR is an imperfect measure of visual quality, and produces better values with JPEG which favours smooth grey-level transitions than with the fractal scheme the strengths of which lie more with precision of edge information.
c) The compression and decompression times of the JPEG scheme are symmetric and very low due to the great amount of man-hours invested in optimising this commercial scheme. As such it is unfair to directly compare these times with the time of the fractal decompression time using a somewhat less than optimal scheme. See Section 3.9 for mathematical analyses of computational complexity.
d) The compression and decompression times of GIF and JPEG are largely independent of the image content, and depend only on the image size. This is not true for the fractal scheme implemented in this research which exploits local self-similarity and featureless areas to benefit both compression ratios and encode/decode sizes.

It may be stated that fractal image compression holds a unique position in the complexity - compression ratio tradeoff. It offers high compression ratios, and promises to be competitive with JPEG at a given SNR point for suitable images. It labours under a severely intensive encoding algorithm, orders of magnitude slower than JPEG, but it has super-fast, simple decompression rivaling that of GIF for speed, and clearly surpassing JPEG in this regard.

6.5. Conclusion

This chapter has shown that if the encoding speed which is fractal image compression's Achilles heel can be addressed, the scheme has many positive features that may then find practical application. The practical component of this research involved the implementation of a parallel algorithm to perform the image compression. The algorithm was adapted to the hardware resources available as described in the previous chapter. Care was taken to optimise the process within these constraints and it was determined that the resource were well utilised. This involved ensuring that all the available processors had minimal idle time, preventing data transfer delays and congestions, and carefully using the different types of available memory for their best suited tasks to minimise access times. Essentially the on-chip memory was used as a fast data cache. This necessitated careful control over the data coherency on a low level. The parallel scheme implemented proved intensive on the programming effort required to ensure synchronicity and minimum idle time. The performance of the developed system clearly demonstrated the convergence of the PIFS approximation of still images. As a first approach, an exhaustive search scheme with no intelligent form of domain pool reduction was implemented. PSNRs consistent with those reported in other works were measured for a compression/decompression cycle on the standard LENA image. The visual fidelity of the attractor approximations produced for this and other test images was subjectively very good. The compression times achieved were competitive with other reported research as could be expected from the processing resources used.

Several improvements to the software were made, including high level pipelining of data transfers to eliminate some delays that had been measured. Some critical tight loops within the code were optimised in directly coded assembly language to exploit the parallel execution units available within each of the processors, and the possibilities for low level pipelining of code flow that this low level parallelism facilitates.

Compression times were then reduced by a factor of up to 5 by such improvements as close-enough encoding, and domain pool reduction. This was achieved with a minimal impact on

measured PSNR or perceived visual quality of the decoded attractors. Specifically, domain pool reduction was achieved by two means. Initial domain selection was stipulated by specifying a domain-skip factor (positional offset between top left corners of domain blocks) of greater than the obvious 1. Intelligent domain pool reduction was achieved by discarding potential domain blocks possessing a low variance of pixel intensities. The assumption is that these are essentially featureless blocks and would be unlikely to lead to useful affine mappings. Similarly, range blocks of low variance were directly encoded by their average grey level. This further reduced compression times, and also benefited decompression speeds and the compression ratios achieved.

Trial implementations of some traditional methods of compression time reduction were attempted. These included adaptive quadtree partitioning to concentrate encoding effort on areas of higher detail, and block classification on localised brightness criteria. It was found that these refinements generally benefited the fidelity of the decoded attractor, but at the cost of significant increase in compression algorithm complexity. They were thus regarded as contrary to the primary aim of this research - namely to maximise compression speed, even at the cost of some loss of fidelity.

Comparisons between fractal image compression and the established JPEG and GIF standards was then presented. It was shown both qualitatively and quantitatively that fractal techniques are indeed promising alternatives offering a compression ratio to fidelity trade-off broadly speaking similar to JPEG.

7. EXTENSIONS AND RECOMMENDATIONS

7.1. Compression of Colour Images

Most of the existing body of literature has concentrated on grey scale images (see Chapter 2). In the early 1990's some significant papers [56, 57] were published dealing specifically with the application of fractal techniques to the compression of colour images. The most obvious approach to multispectral images is to work in the red-green-blue (RGB) colour space. Each of these three colour planes may be treated independently and coded as for a grey-scale image.

Preliminary tests using RGB format colour images were conducted. The test images used were of dimensions 512x512 pixels and 24 bit resolution. (File size 786 432 bytes). The images were split into their three component colours (each of 8 bit resolution), and each encoded separately as for grey-scale. Recombination was done after encoding.

a) BLUE component b) GREEN component c) RED component

Figure 7.1 : Component colours of LENA image in RGB format, each displayed as a grey-level map

Encoding times and compressed file sizes increased by a factor of approximately three over a grey scale image of the same dimensions, as might be expected. This is non-optimal as considering Figure 7.1 a) to c), the three component colours clearly bear strong content resemblance, and may be regarded as the ranges of three different intensity functions over the same domain area. Separately encoding them introduces a lot of redundancy.

Figures 7.2 and 7.3 below respectively show the original colour 512 x 512 x 24 bit LENA image, and the results of a compression/decompression process performed on it. Encoding time was 86 seconds, and resulted in an image which has a high qualitative fidelity to the original. The compression ratio was 22:1.

Figure 7.2 : Original colour 512 x 512 x 24 bit LENA image.

Figure 7.3 : Compressed/decompressed colour 512 x 512 x 24 bit LENA image.

[56] proposed a system based on the traditional block-based coding. The scheme exploits the similarities between the different colour planes by starting with a conventional coding of blocks in the dominant

colour plane. This serves as a prediction for the corresponding block in the other planes. These dependent codes are derived by successive refinement, and thus may be stored more compactly than independently calculated codes. The system developed is also applicable to luminance-chrominance colour space. (YUV format.) Reported compression ratios range from 20:1 for a fixed block scheme, to 60:1 for a quadtree partitioning. [4 ch 2] suggests using YIQ format luminance-chrominance colour space. The Y (brightness) channel (which represents the perceived image brightness) to be encoded at high resolution as for a grey-scale image. The human visual system is much more tolerant of degradation on the chrominance channels (I representing hue and Q representing saturation). They may be encoded at much lower bit rates, and even be truncated to lower than 8 bit intensity resolution. Spatial decimation is also possible with an acceptable impact on fidelity. [58] proposes a similar approach, and makes the observation that if a domain-range mapping pair is chosen for best results in the Y channel, the same (in the spatial sense) geometric mapping can be used in the other planes. Only the brightness and offset affine coefficients are determined uniquely for the three channels. This allows only 12 coefficients to be stored per mapping (compared to 8 for grey scale, and 24 for full independent coding of the three channels). This principle of common geometric transformations was applied in this work, in RGB format, to which it is equally applicable. This was found to yield results nearly indistinguishable from those in Figure 7.3, whilst the compression ratio improved to around 40:1.

7.2. Compression of Video Sequences

The move from still images to video sequences may be regarded as an extension from two-dimensional to three-dimensional space [59]. The two spatial dimensions may be examined for redundancy as for still images. Additionally redundancy may be observed in the temporal dimension (ie common information in successive frames). A typical example of temporal redundancy in a sequence of frames would be the unchanging background behind a moving car, or a talking person. Currently there are several compressed video formats that have become well known largely through their use on the internet, and email file exchange. They generally work by examining individual frames to exploit spatial redundancy based on one of the common still image formats. Temporal redundancy reduction is generally achieved by examining the frames for stationary segments.

7.2.1. Existing Compressed Video Sequence Formats

a) MPEG

The moving picture experts group' (MPEG) were responsible for MPEG compression (file extension .MPG). It is one of the highest resolution formats [60] available, using l,5Mbit/s to transmit 30 frames/s of up to 325 x 240 pixels. MPEG is based on JPEG still image compression to address the spatial redundancy issue. Temporal redundancy reduction is by motion prediction. Each frame is treated either as an 1 (intra) frame (full JPEG encoding), a P (predicted) frame or a B (bi-directional) frame. Blocks in P frames are predicted using difference coding of the best matching block from the previous I or P frame. B frames are positioned between I or P frames and are determined from interpolating between these. At least every 8^{th} frame must be an I frame to ensure that the sequence doesn't diverge from the original.

b) GIF Animated

GIF animated was developed for low frame rate (~5 frames/sec) motion, such as cartoons and other artificial sources. It consists of a sequence of frames each encoded as a still image using GIF. The file extension is also .GIF, and it is decodable with any of the more recent *still* image viewers such as

[1] A joint effort by the International Standards Organisation (ISO) and the International Electro-technical Commission (IEC)

ACDSee[2], and more importantly, most recent internet browsers (without any "plugins"). Fidelity within any particular frame is good, as might be expected from GIF, but the low frame rate limits the range of applications for which it is suitable. It is too jerky for full motion video, and the bitstream type encoding makes it far less suited to "real" images than cartoon type images which feature flat areas of identical pixels, allowing a reasonable compression ratio.

c)AVI

Audio Video Interleave (file extension .AVI) [61] is one special case of the Resource Interchange File Format (RIFF). It is a flexible format allowing for variable frame rates. The format is *interleaved* such that video and audio data are stored consecutively in an AVI file. The audio is based on the Wave (file extension .WAV) format developed for Windows. A video block consists of a single frame, and the following audio block contains the audio associated with that video frame. The video frames can be uncompressed bitmap type images, or may be compressed in one of a variety of formats (as indicated in the header information. Usually it is some form of bitstream compression.

Additionally, the RealMovie format (file extension .RM) can also be found in use on internet sites. However it is intended mainly as a very low bitrate format (28kbit/s), and is generally only used for small (up to 100 x 100 pixel) frame sizes.

7.2.2. Application of Fractal Techniques to Video Sequence Compression

The potentially fast decoding of fractal encoded still images makes it an attractive proposition for frame based video. Obviously the classic drawback to fractal techniques of long encoding time precludes real-time compression, at least with the current state of the art processor systems. As a result it is most suited to broadcast applications, with real time compression only likely for very-low fidelity applications such as video-phone

One of the first papers (1992) on the topic [62] describes a system based on fixed block size fractal compression of an initial frame. Successive frames are coded by transmitting new affme maps only for those blocks that have altered since the previous frame (as determined by difference mapping). Unaltered blocks reuse the previous affine map. When new maps are determined for range blocks in a new frame, the previous frame is used for the domain pool. Results indicate that reasonable fidelity can be had at compression ratios (for the whole sequence) of 76:1 and even higher. [59] proposed a system treating a short sequence of frames as a three-dimensional entity. The standard two-dimensional PIFS is extended to three dimensions, and the sequence is encoded by searching for 3-D (cubic) affine maps within the 3-D space. This could lead to huge search spaces, so a local search pattern is used. A qualitative assessment of the results is difficult, and the search problem intense, but the philosophy is promising. Others such as [63] have developed hybrid systems where blocks are encoded using DCT or fractal techniques.

For the experiments conducted as part of this research it was decided to implement a traditional video compression approach of treating each frame as a still image. The still image compressor developed during this work was the basis. Temporal redundancy is exploited similarly to [62] by only fully encoding every 3rd to 5th frame (designated I or initial frames). Between these are P or projected frames which only use new mappings for blocks that differ from the corresponding block in the previous frame by an RMS value exceeding a selectable threshold. The other blocks containing stationary fragments of the image utilise the same map used in the previous frame. The situation is depicted in Figure 7.4 below.

[2]: Copyright 1994 - 1997 ACD Systems, Ltd, www.acdsystems.com

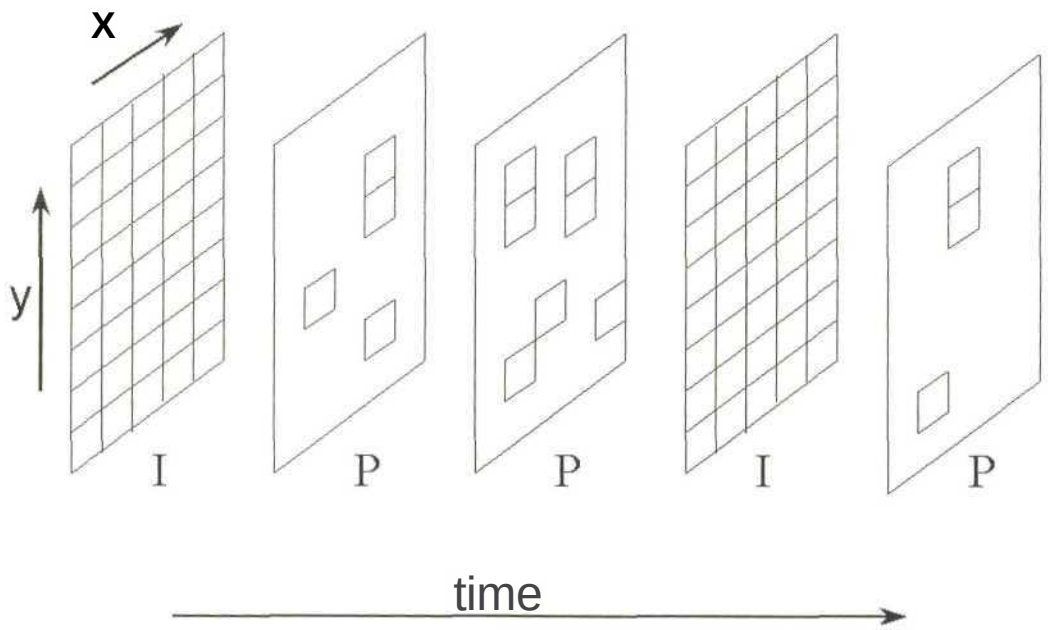


Figure 7.4 : Fractal based video sequence compression scheme.

Practically, the frame displayed at any moment is the content of a two-dimensional array in memory. Each successive frame decoded is overwrites this array. With a suitable source signal, much of a P frame is instantaneously available. The new mappings are simply iterated into the existing array. Many of these new maps take as domain blocks areas of the image remaining from the previous frame. These areas have already been converged to their final state, so the new maps mapping from them converge much quicker (2-3 iterations) than is the case for conventional decoding of a still image.

This form of projected and interpolated frame coding has proven successful with MPEG, but it is even better suited to the iterative decoding of fractal schemes. Any particular frame, either I or P, may be decoded by applying the affine transformations to an initial grey image as for still images (and inserting "borrowed" transformations in the case of P frames). Convergence will result after a number of iterations. In general, successive frames in a sequence are not very dissimilar to one another. Thus, if when decoding any of the frames, the starting point used is the previous frame (and not the grey image), the starting point will already approximate the attractor to a degree. The result is that usually *amaximum of two* iterations per frame is required to track the motion. This may be thought of as a kind of innate exploitation of temporal redundancy exploitation unique to fractal schemes.

These observations make real-time decoding of fractal compressed video sequences highly practicable. The investigations conducted into video sequence compression were regarded as an extension of the main work, and no further optimisation was performed. Nonetheless, the results did demonstrate the feasibility of fractal based video sequence coding.

7.3. Conclusion

The practical work of this research concluded with an application of the developed system to colour images, and later video sequences. Colour image compression by fractal techniques has not received much attention in the existing literature, and it was demonstrated that good visual fidelity may be achieved. The approach used worked in RGB space, but improvements in the compression ratio and encoding speed would be expected if working in one of the luminance-chrominance spaces.

Promising results were observed for compressing video sequences. Temporal redundancy was exploited by transferring mappings for stationary fragments to successive frames. Real-time decoding is feasible, by noting that the original frame sequence can be approximated and tracked by ^*changing* attractor. This changing attractor can be produced by applying iterative decoding where at every iteration (or every second iteration) a slightly different PIFS (representing the different frames) is applied to the continually changing output.

8. CONCLUSION

Digital image representation tends to be inefficient and requires a large amount of storage space or transmission bandwidth for high resolution representation. However, real images contain much redundancy which can be exploited in a variety of means to improve the efficiency of representation. This has become particularly important in view of the volume of images transmitted over the internet, and the limited bandwidth resources available. Many compressed image formats have evolved to address these needs, with GIF and JPEG gaining particular prominence. Fractal image compression represents an attractive alternative to these. It exploits the local partial self-similarity that may be identified within images as a form of redundancy that can be eliminated. This process is based on identifying contractive affine transformations between these areas exhibiting similarity and storing only the parameters of these mappings in the compressed file. Each transformation maps one domain portion of the image to a smaller range portion. Once the image has been completely covered by such range portions, each with an affine mapping, it is said to be encoded. The describing collection of affine transformations is loosely what is called a partitioned iterated function system. The PIFS may be decoded to produce an attractor approximating the original image by iteratively applying the affine transformations to any initial image. The key criterion for assuring convergence of this process is that each of the affine transformations be individually contractive in both the spatial and pixel intensity dimensions'. The theory for this is presented in the context of a three-dimensional image model, and the affine transformations are defined in a metric space. Assessing the contractivity assumes the use of some metric in this space. This work demonstrates that the contractivity criteria are met under the RMS metric. It was used in most of the practical experimentation as it was found to yield results broadly representative of human visual perception.

