INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

THE IMPACT OF COMPUTER ARCHITECTURE FEATURES ON IMAGE PROCESSING APPLICATION EXECUTION TIMES: A CASE STUDY USING MPEG IMAGE SEQUENCE COMPRESSION ON THE IBM SP2

by

Jeremy Alan Salinger

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering: Systems)
in The University of Michigan
2000

Doctoral Committee:

Professor Trevor Mudge, Co-chair
Associate Professor Gregory Wakefield, Co-chair
Professor William Martin
Associate Professor Jeffrey Fessler

UMI Number: 9963887

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

In loving memory of
Grandfather
William Gallancy, z'l
University of Michigan
College of Engineering
Class of 1924

ACKNOWLEDGMENTS

Heartfelt thanks go to my wife, Vicki, and children, Angelica, Dana, Shana, and Mira, without whose encouragement, patience, and understanding this long effort would not have been possible. Many thanks to Noel Brewer and Ramesh Jain who encouraged me to begin this effort and to the faculty at the University of Michigan, particularly Greg Wakefield and Trevor Mudge, for their guidance and insight when I faced difficulties with the research subject. This dissertation would not have reached its current level of cohesion without insightful editorial comments provided at critical times in the writing process by Rudy Schmerl and Paul Mohan. Finally, thanks go to the community of engineers and managers at all the places I have worked, who have contributed to all the projects that provided background for the research reported here.

PREFACE

The research reported in this dissertation grew from 20 years of prior research on image processors by the author performed at several companies. The first two projects, in the 1978 to 1980 time frame, demonstrated the high throughput achievable with circuits dedicated to a specific algorithm. The first implemented a convolution-based filter used in the back projection algorithm of a computerized axial tomography (CAT) scanner for Elscint, Ltd. The second project implemented a pattern recognition algorithm for reading the courtesy amount on checks for Burroughs Corporation (later Unisys).

These early projects were followed, over many years, by work on more general-purpose image processors for industrial and military applications. The most aggressive project was the development of the Advanced Target Cueing and Recognition Engine (ATCURE) at the Environmental Research Institute of Michigan (ERIM). There the author led a team of very creative engineers to develop a heterogeneous computer with components specialized for image processing. The computer includes enough programmability to implement a wide variety of target recognition and tracking algorithms in real time.

The dissertation research began by studying the use of systolic processors, in particular the Intel iWarp, for implementing image processing applications. That effort collapsed when Intel stopped supporting the product. We then turned to the IBM SP series, which had recently become available. The research reported here completes the examination of alternative levels of specialization, from designing image-processing-algorithm specific circuits to the enhancement of general-purpose processors to facilitate image processing.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

PREFACE iv

LIST OF FIGURES vii

LIST OF TABLES ix

CHAPTER

I. INTRODUCTION 1

Problem Description 2
Scope and Limitations 5
Contribution of the Dissertation 11

II. BACKGROUND 14

Image Processing Terminology 14
Image Processing Operations 17
Computer Architecture Implications of Image-Processing Applications 18
Image Algebra 21
MPEG Encoding Applications 27
MPEG Image Sequences 29
Encoding Overview 32
Computational Complexity of Encoding Algorithms 38

III. PREVIOUS COMPUTER ARCHITECTURE RESEARCH 40

Transform Specific Image Processing Hardware 42
SIMD Image Processing Architectures 44
Systolic Array Processors for Image Processing 46
Arrays of Digital Signal Processors 54
MIMD Architectures 60
IBM SP2 Architecture 64
Enhanced Instruction Set Architectures 68

IV. EXPERIMENTAL METHODS AND TOOLS 69
Scope 69
Source Code and Test Images 70
Compilation, Execution, And Timing Analysis Tools 71
Parallelization Tools 72
V. SERIAL PROGRAM ANALYSIS 76
Serial Code Structure Analysis 76
Serial Code Timing Analysis 82
VI. FINE-GRAIN (INSTRUCTION-LEVEL) EXECUTION EFFICIENCY IMPROVEMENTS 87
Instruction Level Timing Analysis 88
Impact of Compiler Exploitation of Instruction Set Architecture 93
Impact of Additional Functional Units 99
VII. COARSE-GRAIN PARALLELIZATION AND EFFICIENCY IMPROVEMENTS 105
Parallel Program Structure Analysis 106
Parallel Program Implementation Technique 111
Parallel Implementation Task Distribution And Control 112
Parallel Program Control 113
Parallel Implementation Communication Protocols 114
Limits Due To Algorithms Used In The Original Source Code 120
Limits Due to Communication Hardware and Software 122
Evaluation of Hypothesized Architecture Improvements 124
VIII. SIMD ISA EXTENSIONS FOR IMAGE PROCESSING ON GENERAL PURPOSE COMPUTERS 126
The Subword Parallelism Approach 126
Effectiveness of MMX ISA Extensions On MPEG2 Program Execution 129
Improved SIMD Extensions For Image Processing Applications 132
Impact of Subword Parallel ISA Extensions on MPEG2 Execution Times 135
IX. CONCLUSIONS 137
REFERENCES 141

LIST OF FIGURES

Figure

1. Computing technology for reducing execution times of image-processing applications 6
2. Video frame display formats 16
3. Four pixels in a result image and their associated neighborhoods in a source image 23
4. Sobel image gradient templates 25
5. High-level block diagram of image sequence compression and decompression 28
6. Image sequence decomposition in MPEG encoding 31
7. MPEG frame types and processing order: interframes (I), predictive frames (P), and bi-directional-predictive frames (B) 32
8. MPEG encoding block diagram 34
9. Pseudo-code for MPEG2 encoding algorithm steps 35
10. Computer throughput enhancement approaches 42
11. ATCURE subsystems and communication structure 48
12. ATCURE Image Processing Subsystem block diagram 49
13. Block diagram of the ATCURE Pipelined Processor Element 52
14. Node architecture from Normile and Wright (1991) 55
15. Parallel processor architecture from Normile and Wright (1991) 55
16. Texas Instruments TMS320C80 Multi-Media Video Processor 58

17. Duncan's MIMD interconnect topologies: (a) bus, (b) ring, (c) mesh, (d) tree, and (e) hypercube. 61

18. System memory distribution alternatives: (a) centralized, (b) distributed shared, and (c) distributed message passing. 62

19. SP2 network topology 65

20. SP2 node CPU architecture 66

21. Images from the sflowg sequence used for program execution time measurements 71

22. Structure of MPEG2 encoding program 77

23. dist1 code for calculating absolute difference of image regions 88

24. dist1 code for calculating absolute difference of image regions 94

25. dist1 code modified to use floating point absolute value and addition instructions 100

26. Execution time improvements for alternative optimization methods 104

27. Parallel implementation block diagram 110

28. Communication patterns for parallel execution 115

29. Message tree used to distribute source images to 16 nodes 118

30. Parallel implementation execution times versus number of nodes used 123

LIST OF TABLES

Table

I. Image-processing applications 17

II. Typical image-processing operations 20

III. Estimated computational performance requirement (millions of operations per second), based upon Akiyama (1994) 39

IV. Exploitable image-processing applications characteristics 41

V. Key parameters used in encoding experiments 78

VI. Unoptimized-implementation execution time for each subroutine by frame encoding type 83

VII. Execution time breakdown for optimized serial code processing 150 frames 84

VIII. Execution time per frame for unoptimized serial implementation 85

IX. Average time to process one frame of each type using optimized serial implementation 85

X. Assembler code from one line of inner loop of original source code 90

XI. Essential operations of inner loop of dist1 for original code 92

XII. Assembler code for one line of inner loop of source code in Figure 24 95

XIII. Essential operations of inner loop of dist1 using __abs instruction 96

XIV. Execution time breakdown for serial implementation that uses abs instruction 97

XV. Average times, by frame type, for executing serial implementation that uses abs instruction 98

XVI. Essential assembler language instructions for inner loop of source code in Figure 25 101

XVII. Execution time breakdown for serial implementation using floating point absolute value instruction 102

XVIII. Execution time breakdown for serial implementation optimization using floating point absolute value (fabs) instruction 103

XIX. Minimum required communication bandwidth for real-time execution 120

XX. Parallel implementation execution times 122

XXI. Subword instruction sets 128

XXII. dist1 implementation using MMX instructions 130

XXIII. dist 1 implementation using improved SIMD instruction set 134

XXIV. Potential execution times (msec per frame) for alternative CPU enhancements 135

CHAPTER I

INTRODUCTION

This dissertation addresses the need to improve the performance of general-purpose computers when executing digital image processing applications. The **hypothesis** advanced by the research reported here for the first time is that **abstractions developed for image algebra and special-purpose image processors can be applied to improve general-purpose computers**. The abstractions suggest improved machine characteristics applicable to many image-processing programs--even those developed using other, more conventional, programming languages. The specific abstractions considered are those included in the image algebras developed by Ritter et al. (1987) at the University of Florida with funding from the Air Force Armament Laboratory (AFATL) and the Defense Advanced Research Projects Agency (DARPA). Ritter et al. incorporated abstract notions of atomic data types and fundamental operations into the AFATL image algebra after evaluating a wide variety of image-processing application programs. The AFATL image algebra was developed to provide a notation for algorithm developers that could more clearly express the transformations developed for these applications. Some of the concepts included in the image algebra were incorporated into special-purpose image-processing computers such as ATCURE (Salinger 1994, Salinger 1995; Salinger and Ackenhusen 1993; Salinger and Hord 1994). The present work demonstrates that concepts developed for the AFATL image algebra can be applied to improve the design of general-purpose computers.

Problem Description

There is a growing demand for general-purpose computers able to execute image-processing application programs efficiently. However, digital image-processing applications exhibit a challenging combination of computational, communications, and storage requirements that often exceed the capabilities of current general-purpose computers. The large amount of information that must be processed, the real-time nature of many of the applications, and the frequent requirement for small processors combine to distinguish this from most other computationally intensive applications. This combination is causing image-processing applications not only to become more significant in themselves but also to compete more vigorously for data processing resources. The research reported here is intended to help implement more cost-effective computers that can satisfy this demand.

The specific application program used to test the hypothesis of this dissertation is a program for compression of video image sequences. Digital transmission of image sequences will soon be ubiquitous in the consumer entertainment and telecommunications industries (Jungen, 1992). Applications include video conferencing, broadcast television, movies-on-demand, games, and education, as well as important military, space exploration, and telemedicine uses. Applications such as these—that capture, store, retrieve, transmit, and display image sequences—are multiplying rapidly. This is because digital video offers many advantages over conventional analog video techniques. These advantages include more efficient and flexible storage and retrieval, more efficient use of available communication bandwidth, flexible enhancement techniques, and reduced sensitivity to noise.

One of the most computationally intense aspects of digital image-sequence storage and transmission can be the image-compression process. Compression reduces the number of bits required to represent the image sequence, greatly increasing the

efficiency of storage and transmission. Programs that encode video sequences are a good choice for testing the hypothesis because they exhibit most of the characteristics of image-processing applications that stress current computing systems. Therefore, computer enhancements that improve image-sequence compression throughput can also improve performance for other image-processing applications.