The successfulness of this approach hinges on identifying the best possible affine transformations. Thoroughness requires an exhaustive, computationally intensive search process. Traditionally, the current state of the art in computing platforms has been insufficiently fast to allow the format to gain commercial appeal, however the repetitive nature of the search process suggests suitability for adaptation to a parallel processor architecture. This gave the main design goal of the project - to implement a parallelised version of the algorithm, capable of performing the compression process in minimum time. It was decided that where appropriate, some absolute fidelity to the original image would be sacrificed if it allowed significant reductions in the encoding time.

Even using modern processors, any practical implementation requires optimisation and algorithm acceleration. A number of (now standard) approaches have been published. Some of these were adopted in this research, including non-unified search spaces implemented by classifying domain and range blocks on local brightness or grey-level texture criteria. Attempts were made to speed up the affine map search procedure reducing by the size of the domain block pool to be searched. Various restrictions were imposed, taking cognisance of limitations imposed by the processor architecture, memory structures, and concerns about an optimal format for the final file containing the affine mapping information. Considerable effort was invested in optimising the code over the hardware resources.

High-end computing architectures have evolved enormously, with move away from fine-grained massively parallel application-specific processors towards reasonably generic RISC processors. The current state of the art in imbedded processors are coarse-grained modestly parallel RISC processors, such as the Texas Instruments TMS320C80 used in this research. This project develops an algorithm for implementing some of the findings of fractal

[1] This has now been shown to be not strictly true - see section 2.4.2 on eventual contractivity.

compression research over this modestly parallel architecture with the aim of addressing the traditionally high compression time.

The TMS320C80 features coarse grained parallelism at a high level. Each of the constituent processors features internal intermediate and low-level parallelism arising from multiple execution units. The different processors are individually semi-application specific. The master controller is used in this work for all floating-point operations, with the bulk of (integer) calculations distributed among the parallel processors. Data transfers are accelerated by a dedicated transfer controller. The state of the art crossbar network linking the processors allows them to be configured to form a high level pipeline or parallel structure, as selected for this work.

For this research the parallel processors were configured in a true asynchronous parallel style, each with dedicated memory cache. They each perform the search for affine maps on a private set of range blocks drawn from a list compiled by the master processor. As such it represents a true BDPA (block data parallel algorithm). This work makes extensive use of sophisticated data content orientated packet transfers, as facilitated by the transfer controller. There is some pipelining of these transfers implemented to eliminate processor idle time. Assembly language was used in time-critical tight loops to best exploit the low level pipelining and parallelism possible within the code of each processor. The master processor compiles the affine maps returned by the parallel processors. It is also responsible for ensuring data coherency and synchronisation issues between itself, the parallel processors and the host machine. A further design refinement was the implementation of a two-pass strategy where the first pass involves the pre-calculation and storage of many parameters which are needed many times during the second pass search for affine maps.

The code was structured to achieve very high levels of processor utilisation, by giving due regard to scheduling data flow over the crossbar network and external memory ports. Data coherency, timing control, run-time code location and data code location were manipulated to achieve maximum throughput. The different types of available memory were used for their best suited tasks to minimise access times. The on-chip memory was used as a fast data cache. This necessitated careful control over the data coherency on a low level. The sympathetic treatment of the design goal, BDFP (block data flow paradigm) processor structure, memory considerations and peculiarities of the software environment has produced a synergistic working solution. After implementation, benchmarking the performance of the compressor confirmed that the idle time of the different processors had been kept to an absolute minimum.

The performance of the developed system clearly demonstrated the convergence of the PIFS approximation of still images. Measured PSNRs were consistent with those reported in other for a compression/decompression cycle on the standard LENA image. Domain pool size reduction achieved by domain skip factor and discarding potential domain blocks possessing a low variance of pixel intensities reduced compression times by a factor of approximately 5, at minimal PSNR penalty. PSNR values of approximately 30dB were measured for a 512 x 512 pixel x 8 bit LENA under these conditions.

Comparisons between fractal image compression and the established JPEG and GIF were undertaken. It was shown both qualitatively and quantitatively that fractal techniques are indeed promising alternatives offering a compression ratio to fidelity trade-off broadly speaking similar to JPEG. GIF is a high resolution format, but suffers from moderate compression ratios. JPEG at the other end of the complexity spectrum offers good resolution and compression ratios by eliminating low energy high frequency information to which the human visual system is relatively insensitive. Fractal techniques were shown to offer a quality to compression ratio tradeoff similar to that achieved with JPEG. Although the decompression has the potential to be faster than JPEG, fractal techniques suffer from a

severely intensive encoding algorithm, orders of magnitude slower than JPEG. This can be addressed by utilising modern high-performance processors such as the TMS320C80. The relevance of this research lies in the observation that, as evidenced by history, it can be expected that this level of processing power will in the very near future be available from commercial desktop persona! computers. This will then allow the technique of fractal image compression to gain acceptance.

It is felt that the design met the original goal. Examining the results revealed several areas where improvements could be made. Further investigation into an adaptive domain and range block partitioning scheme such as quadtree partitioning which allows the economies of covering featureless areas with larger blocks could be beneficial to perceived fidelity. Performing the search in a transformed space could also yield PSNR gains, but it seems that this would probably not improve the compression speed, since the reduced dimensionality of the search is countered by overheads in performing the transformations.

The developed system was applied to colour images. This is an area that has not received very much attention in the existing literature, and it was demonstrated that very good visual fidelity may be achieved. The approach used worked in RGB space, but improvements in the compression ratio and encoding speed would be expected if working in one of the luminance-chrominance spaces. The practical work concluded with an extension of the developed system for compressing video sequences. Temporal redundancy was exploited by transferring mappings for stationary fragments to successive frames. Real-time decoding is feasible, by noting that the original frame sequence can be approximated and tracked by a *changing* attractor. A series of PIFS are used to target this attractor.

Fractal techniques have application in both still images and video sequences. The asymmetry of the encoding/decoding process makes them best suited to broadcast applications such as the internet. Fractal image compression is based on a relatively simple concept, and the algorithm can be adjusted to target a range of points on the fidelity/compression ratio tradeoff curve, allowing adjustment to suit a range of applications. It can offer good fidelity with a range of "real" image, and should be very well suited to animated or cartoon type images which have large flat areas. The decode size independence is another feature of interest in broadcast applications. Computing platforms have reached a performance level allowing fractal image compression to broaden its field of interest from research circles to commercial application. Ideally, the institution of a non-proprietary standard would allow a unified development effort by concerned parties, as was achieved with JPEG. The following man-hours of development would bridge the gap to general acceptance.

REFERENCES

[I] Taub, H., Schilling, D. L., *Principles of Communication Systems, 2$^{n<1}$ Ed,* McGraw Hill, 1986

[2] Lindley, C. A., *Practical Image Processing in C,* McGraw Hill

[3] Wallace, G. K., *The JPEG Still Picture Compression Standard,* Communications of the ACM, Vol 34, No 4, April 1991, pp 30-44

[4] Fisher, Y. (editor), *Fractal Image Compression,* Springer-Verlag, 1995.

[5] Texas Instruments TMS320C80 Technical Reference., TI 1995 (Item SPRU149)

[6] Jacquin, A. E., *Fractal Image Coding: A Review,* Proc. IEEE, October 1993.

[7] Anson, L.F., *Fractal Image Compression,* BYTE, October 1993.

[8] Jacquin, A. E., *Image Coding Based on a Fractal Theory of Iterated Contractive Transforms,* IEEE Transactions on Image Processing, Vol. 1, No. 1, January 1992.

[9] Wohlberg, B., de Jager, G., *A Review of the Fractal Image Coding Literature* IEEE Transactions on Image Processing, vol. 8, no. 12, Dec. 1999

[10] Barnsley, M., *Fractals Everywhere,* Academic Press, San Diego, CA, 1988

[II] Barnsley, M. F., Sloan, A. D., *A Better Way to Compress Images,* BYTE, Jan 1988, pp215 - 223

[12] Kominek, J., Advances in Fractal Compression for Multimedia Applications, Dept Computer Science, Waterloo University, to be published

[13] Davoine, F., Chassery, J., *Adaptive Delaunay Triangulation for Attractor Image Coding,* Proc. 12th International Conference on Pattern Recognition, 1994

[14] Monro, D. M., Dudbridge, F., *Fractal Approximation of Image Blocks,* Proc. ICASSP 3 (1992)485-488

[15] Ramamurthi, B., Gersho, A., *Classified Vector Quantisation of Images,* IEEE Trans. Comm., COM-34:1105-1115, November 1986

[16] Saupe, D., *Lean domain pools for fractal image compression,* in: Proceedings from IS\&T/SPIE 1996 Symposium on Electronic Imaging: Science & Technology ~ Still Image Compression II, Vol. 2669, Jan. 1996.

[17] Uys, R. F. E., *Parallel Implementation of Fractal Image Compression,* Proc. Comsig-98, September 1998 pg 143-148

[18] Fisher, Y., Jacobs, E. W., Boss, R. D., *Fractal Image Compression Using Iterated Transforms,* Technical Report 1408, Naval Ocean Systems Center, San Diego, CA, 1991

[19] Bani-Eqbal, B., *Speeding up fractal image compression.* Proceedings from IS\&T/SPIE 1995 Symposium on Electronic Imaging: Science & Technology Vol. 2418: Still-Image Compression, 1995.

[20] Kominek, J., *Algorithm for fast fractal image compression,* Proceedings of SPIE, Volume 2419, 1995.

[21] Au, O. C, Liou, M. L., Ma, L. K., *Fast fractal encoding in frequency domain,* in Proc. IEEE ICIP-97, Santa Barbara, Oct. 97.

[22] Zhao, Y., Yuan, B., *Image compression using fractals and discrete cosine transform,* Electronics Letters, 30(6): 474-475, March 1994

[23] Wohlberg, B., de Jager , G., *Fast image domain fractal compression by DCT domain block matching,* Electronics Letters 31 (1995) 869-870.

[24] Cardinal, J., *Faster fractal image coding using similarity search in a KL-transformedfeature space* in: Fractals: Theory and Applications in Engineering, M. Dekking, J. L. Vehel, E. Lutton and C. Tricot (eds.), pp. 293-306, Springer-Verlag, London, 1999.

[25] Wohlberg, B., de Jager , G., *On the reduction of fractal image compression encoding time* in : Proc. 1994 IEEE South African Symposium on Communications and Signal Processing COMSIG'94, Oct. 1994.

[26] Saupe, D., Hartenstein, H., *Lossless acceleration of fractal image compression by fast convolution,* ICIP-96 IEEE International Conference on Image Processing, Lausanne, Sept. 1996.

[27] Reusens, E., *Overlapped adaptive partitioning for image coding based on the theory of iterated function systems,* Proc. oflCASSP 1994, Adelaide.

[28] Ho, H.-L., Cham, W.-K., *Attractor image coding using lapped partitioned iterated function systems,* in: Proc. ICASSP'97, Munich, 1997.

[29] Huertgen, B., *Contractivity of fractal transforms for image coding,* Electronics Letters, 29 (1993)1749-1750.

[30] Huertgen, B., Hain, T., *On the convergence of fractal transforms,* Proceedings of ICASSP-1994 IEEE International Conference on Acoustics, Speech and Signal Processing, Vol. 5, pp. 561-564, Adelaide, 1994.

[31] Forte, B., Vrscay, E. R., *Solving the inverse problem for function/image approximations using iterated function systems, I. Theoretical basis,* Fractals 2,3 (1994) 325—334.

[32] Fisher, Y., *Siggraph '92 Course Notes,* Dept. Mathematics, Technicon Israel Institure of Technology, 1992

[33] Oien, G. E., Baharav, Z., Lepsoy, S., Malah, D., Karnin, E., *A new improved collage theorem with applications to multiresolution fractal image coding,* Proc. ICASSP, 1994.

[34] Forte, B., Vrscay, E. R., *Solving the inverse problem for function/image approximations using iterated function systems, II. Algorithm and computations,* Fractals 2,3 (1994) 335—346.

[35] Dudbridge, F., Fisher, Y., *Attractor Optimisation in Fractal Image Encoding*

[36] Armano, G., Giusto, D. D., Vernazza, G., *Recent Results in Adaptive Image Coding* in Proceedings of the 5lh International Conference on Image Analysis and Processing, Positano Italy, 1989

[37] Hamzaoui, R., Müller, M., Saupe, D., *VQ-enhancedfractal image compression,* ICIP-96 IEEE International Conference on Image Processing, Lausanne, Sept. 1996.

[38] Hamzaoui, R., Müller, M., Saupe, D., *Enhancing fractal image compression with vector quantization,* 1996 IEEE Digital Signal Processing Workshop, Loen, Sept. 1996.

[39] Thao, N. T., *A hybrid fractal-DCT coding scheme for image compression,* in Proc. ICIP-96 IEEE International Conference on Image Processing, Lausanne, Sept. 1996.

[40] Oien, G. E., Lepsoy, S., Ramstad, T. A., *An Inner Product Space Approach to Image Coding By Contractive Transformations,* Proc. 1CASSP-91, 1991, 2773-2776

[41] Mazur, B., *Moving from Assembly to* C, Dr. Dobbs Journal, August 1992, pg 72-84

[42] Baglietto, P., Maresca, M., Migliardi, M., Zingirian, N., *Image Processing on High-Performance RISC Systems,* Proc. IEEE, Vol 84 No 7, July 1996. Pg 917 - 930

[43] Duin, P. W., Komen, E. R., *Massively Parallel Architectures For Cellular Logic Image Processing*

[44] Texas Instruments TMS320C80 Online reference CD, TI 1995 (Item SPRCOOIB)

[45] Alexander, W. E., Reeves, D. S., Gloster Jr., C. S., *Parallel Image Processing with the Block Data Parallel Architecture,* Proc. IEEE, Vol 84 No 7, July 1996, pg 947 - 968

[46] Bonomini, F., De Marco-Zompit, F., Mian, G., Odorico, A., Palumbo, D., *An MVP Based Video Decoder for MPEG 2,* Proc. First European DSP Education and Research conference, Sept 25-26, 1996, pg 118-124

[47] Akhan, M. B., Bayik, T., Bahari, E. G., *Faster Scan Conversion Using TMS320C80,* Proc. First European DSP Education and Research conference, Sept 25-26, 1996, pg 110 - 113

[48] Mooshofer, H., Hutter, A., Stechele, W., *Parallelization of a H.263 encoder for the MVP,* Proc. First European DSP Education and Research conference, Sept 25-26, 1996, pg 162-168

[49] Baharav, Z., Malah, D., Karnin, E., *Hierarchical interpretation of fractal image coding and its applications to fast decoding,* Intl. Conf. on Digital Signal Processing, Cyprus, 1993.

[50] Dedera, L., Chmurny, J., *A new fast decoding algorithm for fractal image block coding scheme without spatial contraction,* Journal of Electrical Engineering Vol. 48, No. 11-12(1997), pp. 287-291.

[51] Tremeac, Y. G., Inggs, M. R., *An Example of Rapid Prototyping on the TMS320C80 Multimedia Video Processor (MVP),* Proc. Comsig-98, September 1998 pg 233-236

[52] Cloete, E., Venter, L. M., *Fractal Image Compression* in: SART/SACJ, No. 21, 1998

[53] Rudomin, I., *Genetic Algorithms for Fractal Image and Image Sequence Compression,* http://journey.cem.itesm.iTix/zpaper/zpaper.html

[54] Wakerly, J. F., *Digital Design, Principles and Practices,* Prentice Hall, 1994

[55] Uys, R. F. E., Prentice, J., *Fractal Image Compression Using the TMS320C80 Multimedia Video Processor,* Electron Journal, July 1997 pg 12-14

[56] Huertgen, B., Mols, P., Simon, S. F., *Fractal transform coding of color images,* Proceedings of the International Conference on Visual Communications and Image Processing, SPIE '94, Vol. 2308, pp. 1683-1691, Chicago, Illinois, USA, 1994.

[57] Yan, H., Filippoff, G., *Color image compression based on fractal geometry,* in Proc. 2nd Singapore International Conference on Image Processing, '92, pp 3-5, 1992

[58] http://www.webcom.com/~verrando/university/ifs.html

[59] Barthel, K. U., Voye, T., *Three-Dimensional fractal video coding,* IEEE Int. Conf. on Image Processing (ICIP'95), Washington, D.C., USA

[60] Le Gall, D., *MPEG : A video compression standard for multimedia applications,* Communications of the ACM, Vol 34, No 4 (April 1991) pp 46-58

[61] Aviref
McGowan, J., *An Overview of A VI,* http://www.rahul.net/jfm/avi.htmltfDefinition

[62] Ali, M., Papadopoulos, C, Clarkson, T. G.; *The use of fractal theory in a video compression system,* in Proc. IEEE Data Compression Conference (DCC'92), 24-27 March, 1992.

[63] Nicholls, J. A., Monro. D. M, *Scalable Video by Software,* Proc ICASSP 1996

APPENDIX A. : TEST IMAGES USED IN EXPERIMENTATION

Figure A.I : "LENA" (512x512 pixels x 8 bit - file size 256kb)
Source : www.dip.ee.uct.ac.za

Figure A.2 : "DRIVER" (512x512 pixels x 8 bit - file size 256kb)
Source : Photo and scan : R. Uys

Figure A.3 : "GIRL" (512 x 512 pixels x 8 bit - file size 256kb)
Source : www.dip.ee.uct.ac.za

Figure A.4 : "CAR" (512x512 pixels x 8 bit - file size 256kb)
Source : www.ford.com (International Ford motor corporation website)

Figure A.5 : "POSTGRAD" (512x512 pixels x 8 bit - file size 256kb)
(Image enlarged with block strip to generate a square image for test purposes)
Source : Photo : A. Stylo, Scan : R. Uys

APPENDIX B. : ADDITIONAL INFORMATION

B.1. Fast Decoding

Traditional fractal decoding is an iterative process, and the number of iterates required for adequate results varies with image content, and can be as high as 10. Intuitively this is a sub-optimal approach. There are however several alternative, faster decoding algorithms [9]. Most offer these speed gains independently of the compression process, and do not require any modifications to the compression process. They cannot offer any improvement in the fidelity (either quantitatively or qualitatively) of the decompressed attractor - they simply provide a means of extracting it quicker.

[49] presents a hierarchical interpretation of fractal image compression which relates the compressed mappings to a continuous function from which different scales of the fixed point can be derived. A first iteration of the decoding process is performed conventionally, and then the deterministic algorithm is used to extend this to higher resolution. Decoding times are reported to be an order of magnitude faster. Another well-known scheme [4 appendix C, 9] called pixel chasing is based on the observation that due to the averaging of adjacent pixels that occur in the contraction stage of affine mappings, each "child" pixel in a domain block is related to a unique "parent" pixel in a range block. This in turn is the child of another parent from another mapping. Thus chains of relationships arc constructed and decoding occurs by tracing the chain back to a known pixel value. [4 ch 11] has suggested that it may be to directly determine the fixed point of a PIFS by the inversion of a matrix constructed from the affine coefficients. The affine coefficients and block descriptions used here are discretised parameters derived from a continuous function treatment of the problem.

The second group of improved decompression algorithms rely on symmetric modifications to the compressing algorithm. These may offer both speed and fidelity gains. [50] proposed a new type of fractal compression based on *non-contractive* mappings. Domain and range blocks are of the same size. The choice 1 for all spatial contractivity factors may seem dangerous in the light of the contractive mapping theorem (Section 3.2, Equation 3.5). Decoding is necessarily performed using a pixel chasing type scheme.

B.2. TMS320C80

B.2.1. Texas Instruments DSP Families

At the inception of the project (1997) the TMS320C8x generation was the flagship in the TMS320 family of digital signal processors (DSPs) from Texas Instruments. The TMS320 family has grown from a single device introduced in 1982, the TMS32010, to over thirty different products across six CPU architectures. On-chip hardware multipliers, register files, barrel shifters, ALUs, ROM, RAM, caches, and I/O peripherals, along with massive internal busing (all within a product as programmable as a general-purpose microprocessor), make TI TMS320 devices ideal for a broad range of computation-intensive applications.

- The 'C1x, 'C2x, and 'C5x generations offer a complete line of general-purpose and application-specific fixed-point DSPs.
- The 'C3x and 'C4x generations provide an ensemble of floating-point DSPs.

- The 'C8x generation, of which the TMS320C80 multimedia video processor (MVP) is the first processor, rounds out the TMS320 family and offers video-rate multiprocessing capabilities.

B.2.2. TMS320C80 Parallel Processing System

The TMS320C80 processor itself provides a complete parallel processing system geared towards image processing applications on a single integrated circuit, and includes (See Figure 4.2):

- A 32 bit wide RISC type master processor (MP) with IEEE-754 floating-point unit.
- Four 32 bit wide integer DSP processors (PPO to PP3) running as slaves under control of the master processor
- Each of these 5 processors has 10k bytes of on-chip SRAM associated with it for local data and instruction storage.
- A sophisticated DMA (direct memory access) controller with interfaces to external DRAM, SRAM, and VRAM memory. This transfer controller also arbitrates the transfer of data among the 5 processors and between the processors and off chip RAM.
- These data transfers take place over a network of crossbars which link the processors to allow efficient data sharing.
- All instruction and data paths within the chip are 32 bits wide.
- Video timing control.

Each PP's wide instruction word, three-operand ALU (arithmetic logic unit), and single-cycle multiplier enable it to perform a number of RISC-equivalent operations in a single clock cycle. 10k of the 50k bytes of SRAM are associated with each of the five processors, and communication between the tightly coupled master processor, parallel processors and on-chip SRAM takes place over a high-speed crossbar network that provides fast and simultaneous access to multiple banks of the RAM, allowing efficient data sharing. See Figure 4.2 for illustration of this arrangement. The crossbars allow 5 instruction fetches (1 per processor) and 10 parallel data accesses per cycle. The device is clocked at 40MHz, allowing execution speeds of up to 2 billion RISC instructions per second to be attained. The transfer controller (TC) is a DMA controller that manages all memory traffic. The TC performs packet transfers that move data between on- and off-chip memory. Some of these packet transfers are complex programmable byte-aligned array transfers. The TC also performs instruction- and data-cache servicing for each of the five processors.

The processors on the MVP can be configured for a variety of multiple-instruction, multiple-data (MIMD) operations and are connected by a crossbar network to on-chip SRAMs and to the external memory via the transfer controller. This allows the shared memory to be used efficiently and helps to eliminate processing delays resulting from contention.

B.2.3. The Master Processor

The MP is a 32-bit RISC (reduced instruction set computer) processor with an integral IEEE-754 floating-point unit. The MP's primary role is to perform the general-purpose computations necessary to direct the MVP's on-chip processing resources. The MP must be used for all required floating-point calculations, as it is the only processor on the chip so capable. The MP architecture is designed for efficient execution of C code. As with other RISC processors, all accesses to memory are performed with load and store instructions, and most integer and logical operations are performed on registers in a single cycle.

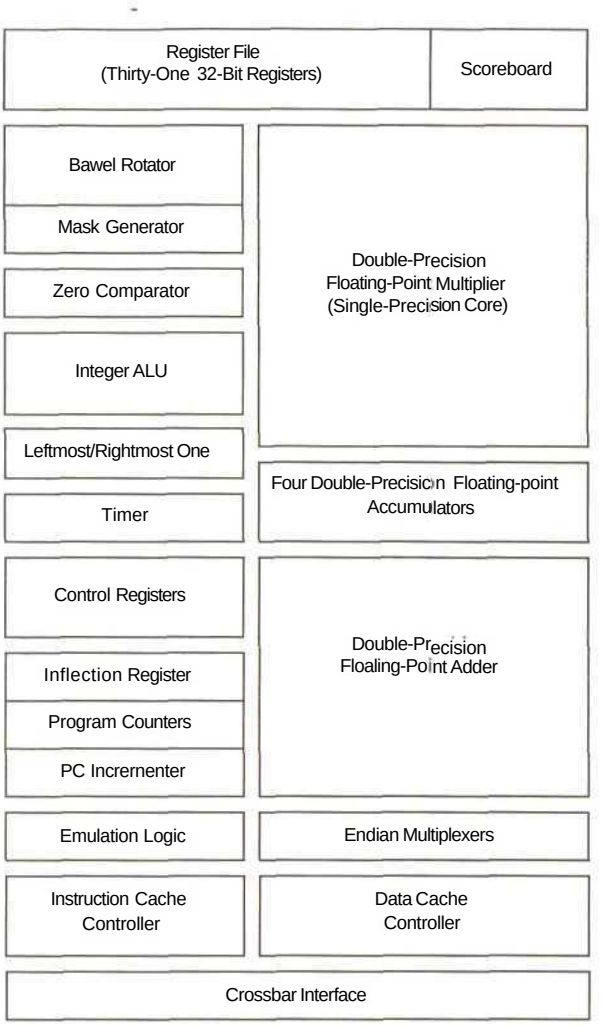


Figure B.I : Block diagram for the master processor.

B.2.3.1. Master Processor Pipelines and Parallelism

To improve performance, the MP uses several heirarchical levels of pipelining. At the hardware level, for example, floating-point instructions are pipelined so a single-precision multiply or any floating-point add instruction can be started on each clock cycle. Moreover, with a special set of parallel multiplication, addition, load, and store instructions, the floating point unit is capable of up to 100 MFLOPS in performance at 50 MHz internal clock rate

On the programmer's level, several instructions can be in successive stages of completion in a given clock cycle as they flow through a pipeline. This is facilitated by the MP having several execution units which can run in parallel. (See Figure B.I)

The pipelining is taken care of by the C compiler, but there is a limit to what the compiler can achieve, so code must be written with this in mind. For optimised performance in this project, time-critical loops were written in assembly language to ensure low level control over this pipelining.

B.2.3.2. Cache Management

The MP's dual-cache architecture allows it to access instruction and data words in parallel during the same clock cycle. When a cache-miss occurs, the transfer controller (TC) automatically updates the cache from external memory. The MP's data and instruction caches each use 4K bytes of on-chip static RAM for storage. They enable the MP to run at speeds approaching those that would be obtained if the MVP's entire external memory ran at the speed of on-chip RAM. If MP code is written to exploit the parallelism that may be achieved with the multiple execution units, it is the responsibility of the programmer to maintain data-cache coherency when several processors may be acting on the same data. (This is the responsibility of the C compiler for code written in C).

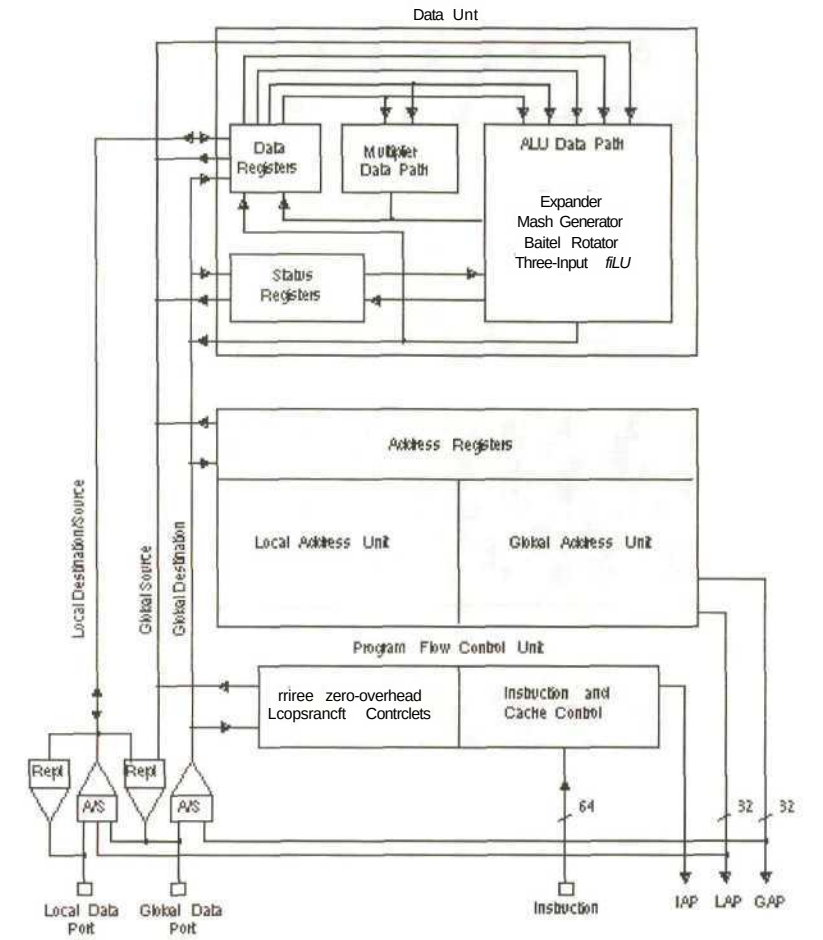
B.2.4. The Parallel Processors

The MVP contains four of these identical processors.

The 4 MVP parallel processors (PP) are each a programmable DSP-like 32-bit integer processor with a 64-bit instruction word that is optimized for imaging and graphics applications. It supports the filtering and frequency domain operations required for image processing. The PP can execute in parallel a multiply, an ALU operation (such as a shift-and-add), and two memory accesses, within one single-cycle instruction. As with the pipe-lining of the MP code, it is up to the programmer to exploit these capabilities. The internal parallelism allows a single PP to achieve over 500 million operations per second for certain algorithms.

The four parallel processors (PPs) provide the MVP's computational power. In order to specify the multiple parallel operations that the PPs can perform, a wide instruction word of 64 bits is used. The instruction has fields that independently control the data unit (with its multiplier and ALU data paths) and the two address units. For this project, the PP's serve as high-speed pixel coprocessors for the RISC master processor.

Figure B.2 shows a block diagram of the PP's four major functional units: the data unit, local address unit, global address unit, and program flow control unit. Each of these units is capable of several RISC-equivalent operations in a single cycle, as specified in the 64-bit instruction word.



Note: IAP =Instnjction address port LAP =LocaJ address port GAP =G)obsJ address port
Repl =Replicate hardware *AS* =AlignAign extend hardware

Figure B.2 : The PP Block Diagram

1) Lds stands for local destination/source bus
2) Gsrc stands for global source bus
3) Gdst stands for global destination bus
4) Repl stands for replicate hardware
5) A/S stands for align/sign-extend hardware

The key areas of the PP can be listed as follows:

- PP Data unit - supports the massive processing associated with algorithms such as frequency domain transforms (for example, DCTs), correlation, and filters. It also supports the bit-field and pixel manipulations

- Global and Local Address units - compute addresses for memory accesses and also control all data movement among the various PP functional units. The two units allow 2 parallel memory accesses per cycle to load data for the data unit to act on. Bus Structure Overview

- Internal buses - transfer data between PP registers and memory, transfer data between the PP functional units, provide instruction addresses, and receive instructions. Each PP has three ports to memory: the instruction port, global port, and local port.

- Program Flow Control Unit - controls the PP instruction pipeline, performs instruction fetching and decoding, performs any necessary handshaking with the transfer controller, and handles interrupt response and prioritization.
- Cache controller - Performs instruction cache management and issues requests to the TC for instructions not in the cache

- Instruction controller - takes the instructions supplied by the cache controller via the cache to generate the control signals that drive the PP. Instructions are processed by a pipeline that consists of three stages: instruction fetch; address unit computations; and data unit, memory access, and/or register-to-register move execution. At any given time, three instructions are in the pipeline (one at each stage) so that the net throughput is one cycle per instruction (assuming there are no stall conditions).

- Loop controllers - The program flow control unit contains three loop controllers that support up to three simultaneous hardware-controlled loops. This allows loop implementation with no execution time overhead.

The PP uses a 64-bit instruction opcode that can specify a number of operations to be performed in parallel by the data unit and address units. The address unit operations are not bound to specific data unit operations. This flexibility greatly enhances the utility of the address units. When programming the PP, it is useful to think of an instruction as consisting of multiple subinstructions (referred to as operations) for the multiplier, ALU data path, global address unit, and local address unit.

B.2.5. The Transfer Controller

The transfer controller (TC) is the sixth programmable processor in the MVP. It is a combined DMA (direct memory access) machine and memory interface that intelligently queues, prioritizes, and services the data requests and cache misses of the five programmable processors on the MVP (the MP and the four PPs). Through the TC, all of the processors can access off-chip memory. This interface supports DRAMs, video RAMs, SRAMs, and ROMs. The support for DRAMs, including timing control and address multiplexing, is relatively new in DSPs. In addition, data-cache or instruction-cache misses are automatically handled by the TC. The TC also supports a host-interface mechanism that allows a host processor to gain access to the MVP system. This facility is used for example when the compression process starts, to allow the host computer to write the image data onto the MVP memory. The functionality is shown in Figure B.3.