Various approaches have been developed for compression of images. Their benefits depend partly on the types of images to be compressed. Different imaging modalities have different characteristics that can be exploited in image compression. For example, still panchromatic (i.e., monochrome) images such as those from a satellite, X-rays, and synthetic aperture radar, all have different spatial characteristics. Compression techniques for these images exploit the redundancy of data within the image. Similarly, standard color, multispectral, and hyperspectral images all have different amounts of spectral information for each location in the scene. Standard color images have three color channels, multi-spectral images have five to twenty spectral bands, while hyperspectral images can have over 100 spectral bands recorded for each pixel. Compression techniques for these types of images can exploit the redundancy between the spectral bands in addition to the spatial redundancy in the image. Finally, video image sequences include the time-dependent variations in the scene. Compression techniques for image sequences can exploit the redundancy from the similarity of images at different time steps as well as the spatial and spectral redundancy in the sequence.

The introduction of digital television for broadcast applications is likely to cause the compression scheme used in that application to dominate the field. In the U.S. the standard is likely to be the MPEG standard (MPEG 1994), developed by the Motion Pictures Expert Group and adopted by the American National Standards Institute (ANSI). The MPEG standard was developed for storage, transmission, and display of color video image sequences for various applications. The widespread use of the MPEG standard and other image-compression techniques is limited by, among other factors, the ability to

execute the compression and decompression algorithms on general-purpose microprocessors at standard video frame rates. Currently, special-purpose hardware provides cost-effective encoding and decoding of image sequences in most commercial applications. However, to achieve the broadest use, more cost-effective encoding will be required. It would be much less expensive to use general-purpose processors if they could meet the throughput requirements. This is but one example of the benefits that could accrue if general-purpose computers were more efficient at executing image-processing programs.

The design of computers versatile enough to be efficient for different applications requires difficult tradeoffs. The diversity of applications complicates the design problem. This is true even in the narrow field of image sequence compression. For example, a general-purpose processor for use in personal entertainment systems must be small and lightweight, and consume very little power. In contrast, archival systems can be much larger with much more latency, even though their throughput must be high. The desired reduction in bandwidth for a particular application can vary depending on whether the transmission is over a local connection (e.g., from local disk to a computer) or wide area (e.g., from a Martian tele-robotic rover to an earth station). For example, more efficient use of the available bandwidth benefits transmission via RF broadcast, fiber optic, or coaxial communications links but this gain comes at a cost of additional computational complexity and a larger memory requirement. Similarly, low latency is required for interactive applications such as teleconferencing, but this requirement limits the use of computationally complex and memory-intensive compression techniques. Different tradeoffs must be made for storage applications which can be for long-term archival purposes (years) or for rather short-term (hours or days) video-on-demand. These differences in image-compression system requirements result in different tradeoffs between computational complexity, bandwidth, latency, memory requirements, and image qualities.

Scope and Limitations

The research reported herein falls within the context of innovations to accelerate execution of image-processing applications. Generally applicable improvements are important for several reasons. There are many situations where fast execution is desired without requiring the effort to hand tune software for greatest speed. For example, there are many research projects for medical, industrial, and military applications. Often, a lot of experimentation is required during this research to find an algorithm that solves the problem. If the execution time is long, then alternative algorithm evaluations cannot be as thorough. Another reason generally applicable improvements are needed is that software developers often must be able to deliver their software to run on a variety of machines. When the number of target machines is large the cost to tuning the code for all of them becomes prohibitive. Finally, new processors are being developed and introduced regularly. For many applications it is impractical to require that an application programmer know the specifics of each computer architecture to achieve efficient program execution or to re-tune the algorithm each time a new generation of computers is introduced.

The research to provide generally applicable improvements that will allow image-processing application programs to run faster on general-purpose computers can be divided into three major approaches: more efficient algorithms, improved compilers, and faster computers (Figure 1).

Methods to Achieve Faster Execution of Image Processing Programs

1. More Efficient Image Transform Algorithms
2. Compilers that Produce Object Code that Runs Faster
3. Computers that Run Image Processing Code Faster
   a) Faster Clock Speeds
   b) More Efficient Processors
   c) Parallel Processors

Figure 1. Computing technology for reducing execution times of image-processing applications

A topic of intense research for over 30 years has been the development of more efficient algorithms for performing the transforms that are used in image processing (item 1 in Figure 1). This work is important because it reduces the overall amount of computing that must be done to accomplish each image transform. This, in turn, allows programs to run more efficiently on a wide variety of computers if they use the improved algorithms. On the other hand, **improvements in each algorithm area affect only the applications that use the specific transform for which an improved algorithm has been developed**.

The research reported here does not address the development of new algorithms. Instead, we seek to improve the execution of programs that incorporate current and future algorithms. The algorithms implemented in a previously developed MPEG2 image-compression program were used to represent those common to image-processing applications. One result of the research was to show that the MPEG program is representative in that it includes many of the types of transformations previously found to be common in image-processing application programs.

One way to improve the execution of programs that incorporate current and future algorithms would be to develop improved compilers (item 2 in Figure 1). If a compiler can translate the source code so that it takes better advantage of the available computer

resources, then the programs compiled with that compiler will run faster. Some compiler innovations can improve the execution of programs on a variety of computers while other innovations are applicable only for computers that include specific features that the improvement relies upon. Much research has been done on improving the ability of compilers to analyze and manipulate source code so that it runs more efficiently using the resources available on most computers. However, there has been relatively little research on how to get compilers to use hardware features not necessarily available on all computers (such as specialized or unusual instructions). A contribution of the research reported here was to demonstrate that the ability of a compiler to take advantage of the available hardware resources within a particular computer has a significant impact upon the execution time of an image-processing application program. However, there was no attempt to develop improved compilers.

Another topic of intense research is the improvement of computers so that they can execute the machine code produced by the compilers faster (item 3 in Figure 1). For a general introduction to the field of computer architecture, the reader should refer to one of the many fine textbooks on the subject such as Baer (1980), Almasi and Gottlieb (1989), or Siework, Bell, and Newell (1982).

As indicated in Figure 1, three hardware approaches can be used to increase available computer power. One is to increase the clock rates of existing architectures. Another is to provide more computationally efficient processors. The last approach is to use multiple processors. The problem addressed in the research reported here (as indicated by the bold text in Figure 1) is to identify specific examples of the second and third approaches that provide significant improvements for image-processing performance in general and for MPEG2 encoding in particular. In each case, we sought small changes that produce significant improvement. There were two reasons to adopt this philosophy. First, there are many more opportunities to implement small changes to current computers as opposed to implementing major changes. Small changes can be

adopted by a variety of processors such as the PowerPC and SPARC. Second, for this dissertation we chose to adopt the philosophy used in Reduced Instruction Set Computer (RISC) processor. That is, a small number of simple instructions can be executed fastest. Simpler instructions require less control and hardware allowing higher clock speeds. The net result is faster execution. We therefore sought to identify a small set of changes that provide a significant improvement in execution times.

To reiterate, the hypothesis states that abstractions included in the AFATL image algebra can be used to suggest important characteristics that could improve computer performance. These include support for abstract notions such as images as fundamental data structures, templates as operands, and template operations as fundamental operations. These concepts are explained in greater detail in Chapter II and Chapter VII.

The experiments to test the hypothesis were limited to execution of a program that performs image sequence compression using the MPEG 2 encoding standard. The MPEG2 standard provides a variety of options that allow encoding to be tailored to specific applications with different assumptions about the capabilities of the decoding system. The options provide flexibility in the size of the images, the communication fault tolerance, and the target compression ratio. The computational intensity increases as each additional option is implemented. The research reported here focused on one of the most computationally challenging applications of MPEG, live-broadcast and video-conferencing applications. They are computationally challenging because they require low-latency, high-quality video compression. General-purpose processors that can satisfy the demands of these two application areas are also likely to satisfy the demands of archiving and other MPEG encoding applications.

As mentioned previously, currently available general-purpose computers are not capable of executing MPEG 2 encoding programs in real time for broadcast-quality images. The research reported here identified potential improvements to the design of computers so that they could satisfy this need.

The IBM Power Parallel SP2 computer was chosen to represent general-purpose computers for this research. This choice was deemed valid because its hardware components are the same, or of the same product line, as some of the most popular personal computers and workstations. The Power Parallel 2 CPU used in the SP2 is part of the microprocessor family that includes the Motorola PowerPC used in many personal computers. Its nodes are essentially the same as the popular IBM workstations.

The IBM SP2 can be used to study how to make individual processors more efficient and how to connect a collection of processors so they can efficiently operate in parallel to execute application programs faster. It is a distributed-memory, Multiple-Instruction, Multiple-Data-Stream (MIMD) multi-computer. That is, it includes multiple nodes, each having its own local memory, and each running its own program. Each node of the SP2 includes a Power Parallel 2 central processing unit (CPU), local random access memory (RAM), and a local hard disk drive. The nodes of the SP2 are connected via a communication network arranged so that each node can communicate directly with any other node. The parts of the application program running on different nodes transfer information via messages transmitted across this communication network. The flexibility provided by these features has helped make the SP2 one of the fastest selling scaleable computers, and therefore, representative of high-performance general-purpose computers.

The SP2 provides an excellent platform for investigating both the factors that limit the efficiency of individual processors and the factors that limit how fast programs run on parallel processors. To eliminate or avoid the factors that limit execution time requires exploitation of the fine-grained and coarse-grained parallelism available in the applications. Fine-grained parallelism features refer to the possibility of executing small tasks, such as individual instructions, simultaneously. Exploiting the fine-grained parallelism of an application program depends on the low-level characteristics of a computer.

Low-level characteristics are those that define the individual instructions of the processor, that would allow multiple instructions to run simultaneously, or reduce delays due to conflicts for resources between instructions. The research studied three alternative approaches to improving fine-grained parallelism. These were (1) special-purpose RISC instructions, (2) additional functional units, and (3) single-instruction multiple-data-stream (SIMD) functional units. This study also demonstrated the importance of compiler performance.

In contrast to fine-grained parallelism, coarse-grained parallelism refers to the possibility of executing large tasks, such as subroutines or processes, simultaneously. Exploiting the coarse-grained parallelism of an application program depends on the high-level characteristics of a computer. High-level characteristics of a parallel computer would allow the application to run using multiple nodes more efficiently. The features of the computer that most affect the improvement possible with coarse-grained parallelization include the latency of communications, the connectivity, and mass-storage access characteristics.

Several coarse-grained parallelization approaches were implemented on the SP2 to help identify those features of the computer that limit the possible speedup of the application with this approach. The coarse-grained approaches studied looked at both data-parallel and functional-parallel implementations. *Data-parallel* approaches subdivide the data among nodes, for example by frame, by group of frames, or by areas within a frame. *Functional parallelism* decomposes the problem by its steps so that each node executes a different step or group of steps. For MPEG 2 encoding, some of these steps include motion estimation, Discrete Cosine Transform (DCT), quantization, and run-length encoding.

To study functional parallel implementations, the time required for execution of the program was measured using several levels of granularity for the parallelization and several task distribution approaches. The timing analysis measured the time for

communications, operating system functions, disk access, and calculations. Parallel communication was implemented using the Linda language (Gelernter 1988; Carriero and Gelernter 1989). A contribution of the research was to identify an efficient decomposition of the application for this computer architecture. Features of the Image Processing Subsystem of the ATCURE computer were evaluated to suggest improvements that might significantly improve the parallel execution times. Once the fastest implementation on the SP2 was developed, the timing studies were analyzed to evaluate the likely impact of each of the features suggested by the ATCURE architecture. The analysis identified the types of changes to the communications structure that would most significantly enhance the throughput of the machine for image-processing programs in general.