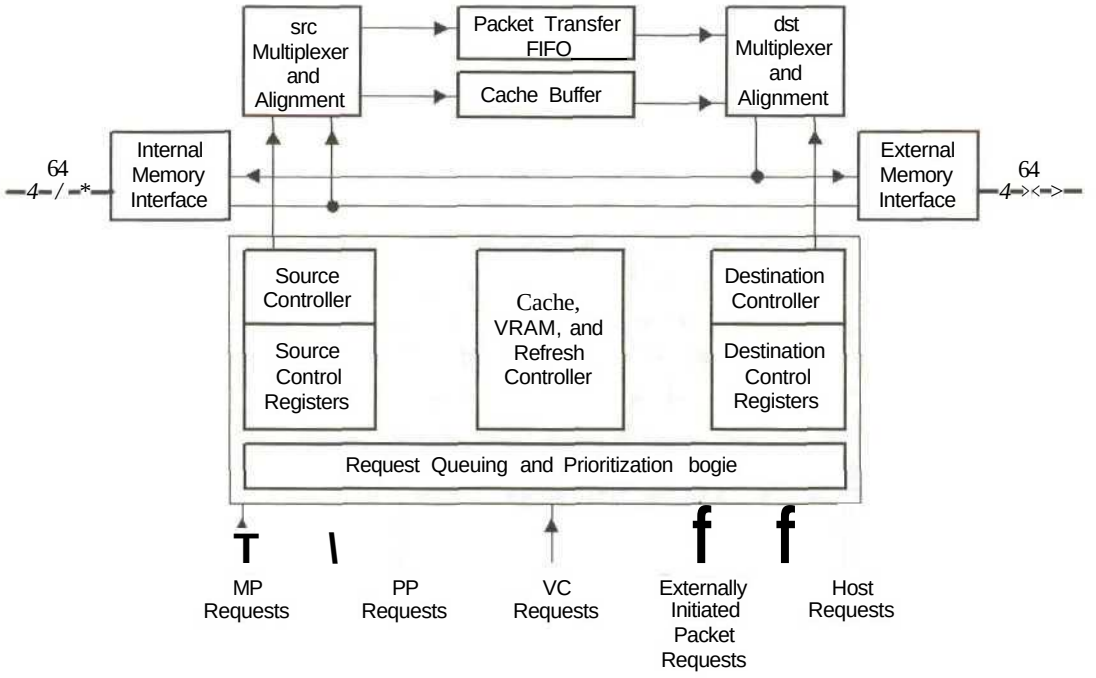


Figure B.3 : Transfer Controller Block Diagram

The transfer controller is the central component of the implementation of so called packet transfers (PT). This represents the most sophisticated form of content orientated data transfers between on-chip and off-chip memory. These data transfers are specifically requested by the PPs or the MP in the form of linked-list of parameters, which are interpreted by the TC. These requests allow multidimensional blocks of information to be transferred between a source and destination, either of which can be on-chip or off-chip. The TC can transfer data between a source and a destination that have different logical dimensions. In graphics and imaging, for example, it is common for the TC to fetch data from an image region as a two-dimensional X/Y array and bring the data on-chip for processing as a linear array. After processing, the results stored in a linear array can then be stored off-chip as an X/Y array. The ability of the TC to make these transformations autonomously greatly improves the efficiency of processing by the PPs or MP.

Packet transfers may be initiated by the MP, PPs, VC, or external devices as requests to the TC. The TC services the requests using the fixed and round-robin prioritizations. Once a processor has submitted a request for a transfer, it can continue program execution. The packet transfer is completed by the TC without the need for additional cycles by the requesting processor. Packet transfers formed a vital part of the code developed for this project.

B.2.6. C Compiler Description

Since the MP and PP's have different instruction sets, different compilers must be used. Each source code module (there may be several separately defined functions called from one processor's code) must be separately compiled. Command line options control the operation of both the shell and the programs it runs. At this stage it is possible to specify options pertaining to type checking on pointers and other syntactical options. The compiler package includes an optimization program which may be invoked at this stage to improve the execution speed and reduce the size of C programs by simplifying loops, rearranging statements and expressions, and allocating variables into registers. The optimizer runs as a

separate pass between the parser and the code generator. Totally then the compilers are of the three pass type. (Parser, Optimiser, Code generator)

There are four levels of optimization: 0, 1, 2, and 3. [51] These levels control the type and degree of optimization, and essentially describe predefined scenarios of compilation options. As with most aspects of this processor, the optimiser can be confused, and care must be taken when writing code, bearing in mind the peculiarities of the optimisation process, to avoid ambiguities which lead to syntactically but logically incorrect optimisations.

Aliasing occurs when a single object may be accessed in more than one way, such as when two pointers point to the same object or when a pointer points to a named object. Aliasing can disrupt optimization because any indirect reference could potentially refer to any other object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program. The compiler assumes that if the address of a local variable is passed to a function, the function might change the local variable by writing through the pointer, but that it will not make its address available for use elsewhere after returning. For example, the called function cannot assign the local's address to a global variable or return it.

Some assistance may be had by the programmer from the interlist utility. This utility interlists C source statements into the assembly language output of the compiler. This enables one to inspect the assembly code generated for each C statement. The interlist utility runs as a separate pass between the code generator and the assembler. There is as always a penalty for convenience - this extra pass prevents the code generator from scheduling code across a C statement boundary. This will limit parallelism in the PP and also will prevent optimal delay slot filling in both the MP and the PP. This feature proved useful during code development.

B.2.7. Runtime-Support Functions

Some of the tasks that a C program may need to perform (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C language itself. However, the ANSI C standard defines a set of runtime-support functions that perform these tasks. The TMS320C8x C compilers include a set of libraries that implement the complete ANSI standard library. In addition to the ANSI-specified functions, the TMS320C8x runtime-support libraries include routines that allow a user to have direct access to specific MVP master processor commands, and direct C language I/O requests to a user-specified device.

The runtime-support libraries can be edited by the user. (This source code must reverse-compiled, edited, and be "built" into libraries using a build utility provided in the development package.) This provides a unique form of low-level control not found in conventional C development environments. The runtime-support libraries contain macro definitions type definitions, for example the parameters governing floating point precision and format can be accessed and altered for special applications. The memory maps of the transfer controller and other processor control registers are also defined here, as are the formats for packet transfers.
This feature was used extensively in the project to customise the behavior of packet transfers.

B.2.8. Common Object File Format

The assemblers and linker create object files that can be executed by a TMS320C8x device. The format that these object files are in is called common object file format, or COFF. The COFF format is a modular structure. Both the assembler and the linker provide directives that

allow the modules, or sections to be created and manipulated. This format is used because it provides a means of structuring the code and managing the sections to reflect the memory map definitions set up during the linking stage. As such, a holistic approach to code development must be adopted where the code structure is sympathetic to the hardware architecture, and the logical appearance of hardware structures and boundaries as defined by the user at link time.

The elements of a COFF object file describe the file's sections and symbolic debugging information. These elements include:

- A file header
- Optional header information
- A table of section headers
- Raw data for each initialized section
- Relocation information for each initialized section (reflecting the hardware enforced memory map, and the restrictions defined by the user at link time)
- Line number entries for each initialized section
- A symbol table
- A string table

APPENDIX C. : ADDITIONAL INFORMATION ON IMAGE FORMATS

There are numerous image formats used for image storage on computers. They tend to vary depending on which computer platform is used. Again no attempt is made at an exhaustive list, only those formats in common use for continuous tone images are described. Generally these are raster type formats. It is worth noting that there are several other specialised categories of image formats including :

- Vector type formats for storage of line drawings in such applications as contour plotting and CAD
- Printer post-script types for sending images to printers. These are usually specific to a particular printer manufacturer and are not "seen" by the user.
- Within certain specialist sectors there are application specific image formats designed to meet particular needs. An example of this is DICOM which is used in the USA for radiography and microbiology.

C.1. Windows Bitmap (BMP) - PC FORMAT

The BMP bitmap format originally came into use with Windows and has gained much support from software providers. The BMP format comes in a compressed form using RLb (run length encoding) and an uncompressed form. Advantages of this format include that it is well supported by many Windows applications and can handle 24-bit colour images. In uncompressed form however, BMP files can be large in size.

C.2. Computer Graphics Metafile (CGM) - PC FORMAT

The CGM standard created by ANSI is a widely used international graphics format. It was developed to provide a basis for the storage and interchange of graphics. CGM files are generally well compressed and are used on a range of platforms.

C.3. Device Independent Bitmap (DIB) - PC FORMAT

DIB is a format that has been made popular by Windows and is similar to the BMP format. They can be stored in either a compressed or uncompressed format. DIB format files are really only supported by the Windows operating system.

C.4. DXF format (DXF) - PC & MAC FORMAT

The DXF format was developed to allow the interchange of files between AutoCAD applications. AutoCAD is perhaps the most widely used CAD (Computer Aided Design) package and the DXF format has become the standard format used with virtually all CAD packages supporting it.

C.5. Graphic Interchange Format (GIF)- PC & MAC FORMAT

The GIF format was developed by CompuServe as a method of sending compressed files over a network. It is however, rapidly gaining acceptance as a file format standard with many software packages now supporting it. GIF utilises LZW (Lempel-Ziv-Welch) bitstream compression as used in many file compression utilities such as PKZIP. The compression ratio achieved using GIF files is usually better other 8 bit formats. The GIF standard supports two different forms GIF87a and GIF89a.

Advantages of this format are that it has a good compression ratio and is widely supported across a variety of platforms. Disadvantages include that it can be slower to load than most than bitmap formats and it does not support true colour 24-bit images.

C.6. Joint Photographic Expert Group (JPEG) - PC FORMAT

The JPEG format was designed by the Joint Photographic Expert Group, its aim being to achieve maximum image compression rates through using 'lossy' compression techniques. This involves the permanent loss of information so that once an image has been compressed and then decompressed, it will not be identical to the original image. Usually though, this is not discernible by the human eye. Advantages are that whilst most other methods of image compression achieve up to 3x compression rates but the JPEG technique can achieve 20x or more. Disadvantages are that the flexibility of this format can lead to compatibility problems. As the image is compressed when saved, further manipulation of JPEG images can result in deterioration which may not be acceptable in certain circumstances.

The operation of JPEG may be summarised into the following steps :

a) Transform the image into a suitable colour space

This is not relevant for grey scale, but for colour raw images in Red-Green-Blue (RGB) space are transformed to Luminance-Chrominance (YcbCr or YUV) space. The luminance component is then treated the same as a grey scale image. The other two dimensions are the colour information.

b) Down-sampling of chrominance components

The human visual system is far less sensitive to high frequency chrominance information than high frequency luminance information. Thus the luminance component is retained at maximum resolution while the chrominance information is decimated by a factor of 2 in each spatial dimension by averaging adjacent pixels. This immediately gives a 75% reduction in two of the three components.

c) DCT coding

For each of the components, the pixels are divided into 8 x 8 pixel blocks and a two-dimensional discrete cosine transformation (2D-DCT) is performed over the block. This yields 64 DCT coefficients. This first one represents the DC value of the block (average brightness) and the others the AC components.

d) Coefficient Quantization

Most of the image energy is concentrated in the lower frequencies. The upper frequencies contain progressively less information. Thus they may be quantized at progressively lower

resolution without significant impact on the overall information content. This is the lossy stage of JPEG, and is the major influence on the compression ratio.

e) DC coding
There is a strong correlation between the DC coefficients of adjacent blocks. Thus the values may be efficiently stored by storing the difference between them, rather than the actual intensity values.

f) Entropy coding
The reduced coefficients are encoded using Huffman coding. This is a lossless step.

g) Attach header information
Applicable formatting information is appended.

C.7. PCX format (PCX) - PC FORMAT

The PCX format was originally created by Zsoft for use with their software packages. Like the BMP format it utilises RLE (Run length encoding) for the compression of images although the ratios achieved are relatively low. Run length coding is a semantic-dependent method that involves mapping a sequence of pixel values into a sequence of pairs (c_k, l_k) where c^ represents the value and l_k the length of the run or sequence of pixels of that value. Although PCX is one of the oldest formats, it is still commonly used but is now starting to be replaced by newer formats.

Its advantages of simplicity and speed of encoding/decoding tend to be outweighed by the relatively low compression ratios and portability problems between different machines. Since PCX was first created, there have been a number of enhancements to the format to cope with advances in available video graphics technology. The original specification could only handle monochrome images, but later improvements catered for various levels of colour display, with modern formats able to store 256-colour and 24-bit colour images.

PCX files are very good for storing bitmap images in a compact form, when using computers of a similar graphics capability. On the other hand, in situations where images have to be transferred between machines which have different types of display, or especially between different platforms (PC to Macintosh, for example), then other file formats (such as TIFF) may be more appropriate.

C.8. PICS format (PICS) - MAC FORMAT

This is a format for storing sequences of PICT/PICT2 images to be used, for example, in animations. The resulting file sizes can be extremely large.

C.9. PICT/PICT2 - MAC FORMAT

PICT is a format which can be used for storing bitmapped and vector images. PICT2 is an enhancement of PICT and can be used storing 8-bit and 24-bit colour images. Although it is principally a Macintosh format, it can be read by some PC applications.

C.10. Portable Network Graphics (PNG)

PNG is a non-lossy image compression format which stores metadata within the file. The metadata can include descriptions, keywords, classification codes and additional data. This format was developed to get around the fact that some of the other formats (for example GIF) are proprietary.

C.11. Targa (TGA) - PC FORMAT

The Targa format was created to support graphic cards developed by Truevision Inc. It is a popular format used for high resolution images and is useful when transferring 24-bit files. Although this format supports both compressed and uncompressed files usually no compression is used resulting in large file sizes.

C.12. Tagged Image File Format (TIF/TIFF) - PC & MAC FORMAT

Tagged Image File Format (TIFF) is a bitmap (pixel-based) graphics standard which was created jointly by the Aldus and Microsoft corporations chiefly for the Desktop Publishing industry. The reason for developing it was that they felt that the industry needed an image file format that could withstand the incessant evolutionary improvements of computer equipment. In contrast to, say, the PCX standard, TIFF was designed so that it would function on a variety of computer platforms.

It is not a single format and there are several versions as follows:

- TIFF B - black and white images
- TIFF G - greyscale
- TIFF P - storing colour palettes with the image
- TIFF R - 32-bit colour

TIF/TIFF files may be compressed using LZW (Lempel-Ziv-Welch) to achieve high compression ratios. LZW is rather more intricate and certainly more efficient, than the simple Run-Length Encoding compression used in other image formats (such as PCX)

The main disadvantage of the format is the number of different versions used can result in incompatibility problems. One reason why TIFF has become so popular is the fact that it is flexible and extensible. According to the standard, a wide variety of information can be included in a TIFF file, but not all of this need be used. A program can simply utilise what information is required to adequately store or retrieve the images that it works with. In addition, the program can choose to organise the information in a variety of ways, rather than having to stick to a fixed format.

Information in a TIFF file is grouped into small sections. Each section starts with a special code, or tag, that details what information is contained in the section. There is a standard to define what each tag means and how to read that section's data. For example, each TIFF file has

a tag entry that specifies the height of the image and one that specifies its width, and another to specify the colour palette.

Monochrome TIFF - the original format - stores only black and white images, but the black and white pixels can be 'dithered' - that is to say that a variety of dot patterns can be used to simulate different shades of grey fairly accurately. Grey-scale TIFF on the other hand, contains 256 proper shades of grey, giving a better definition for some images. At the other end of the scale, a Colour TIFF image can contain as many as 16.8 million colours.

C.13. Windows Metafile (WMF) - PC FORMAT

This is a Windows file format that is not commonly used elsewhere. It is a vectorformat which allows the image to be rescaled proportionally. It is an extremely useful format for the interchange of image between applications within the Windows environment. This is the structure used by the "draw" functions of MS Word and MS Powerpoint.

C.14. References :

On line :

[I] http://www.cica.indiana.edu/graphics/image.formats.html

[2] http://www.alpeda.shef.ac.uk/fr_1195.htm

APPENDIX D. : SAMPLE CODE

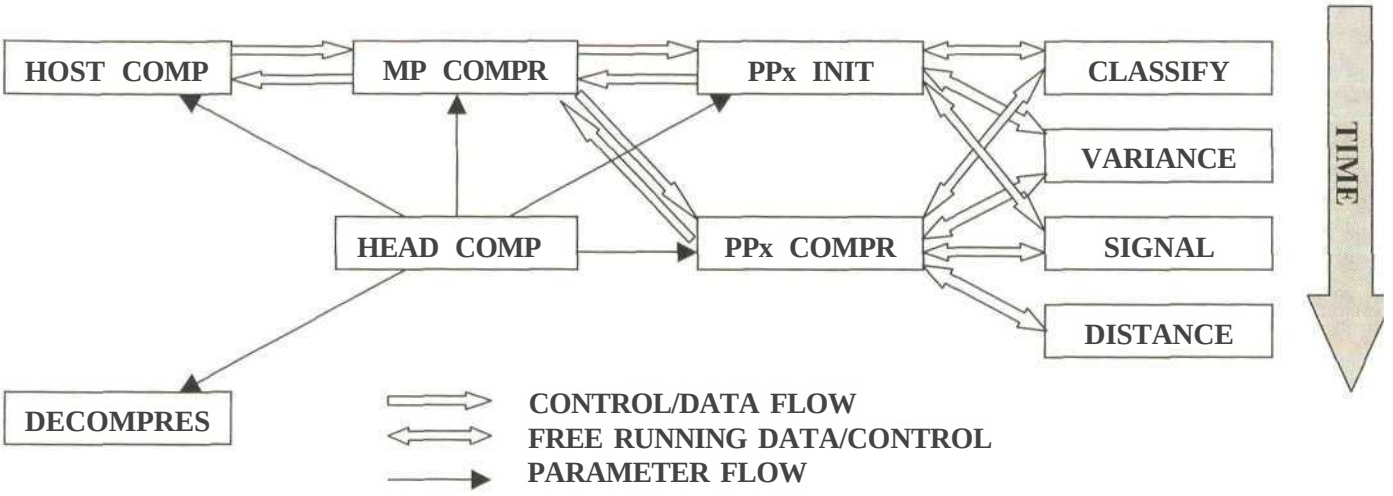


Figure D.I : Interrelationship between different code modules

Figure D.I above illustrates the interdependence between the main code modules, with the time dependence indicated. Definitions for the module names appears below :

a) HEADER FILES

MODULE NAME	*FUNCTION*	*SAMPLE LISTING*
HEADCOMPR.H	List of definitions of constants, parameters and communication protocols	Page D-5

b) MAIN MODULES

MODULE NAME	*FUNCTION*	*SAMPLE LISTING*
HOST_COMPR.CPP	Code running on host processor during compression operation	Page D-6
MPCOMPR.C	Code running on master processor during compression operation	Page D-7
PPxINIT.C	Code running on parallel processor x (for x:-> 0..3) during precalculation phase of compression operation	Page D-19
PPxCOMPR.C	Code running on parallel processor x (for x:-> 0..3) during affine map search phase of compression operation	Page D-22

*** NOTE : sample code is presented only for PPO - ie PPOJNIT.C and PP0JCOMPR.C. The code running on the other PPs is similar*

c) SUB-UNITS RUNNING ON PARALLEL PROCESSORS

MODULE NAME	*FUNCTION*	*SAMPLE LISTING*
CLASSIFY.C	Code performing classification of domain and range blocks	Page D-26
VARIANCE.C	Code computing variance of pixels within a specific block	Page D-28
SIGNAL.S	Routine called to facilitate time synchronisation with master processor	Page D-31
DISTANCE.C	Code computing metric distance between two blocks with respect to some defined metric	Page D-29
PACKET TRANS.	Packet transfer definition assembly code	Page D-32

Note that the packet transfer definition is an extraction from a library built for the purposes of this project. It is presented because it is central to the operation of the code.

In addition to the code, sample listings of the memory map difining files, assembly/link batch files and other such management files. The details are as in the table below. The batch file makes calls to the source code files and the other .CMD files listed in this table:

FILENAME	*FUNCTION*	*SAMPLE LISTING*
MVP_COMPR.BAT	Batch file governing the compilation of the C source files, assembly of these and the assembly modules, linking using the memory map information and creation of the COFF file. Many command line options are stipulated.	Page D-3
MPLNK.CMD	Memory map information for code running on the Master Processor.	Page D-4
PPLNK.CMD	Memory map information for code running on the Parallel Processors.	Page D-4
EXAMPLE.CMD	Overall memory directives and additional options about C code compilation.	Page D-4

D.1. MVP_COMPR.BAT

```
ppcl -g -tp pp0_init.c  ppl_init.c pp2_init.c pp3_init.c signal.s

ppcl -g -tp pp0__compr.c  ppl_compr.c pp2_corapr.c pp3_compr.c

ppcl distance.c

ppcl variance, c

ppcl classify.c

ppcl transform.c

ppcl poll.s

mpcl -g mp_compr

mvplnk -r -h -g$distance distance.o -1 pp_rts.lib -o distance.out -m distance.map

mvplnk -r classify.o -1 pp_rts.lib -o classify.out -m classify.map

mvplnk -r transform.o -1 pp_rts.lib -o transform.out -m transform.map

mvplnk -r variance.o -1 pp_rts.lib -o variance.out -m variance.map

mvplnk -r poll.o -1 pp_rts.lib -o poll.out -m poll.map

mvplnk pp0_init.o -1 pplnk.cmd -t tsk_i0  -o pp0_init.out  -m pp0_init.map

mvplnk ppl_init.o -1 pplnk.cmd -t tsk_il  -o ppl_init.out  -m ppl_init.map

mvplnk pp2_init.o -1 pplnk.cmd -t tsk_i2  -o pp2_init.out  -m pp2_init.map

mvplnk pp3_init.o -1 pplnk.cmd -t tsk_i3  -o pp3_init.out  -m pp3_init.map

mvplnk pp0_compr.o -1 pplnk.cmd -t tsk_c0  -o pp0_compr.out  -m pp0_compr.map

mvplnk ppl_compr.o -1 pplnk.cmd -t tsk_cl  -o ppl_compr.out  -m ppl_compr.map

mvplnk pp2_compr.o -1 pplnk.cmd -t tsk_c2  -o pp2_compr.out  -m pp2_compr.map

mvplnk pp3_cornpr.o -1 pplnk.cmd -t tsk_c3  -o pp3_compr. out  -m pp3_compr .map

mvplnk   mp_compr.obj distance.out variance.out classify.out transform.out poll.out
pp0_init.out ppl_init.out pp2_init.out pp3_init.out pp0_compr.out ppl_compr.out
pp2_compr.out pp3_compr.out signal.o example.cmd -1 mplnk.cmd -o mvp_compr.out -m
mvp_compr.map
```

D.2. MPLNK.CMD

```
-c
-x
-heap  0x200000
-stack 0x5000
-l mp_rts.lib


MEMORY
{
    EXTMEM    :  0=0x80000000   1 = 0x00800000
}

SECTIONS
{
    .text     :  > EXTMEM
    .ptext    :  > EXTMEM
    .bss      :  > EXTMEM
    .const    :  > EXTMEM
    .switch   :  > EXTMEM
    .sysmem   : > EXTMEM
    .stack    :  > EXTMEM
    .cinit    :  > EXTMEM
    .pcinit   :  > EXTMEM
}
```

D.3. PPLNK.CMD

```
-c
-x
-heap  0x200000
-stack 0x5000
-l mp_rts.lib


MEMORY
{
    EXTMEM    :  0=0x80000000   1 = 0x00800000
}

SECTIONS
{
    .text     :  > EXTMEM
    .ptext    :  > EXTMEM
    .bss      :  > EXTMEM
    .const    :  > EXTMEM
    .switch   :  > EXTMEM
    .sysmem   : > EXTMEM
    .stack    :  > EXTMEM
    .cinit    :  > EXTMEM
    .pcinit   :  > EXTMEM
}
```

D.4. EXAMPLE.CMD

```
-u _exit
-l mp_cio.lib
-l mp_task.lib
-l mp_int.lib
-l mp_ptreq.lib
-l SDBDRVS.LIB
-l SRPC.LIB

SECTIONS
{
    .cio      :  > EXTMEM
    .bypass   :  { *(.bypass) . = align(64); } align(64) > EXTMEM
}
```

D.5. HEAD_COMPR.H

```
/*****************************************************************************/
/*                                                                           */
/*      head_compr.h                                                         */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/

#define TABLE_MAX                64
ttdefine IMAGESIDE               512
#def±ne BLOCKSIDE                8
•define BLOCKMAX                 32
ttdefine BLOCKMIN                4
#define RPCSERVER                1 /* 1 = boot from host, 0 = run from debugger */
#define SGAIN                    60
#define INITDIST                 65000
#define DISTANCETOL              10
#define VARIANCETOL              100
#define DVARIANCETOL             100
#define DOMAIN^SKIP              8
#define DOM4_SKIP                8
ttdefine DOM8_SKIP               8
#define D0M16_SKIP               8
#define BLOCKSPERPP              1024
#define DECFACTOR                2
#define ZERO                     0
```

D.6. HOST_COMPR.C

```
/*****************************************************************************/
/*                                                                           */
/*  HOST_COMPR.C                                                             */
/*  runs on host Pc processor to handle data transfers onto s off SDB        */
/*                                                                           */
/*****************************************************************************/
#include <hsdbdrvs.h>    /*  sdbapi.h  */
((include <sdbrpc.h>
#include <stdio.h>
#include "c:\c8xcode\share\share.h"


char *imagefile;        /* ptr to input bitmap */
char *SDB_addr;         /* address of malloc'ed space on SDB */


void main()
{
  /* HANDLE SDB_Open(); */
  FILE *imagefile;

  Rpclnit("c:\c8xcode\mvp_compr.out",1); /* wait for server to boot. */
                                         /* 1 means boot coff file onto board */

  if (!System_Alive()) {
    printf("\nError in server.\n");
    exit(1);

  }

  if ( (imagefile = fopen ("c: Wc8xcode\\lenabw. raw", "r")) == NULL)
    printf("cannot open c:\\c8xcode\\lenabw.raw\n");
    else printf ( "c : Wc8xcode\\lenabw. raw successfully opened\n");


  /* get address of malloc'ed space from SDB */
  System_WriteBlock(0x800104e0, imagefile, 262144, CLIENT_NOSWAP, 0);

  /* SDB_FromFile(HANDLE, imagefile, SDB_addr); */

  RpcQuit();
}
```