Contribution of the Dissertation

The hypothesis suggests an approach that can be used when designing new computers. Previous work to improve the performance of general-purpose computers when executing image processing application programs has not taken into consideration the theoretical and practical background from the development of image algebra or the practical experience from the development of special purpose image processing computers. For example, no published articles were found to suggest that subword parallel instruction sets such as MMX or Altivec were derived from evaluation of image algebra, image processing languages, or special purpose image processing computers.

The research reported in this dissertation showed that the development approach suggested by the hypothesis does work, that significant improvements can be made using the theoretical and practical background cited. In particular, four methods for improving the computer architecture of the Power2 processor and the SP2 communication network were identified. These include (1) increasing the number of integer operations that can

be performed simultaneously either through additional integer functional units, improved conversion from integer to floating point numbers, or subword parallelism, (2) adding reduction operations to the architecture such as add-maximum, and multiply-maximum to supplement the already available multiply-accumulate capability, (3) provide for multiple destinations for messages across the communication network, particularly for transfer of large messages such as images, and (4) provide improved data retrieval efficiency when transferring subregions in multi-dimensional arrays such as images.

The research predicts that the same computer architecture changes just suggested would also improve the performance of a wide variety of general purpose computers, not just the Power2 and the SP families. In particular, improvements (1) and (2) are applicable to any general purpose microprocessor, and improvements (3) and (4) are applicable to any distributed memory (i.e., message passing), multiple-instruction, multiple-data-stream computer.

In addition, the research reported here predicts that there are likely to be additional implications for computer architecture improvements that can be derived from image algebra and special purpose image processors. The image algebra literature includes a host of theorems that might suggest ways to improve the efficiency of general purpose computers when executing image processing applications. In particular, the research reported here focused on translation independent templates (explained in Chapter II). Although these are very common, there has also been some very interesting theoretical work showing how to use translation-dependent templates to implement the two dimensional fast Fourier transform. Future work should consider the theoretical and practical aspects of implementing translation dependent templates to determine their implications for computer architecture development.

The remainder of this dissertation is divided into eight chapters. Chapter II provides background information for those who are unfamiliar with image processing in general and MPEG encoding in particular. It also includes definitions of some of the

computer science terms used in later chapters. Chapter III includes a summary of previous work on computers for image processing. It demonstrates the complexity of the problem and that previous researchers have not applied the approach used here for general-purpose computers. It also includes an introduction to the computer architecture of the IBM SP2. An understanding of this architecture is required for the material presented in later chapters. Chapter IV includes a description of the experimental methods and tools used in the research. It also explains the limits imposed upon changes to the MPEG program source code to facilitate the experiments. Chapter V provides detailed descriptions of the experiments that studied the execution of the serial code and the impact of alternative compiler optimization options. Chapter VI provides a detailed description of the experiments that studied the fine-grained architecture implications of the hypothesis. Chapter VII provides similar details about the studies of the coarse-grained architecture. Chapter VIII discusses the implications of the experimental results for developments currently under way to improve several popular microprocessors by providing sub-word parallel processing capabilities. Chapter IX provides a brief summary of the research and its conclusions.

CHAPTER II

BACKGROUND

Image processing is a broad discipline used in many applications. As with any technical discipline, many terms used are unfamiliar to those working in other, even closely related, disciplines. This chapter introduces terminology and concepts used in subsequent descriptions of the experiments and analysis. The material on image-processing terminology is provided for those unfamiliar with the field. The detailed description of MPEG encoding is provided to demonstrate that the operations used by this process are representative of general image processing and, therefore, an MPEG encoding program is a valid benchmark for testing the hypothesis of this dissertation.

Image Processing Terminology

The general topic advanced here is the design of computers efficient at executing image-processing application programs. Images can represent many physical phenomena and have many formats. Most often, they represent a three-dimensional scene as a two-dimensional image. However, some images, such as those from magnetic resonance imaging (MRI) or computerized X-ray tomography (CT), can represent a scene as a three-dimensional image, where each location in the image corresponds to a location in a volume.

A simple monochrome image is usually represented as a two-dimensional array of integers. Each entry in the array represents the intensity of the image for a small area called a *picture element*, *pixel*, or *pel*. In a monochrome image the value for each pixel represents the intensity of the image in the associated location. The pixel intensity,

however, can represent a diverse range of physical phenomena. For example, pixels in camera images represent the optical characteristics of a scene while ultrasonic and x-ray images represent the density of materials.

Color images usually have three values associated with each pixel. There are several standards for representing color. RGB represents color as the intensities for red, green and blue that would have to be combined to get an image that looks like the original to a human eye. Several color representation schemes include two color coordinates and an intensity (Ballard and Brown 1982, 31-34). The IHS system includes an intensity, hue, and saturation. The intensity alone provides a black and white image while the hue and saturation define the color. The National Television Systems Committee (NTSC) standard used in commercial television uses the YIQ basis. The Y value is the weight along a black-white axis, the I value is a weight along a red-cyan axis, and the Q value is a weight along a magenta-green axis. YIQ is a linear transform from RGB as follows:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ 0.60 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Multispectral images are often used to represent the light beyond the visible range observed from a scene. They may have dozens of values associated with each pixel. Hyperspectral images go a step further by representing an entire spectrum for each pixel. They can include hundreds of values to represent the spectrum of each pixel. Multispectral and hyperspectral images are used when the application requires better discrimination between different types of materials in a scene.

While monochrome, color, multispectral, and hyperspectral images each add more information about the optical characteristics in a scene, video and film sequences add the

time dimension to the recorded information. Video and film images are usually monochrome or color. Standard video is recorded at 30 frames per second while film is usually at 24 frames per second.

There are two video display formats, progressive and interlaced (Figure 2). *Progressive* frames have all the rows of each image collected and displayed in sequence. *Interlaced* frames are divided into two fields. The top field includes the odd rows of the image while the bottom field contains the even rows. The fields are collected and displayed 1/60th of a second apart.

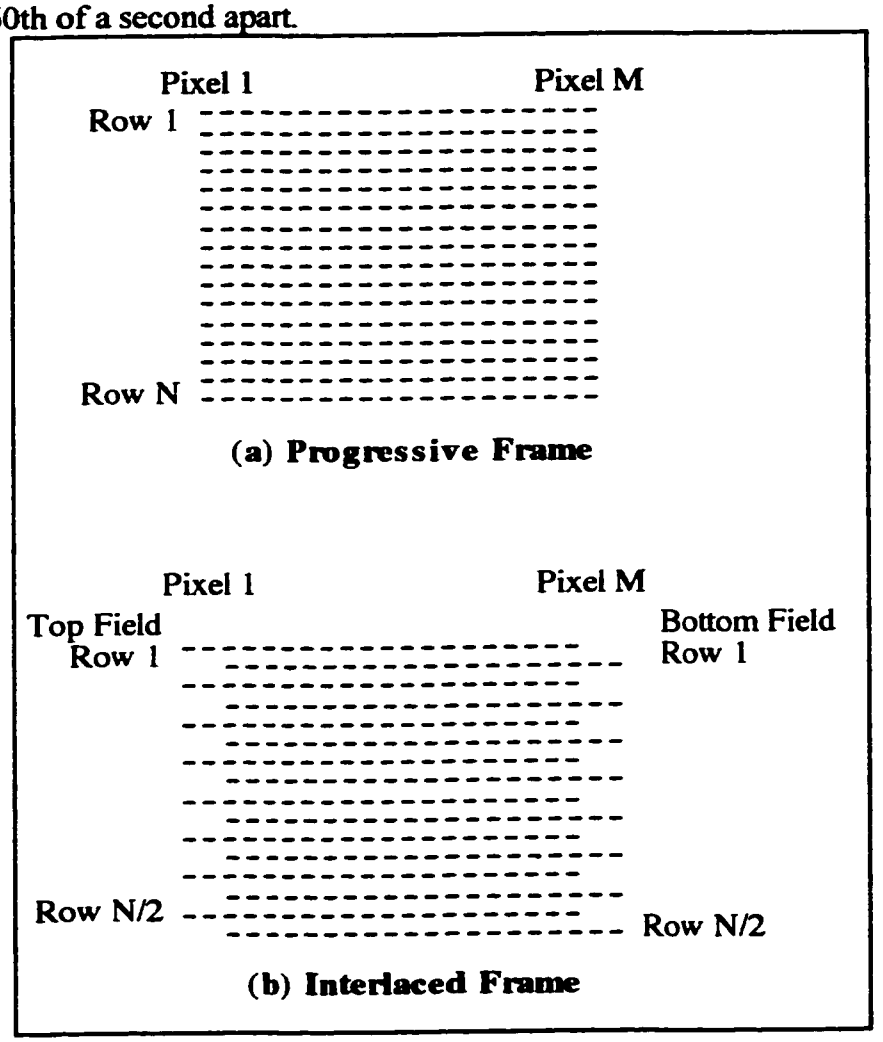

Figure 2. Video frame display formats

Image Processing Operations

Image-processing applications may be divided into three general areas (Table I). *Graphics* applications generally deal with creating images from other types of information such as computer models of objects. The models are usually a stored description of the shapes, sizes, orientations and locations of objects in a scene. *Image transformation* applications generally have images as their input and produce images as their output.[1] For example, in magnetic resonance imaging (MRI) and computed tomography (CT), image processing is used to reconstruct the image from the acquired raw data, usually for eventual display to a physician. Similarly, synthetic aperture radar (SAR) requires an image formation process before it can be interpreted by an analyst (Carrara, Goodman and Majewski 1995). *Interpretation* applications, generally, extract information from images. The input is an image and the result is either a very short description of the relevant features of the image or a decision based upon the interpretation of the image. For example, in target recognition, image processing is used to find objects and determine if they are of a particular class as part of a guidance and control system. Similarly, industrial machine vision systems are usually used to evaluate images automatically for inspection.

Graphics	**Image Transformations**	Interpretation
Rendering Animation	**Enhancement** **Compression** **Reconstruction**	Detection Identification Tracking

Table I. Image-processing applications

---

[1] In this context we use "image" loosely to include data collected from a scene via a sensor that can be used to produce a picture of the scene. This includes, for example, the Fourier transform or projections that include all the information needed to create a picture of the scene.

The research reported here identified enhancements to general-purpose computers that improve their performance while performing image transformations (the bold items in Table I). However, there is enough similarity among all three areas of image processing that computer enhancements to improve execution times for one type of application often can improve the execution times for other applications.

Computer Architecture Implications of Image-Processing Applications

Image processing poses a unique and challenging problem to the design of computers. There are several characteristics of image-processing application programs that, when taken together, make the computational challenge very difficult (Salinger and Ackenhusen 1993). The large number of input and reference data that must be processed, the real-time nature of the problem, and the physically small processors often required, combine to distinguish image processing from other computationally intensive applications.

The large data structures processed in these applications fall into three categories, images, matrixes/vectors, and symbolic information. These distinct data types highlight the heterogeneous nature of many image-processing applications. They suggest that there are four types of functions that a computer must handle well to provide a general platform for image processing. These are:

1. Input/output (of images and reduced data),
2. Image manipulation,
3. Vector and matrix manipulation (signal processing), and
4. Symbolic processing.