D.7. MP_COMPR.C

```
/*****************************************************************************/
/* MP_compr.C                                                                */
/*                                                                           */
/*    MAIN!) - This routine will calculate a table of function results.      */
/*             This routine will calculate the results, by initiating        */
/*             one function per PP. The MP will wait for the results,        */
/*             and will print them in a table.                               */
/*                                                                           */
/*****************************************************************************/
•include <stdio.h>
•include <mvp.h>           /* includes defs for NOCHACHE_* etc */
•include <sdbdrvs.h>
•include <stdlib.h>
•include <sserver.h>
/*•include <math.h>*/
•include "head_compr.h"
/••include <sdbrpc.h>*/
                          /*************************************************/
I* Define PP control macros                                                  */
/*****************************************************************************/
#dafine INTER_PPS          0xOOOFOOOO

#define RESET_PPS          asm("\tcmnd  0x8000000F")
#define START_PPS          asm("\tcmnd  0x3000000F")
#define UNHALT_PPO                  asm("\tcmnd  0x20000001")
#define UNHALT_PP1                  asm("\tcmnd  0x20000002")
#define UNHALT_PP2                  asm("\tcmnd  0x20000004")
#define UNHALT_PP3                  asm("\tcmnd  0x20000008")
•define PPSTART_VEC(pp)    Mint +*) (OxOlOOOlbO I ( (pp) << 12))


/* Allocate space for the communication buffers in their own section. This  */
/* will help avoid cache errors.                                             */
/*****************************************************************************/
•pragma DATA_SECTION(xnputimage,  ".bypass");
•pragma DATA_SECTION(decimated,   ".bypass");

•pragma DATA_SECTION(dom41ists,        ".bypass");
•pragma DATA_SECTION(dom4classes, ".bypass");
•pragma DATA_SECTION(dom4rots,    ".bypass");
•pragma DATA_SECTION(dom4flips,   ".bypass");
•pragma DATA_SECTION(dom4tots,    ".bypass");
•pragma DATA_SECTION(domdom4tots, ".bypass");
•pragma DATA_SECTI0N(dom4number,  ".bypass");

•pragma DATA_SECTION(dom81ists,        ".bypass");
•pragma DATA_SECTION(dom8classes, ".bypass");
•pragma DATA_SECTION(dom8rots,    ".bypass");
•pragma DATA_SECTION(dom8flips,   ".bypass");
•pragma DATA_SECTION(dom8tots,    ".bypass");
•pragma DATA_SECTION(domdom8tots, ".bypass");
•pragma DATA_SECTION(dom8number,  ".bypass");

•pragma DATA_SECTION(doml61ists,  ".bypass");
•pragma DATA_SECTION (domUclasses,".bypass");
•pragma DATA_SECTION(doml6rots,   ".bypass");
•pragma DATA_SECTION(doml6flips,  ".bypass");
•pragma DATA_SECTION(doml6tots,   ".bypass");
•pragma DATA_SECTION(domdoml6tots,".bypass");
•pragma DATA_SECTION(doml6number, ".bypass");

•pragma DATA_SECTION(classO,           ".bypass");
•pragma DATA_SECTION(classl,      ".bypass");
•pragma DATA_SECTION(class2,      ".bypass");
•pragma DATA_SECTION(class3,      ".bypass");
•pragma DATA_SECTION(returnO,          ".bypass");
•pragma DATA_SECTION(returnl,     ".bypass");
•pragma DATA_SECTION(return2,     ".bypass");
•pragma DATA_SECTION(return3,     ".bypass");
•pragma DATA_SECTION(comm_ppO,    ".bypass");
•pragma DATA_SECTION(comm_ppl,    ".bypass");
•pragma DATA_SECTION(comm_pp2,    ".bypass");
#pragma DATA_SECTION(comm_pp3.    ".bypass");
```

```
#pragma DATA_SECTION(PP0_BLOCKS,  ".bypass");
#pragma DATA_SECTION(PP1_BLOCKS,  ".bypass");
#pragma DATA_SECTION(PP2_BL0CKS,  ".bypass");
#pragma DATA_SECTION(PP3_BL0CKS,  ".bypass");
#pragma DATA_SECTION(PPOJDONE,    ".bypass");
#pragma DATA_SECTION(PP1_DONE,    ".bypass");
#pragma DATA_SECTION(PP2_D0NE,    ".bypass");
#pragma DATA_SECTION(PP3_D0NE,    ".bypass");
/*#pragma DATA_SECTION(classmap,    ".bypass");*/

/*************************************************************************/
/* Define the communication variables. They are defined shared so that   */
/* both MP and PP code can reference them.                               */
/*************************************************************************/

/*shared classification      classmap[4][4][4];*/
shared int                              PP0_DONE,  PP1_DONE,  PP2_DONE,  PP3_DONE;

shared unsigned char   *addrtemp;
unsigned char                    **helper;

shared unsigned char  commppO[64] , comm_ppl[64] , comm_pp2[64], comm_pp3[64];
              /* direct mem - mem transfer communication variables */

shared int                              PP0_BLOCKS,  PP1_BLOCKS,  PP2_BLOCKS,

PP3_BLOCKS;

shared unsigned char  **classO, **classl, **class2, **class3;

unsigned char                    * spacer;

shared unsigned char  *returnO, *returnl, *return2, *return3;
shared unsigned char   *inputimage;   /* points to unprocessed data */
shared unsigned char          *decimated;    /* points to a predecimated version */
shared unsigned char          **dom41ists,  **dom81ists,  **doml61ists;

shared unsigned char          *dom4classes,  *dom8classes,  *doml6classes;
                      /* points to list of domain classifications */

shared unsigned char          *dom4rots,  *dom8rots,  *doml6rots;
                      /* points to list of flips needed to force into class */

shared unsigned char          *dom4flips,  *dom8flips,  *doml6flips;
                      /* points to list of rotations needed to force into class */

shared short                     *dom4tots,  *dom8tots,  *doml6tots;
                      /* points to list of summation of block pixel values */

shared long                           *domdom4tots,  *domdom8tots,
*domdoral6tots;
                      /* points to list of summation of block pixel values^2 */

shared int                            dom4number,  dom8number,  doml6number;

unsigned char                    *fracmap;      /* the compiled fractal mapping */


/* Entry points to the functions which will be run on the PPs.           */
/*************************************+***********************************/
extern int ep_tsk_iO, ep_tsk_il, ep_tsk_i2, ep_tsk_i3;
extern int ep_tsk_cO, ep_tsk_cl, ep_tsk_c2, ep_tsk_c3;

/**************+******************+*******+*******************************/
/* MAIN() - Driver routine for this example.                             */
/*************************************************************************/
main()
{
      int                    k, n, i;
   unsigned char      test;
   int                      pixels,  blocks;
   int                      index,  position;
   long                            temp,  positionl,  position2;
   int                      mapindex;
   int                      bytes;
  • /*int                             blocknumber;
```

```
   short                              domtot;
   long                          domdomtot;*/

   if (IRPCSERVER)
       printf("\nstarted - setting pp entry points\n");

       /*************************************************************************/
   /* Set PPs entry point vectors to function they will perform.           */
   /it*************************************************************************/
   /*PPSTART_VEC{0) = &ep_tsk_cO;
   PPSTART_VEC(1) = &ep_tsk_cl;
   PPSTART_VEC(2) = &ep_tsk_c2;
   PPSTART_VEC(3) = &ep_tsk_c3;*/


   if (RPCSERVER == 1)(
       NOCACHE_USHORT(*(volatile USHORT*)OxE0000180)  = 0x0000; /* ENABLE0 */
       NOCACHE_USHORT(*(volatile USHORT*)OxE0000182)  = 0x0000; /* ENABLE1 */
       NOCACHE_USHORT(*(volatile USHORT*)0xE00O0184)  = 0x0000; /* ENABLE2 */
      NOCACHE^USHORT(*(volatile USHORT*)OxE0000186) = 0x0000; /* ENABLE3 */  /* *****
V
       NOCACHE_USHORT{*(volatile USHORT*)0xE0OO0188)  = 0x0000; /* CLFLAG0 */
       NOCACHE_USHORT(*(volatile USHORT*)0xE000018C)  = 0x0000; /* CLFLAG1 */
   }

   if (RPCSERVER == 1) Rpclnit(15);

   /* Allocate memory */
   pixels = IMAGESIDE*IMAGESIDE;
   if ((inputimage = (unsigned char*) malloc(pixels)) == NULL)  {
       helper =  (unsigned char**)0x807FFFFC;
       *helper =  (unsigned char*)l;
       if (IRPCSERVER) printf("\nmalloc failed on inputimage\n");
      exit(l);   /* terminate if out memory */
   )
       pixels = ( (IMAGESIDE/2)*(IMAGESIDE/2) ) ;
   if ((decimated = (unsigned char*) malloc(pixels)) == NULL)  {
       helper =  (unsigned char**)0x807FFFFC;
       *helper =  (unsigned char*)2;
      if (IRPCSERVER) printf("\nmalloc failed on decimatedXn");
      exit(l);
   }
   pixels = ((IMAGESIDE/BLOCKSIDE)*(IMAGESIDE/BLOCKSIDE)*7);
   if ((fracmap = (unsigned char*) malloc(pixels)) == NULL)  {
       helper =  (unsigned char**)0x807FFFFC;
       •helper =  (unsigned char*)3;
      if (IRPCSERVER) printf("\nmalloc failed on fracmapXn");
      exit(l);
   }

   helper =  (unsigned char**)Ox807FFFFO;     /* place address of inputimage at a */
   *helper = inputimage;                      /* known location for host to retrieve */

   helper =  (unsigned char**)0x807FFFF4;     /* place address of decimated at a */
   *helper = decimated;                       /* known location for host to retrieve */

   helper =  (unsigned char**)0x807FFFF8;     /* place address of fracmap at a */
   *helper = fracmap;                                /* known location for host to
retrieve */

   pixels = ((IMAGESIDE/2)/DOM4_SKIP)*((IMAGESIDE/2)/DOM4_SKIP);
   if ((dom41ists = (unsigned char**) malloc(pixels*4)) == NULL)  {
      if (IRPCSERVER) printf("\nmalloc failed on dom41ists\n");
      exit(1);
   }

   if ((dom4classes = (unsigned char*) malloc(pixels)) == NULL)  {
      if (IRPCSERVER) printf("\nmalloc failed on dom4classes\n");
      exit(1);
   }
   if ((dom4rots = (unsigned char*) malloc(pixels)) == NULL)  {
      if (IRPCSERVER) printf("\nmalloc failed on dom4rots\n");
      exit(1);
   }
   if ((dom4flips = (unsigned char*) malloc(pixels)) == NULL)  {
      if (IRPCSERVER) printf("\nmalloc failed on dom4flips\n");
```

```
   exit(l);
}
pixels = pixels*2;   /* 2 bytes for short type */
if ((dom4tots = (short*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on dom4tots\n");
   exit(l);
}
pixels = pixels*3;   /* 4 bytes for long type */
/* check the extra space -  seems to crash - run in debugger */
if ((domdom4tots = (long*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloo failed on domdom4tots\n");
   exit(l);
}


pixels = ( (IMAGESIDE/2)/DOM8_SKIP)*((IMAGESIDE/2)/DOM8_SKIP) ;
if ((dom81ists = (unsigned char**) malloc(pixels*4)) == NOLL) {
   if (IRPCSERVER) printf("\nmalloc failed on dom81ists\n");
   exit(l);
}
if ((dom8classes = (unsigned char*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on dom8classes\n");
   exit(1);
}
if ((dom8rots = (unsigned char*) malloc(pixels)) == NULL) {
   if {IRPCSERVER) printf("\nmalloc failed on dom8rots\n");
   exit(1);
}
if ((dom8flips = (unsigned char*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on dom8flips\n");
   exit(1);
}
pixels = pixels*2;   /* 2 bytes for short type */
if ((dom8tots = (short*) malloc(pixels)) == NULL) (
   if (IRPCSERVER) printf("\nmalloc failed on dom8tots\n");
   exit(1);
}
pixels = pixels*3;   /* 4 bytes for long type */
/* check the extra space -  seems to crash - run in debugger */
if ((domdom8tots = (long*) malloc(pixels)) == NULL) (
   if (IRPCSERVER) printf("\nmalloc failed on domdom8tots\n");
   exit(1);
}


pixels = ((IMAGESIDE/2)/DOM16_SKIP)*((IMAGESIDE/2)/DOM16_SKIP);
if ((doml61ists = (unsigned char**) malloc(pixels*4)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on doml61ists\n");
   exit(l);
>
if ((doml6classes = (unsigned char*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on doml6classes\n");
   exit(l);
}
if ((doml6rots = (unsigned char*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on doml6rots\n");
   exit(1);
}
if ((doml6flips = (unsigned char*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on doml6flips\n");
   exit(l);
}
pixels = pixels*2;   /* 2 bytes for short type */
if ((doml6tots = (short*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on doml6tots\n");
   exit(l);
}
pixels = pixels*3;   /* 4 bytes for long type */
/* check the extra space -  seems to crash - run in debugger */
if ((domdoml6tots = (long*) malloc(pixels)) == NULL) {
   if (IRPCSERVER) printf("\nmalloc failed on domdoml6tots\n");
   exit(l);
}

helper = (unsigned char**)0x807FFFB4;     /* place address of dom41ists at a */
```

```
   *helper = (unsigned char *)dom41ists;  /* known location for host to retrieve */

   helper = (unsigned char**)0x807FFFB8;     /* place address of dom81ists at a */
   *helper = (unsigned char *)domSlists;  /* known location for host to retrieve */

   helper = (unsigned char**)0x807FFFBC;     /* place address of doml61ists at a */
   *helper = (unsigned char *)doml61ists; /* known location for host to retrieve */


   pixels = ((IMAGESIDE/BLOCKSIDE)*(IMAGESIDE/BLOCKSIDE)*4); /* x4 because 4 bytes for
int type */
   pixels = pixels/4;   /* but 4 processors */
   /*pixels = pixels/2;*/   /* but 4 processors */
   if ((classO = (unsigned char**) malloc(pixels+4)) == NULL) {
       if (IRPCSERVER) printf("\nmalloc failed on classOW);
      exit(l);
   }

   if ((classl = (unsigned char**) malloc(pixels+4)) == NULL) {
      if (IRPCSERVER) printf("\nmalloc failed on classlW);
      exit(l);
   }

   if ((class2 = (unsigned char**) malloc(pixels+4)) == NULL) {
      if (IRPCSERVER) printf("\nmalloc failed on class2\n");
      exit(l);
   }

   if ((class3 = (unsigned char**) malloc(pixels+4)) == NULL) {
      if (IRPCSERVER) printf("\nmalloc failed on class3\n");
      exit(l);
   }

   helper = (unsigned char**)0x807FFFDO;     /* place address of classO at a */
   •helper = (unsigned char *)classO;     /* known location for host to retrieve */

   helper = (unsigned char**)O0x807FFFD4;     /* place address of classl at a */
   *helper = (unsigned char *)classl;     /* known location for host to retrieve */

   helper = (unsigned char**)0x807FFFD8;     /* place address of class2 at a */
   *helper = (unsigned char *)class2;     /* known location for host to retrieve */

   helper = (unsigned char**)0x807FFFDC;     /* place address of class3 at a */
   *helper = (unsigned char *)class3;     /* known location for host to retrieve */


   pixels = 100;
   if ((spacer = (unsigned char*) malloc(pixels)) == NULL) {
       if (IRPCSERVER) printf("\nmalloc failed on spacerOW);
      exit(l);
   }


   pixels = ((IMAGESIDE/BLOCKSIDE)*(IMAGESIDE/BLOCKSIDE)*7); /* no. blocks x 7 */
   pixels = pixels + 100;
   if ((returnO = (unsigned char*) malloc(pixels)) == NULL) {
       if (IRPCSERVER) printf("\nmalloc failed on returnOW);
      exit(l);
   }

   if ((returnl = (unsigned char*) malloc(pixels)) == NULL) {
      if (IRPCSERVER) printf("\nmalloc failed on returnl\n");
      exit(l);
   }

   if ((return2 = (unsigned char*) malloc(pixels)) == NULL) {
      if (IRPCSERVER) printf("\nmalloc failed on return2\n");
      exit(l);
   }

   if ((return3 = (unsigned char*) malloc(pixels)) == NULL) {
      if (IRPCSERVER) printf("\nmalloc failed on return3\n");
      exit(l);
   }

   helper = (unsigned char**)0x8O7FFFC0;     /* place address of returnO at a */
```

```
   •helper = returnO;                              /• known location for host to
retrieve */

   helper = (unsigned char**)0x807FFFC4;     /* place address of returnl at a */
   •helper = returnl;                              /• known location for host to
retrieve */

   helper = (unsigned char**)0x807FFFC8;     /* place address of return2 at a */
   *helper = return2;                              /* known location for host to
retrieve */

   helper = (unsigned char**)0x807FFFCC;     /* place address of return3 at a */
   •helper = return3;                              /* known location for host to
retrieve */


   pixels = IMAGESIDE*IMAGESIDE;
   n = 0;
   while (n < pixels) {
        *(inputimage+n) = (unsigned char)0;
        n = n + 1;
   }
   if (RPCSERVER != 1) printf("\nlmage initialisation complete.\n") ;
   flush(inputimage, pixels);

   /•a*************************************************************************/
   /* Start the RPC server                                                    */
   /*************************************************************************/

   if (RPCSERVER == 1) RpcServerStart(l);
                         /* return 0 for debugging */
                              /* or return 1 as a response to client's RpcInitO */

   /* delay */
   pixels = IMAGESIDE * IMAGESIDE;
   n = 0;
   k = 0;
   while (n < pixels) {
        k = k + n;
      n = n + 1;
   )
   /*
   if (RPCSERVER == 1) Server_Sync(); */ /* confirm coff boot */

   /*************+*************************************************************/
   /* Wait for data to be transferred to 'C80                                 */
   /************.*********.*****************,**,***.************************/

   if (RPCSERVER != 1)(
        printf("address of inputimage is 0x%x\n",inputimage);
   }


   /* delay */
   pixels = IMAGESIDE * IMAGESIDE;
   n = 0;
   k = 0;
   while (n < pixels) {
        k = k + n;
      n = n + 1;
   )

   if (RPCSERVER == 1) Server_Sync();  /* wait for host to complete data transfer */


   /* invert the image data on the MP                                         */
   /*************************************************************************/

   if (RPCSERVER != 1){
        pixels = IMAGESIDE*IMAGESIDE;
        n = 0;
        while (n < pixels) {
                *(inputimage+n) = (unsigned char)0;
        n = n + 1;
        }
```

```
    printf("\nlmage initialisation complete.\n") ;
    flush(inputimage, pixels);
}


/*****************************************************************************/
/* paste a different patch                                                   */
/*****************************************************************************/
/*printf("\nPerforming patch paste on MP\n");*/

if (RPCSERVER != 1){
    n = 0;
    while (n < BLOCKSIDE) {
           k = 0;
    while (k < BLOCKSIDE) {
                positionl = ((n*IMAGESIDE) + k);
                test = ((n*BLOCKSIDE) + k) ;
                *(inputimage+positionl) = (unsigned char)4;
                k = k + 1;
    }
    n = n + 1;
           }
    flush(inputimage, pixels);
    printf("\npatch paste complete.\n");
}

/*****************************************************************************/
/* Do a pre-decimation of the input image                                    */
/*****************************************************************************/
/*printf("\nPerforming pre-decimation on MP\n");*/
n = 0;
while (n < IMAGESIDE) {
     k = 0;
   while (k < IMAGESIDE) {
            positionl = ((n*IMAGESIDE) + k);
            temp =  (*(inputimage+positionl) + *(inputimage+positionl+1)
       + *(inputimage+positionl+IMAGESIDE) + *(inputimage+positionl+IMAGESIDE+1));
            temp = (temp/4);
            position2 = (((n/2)*(IMAGESIDE/2)) +(k/2));
            *(decimated+position2) = (unsigned char)temp;
            k = k + 2;
   }
   n = n + 2;
}

if (RPCSERVER != 1)(
    printf("XnPredecimation complete.\n");
}

/*if (RPCSERVER)
    Server_Sync();*/  /* signal that processing is complete */


/*****************************************************************************/
/* set up lists of addresses of blocks for pp's to process                   */
/*****************************************************************************/
PP0_BLOCKS = ZERO;
PP1_BLOCKS = ZERO;
PP2_BLOCKS = ZERO;
PP3_BLOCKS = ZERO;

mapindex = 0;

pixels = IMAGESIDE*IMAGESIDE;
blocks = ((IMAGESIDE/BLOCKSIDE)*(IMAGESIDE/BLOCKSIDE));
n = 0;
while (n < IMAGESIDE/4) {
    k = 0;
    while (k < IMAGESIDE) {
           position = (n * IMAGESIDE) + k;
    *(classO+mapindex) = (unsigned char *)(inputimage+position);
     PP0_BLOCKS += 1;
    *(classl+mapindex) = (unsigned char *)(inputimage+position+OxlOOOO);
     PP1_BLOCKS += 1;
    *(class2+mapindex) = (unsigned char *)(inputimage+position+0x20000);
     PP2_BLOCKS += 1;
```

```
    *(class3+mapindex) = (unsigned char *)(inputimage+position+0x30000);
     PP3_BL0CKS += 1;
     k = k + BLOCKSIDE;
     mapindex = mapindex + 1;
   }
   n = n + BLOCKSIDE;
)
/*
*(classO+mapindex) = (unsigned char *)classO;
*(classl+raapindex) = (unsigned char *)classl;
*(class2+mapindex) = (unsigned char *)class2;
*(class3+mapindex) = (unsigned char *)class3;
*/

/*n = 0;
while (n < IMAGESIDE) {
     k = 0;
     while (k < IMAGESIDE) {
            position = n * IMAGESIDE + k;
     *(classO+mapindex) = (unsigned char *)(inputimage+position);
     *(classl+mapindex) = (unsigned char *)(inputimage+position+BLOCKSIDE);
     *(class2+mapindex) = (unsigned char *)(inputimage+position+(2*BLOCKSIDE));
     *(class3+mapindex) = (unsigned char *)(inputimage+position+(3*BLOCKSIDE));
      k = k + (4*BLOCKSIDE);
      mapindex = mapindex + 1;
   }
   n = n + BLOCKSIDE;
} */


PP0_DONE = 0;
PP1_DONE = 0;
PP2_DONE = 0;
PP3_DONE = 0;

PPSTART_VEC(0) = sep_tsk_i0;
PPSTART_VEC(1) = sep_tsk_il;
PPSTART_VEC(2) = Sep_tsk_i2;
PPSTART_VEC(3) = Sep_tsk_i3;

/*if (RPCSERVER == 1)
     Server_Sync();*/

/********** + **********.********** + .**********.*-******/
/* Reset the PPs to get them in a known state.                    */

IE 1= 0x24001801; /*0x24001801;*/
IE |= 0x40000000;

RESET_PPS;
if (!RPCSERVER)
     printfC'sent reset to PP's, PPERROR is 0x%x\n", PPERROR) ;

RESET_PPS;
if (!RPCSERVER)
     printfC'sent reset to PP's, PPERROR is 0x%x\n", PPERROR);

/*************************************************************************/
/* Clear INTPEN flag. Each PP writes 1 into INTPEN, when done            */
/*************************************************************************/
INTPEN = ~0;

if (!RPCSERVER)
     printf("cleared INTPEN. value : 0x%x, PPERROR is 0x%x\n",INTPEN, PPERROR);

/*************************************************************************/
/* Ensure all PP's were halted by the reset, then start PP's             */
/*************************************************************************/


while ((PPERROR & INTER_PPS) != INTER_PPS) {
     if (!RPCSERVER)
     printf("reset failed : INTPEN is 0x%x, PPERROR is 0x%x\n",INTPEN, PPERROR);
}

      /* because when each PP halts, the corresponding bit (16 to 19) */
```

```
        /* in PPERROR will become set.  When all 4 are set, the AND of  */
        /* PPERROR with 0x000f0000 will give 0x000f0000, so the while   */
        /* statement halts
                                                   */


    START_PPS;
    if (IRPCSERVER)
         printf("\nPP start cmnd sent, PPERROR is 0x%x\n", PPERROR);

    if (IRPCSERVER) {
         printf("PPERROR : 0x%x, INTPEN : 0x%x\n",PPERROR, INTPEN);
    }

    /**************************************************************************/
    /* Print table header.                                                  */

    if (IRPCSERVER) {
         printf("      k       pp0      ppl      pp2      pp3      \n");
         printf ("      ________________________________________\n");
    }                 ~
    /it*********************************************************************/
    /* Wait for all PP's to signal that they have completed.               */
    /**************************************************************************/

                         /* INTER_PPS */

    while ((INTPEN & INTER_PPS) != INTER_PPS){
         helper = (unsigned char**)0x807FFFEC; /* place INTPEN at a known */
          *helper = (unsigned char *)INTPEN;    /* location for host to retrieve */
       flush((unsigned char*)0x807FFFEC, 4);
       helper = (unsigned char**)0x807FFFE8; /* place PPERROR at a known */
          *helper = (unsigned char *)PPERROR;   /* location for host to retrieve */
       flush((unsigned char*)0x807FFFE8, 4);
    }


/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

    PP0_DONE = 0;
    PP1_DONE = 0;
    PP2_DONE = 0;
    PP3_DONE = 0;

    PPSTART_VEC(0) = &ep_tsk_c0;
    PPSTART_VEC(1) = &ep_tsk_cl;
    PPSTART_VEC(2) = &ep_tsk_c2;
    PPSTART_VEC(3) = &ep_tsk_c3;
    IE |= 0x24001801; /*0x24001801;*/
    IE |= 0x40000000;

    RESET_PPS;
    if (!RPCSERVER)
         printf("sent reset to PP's, PPERROR is 0x%x\n", PPERROR);

    RESET_PPS;
    if (IRPCSERVER)
         printf("sent reset to PP's, PPERROR is 0x%x\n", PPERROR);


    /**************************************************************************/
    /* Clear INTPEN flag. Each PP writes 1 into INTPEN, when done           */
    /*****************************+********************************************/
    INTPEN = ~0;

    if (IRPCSERVER)
         printf("cleared INTPEN. value : 0x%x, PPERROR is 0x%x\n",INTPEN, PPERROR);


    /* Ensure all PP's were halted by the reset, then start PP's            */
    /**************************************************************************/


    while ((PPERROR S INTER_PPS) != INTER_PPS) {
         if (IRPCSERVER)
         printf("reset failed : INTPEN is 0x%x, PPERROR is 0x%x\n",INTPEN, PPERROR);
```

```
}

      /* because when each PP halts, the corresponding bit (16 to 19) */
      /* in PPERROR will become set.  When all 4 are set, the AND of  */
      /* PPERROR with OxOOOfOOOO will give OxOOOfOOOO, so the while   */
      /* statement halts
                                                    */


START_PPS;
if {IRPCSERVER)
    printf("\nPP start cmnd sent, PPERROR is Ox%x\n", PPERROR);

if (IRPCSERVER) {
    printf("PPERROR : Ox'tx, INTPEN : Ox%x\n", PPERROR, INTPEN);
}

/*************************************************************************/
/* Print table header.                                                   */

if (IRPCSERVER) {
    printf("      k       ppO      ppl     pp2     pp3     \n");
    printf("      _______________________________________\n");
}
/*************************************************************************/
/* Wait for all PP's to signal that they have completed.                 */



                    /* INTER_PPS */

while ((INTPEN & INTER_PPS) != INTER_PPS)(
    helper = (unsigned char**)0x807FFFEC; /* place INTPEN at a known */
    *helper = (unsigned char *)INTPEN;    /* location for host to retrieve */
   flush((unsigned char*)0>:807FFFEC, 4);
   helper = (unsigned char**)0x807FFFE8; /* place PPERROR at a known */
    *helper = (unsigned char *)PPERROR;   /* location for host to retrieve */
   flush((unsigned char*)0x807FFFE8, 4);
}


/*if (RPCSERVER)
    Server_Sync();*/  /* signal that processing is complete */

helper = (unsigned char**)0x807FFFEC; /* place INTPEN at a known */
*helper = (unsigned char *)INTPEN;    /* location for host to retrieve */
flush((unsigned char*)0x807FFFEC, 4);

helper = (unsigned char**)0x807FFFE8; /* place PPERROR at a known */
*helper = (unsigned char *)PPERROR;   /* location for host to retrieve */
flush((unsigned char*)0x807FFFE8, 4);


/* compile returned mappings from pp's into fracmap */

/*n = 0;
while (n < 7168) (
    *(fracmap+n) = *(returnO+n);
    *(fracmap+7168+n) = *(returnl+n);
    *(fracmap+14336+n) = *(return2+n);
    *(fracmap+21504+n) = *(return3+n);
   n = n + 1;
}*/


PPOJBLOCKS = BLOCKSPERPP * 7;
PP1_BLOCKS = BLOCKSPERPP * 7;
PP2_BLOCKS = BLOCKSPERPP * 7;
PP3_BLOCKS = BLOCKSPERPP * 7;

n = 0;
while (n < PP0BLOCKS) (
    *(fracmap+n) = *(returnO+n);
   n = n + 1;
)
bytes = PP0__BLOCKS;
```

```
    n = 0;
    while (n < PP1BLOCKS) {
         *(fracmap+bytes+n) = *(returnl+n);
       n = n + 1;
    }
    bytes = bytes + PP1_BLOCKS;

    n = 0;
    while (n < PP2_BLOCKS) {
         *(fracmap+bytes+n) = *(return2+n);
       n = n + 1;
    }
    bytes = bytes + PP2_BLOCKS;

    n = 0;
    while (n < PP3_BLOCKS) {
         *(fracmap+bytes+n) = *(return3+n);
       n = n + 1 ;
    }
    bytes = bytes + PP3_BLOCKS;

    helper = (unsigned char**)0x807FFFE4; /* place number of bytes in frac file */
    •helper = (unsigned char *)bytes;   /* at a known location for host to retrieve */
    flush((unsigned char*)0x807FFFE4, 4);


    /*flush(fracmap, 28672);*/

   /* n = 0;
    while (n < 35) {
         *(fracmap+n) = *(returnO+n);
       n = n + 1;
    >*/

/*   if  (RPCSERVER != 1){
       n = 0;
       while (n < 5) (
              n = n * 7;
              printf("\n%d\t%d\t%d\t%d\t%d\t%d\t%d\n",*(fracmap+n), *(fracmap+n+1),
*(fracmap+n+2), *(fracmap+n+3), *(fracmap+n+4), *(fracmap+n+5), *(fracmap+n+6));
       n = n/7;
       n = n + 1;
       }
    }*/

    if (RPCSERVER)
         Server_Sync();  /* signal that processing is complete */


    /* Read results generated by the PP's and print them out.              */
    /*****************************************************************************/
    if (!RPCSERVER) {
         k = 0;
         while (k < 5) {
         /*****************************************************************************/
                /* Note that results are read directly from memory, because the PP's*/
                /* did not write them into cache.                                    */

         printf(" %6d  %6d  %6d  %6d  %6d\n", k,
                NOCACHE_UCHAR(comm_pp0[k]), NOCACHE_UCHAR(comm_ppl[k]),
                NOCACHE_UCHAR(comm_pp2[k]), NOCACHE_UCHAR(comm_pp3[k]));
         k = k + 1;
         }
         }


    /*Server_Sync();*/
    /* free the memory allocated for the bitmaps */
    free(inputimage);
    free(decimated);
    free(fracmap);

    free(dom41ists);
    free(dom4classes);
    free(dom4rots);
```

```
    free(dom4flips);
    free(dom4tots);
    free(domdom4tots);

    free(dom81ists);
    free(dom8classes);
    free(dom8rots);
    free(dom8flips);
    free(dom8tots);
    free(domdom8tots);

    free(doml61ists);
    free(doml6classes);
    free(doml6rots);
    free(doral6flips);
    free(doml6tots);
    free(domdoml6tots);

    free(class0);
    free(classl);
    free(class2);
    free(class3);
    free(spacer);
    free(return0);
    free(returnl);
    free(return2);
    free(return3);

}
```

D.8. PPOJNIT.C

(PP1JNIT.C, PP2JNIT.C and PP3JNIT.C are similar)

```
/*************************************************************************/
/* ppO_init.C                                                            */
/*                                                                       */
/*    MAIN() - This main routine will read a global array, compute       */
/*             the factorial of the arrays elements, and write the results*/
/*             back to the global array. It then will signal the MP that  */
/*              it has completed.                                        */
/*                                                                       */
/*************************************+************************++*****/
#include <mvp.h>
#include <stdlib.h>
((include "head_compr .h"


extern sharedpp void                    signal_done();    /* Routine that signals MP
*/

extern sharedpp long                    variance(unsigned char *onchipr, unsigned char
       *mapping, int  side);

extern sharedpp unsigned char classify(unsigned char *block,

              unsigned char *rotation,
                                  unsigned char *flip,
                                  int side);

extern shared unsigned char   *inputimage;
extern shared unsigned char   *decimated;

extern shared unsigned char   **dom41ists;
extern shared unsigned char   *dom4classes;   /* points to list of domain
classifications */
extern shared unsigned char   *dom4rots;      /* points to list of flips needed to
force into class */
extern shared unsigned char   *dom4flips;     /* points to list of rotations needed to
force into class */
extern shared short                          *dom4tots;      /* points to list of
summation of block pixel values */
extern shared long                           *domdom4tots;         /* points to list
of summation of block pixel values"2 */
extern shared int                                   dom4number;

unsigned char                                       *ppO_nextran;     /* Global
pointer to next range block  */
unsigned char                                       *ppO_nextdom;     /* Global
pointer to next domain block */
extern shared unsigned char   comm_ppO[TABLE_MAX];         /* Global Argument array
V

unsigned char                 *onchipd;             /* On-chip copy of domain block
*/
unsigned char                      *onchipr;        /* On-chip copy of range block
                */
unsigned char                 *mapping;             /* list of affine mapping coeffs
*/

main()
{
       int                          k, n, kk, nn, index, offset, count;
   long                             pixels;
   unsigned char       domorient, brightness, contrast;
   int                         ranxpos, ranypos, domxpos, domypos;
   short                            domtot;
   long                        domdomtot;
   int                         blocknumber, currentside;
   int                         domblocksperline;
   int                         ppnumber;
   unsigned char       *base;
   int                         toggle;
   int                         domcounter;
   long                        temp;
   /*********....*+*************************************************************/
   /* Block COPV araument array on-chio, for faster access.                 */
```

```
/*****************************************************************************/

currentside = 4;

pixels = currentside*currentside;
if ((onchipr = (unsigned char*) malloc(pixels)) == NULL) {
     comm_ppO [ZERO] = 1;
     exit(l);   /* terminate if out memory */
}
if ((onchipd = (unsigned char*) malloc(2*pixels)) == NULL) {
     comm_ppO[ZERO] = 2;                  /* allow for two domain blocks */
     exit(1);   /* terminate if out memory */
}
if ((mapping = (unsigned char*) malloc(5)) == NULL) {
     comm_ppO[ZERO] = 3;
     exit(l);   /* terminate if out memory */
}

/* Obtain next range block                                                  */
/*****************************************************************************/

ppnumber = 0;
domcounter = ZERO;
     ppO_nextdom = decimated;
/•offset = (IMAGESIDE/8)*ppnumber;*/ /*number of lines */
offset = ZERO;

index = offset * (IMAGESIDE/2);


memtrans(onchipd, ppO_nextdom, currentside, (IMAGESIDE/2));
  /* fetch first block to start pipeline */
toggle = ZERO;     /* first block was fetched into lower space */
     n = ZERO;
     while (n < (IMAGESIDE/2)){
     k = ZERO;
     while (k < (IMAGESIDE/2)){
     while (COMM s (1 << 29));  /* poll for transfer */

       kk = k + DOM4_SKIP;
       nn = n;
       if (!(kk < (IMAGESIDE/2))){
             kk = ZERO;
          nn = n + DOM4_SKIP;
       }
       if (nn < (IMAGESIDE/2)){
             index = ((nn + offset) * (IMAGESIDE/2)) + kk;
             base = onchipd+((1-toggle)*pixels);
             memtrans(base, (ppO_nextdom+index), currentside, (IMAGESIDE/2));
          /*asm("\tnop");
             asm("\tnop");
                    while (COMM & (1 << 29));*/
       }

       index = ((n + offset) * (IMAGESIDE/2)) + k;

       base = onchipd+(toggle*pixels);

•


       temp = variance(base, mapping, currentside);

      if (temp > DVARIANCETOL){
P
             count = ZERO;

             domtot = ZERO;
             domdomtot = ZERO;
             /*while (COMM & (1 << 29));*/
                          while (count < pixels) {
                          domtot = domtot + (long)(*(base + count));
                          domdomtot = domdomtot + ((long) (Mbase +
count)))*((long)(*(base + count)));
                          count++;
             }
```

```
                /*blocknumber = {(n + offset)/D0M4_SKIP *  (IMAGESIDE/(2*DOM4_SKIP))) +
k/D0M4_SKIP;*/
                               *(dom4tots + domcounter) = domtot;
                               *(domdom4tots + domcounter) = domdomtot;
                *(dom41ists + domcounter) =  (unsigned char *)(decimated + index);
             /**(dora4classes + domcounter) = classify(base,   (dom4rots + domcounter),
                                                 (dom4flips + domcounter), currentside);*/

                domcounter += 1;
           )

           toggle = 1 - toggle;
           k = k + DOM4_SKIP;
           /*whxle (COMM & (1 << 29));*/
         }
         n = n + DOM4_SKIP;
         }


   dom4number = domcounter;  /*((IMAGESIDE/2)/DOM4_SKIP)*((IMAGESIDE/2)/DOM4_SKIP);*/
   /* note domcounter was incremented at the end of the loop but this will compensate
*/
   /*  for the first entry having index 0 */
   free(onchipr);
   free(onchipd);
   free(mapping);
   signal_done();
}
```