A computer often needs to be able to do all four of these types of functions efficiently when executing image-processing application programs.

The three categories of data (images, matrix/vector, and symbolic) have different implications for optimizing a general-purpose computer. They are distinguished by:

1. The way they are usually organized in memory,
2. The operations that are performed on them,
3. The way their values are represented, and
4. The control structures commonly used with them.

For example, images are most often (but not always) represented as integers. They are stored in memory so that adjacent pixel elements are in predictable locations relative to each other. They typically are stored in arrays of from 128x128 to 2kx8k pixels. The operations performed often involve creating new pixel values based on pixels with the same indexes in two other images or based on neighboring pixels in one image. Rarely are random access, shuffling, or sorting performed. Exceptions would include the Cooley-Tukey Fast Fourier Transform and Hough Transform. Table II summarizes the types of transformations often performed on images. The operations performed are often logical operations based on the values of the pixels rather than just mathematical operations.

Unlike images, matrix and vector data are often stored in floating point format. As with images, the storage location corresponds to the index(es) of each element of the matrix or vector. In practical image-processing applications, the dimensions of the matrixes typically range from 1x1 (scalars), to 64x64 (matrixes), or 1024x1 (vectors). In image processing applications, the number of elements in any one matrix or vector is

rarely more than 1024. The operation performed most often on this kind of data is the multiply-accumulate function.

Memory Operations	Neighborhood Operations	Multi-Image Operations	Whole Image Transformations
Subsampling	Convolution	Add	Histogram
Indirect Access	Dilation	Multiply	Run-Length Encoding
Move	Erosion	Subtract	FFT
Interlace	Maximum	Maximum	DCT
	Minimum	Minimum	Hough Transform
	Logical Or	Select	

Table II. Typical image-processing operations

Unlike images, vectors, or matrices, symbolic data are often stored in linked lists. Tree structures, graph structures and semantic nets are often used. The relationships between locations in the image are no longer reflected in the storage locations in memory. The data are no longer iconic (image-like). The data processing develops and interprets a compact description of the scene. The operations often involve matching an unknown graph to a set of models or templates (Barr 1982; Rich 1983). Search functions often dominate the operations on this type of data.

A large body of previous research into general-purpose and scientific computing has addressed the symbolic, signal processing, and the Input/Output (I/O) aspects. It is, however, equally important to examine closely the image-transformation capability of a computer architecture to improve its generality in image processing. Image manipulation tends to be the most computationally intensive, requiring the most accesses to data, with access patterns that are not always linear, such as in neighborhood processing.

Image Algebra

Part of the hypothesis of this dissertation is that abstractions and theoretical developments in the field of image algebra suggest machine characteristics that can improve the performance of computers when executing many image-processing programs. The theoretical developments in image algebra have fostered attempts to define image-processing languages (Ritter 1987; Hatfield, Miner and Wilkes 1991) or a library of subroutines that could be called by image-processing programs. However, no such language has been widely accepted.

The AFATL image algebra was developed after a thorough review of image transformation algorithms used in a variety of military and commercial applications. The algebra defines a set of operations with images and templates as the operands. In the AFATL image algebra, an *image*, A, is defined as a set of coordinates with a value associated with each coordinate,

$$A = \{(x, a(x)): \quad x \in X, \quad a(x) \in F\}.$$

The set X is a subset of an n-dimensional coordinate system, where n is some integer. For example $X \subset \Re^n$ would be a set of coordinates that is a subset of the coordinate system

$$\Re^n = \{(x_1, x_2, \cdots x_n): \quad x_i \in \Re, \quad i = 1, 2, \cdots n\}$$

where $\Re$ is the set of real numbers. Similarly, $X \subset \mathbf{Z}^n$ would be a subset of the coordinates with integer values.

The set F is the set of possible values at each location in the image. The set F can be derived from the sets $\mathbf{Z}$, $\Re$, $\mathbf{C}$, and $\mathbf{Z}_{2^k}$, respectively the set of integers, real numbers, complex numbers, and binary numbers of fixed length k. Thus, the image abstraction in the AFATL image algebra is a mapping from each location to a value, $a: X \rightarrow F$.

Unary and Binary Operations On Images

With this definition of an image, the AFATL image algebra defines any *unary operation* (i.e., having one operand) on an image as a function f that is applied to the value of each pixel of the image. That is

$$f(A) = \{(x, c(x)) : \quad c(x) = f(a(x)), \quad x \in X\}.$$

Thus a trigonometric function applied to an image would be

$$\sin(A) = \{(x, c(x)) : \quad c(x) = \sin(a(x)), \quad x \in X\}.$$

We can also define a set of *binary operations* (i.e., having two images as operands) on images.

$$C = A + B = \{(x, c(x)) : \quad c(x) = a(x) + b(x), \quad x \in X\}$$

$$C = A \times B = \{(x, c(x)) : \quad c(x) = a(x) \times b(x), \quad x \in X\}$$

$$C = A \vee B = \{(x, c(x)) : \quad c(x) = a(x) \vee b(x), \quad x \in X\}$$

$$C = A^B = \left\{(x, c(x)) : \quad c(x) = \begin{cases} a(x)^{b(x)} \text{ if } a(x) \neq 0 \\ 0 \quad \text{otherwise} \end{cases}, \quad x \in X\right\}$$

$$C = \log_A B = \{(x, c(x)) : \quad c(x) = \log_{a(x)} b(x), \quad x \in X\}$$

where $\vee$ stands for the maximum operation. The logarithm function is defined only for images A and B in which $a(x) > 0$ and $b(x) > 0$ for all $x \in X$. One binary operation is distinguished in that its result is a scalar.

$$A \bullet B = \sum_{x \in X} a(x) \times b(x)$$

The *characteristic function* χ is used to implement relationship tests. For example

$$\chi_{\geq B} = \left\{(x, c(x)) : \quad c(x) = \begin{cases} 1, \text{ if } a(x) \geq b(x) \\ 0, \text{ otherwise} \end{cases}, \quad x \in X\right\}.$$

Similar definitions are defined for the other relationship tests $\leq$, $>$, $<$, $=$, and $\neq$.

Template Operations

The template abstraction provided in the AFATL image algebra unifies and generalizes the concepts of templates, masks, windows, and neighborhood functions into one mathematical entity. The definition of a template begins by considering the set of points Y in the result or output image. For each of the points in the output image, $y \in Y$ a template defines a set of points (called a neighborhood) in the source image. In other words, this part of the definition of a template defines a mapping from each location in the output image onto a set of points in the input image,

$$\tau : Y \rightarrow 2^X$$

or

$$\tau(y) \subset X, \quad \forall y \in Y.$$

where 2^X is the power set (set of all subsets) of X, the coordinates in the source image. An example would be the Von-Neuman neighborhood often used in edge detection (Figure 3).

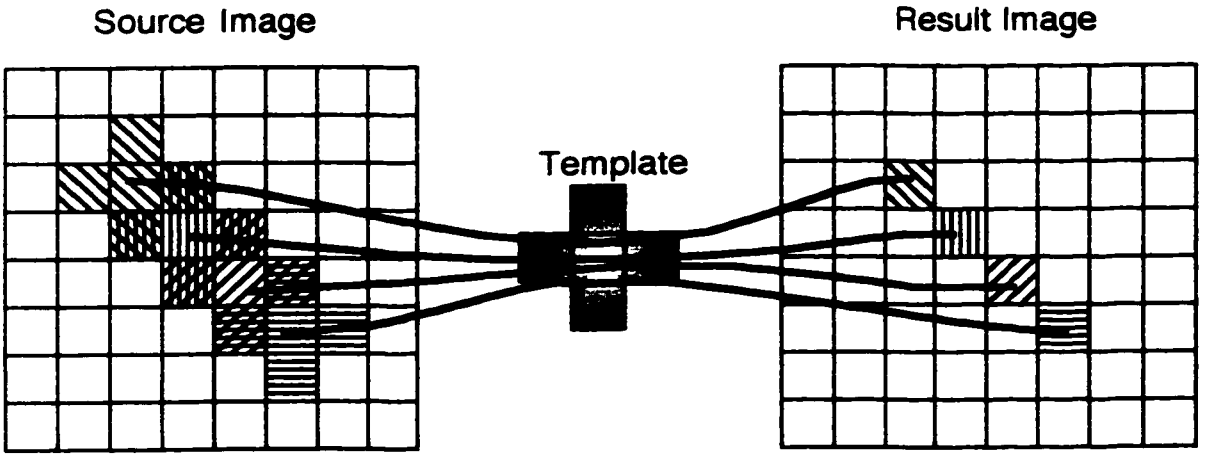

Figure 3. Four pixels in a result image and their associated neighborhoods in a source image

The second part of defining a template is to define a set of weights. The weights assign a value for each location in a neighborhood. So, for each element of the neighborhood $x \in \tau(y)$ there is a value $t_y(x) \in \Re$. Putting the neighborhood definition

and the weights together, the complete definition of a template assigns a set of points in the source image for each point in the result image and for each of these points in the source image the template also assigns a weight.

Note that a template need not have constant configuration or shape. A *position-dependent template* can have a different shaped neighborhood in the source image or a different set of weights associated with each location in the result image. Alternatively, a *data-dependent template* can have weights that are a function of the values in the source image. This versatility permits the definition of templates with the coefficients required to perform complex transformations such as the fast Fourier transform (FFT).

With the above definition of a template, the image algebra defines three template operations,

$$C = A \textcircled{\vee} T = \left\{ (x, c(y)) : \quad c(y) = \bigvee_{x \in \check{t}(y)} \left[a(x) \times t_y(x) \right], \quad y \in Y \right\} \qquad (1)$$

$$C = A \boxed{\vee} T = \left\{ (x, c(y)) : \quad c(y) = \bigvee_{x \in \check{t}(y)} \left[a(x) + t_y(x) \right], \quad y \in Y \right\} \qquad (2)$$

$$C = A \oplus T = \left\{ (x, c(y)) : \quad c(y) = \sum_{x \in \check{t}(y)} \left[a(x) \times t_y(x) \right], \quad y \in Y \right\}. \qquad (3)$$

The template operations define two stages of calculations to produce the value for each pixel in the result image. The first stage is an operation between each pixel in a neighborhood and the associated weight in the template. In Equations 1 and 3 the first operation is multiplication, in Equation 2 the first operation is addition. The second stage is an operation to combine weighted partial results to produce a single value for each pixel in the result image. This operation is referred to as the *reduction operation*. In Equations 1 and 2 the reduction operation is the maximum while in Equation 3 the reduction operation is summation.

A simple example of the use of templates would be to define the Sobel image gradient operator (Ballard and Brown 1982). The Sobel gradient operator uses two

templates; each has the same shape but different weights (Figure 4). The gradient, G, of an image, A, is calculated using the formula in Equation 4.

$$G = \{(A \boxed{V} T_1)^2 + (A \boxed{V} T_2)^2\}^{1/2} \quad (4)$$

T_1

-1	0	1
-2	0	2
-1	0	1

T_2

-1	-2	-1
0	0	0
1	2	1

Figure 4. Sobel image gradient templates

It is important to note that the operations defined in the AFATL image algebra form a complete set of operations, able to express any gray-scale image-to-image transformation (Ritter and Wilson 1987). This suggests that a necessary condition for a computer to efficiently execute image processing application programs is to efficiently execute the operations defined by the image algebra.

Analysis of the research that supports the image algebra developments suggests some characteristics that can be exploited in designing a computer for image processing. These characteristics are different from those that would be required in the design of general-purpose processors for scientific or other applications. These include:

- Images are formed from large numbers of pixels; they are stored as large, very regular data structures.
- Image data can be stored so that they are usually accessed in some predictable order.
- Images are formed from rather low precision data, typically 8 to 12 bits per channel.

- Frequently, similar operations are executed on all the pixels selected from an image.
- The computations are most often integer operations.
- Neighborhood operations are about as common as multi-image operations.
- There are often long sequences of operations that do not depend on global (image-wide) analysis of the data.
- The control structure and data access sequences are very regular. Many of the computationally intense operations performed on images have very predictable patterns for the data access and program flow.
- Many applications require the system to keep up with a data stream. Some applications can tolerate long latencies (e.g., video on demand and archiving) while others require small latency (e.g., video conferencing).

These characteristics are important to consider when looking for ways to execute image-processing programs in parallel. The processor that can take advantage of some of these characteristics will be more efficient when executing image-processing application programs. From the analysis of the research that supported the development of the AFATL image algebra, we can identify the most important characteristics of general purpose processors that should be improved to achieve faster execution of image processing application programs.

- Improve performance when executing integer operations whose operands have modest (8-16 bits) precision.
- Improve performance when executing reduction operations such as multiply-accumulate, add-maximum, and multiply-maximum with integers as operands.
- Improve performance generating addresses to sequence through small regions of very large multi-dimensional arrays of data.
- Reduce data access times for repeated sequencing through large multi-dimensional arrays of data.

- Improve performance when executing short sequences of instructions whose number of repetitions can be calculated beforehand.
- Improve performance when executing the same sequence of instructions for each of many regions in an image where the calculations for each region are independent of the calculations in the other regions.

The experiments and analysis reported in Chapter VI through Chapter VIII look at the potential improvements that could be achieved by following these guidelines.

MPEG Encoding Applications

As will be shown in the rest of this chapter, the MPEG encoding application has many of the attributes of image processing applications listed in the previous section. This makes the MPEG encoding application a reasonable representative for testing the impact of the general-purpose computer-architecture improvements suggested by the image algebra and specialized image processing computers.

Recall that image-compression techniques are used to reduce the amount of data required to represent an image so that it can be stored or transmitted more efficiently. All image-compression techniques look for redundant information in an image and then encode the image so that the redundancy is reduced or eliminated. Whereas the data for an uncompressed image are organized row by row in memory, the data for a compressed image are organized as a bit stream, whose interpretation is much more complex (Figure 5).

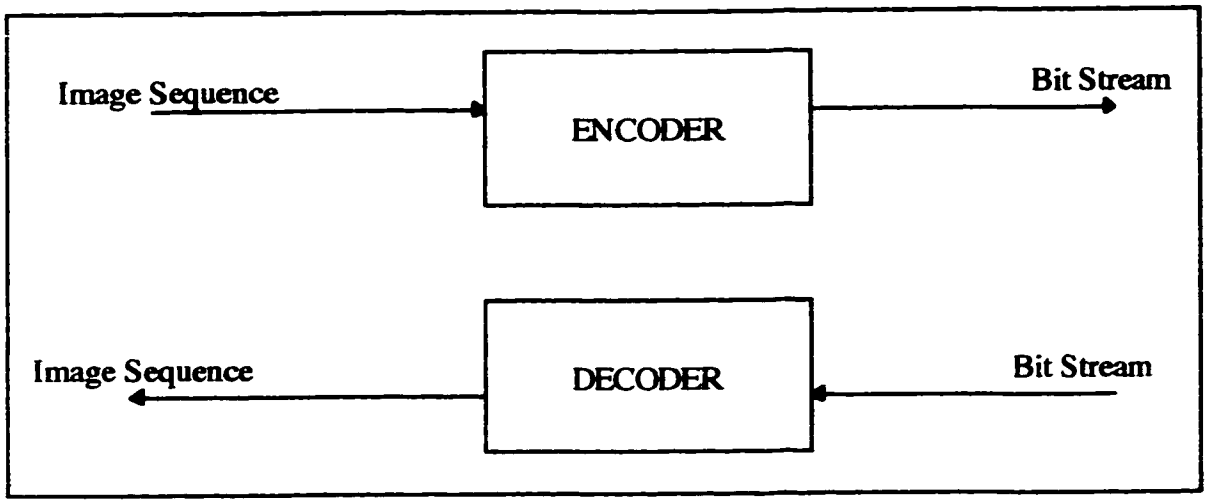

Figure 5. High-level block diagram of image sequence compression and decompression

Image-compression techniques can be lossless or lossy. A *lossless* technique maintains all of the information required to exactly recreate the original image. A *lossy* technique, such as MPEG, cannot exactly recreate the original image. The lossy techniques provide higher compression ratios (the ratio of original data to compressed data) at the expense of image quality.

MPEG encoding (ISO 1992; 1993) is useful for evaluating the performance of a computer on image-processing operations because (1) it is likely to be used in many applications, and (2) it has many of the attributes that make image-processing applications difficult to execute efficiently. It includes intense fixed-point processing, neighborhood operations with very regular access patterns, floating point transforms with irregular access patterns, use of global knowledge to control local decisions, and intensive time-critical input/output operations.

The MPEG standards are emerging as an important mechanism that encourages compatibility between equipment used in a variety of entertainment and communications applications. The standard, ISO/TEC JTC1/SC29/WG11 N0702, specifies features important in capture, storage, retrieval, communication, and playback. The standard permits a wide variety of alternatives in the compression algorithm. These alternatives

permit tradeoffs between image fidelity, compression ratio, computational complexity, and storage requirements. Important features include that it:

- accommodates both interlaced and progressive scan video formats,
- accommodates a wide range of picture qualities and dimensions,
- allows different chrominance sampling formats,
- supports 3:2 pull down to represent ~24 frames per second (fps) film as ~30 fps video,
- allows both constant and variable bit rate compressed data communication channels,
- includes a low delay mode for video-conferencing,
- facilitates random access to support channel acquisition and channel hopping,
- can be used for editing of encoded video, and
- allows fast-forward and fast-reverse playback of recorded bit-streams.

There are two MPEG standards. MPEG-1 is targeted for computers, games, and set-top boxes. It was developed for a video resolution of 352-by-240 at 30 frames per second. The compression standard seeks to provide video quality with a communications bandwidth of 150 kilobytes per second. MPEG-2 is targeted for high-bandwidth broadcast applications, a wider variety of image qualities and sizes, and greater resilience to transmission errors.

MPEG Image Sequences

Figure 6 shows how a movie or video is divided into smaller parts in MPEG encoding. Note that this is how the data is organized as input to the encoder or output from the decoder (the left side of Figure 5). The entire set of images is divided into sequences. The selection of a sequence length depends on the application. The sequence

length affects where a decoder can begin access to a movie or video for implementing features such as channel hopping.

Each sequence of images is broken into groups of pictures. The length of a group of pictures, which can change during a sequence, is chosen during encoding. It affects the ability of a decoder to perform random access within the sequence and to perform reverse playback.

Each group of pictures consists of a set of frames. The size of the frames in a sequence is constant. As mentioned previously, frames can be progressive or interlaced. Interlaced frames have a top field and a bottom field. Each field consists of half the rows in the image; the top field contains the odd rows of the image and the bottom frame contains the even rows. In cameras and displays that use fields, the fields are usually collected and displayed one-half frame-time apart.

Frames or fields are divided into slices, each of which is 16 rows high. A slice does not have to cover the entire width of a frame. In fact, in applications such as multi-media presentations, the slices may cover only part of the entire picture area.

Slices are divided into macroblocks that consist of 16 x 16 pixel areas. Macro-blocks are divided into blocks in different ways depending on the chrominance representation scheme of the sequence. There is always one luminance value per pixel, but pairs of chrominance values (Cr and Cb) may be associated with either 1, 2, or 4 pixels. For many applications, the lower resolution can be used for the chrominance data because the chrominance information in a display does not need to be as dense as the luminance information to provide the desired image quality. So, for example, in the 4:2:0 recording scheme, only one pair of chrominance values is recorded for each 2x2 pixel area.

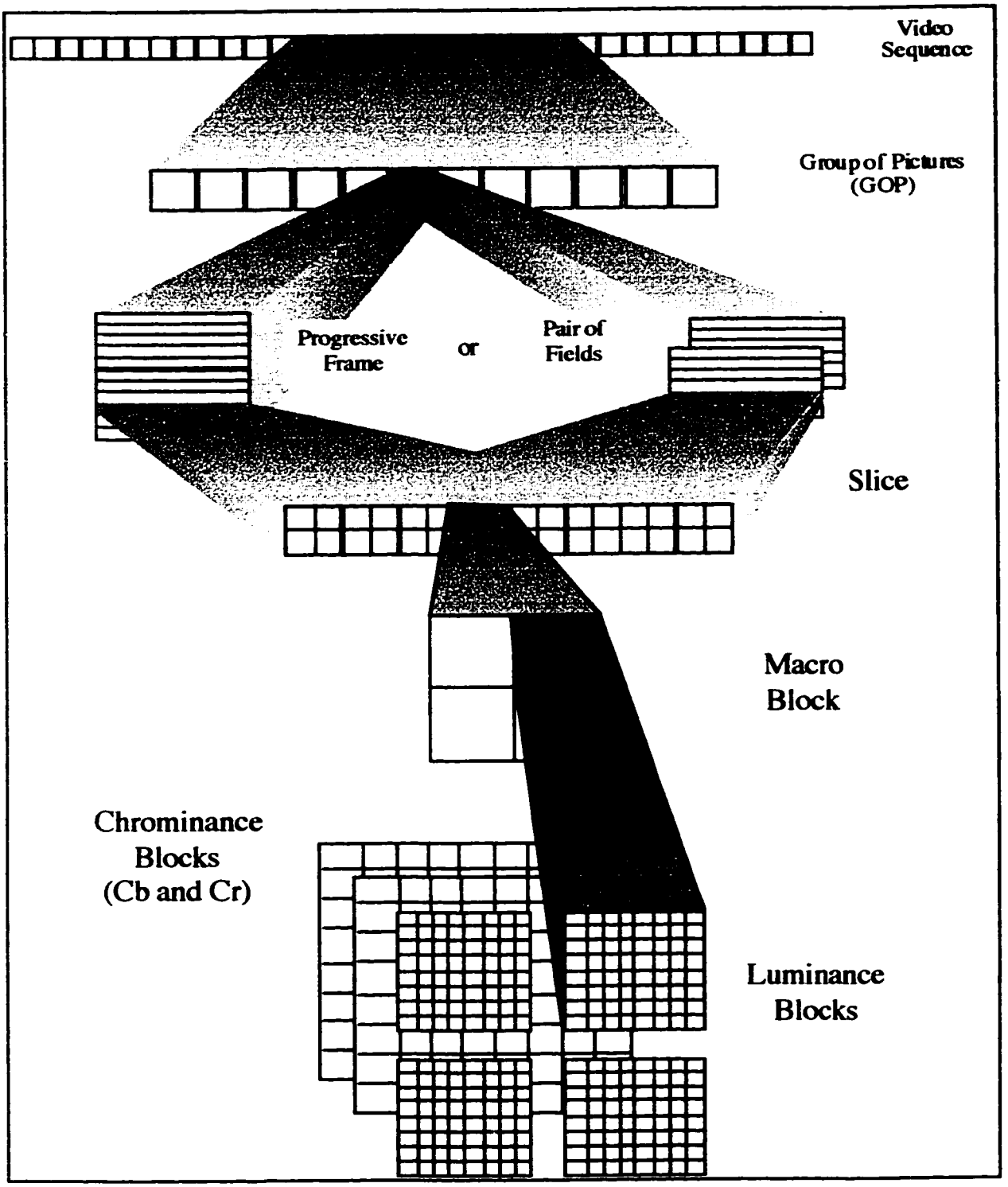

Figure 6. Image sequence decomposition in MPEG encoding

Encoding Overview

MPEG encoding takes advantage of the redundancy between subsequent images in a video or movie sequence to achieve improved compression ratios over approaches that compress frames individually. An MPEG encoded image sequence consists of three types of frames: intra-frames (I-frames), predictive-frames (P-frames), and bi-directional-predictive-frames (B-frames). The difference lies in the way information from other frames is used in the encoding and decoding of each new frame (Figure 7). Intra-frames (frame 1 in Figure 7) do not use any information from other frames.