D.9. PP0_COMPR.C

(PP1_C0MPR.C, PP2_C0MPR.C and PP3_C0MPR.C are similar)

```
/*************************************************************************/
/* ppO_compr.C                                                           */
/*    ~                                                                  */
/*    MAIN() - This main routine will read a global array, compute       */
/*             the factorial of the arrays elements, and write the results*/
/*             back to the global array. It then will signal the MP that  */
/*              it has completed.                                        */
/*                                                                       V
/-A************************************************************************/
#include <mvp.h>
(•include <stdlib.h>
#include "head_compr.h"


extern sharedpp void                signal_done();    /* Routine that signals MP
*/

extern sharedpp long                        distance(unsigned char         *onchipd,

               unsigned char  *onchipr,
                                    unsigned char        •mapping,
                                    short                       domtot,
                                    long                        domdomtot,
                                    int                         pixels,
                                    int                         side);

extern sharedpp long                        variance(unsigned char *onchipr,

               unsigned char  *mapping,
                                    int                         side);

extern sharedpp unsigned char classify(unsigned char        *block,

               unsigned char  *rotation,
                                    unsigned char        *flip,
                                    int                         side);

extern sharedpp unsigned char transformfunsigned char *blockb,

                unsigned char *block,
                                     unsigned char *flipa,
                                     unsigned char *flipb,
                                     int                        side);


extern shared unsigned char   *inputimage;
extern shared unsigned char   *decimated;

extern shared unsigned char   **dom41ists;
extern shared unsigned char   *dom4classes;   /* points to list of domain
classifications */
extern shared unsigned char   *dom4rots;      /* points to list of flips needed to
force into class */
extern shared unsigned char   *dom4flips;     /* points to list of rotations needed to
force into class */
extern shared short                         *dom4tots;      /* points to list of
summation of block pixel values */
extern shared long                          *domdom4tots;   /* points to list of
summation of block pixel valuesA2 */

extern shared unsigned char   **dom81ists;
extern shared unsigned char   *dom8classes;   /* points to list of domain
classifications */
extern shared unsigned char   *dom8rots;      /* points to list of flips needed to
force into class */
extern shared unsigned char   *dom8flips;     /* points to list of rotations needed to
force into class */
extern shared short                         *dom8tots;      /* points to list of
summation of block pixel values */
extern shared long                          *domdom8tots;   /* points to list of
summation of block pixel valuesA2 */
extern shared int                           dom8number;     /* number of 8x8 domain
blocks to be used */
```

```
extern shared unsigned char    **doml61ists;
extern shared unsigned char    *doml6classes;   /* points to list of domain
classifications */
extern shared unsigned char    *doml6rots;      /* points to list of flips needed to
force into class */
extern shared unsigned char    *doml6flips;     /* points to list of rotations needed
to force into class */
extern shared short                     *doml6tots;      /* points to list of
summation of block pixel values */
extern shared long                      *domdoml6tots;   /* points to list of
summation of block pixel values^2 */


unsigned char                                  *pp0_nextran;     /* Global
pointer to next range block  */
unsigned char                                  *pp0_nextdom;     /* Global
pointer to next domain block */
extern shared unsigned char   comm_pp0[TABLE_MAX];          /* Global Argument array
*/
extern shared int                              PP0_DONE;
extern shared int                              PP0_BLOCKS;
extern shared unsigned char    **class0;
extern shared unsigned char    *return0;

unsigned char               *onchipd;              /* On-chip copy of domain block
V
unsigned char                   *onchipr;          /* On-chip copy of range block
                       */
unsigned char               *mapping;              /* list of affine mapping coeffs
*/
unsigned char                                  •scratch;

unsigned char                                  *ranrot;
unsigned char                                  *ranflip;
unsigned char                                  ranclass;

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-_=*/

main()
{
        int                         k, n, index;
   long                             pixels, s, bestdistance, addrmarker,
returnmarker, dommarker;
   unsigned char      domorient, brightness, contrast;
   int                        ranxpos, ranypos, domxpos, domypos;
   int                         ppnumber;
   long                        t emp;
   /*************************************************************************/
   /* Block copy argument array on-chip, for faster access.                 */
   /*************************************************************************/


   pixels = BLOCKSIDE*BLOCKSIDE;
   if ((onchipr = (unsigned char*) malloc(pixels)) == NULL) {
        comm_pp0[ZERO] = 1;
        exit(l);   /* terminate if out memory */
   }
   if ((onchipd = (unsigned char*) malloc(2*pixels)) == NULL) {
        comm_pp0[ZERO] = 2 ;                 /* allow for two domain blocks */
        exit(l);   /* terminate if out memory */
   }
   if ((mapping = (unsigned char*) malloc(5)) == NULL) {
        comm_pp0[ZERO] = 3;
        exit(l);   /* terminate if out memory */
   }
   if ((scratch = (unsigned char*) malloc(pixels)) == NULL) {
        comm_pp0[ZERO] = 4;
        exit(l);   /* terminate if out memory */
   }

   /*************************************************************************/

                                  /*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
   /* Obtain next range block                                               */
```

```
   /*****************************************************************************/

   ppnumber = 0;

/*****************************************************************************/
   addrmarker = ZERO;
   returnmarker = ZERO;
   dommarker = ZERO;
   while (PP0_DONE == ZERO) {
        ppO_nextran =  (unsigned char *)*(classO+addrmarker);
        ppO_nextdom = decimated;


      index = ppO_nextran - inputimage;

      ranypos = index/IMAGESIDE;
      ranxpos = index%IMAGESIDE;

       memtrans(onchipr, ppO_nextran, BLOCKSIDE, IMAGESIDE);
      asm!"\tnop");
       bestdistance = INITDIST;
      while (COMM & (1 << 29));

      temp = variance(onchipr, mapping, BLOCKSIDE);
      if (temp < VARIANCETOL){
       domxpos = ZERO;
         domypos = ZERO;
         domorient =  (unsigned char)ZERO;
         brightness = *(mapping+3);
         contrast = 128;  /* value of ZERO scaled to centre of UCHAR range d*/
      }



      if (temp >= VARIANCETOL){

       ranclass = classify(onchipr, ranflip, ranflip, 8);
        dommarker = ZERO;
              while (dommarker < dom8number){

            if (ranclass == *(domSclasses+dommarker)){

               ppO_nextdom =  (unsigned char *)*(dom81ists+dommarker);
                      memtrans(onchipd, ppO_nextdom, BLOCKSIDE, (IMAGESIDE/2));
               asm("\tnop");
                      asm("\tnop");
                             while (COMM s (1 « 29));
               /*transform(onchipr, scratch, (dom8flips+dommarker), ranrot, 8);*/
                      s = distance(onchipd, onchipr, mapping, *(dom8tots+dommarker),

       *(domdom8tots+dommarker), pixels, BLOCKSIDE);

                      if (s < bestdistance){
                             bestdistance = s;
                      index = ppO_nextdom - decimated;
                      domypos = index/(IMAGESIDE/2);
                      domxpos = index%(IMAGESIDE/2);
                             domorient =  (unsigned char)ZERO;
                             brightness = *(mapping+3);
                             contrast = *(mapping+4);
               }

               if (S < DISTANCETOL){
                      dommarker = dom8number;
               }


            }
            dommarker = dommarker + 1;
               }

      }


       /* write results back only once to external memory */
```

```
      *(returnO+returnmarker) =  (unsigned char)(ranxpos/2);
      *(returnO+returnmarker+1) = (unsigned char)(ranypos/2);
      *(returnO+returnmarker+2) =  (unsigned char)domxpos;
      *(returnO+returnmarker+3) =  (unsigned char)domypos;
      *(returnO+returnmarker+4) =  (unsigned char) dornorient;
      *(returnO+returnmarker+5) =  (unsigned char)brightness;
      *(returnO+returnmarker+6) =  (unsigned char)contrast;

      if  (iRPCSERVER){
        commjppO [ ZERO] = (unsigned char) (ranxpos/2);
        comm_ppO[l] = (unsigned char)(ranypos/2);
               comm_ppO[2] =  (unsigned char)domxpos;
               comm_ppO[3] =  (unsigned char)domypos;
               comm_ppO[4] =  (unsigned char)domorient;
               comm_ppO[5] =  (unsigned char)brightness;
               comm_ppO[6] =  (unsigned char)contrast;
      }



        /* Return the coeffs of the best affine mapping for that range block & signal
done */
        /*****************************************************************************/

               returnmarker = returnmarker + 7;
        addrmarker = addrmarker+1;

        if (addrmarker == BLOCKSPERPP)  PPO^DONE = 1;
   }
   PP0_BLOCKS = BLOCKSPERPP * 7;
   free(onchipr);
   free(onchipd);
   free(mapping);

   signal_done();
}
```

D.10. CLASSIFY.C

```
#include "head_compr.h"
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
/* compute the rel brigtness of quads of block & return class & rotation to normalise
*/
unsigned char classify(unsigned char *block, unsigned char *rotation,
                                                  unsigned char *flip,  int
side)
{
        /* first compute the average grey level of the range block */
   int                        quad[4];
   int                        order[4] ;
   int                        temp;
   unsigned char        clas;
   /•extern shared classification     classmap[4][4][4];*/
        register int   n, k;


   quad[0]  = 0;
   quad[l]  = 0;
   quad[2]  = 0;
   quad [3]  = 0;

   order[0]  = 0;
   order[l]  = 1;
   order[2]  = 2;
   order[3]  = 3;

        n = 0;
   while  (n <  (side/2))  (
        k = 0;
        while  (k <  (side/2))  {
          quad[0]  += *(block +  ((n*side)  + k)   );
          quad[l]  += *(block +  ((n*side)  +  (k+side/2))   );
          quad[3]  += *(block +  (((n+side/2)*side)  + k)   );
          quad[2]  += * (block +  (( (n+side/2)*side)  +  (k+side/2))   );
                      k = k + 1;
        }
        n = n + 1;
        }


   for  (n = 3;  n > 0;  n--){
        for  (k = 0;  k <= n-1;  k++){
        if (quad[k]  > quadtk+1])  {
                temp = order[k];
             order[k] = order[k+l);
             order[k+1]  = temp;
             temp = quad[k];
             quad[k] = quad[k+1];
             quad[k+1)  = temp;
          }
      }
   }


   switch(order[0])  {
        case 0 : switch(order[1])  {
                            case 1 : switch(order[2])  {
                                          case 2 :  *flip = 0;  return(0);
                            case 3 :  *flip = 0;  return(l);
                                    }
                 case 2 : switch(order[2] )  {
                                          case 1  :  *flip = 0 ;  return(0);
                            case 3 :  *flip = 0;  return(1);
                                    }
                 case 3 : switch(order[2])  (
                                          case 1  :  *flip = 0;  return(O);
                            case 2 :  *flip = 0;  return(1);
                                    }
      }
      case 1 : switch(order[1])  {
                            case 0 : switch(order[2])  {
                                          case 2 :  *flip = 4;  return(1);
                            case 3 :  *flip = 4;  return(O);
```

```
                                          }
                 case 2 : switch(order[2]) {
                                              case 0 : *flip = 3; return(2);
                                 case 3 : *flip = 2; return(0);
                                          }
                 case 3 : switch(order[2]) (
                                              case 0 : *flip = 6; return(2);
                                 case 2 : *flip = 2; return(l);
                                          }
         }
         case 2 : switch(order[1]) {
                                 case 0 : switch(order[2]) {
                                              case 1 : *flip = 1; return (1);
                                 case 3 : *flip = 1; return(2);
                                          )
                 case 1 : switch(order[2]) {
                                              case 0 : *flip = 6; return(O);
                                 case 3 : *flip = 7; return(2);
                                          }
                 case 3 : switch(order[2]) {
                                              case 0 : *flip = 3; return(0);
                                 case 1 : *flip = 7; return(l);
                                          }
         }
         case 3 : switch(order[1]) {
                                 case 0 : switch(order[2]) {
                                              case 1 : *flip = 1; return(0);
                                 case 2 : *flip = 4; return(2);
                                          }
                 case 1 : switch(order[2]) {
                                              case 0 : *flip = 6; return(l);
                                 case 2 : *flip = 2; return(2);
                                          }
                 case 2 : switch(order[2]) {
                                              case 0 : *flip = 3; return(l);
                                 case 1 : *flip = 7; return(0);
                                          }
         }
         default : clas = 4; *flip = 8;   /* error condition */
     }
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

D.11.VARIANCE.C

```
#include "headcompr.h"

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;=-=-=-=-=_=-=-=-=_=-=-=-=_=_=_=*/
/* compute the variance over a block of size sidexside pixels */
long variance(unsigned char *onchipr, unsigned char *mapping, int side)
{
        /* first compute the average grey level of the range block */
        long    blocktot;
   long vartot;
   short       pixels;
   register int n;

   pixels  = side * side;

   blocktot = 0;
   vartot = 0;
   n = 0;
   while (n < pixels){
        blocktot = blocktot + (long) (*(onchipr + n)) ;
      vartot = vartot +(*(onchipr + n))  *  (*(onchipr + n));
      /* vartot = vartot + pow(* (onchipr + n),2);V
      n = n + 1;
   }

   vartot = vartot - (blocktot * blocktot)/pixels;
   /*vartot = vartot - ( (pow(blocktot,2))/pixels);*/
   vartot = vartot/(pixels - 1);

   if (vartot < 0) vartot = (-1)*vartot;

   blocktot = (blocktot/pixels)/2 + 128;
   if (blocktot > 255) blocktot = 255;
   if (blocktot < 0) blocktot = 0;
   *(raapping+3) = blocktot; /*/pixels;*/                  /* return avg greylevel */

   return(vartot);
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

D.12. DISTANCE.C

```
#include "head_compr.h"
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
/* compute the distance between two blocks each of size sidexside pixels */
long distance(unsigned char *onchipd, unsigned char *onchipr, unsigned char 'mapping,
                                                short domtot, long domdomtot,int pixels,
int side)
{
        /* first compute the average grey level of the range block */
        long rantot;
        long domrantot;
        long ranrantot;
        long offset;
        long scalefactor;
        long top;
        long bottom;
        long prelim;
        long total, terml, term2, term3;

   /* loop thru block to compute prelim results */
   /* these results will be used to optimise affine mapping */

        register int n,k;
   /*domtot = 0;
   domdomtot = 0;*/
        rantot = 0;
        domrantot = 0;
        ranrantot = 0;

        n = 0;
        while (n < pixels) {
                /*domtot = domtot + (long)(*(onchipd + n));
        domdomtot = domdomtot + ((long)(*(onchipd + n)))*((long)(*(onchipd + n)));*/
        rantot = rantot + (long) (*(onchipr + n) ) ;
        domrantot = domrantot + ((long)(*(onchipr + n)))*((long)(*(onchipd + n)));
        ranrantot = ranrantot + ((long)(*(onchipr + n)))*((long)(*(onchipr + n)));
      n++;
        }

        top = (long)((domrantot * pixels) - (rantot * domtot));
        bottom = ((domdomtot * pixels) - (domtot * domtot));
        top = top * SGAIN;              /* avoid rational values */
        if (bottom == (long)0) {
        scalefactor = (long)0;
        offset = (long)(rantot/pixels);
        }

        else {
        scalefactor = (long)(top/bottom);
        if (scalefactor > (long)127) {
        scalefactor = (long)127;
        }
        if (scalefactor < (long)(-128)) {
        scalefactor = (long)(-128);
        }
        offset = (long)((SGAIN*rantot) - (scalefactor * domtot)) / pixels;
        offset = offset/SGAIN;
        }

        if (offset > 255) {
        offset = 255;
        }
        if (offset < -256) {
        offset = -256;
        }

        terml = ranrantot;
        term2 = scalefactor*((scalefactor*domdomtot)/SGAIN - (2*domrantot) +
 (2*offset*domtot))/SGAIN;
        term3 = offset*((pixels*offset) - (2*rantot));
        total = (terml + term2 + term3)/pixels;
```

```
        /* loop thru again to compute rras error */
        /*total = 0;
        n = 0;
        while (n < pixels) {
        prelim = (long) (Monchipd + n) ) - (long) (*(onchipr + n));
        total = total + prelim;
        n++;
        } */


        if (total < 0){
        total = (-1) * total;
        }

        offset = offset/2;
        offset = offset + 128;                          /* DC shift to UCHAR */
        *(mapping + 3) = (unsigned charloffset;
        scalefactor = scalefactor + 128;
        *(mapping +4) = (unsigned char)scalefactor;

        return(total);
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

D.13. SIGNALS

```
        .global $signal_done

$signal_done:  d7 = comm & 7                /extract the PP number.
               d7 = 0xl\\d7                 /convert the PP number to the
                                                          /corresponding PP
command word
                                  /designator bit.
               d7 = 0x40002100 |d7          /message interrupt to MP
                                  cmnd = d7            ;submit
                       nop                /valid instructions in delay slots
                       nop
               nop
```

D.14. PACKET_TRANSFER.S

```
        .global $PP_dim_dim
                        .include "c:\c8xtools\lib\packetpp.i"

SPP_dim_dim:

                        ; when a function is called the 1st

                        ; 4 arguments are placed in dl, d2, d3
                                        ; d4 in that order
                                        ; so dl holds src address, d2 holds
                                        ; dst address
                                        ; NB  sp = a6 or al4
                dO = d7                    /preserve d7

                d7 = 0xl\\31               ; Prepare PT_Options.
                  || *(a8 = pba + 0x200) = a8 ; Set Next-Entry Address to itself.

                *a8.sPT_Options = d7          ; Dimensioned-to-Dimensioned transfer
                                              ; with stop bit set.
                *a8.sPT_SrcStartAddress = dl  ; src Start Address passed as 1st
                                              ;argument
                *a8.sPT_DstStartAddress = d2  ; dst Start Address passed as 2nd
                                              ;argument
                 d7 = 0x00000008
                *a8.sPT_SrcBACount = d7       ; src B Count = 0, A Count = 8.

                d7 = 0x00070008
                *a8.sPT_DstBACount = d7       ; dst B Count = 7, A Count = 8.

                d7 = 0x7
                *a8.sPT_SrcCCount = d7        ; Source C Count = 7.

               d7 = 0x0
                *a8.sPT_DstCCount = d7        ; dst C Count = 0.

               d7 = 0x0                       ; 0x8   ; d7 = OxlWlO
               *a8.ePT_SrcBPitch = d7         ; Src B pitch = 0

                d7 = 0x8
               *a8.ePT_DstBPitch = d7         ; dst B Pitch = 8.

               d7 = OxlWlO
                *a8.ePT_SrcCPitch = d7        ; src C pitch = 512

               d7 = 0x0
                *a8. ePT_DstCPitch = d7       ; dst C Pitch = 0.

                nop
                nop
               comm = comm I 0xl\\28          ; submit request
               d7 = dO                        ; restore d7

         ; now poll transfer
                nop
                nop

Poll:           al5 = comm S 0xl\\29          ; Test Q bit (COMM register bit 29)
                br =[nz] Poll                 ; Continue in Poll loop as long as
                                              ; Packet Transfer Request is queued
                nop                           ; Branch delay slot 1
                nop                           ; Branch delay slot 2

                ;a8 = dl
                ;br = iprs
                ;nop
                ; nop
```