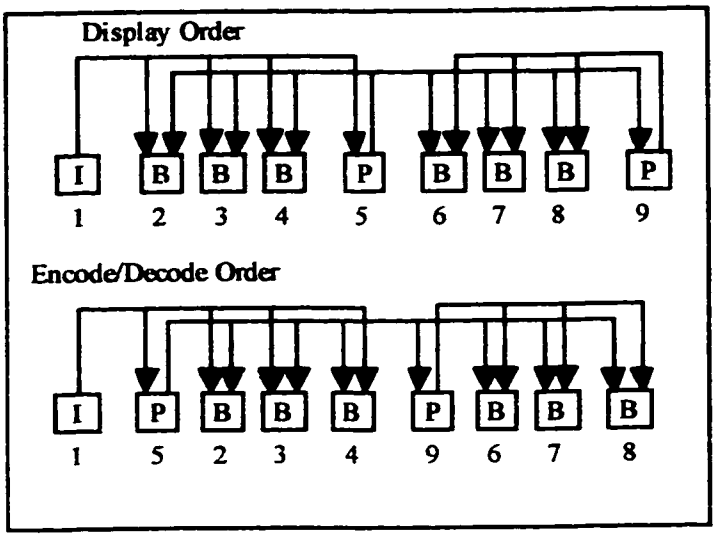

Figure 7. MPEG frame types and processing order: interframes (I), predictive frames (P), and bi-directional-predictive frames (B)

Predictive frames (such as frame 5 in Figure 7) can use information from a previous frame (called the *reference frame*) to reduce redundancy. For each macro-block in the current frame, the encoding method finds a 16 by 16 pixel region in the reference frame similar to the current macro-block. The search for the closest match is called *motion estimation*. The size of the search area is selected by the encoder. The matching can include interpolation to one-half pixel within the reference image. If the two regions are similar enough, the difference can be encoded with fewer bits than the original data.

Since the decoder will have previously decoded the reference block, the decoder can decode the difference data and add them to the reference image data to reconstruct the image.

Bi-directional-predictive frames (frames 2, 3, and 4 in Figure 7) can use information from a previous and a subsequent frame to reduce redundancy. The encoding of this type of frame allows a macro-block to be encoded using an interpolation between a previous frame and a subsequent frame.

As shown in Figure 7, the order of frames in the bit stream is not the same as the display order. The images are reorganized so that a decoder always has previously decoded any frames needed as reference frames. The decoder then reshuffles the images for display purposes. So, as indicated in Figure 7, if frame 1 is an I-frame, then it is encoded first without reference to any other frames. If frame 5 is the first P-frame after frame 1, then it is encoded second using the information from frame 1 to reduce the number of bits that must be transmitted. In this case, frame 5 would be transmitted or stored and subsequently decoded second. In this example, frames 2, 3 and 4 from the display order are B-frames. Since they are encoded and decoded using information from both frames 1 and 5, they are encoded, stored or transmitted, and then decoded in their display order but after frames 1 and 5. This process can then be repeated. The ratio of I-frames to B- and P-frames is determined by the encoder. The ratios in the number of I-, B-, and P-frames have a significant impact on the amount of compression that can be achieved. Fewer I- and P-frames produce higher compression ratios so fewer bits need to be transmitted or stored. On the other hand, if there are more B frames, then more frame storage is required at the decoder, transmission errors propagate to more frames, and rewinding or channel hopping can occur at fewer places.

Figure 8 shows a simplified signal processing flow diagram for a basic MPEG encoding algorithm. The input is digital video organized as shown in Figure 6. Motion estimation and compensation only occur for B- and P-frames. Motion estimation is the

process that searches for the best match for each macroblock between the current image and the reference image(s). The result of motion compensation is a predicted image that is a linear combination of the reference images. The predicted image is subtracted from the current image to produce a prediction error image.

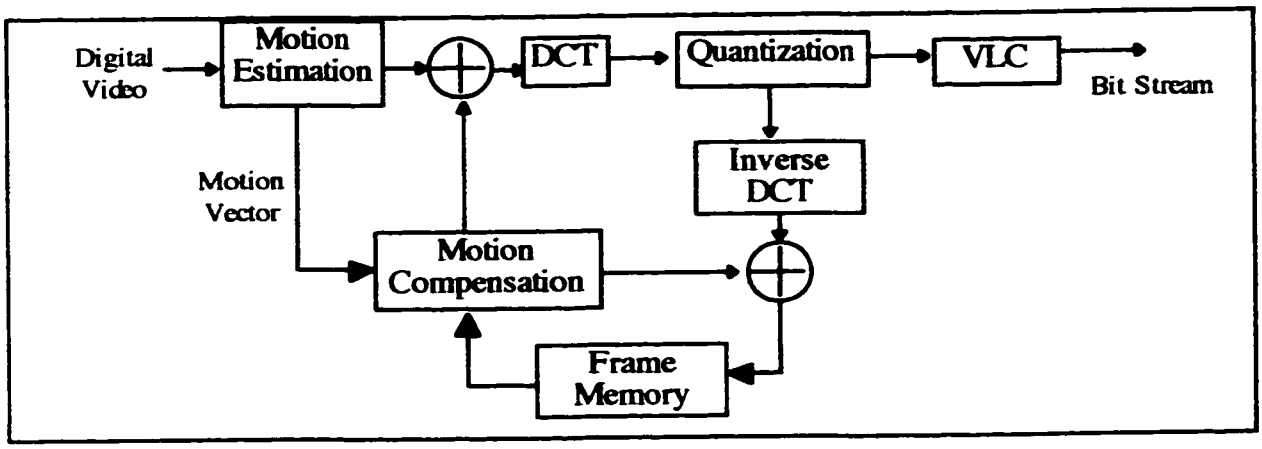

Figure 8. MPEG encoding block diagram

The next steps in MPEG encoding take advantage of the characteristics of the human visual system to reduce the number of bits transmitted by eliminating some information that cannot normally be perceived. As shown in Figure 8, this is done by performing a Discrete Cosine Transform (DCT) on each 8 x 8 pixel block. Up to this point the process is completely reversible. However, the next step, quantization, effectively throws away small coefficients. When the image in the block is fairly smooth or when a reference block closely matches the current block, many of the DCT coefficients will be near zero. A quantization table is used to specify which coefficients are set to zero. Quantization is the step that makes MPEG encoding lossy; the reconstructed image is not exactly the same as the original image.

The variable length coding (VLC) step reduces the number of bits used to represent the quantized DCT coefficients. Variable length coding uses fewer bits to

represent the most common patterns in the data. Progressively more bits are used to represent the less common patterns. A standard codebook is used for the variable length encoding. For example, the most common non-zero DCT coefficient is a 1. Rather than using a whole byte (8 bits) to represent this value, the table designates a 3-bit pattern, 100. Similarly, a string of 3 zeros followed by a 2, which would normally be stored as four 8-bit bytes, is represented by the 9-bit binary pattern 001001100.

```
for each macro block,
 if the frame is B-frame or P-frame, then do motion estimation and
    compensation which includes
    determine the 16 x 16 pixel region in the reference image(s)
       (to 1/2 pixel in each direction) that provides the best
       match in luminance (the L1 norm is usually used).
    Subtract the two macroblocks,
    separate the difference into blocks,
 else
    separate the macroblock into blocks,

 perform the DCT on each block.

quantize the DCT output.

convert each resulting 8 x 8 array into a 64 x 1 array.

variable length encode each 64 x 1 array.

 concatenate the vector length encoded data and the motion
     vectors with appropriate headers to form the bit-stream.

 computer inverse DCT of the quantized coefficients.

 add result to the previous reference frame (using the motion
     vectors for B and P frames) to get a new reference frame.
```

Figure 9. Pseudo-code for MPEG2 encoding algorithm steps

The remaining blocks in Figure 8 (inverse DCT and frame memory) are used to maintain the reference images. The inputs to the frame memory are the images that

would be reconstructed by a decoder. These are used as the reference images when subsequent images are encoded. Use of decoded images as reference frames is necessary to reduce any error buildup that might cause the reconstructed images to diverge from the original images.

The central encoding algorithm represented by the signal flow diagram of Figure 8 can be summarized in Figure 9 using pseudo-code.

Header Information

For a decoder to properly reconstruct an image sequence, it must receive some information about the format of the images. This information is sent in three types of headers, one at the beginning of the entire sequence, one at the start of each group of pictures, and one at the start of each frame.

The header at the start of the bitstream for an image sequence includes the following:

1. Horizontal and vertical size,
2. Frame rate (one of eight standard frame rates), and
3. Quantizer matrix values.

Optional sequence header information includes:

1. profile (one of five alternatives) and level indication (one of four alternatives),
2. progressive sequence flag,
3. chroma format selection (4:2:0, 4:2:2, or 4:4:4),
4. horizontal and vertical size extension (adds 2 more bits to each size dimension),
5. bit rate extension (allows not-standard frame rates).

The header at the start of each Group of Pictures (GOP) contains the following information:

1. time code (hour, minute, second, picture),
2. broken link and closed GOP flags (used in random access and editing).

The header for each picture contains the following information:

1. Temporal reference (picture count modulo 1024),
2. Picture coding type (intra-coded, predictive-coded, or bi-directionally predictive-coded).

Extensions

The MPEG 2 standard defines extensions that provide greater scalability than the MPEG-1 standard. These scalability extensions are for data partitioning, signal-to-noise ratio (SNR) scalability, spatial scalability, and temporal scalability. Combinations of these are supported to produce hybrid scalability.

Data partitioning allows multiple storage or communication channels to be used to increase the bandwidth available for an image sequence. The bitstream is partitioned so that more critical data (headers, motion vectors, DC coefficients) are transmitted using a more reliable channel, and less critical data are transmitted in a less reliable channel. Partitioned data may not be decoded without a decoder intended for decoding partitioned data.

The SNR scalability extension uses a second channel to provide higher precision for the DCT coefficients, thus improving the SNR through the communication channel. Each channel provides the same picture resolution. This would allow smaller displays to

use a reduced complexity decoder, while larger displays would use the entire data set. This extension also provides a mechanism for providing additional chrominance information in the second channel so that a 4:2:0 signal in the lower layer could be enhanced to 4:2:2 or 4:4:4.

The spatial scalability extension provides a base layer with a lower picture resolution and an enhancement layer that provides the corrections to produce a higher resolution picture. During decoding, the base layer is resampled to the resolution of the enhancement layer, and then the two are added. This would allow smaller displays to use a reduced complexity decoder while larger displays would use the entire data set. In a lossy communications channel, the lower resolution image can be sent through the lower error rate channel so that the lower-resolution information can be displayed even when some of the high-resolution information is lost. Also, the enhanced area can be set to cover only part of the entire frame for applications that need high spatial resolution only in specific areas.

Temporal scalability is accomplished by sending additional frames in the enhancement layer. I-frames, P-frames, or B-frames can be included. They must be the same resolution as the base layer and can rely on the base layer or enhancement layer as the basis for predictions.

Computational Complexity of Encoding Algorithms

Normile and Wright (1991) analyzed the computational complexity for several algorithms used in MPEG encoding. They used the Lee algorithm to perform the DCT and IDCT calculations. With this algorithm, one-dimensional 8-point transforms were performed in 64 processor cycles. This was applied to each row and then to each column in each 8 x 8 pixel block for a total or 16 * 64 = 1024 cycles per block. For decoding, the

IDCT result is added to the previous frame using two cycles per pixel or 128 cycles per block.

Normile and Wright found that decoding of raw events into frequency domain blocks (Huffman decoding) and inverse zigzag scanning required one cycle per element to initially clear the block to zero and then 10 cycles per event for inverse quantization and placement into the correct block position. They also found that YUV to RGB conversion required 12 cycles per pixel or 12*64=768 cycles per block.

Akiyama et al. (1994) estimated the computational power necessary for various parts of the MPEG1 and MPEG2 algorithms.

Function	MPEG1	MPEG2
Motion Estimation		109,000
Motion Compensation	75.0	299.8
Prediction	10.2	81.5
DCT, Quantization, Inverse Quantization, Inverse DCT	34.6	138.2
Variable Length Encoding	3.9	15.4

Table III. Estimated computational performance requirement (millions of operations per second), based upon Akiyama (1994)

CHAPTER III

PREVIOUS COMPUTER ARCHITECTURE RESEARCH

Image processing provides a unique combination of characteristics that, taken together, can tax the capabilities of the highest performing computer architectures. Many computer architecture innovations have been tested in the quest for improvements in the efficiency with which computers execute these applications. The variety of architectural approaches previously implemented spans the gamut from hardware designed for specific image-processing operations to architectures that use innovative combinations of general-purpose processors. The following sections demonstrate the diversity of approaches that have been developed and how the current work fits within the general body of research on the efficient use of computers for image processing. The focus is on image processors in general, with an emphasis on hardware that has been used for MPEG encoding.

To provide a framework for studying each of the examples one can consider how well each takes advantage of the exploitable characteristics of image processing mentioned in Chapter II on page 25 and repeated in Table IV. In the following sections, the effectiveness of each design feature intended to improve image-processing application execution times will be explained by how well it exploits the items in this list.

Characteristic	Explanation
Data storage regularity	Images are formed from large numbers of pixels; they are stored as large, very regular data structures. The address of a particular pixel is calculated with a simple formula.
predictable data access patterns	Image data can be stored so that they are usually accessed in some predictable order. The next pixel to access rarely depends upon the results of a calculation using the previous pixel.
low precision data	Images are formed from low-precision fixed-point data—typically 8 to 12 bits per channel.
spatial repetition	Frequently, similar operations are executed on all the pixels selected from an image.
low precision math	Consistent with the low precision of the input data, the computations are mostly integer or fixed-point operations.
neighborhood operation: multi-image operation balance	Neighborhood operations are about as common as multi-image operations.
locality of operations	Neighborhood operations produce each result pixel based upon the values of a small number of associated input pixels. Many multi-image operations produce each result pixel based only upon the values of the corresponding pixels in each of two source images. There are often long sequences of operations that do not depend on global (image-wide) analysis of the data.
predictable instruction sequences	The control structure is very regular. Many of the computationally intense operations performed on images have very predictable patterns for the program flow.
latency	Many applications require the system to keep up with a data stream. Some applications can tolerate long latencies (e.g., video on demand and archiving) while others require small latency (e.g., video conferencing).

Table IV. Exploitable image-processing applications characteristics

Various classification schemes are used to categorize different computer architectures. The classification scheme developed by Flynn (1966, 1972) is used most often. This taxonomy has been enhanced by various authors including Duncan (1990) and Siewiorek, Bell, and Newell (1982). Figure 10 shows the taxonomy that will be used

here. Alternatives further to the left are those approaches that provide greater performance, although at the expense of flexibility. Alternatives further to the right in Figure 10 are those that provide greater flexibility, although at the expense of performance per unit of hardware.

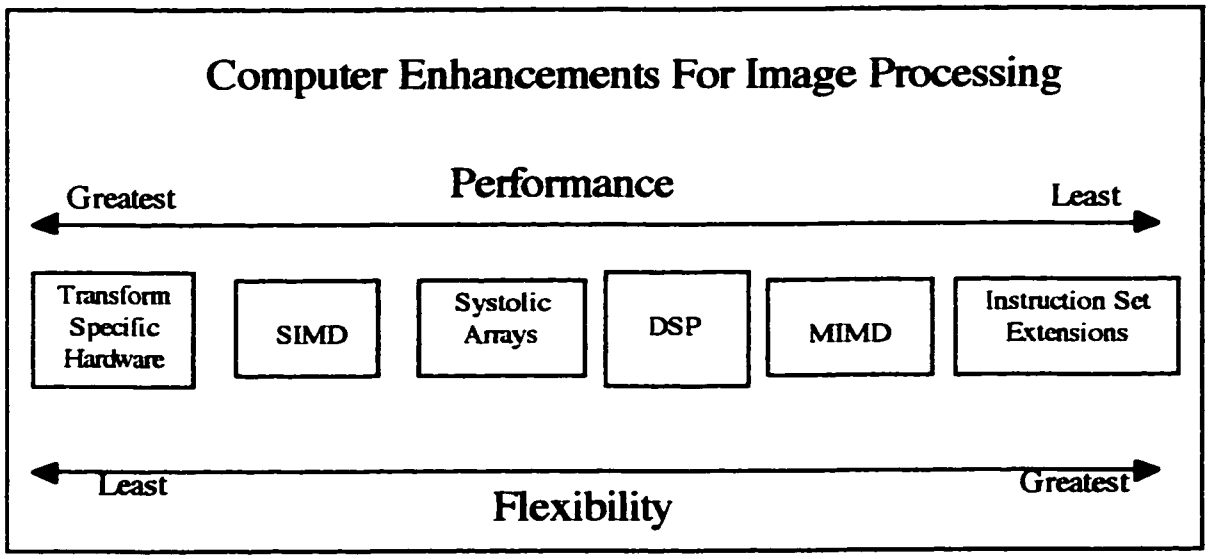

Figure 10. Computer throughput enhancement approaches

Transform Specific Image Processing Hardware

Transform specific hardware implements a specific function or algorithm. As indicated in Figure 10, this approach provides the fastest throughput for the intended application, but at the expense of flexibility. Transform specific hardware is not applicable for programs that do not use the specific operation implemented. This type of processor is most often embedded into a system that is used for a specific application. The processors are often used as accelerators under the control of a general-purpose microprocessor.

Examples of transform specific image processors include the Max-Video circuit-board set from Data Cube (Siegel 1985) and the CRS 1000 (Alsford 1985). Each set

provides circuit boards for operations such as convolution, median filtering, dilation, warping, histogram extraction, or the Hough transform.

Key to each of these circuit-board sets are the use of a frame memory and specialized image data paths. The frame memories contain their own address generation logic so they can be programmed to support image capture and display as well as to efficiently supply data to the other circuit boards for processing. The use of programmable address generators is an example of a way to exploit the predictable data access patterns and data storage regularity characteristics listed in Table IV.

The specialized image data paths used by these circuit-board sets provide very high bandwidth transfers of data between the circuit boards in a system. Typically the data paths allow the circuit boards to be pipelined together so that, as each circuit board produces the results of one calculation, the result is transferred immediately to the next circuit board for execution of the next operation. These data paths are 8 to 16 bits wide, taking advantage of the low data precision characteristic listed in Table IV. The use of multiple transform specific circuit boards connected in combinations to execute complete algorithms is an example of a method to exploit the very predictable instruction sequence characteristic of image processing.

Specialized components and circuit boards for MPEG encoding have also been developed. Matsushita Electric's approach to MPEG encoding and decoding uses multiple specialized chips (Akiyama et al. 1994; Araki et al. 1992; ISSCC 1994; Araki 1994). The chip set executes the encoding algorithm in two parts, one for prediction and motion compensation, and the other for discrete cosine transform, quantization, inverse discrete cosine transforms, and variable-length encoding. The VDSP2 is the special-purpose DSP component designed to perform the first of these two parts of the algorithm. The VDSP was designed for the MPEG2 main profile at the main level for broadcast resolutions. The implementation of MPEG2 encoding exploits the regular control

structure with functional parallelism at the macro-block level and the regular data access sequences with data parallel processing at the block level.

SIMD Image Processing Architectures

As indicated in Figure 10 on page 42, *Single Instruction, Multiple Data Stream* (*SIMD*) computers are not as specialized as transform specific hardware, however they are still very specialized. This approach is particularly applicable to those computations that must be performed on very large data structures such as images. They have many processors connected together in a communications network. Each processor is assigned part of the data. The name, SIMD, derives from the fact that all of the processors execute the same instruction simultaneously, each on its local piece of the data. This makes them applicable for many image-processing functions where the same operation is repeated everywhere in the image. Thus, the SIMD approach primarily focuses on exploiting the spatial repetition characteristic listed in Table IV.

Unger (1958) first suggested a mesh of single-bit processing elements for image processing. Working architectures include the DAP, CLIP4 and CLIP5, MPP, and GAPP (Fontain 1985). The Pixie is a one-dimensional meshed array (Schmitt and Wilson 1988). The GRID (Pass 1985) is a pyramid of processors. Pyramid processors expand the concept of a meshed array to include a heterogeneous hierarchy where the processors in each level can be specialized differently.

These SIMD processors all include an array of extremely simple processing elements. The use of extremely simple processing elements allows each integrated circuit to contain from 64 to 1024 processing elements. Often the extremely simple processing elements in these systems work on only one bit of data at a time. Data with more than one bit of precision must be processed serially. This requires that practically all mathematical operations be divided into many simple instructions.

Each element has local memory and communication with a small set of neighbor processors. Small images are distributed one pixel per processing element. Larger images can be distributed with a patch assigned to each processing element or the image can be broken into tiles with one pixel from each tile stored in each processing element.

Instruction sequencing is controlled by a central instruction control unit. All the processors in the array execute the same instruction simultaneously. There are often control mechanisms to allow a processor to skip execution of a particular instruction based on a locally stored control bit.

SIMD processors are scaleable to some extent. Additional processors increase execution speed up to the point where there is one processor per pixel in the image. Beyond that limit there is no gain. For a given size array, the execution time depends on the length of the algorithm. Since each processing element executes the entire algorithm, if the algorithm becomes more complicated the execution time increases. However, adding more processing elements can keep the execution time constant as the size of the image increases. Functional parallelism can be performed if multiple arrays are used with each array doing part of the algorithm. This is practical only if it is efficient to pass results between arrays. The PREP uses this approach.

One weakness of the meshed arrays is that they are less efficient at extracting summary or global information from an image than performing parallel operations on all the pixels in an image. Finding the average intensity or creating a histogram of the intensities in an image causes part of the processing elements to be idle during the calculation. This is not a significant detriment as long as global summary calculations are only a small part of the program execution time, as indicated by the locality of operations property in Table IV. SIMD machines also have trouble moving the image into and out of the array. The arrays of one-bit processors, in particular, must have logic at the boundary of the array to convert data from parallel to serial representation. Another common problem is inadequate local storage. There is not enough storage for

temporary images. Lookup tables to accelerate complex calculations are also difficult, partly because of lack of local storage for the tables and often because all addresses must be supplied by the control unit.

Another difficulty with SIMD meshed-array processors is incorporating fault tolerance. If one processing element is faulty, an entire column or row is often disabled so that the array maintains its regularity. Some systems incorporate redundant processing elements for each location to overcome this problem.

Systolic Array Processors for Image Processing

As suggested in Figure 10 on page 42, systolic arrays have greater flexibility but lower performance than SIMD processors. While SIMD computers divide the data among many processors that all execute the same instructions, systolic arrays divide the instructions among many processors. Each processor is assigned part of the algorithm. Then the data are passed from processor to processor. Each processor does its part of the algorithm and then passes the results to the next processor. Typical systolic arrays (sometimes referred to as pipelined processors) are the Cyto-HSS (Lougheed 1985), DIP-1 (Duin and Gerritsen 1985), and Warp (Webb and Kanade 1986).

As with SIMD, systolic arrays are valuable when the same instruction sequence must be executed many times on a large data set. The difference lies in the scalability. For SIMD, if the data set gets larger it is possible to keep the execution time constant by adding processors. However, with SIMD, if the algorithm gets more complex and takes more steps then the execution time will get longer. In contrast, with systolic arrays, if the data set gets larger, the execution time gets longer. However, if the algorithm gets more complex so that it takes more steps, then it is possible to add more processors to keep the execution time constant.

Systolic arrays are scaleable; additional processors can reduce the execution time down to a limit caused by the number of sequential instructions in the algorithm. Once the data must be removed from the pipeline for analysis, the remaining processing elements are idle. For a given processor configuration and algorithm the processing time is a function of the image size. Larger images take longer to pass through the pipeline. However, if the algorithm becomes more complicated, additional processing elements can be added.

ATCURE

ATCURE (Salinger and Ackenhusen 1993; Salinger and Hord 1995; Salinger 1994; Adams 1991) provides a good illustration of the features of systolic array processors for image processing. ATCURE was used in the research reported here to test the hypothesis that features derived from image algebra and implemented in specialized image processors can be applied effectively in general-purpose processors. ATCURE was chosen because it was developed using image algebra abstractions to guide the design and because of our familiarity with the system.

ATCURE is the fourth generation of a series of synchronous pipelined processors specifically designed for image processing. While the first three generations, the Cytocomputer, Cyto II, and Cyto-HSS, were designed specifically to support mathematical morphology, ATCURE generalized the image-processing pipeline approach by expanding the instruction set of each processing element and introducing the braided pipeline bus structure.

ATCURE implements a modular heterogeneous processor architecture to provide exceptional versatility and performance in a small package. To provide this versatility the system uses an open architecture that incorporates carefully selected modularity and

programmability features. The open architecture includes use of standard busses, operating systems, and interfaces.

The functionality of the system is divided into modules that can be duplicated or removed as the application requires. As new processors become available, or if special-purpose processors are developed to accelerate specific types of operations, they can be incorporated into the system.

The ATCURE architecture includes four distinct subsystems; one each specialized for processing of images, numeric data, symbolic data, and external communications. Figure 11 depicts how the subsystems would relate in a typical system. The subsystems reflect the four types of functions a computer must provide for general-purpose image processing listed on page 18. Each subsystem includes processors that exploit the characteristics of one of these major application program domains. Each is optimized for the data types, control structures, and operations mix typical of that domain; each is configurable to include parallel processors connected via local busses.

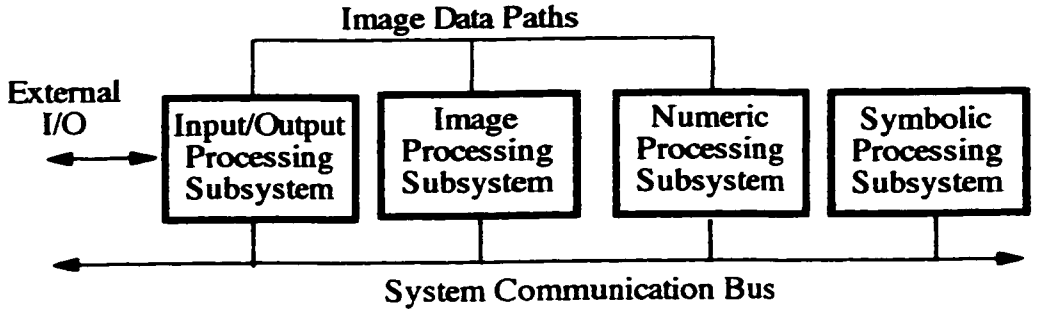

Figure 11. ATCURE subsystems and communication structure

The image-processing subsystem consists of a pipeline of elements operating as a systolic processor. The numeric and symbolic processing subsystems consist of MIMD configurations of commercially available components for digital signal processing and symbolic processing respectively. The external communication subsystem includes general-purpose processors to control specialized interface circuits for sensors, displays,

and communications with other systems. A general-purpose bus and special image busses provide communication paths between the subsystems. This use of separate busses within and between the subsystems allows feedback and feed forward interactions among the diverse processors.

Image Processing Subsystem Structure

A block diagram of the ATCURE Image Processing Subsystem (IPS) is shown in Figure 12. An important characteristic of the IPS is that it treats images as operands, either whole images or selected regions within images. It efficiently performs the four types of operations listed in Table II on page 20 using a programmable pipeline that exploits most of the characteristics listed in Table IV on page 41.

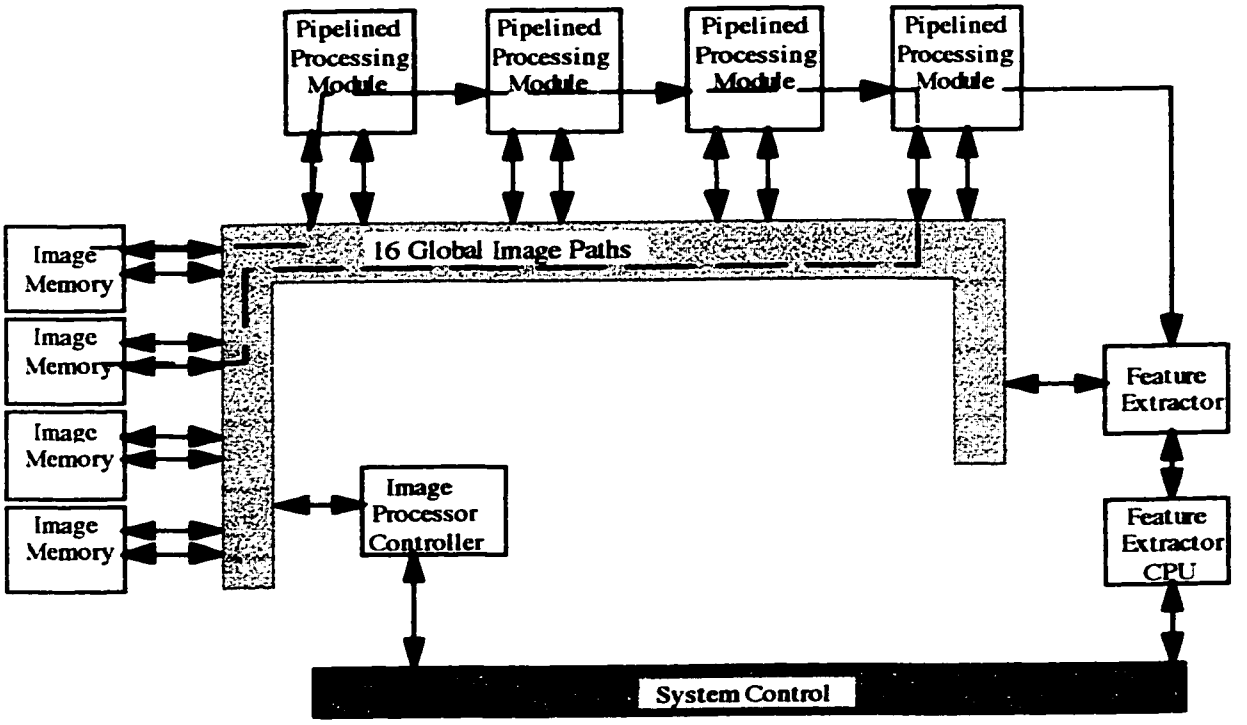

Figure 12. ATCURE Image Processing Subsystem block diagram

To perform most transformations the IPS circulates images via a complex of image data paths that run between Image Memories, Pipelined Processing Modules, and a Feature Extractor. To perform a circulation the Image Processor Controller loads the control registers of each of these elements with the instruction it is to execute. The sequence of instructions executed in one circulation can include a combination of memory, neighborhood, multi-image, and whole-image operations. After the instructions for one circulation are loaded, the designated images are passed from the image memories, through the pipelined processing modules and feature extractor, back to the image memories. The dashed line in Figure 12 indicates one possible path that image data could take. Once a circulation is complete the instruction load and execution cycle repeats with the next set of transformations required for the application program.

The Intelligent Image Memories support memory operations such as those listed in the first column of Table II. These memory operations move data from one location to another but do no arithmetic operations. An example would be to convert a frame of data from interlaced to progressive format.

The image memories also act as the source and destination for images used in neighborhood, multi-image, and whole-image transformations. For all of these operations large sets of pixels from each image must be retrieved, operated upon, and stored. Often a window within the images is to be operated upon rather than the entire image. Usually the order in which the data is to be retrieved is known a priori. The sequence of operations performed on each pixel retrieved is identical.

The image memories exploit the orderly nature of image read and write operations. Each image memory can store or retrieve two images simultaneously. They support various scan patterns, lookup-table operations and indirect addressing. The scan patterns are formed by using a nested loop of seven counters. This facilitates scanning of images in a raster fashion row-wise or column-wise, scanning of a window within an image, and subsampling of an image. Indirect addressing uses the content of an image

memory as the sequence of addresses to access other data in the image memories. This can be used as a lookup table where the value of each pixel is used to retrieve a different value from a table. It can also be used to facilitate complex access patterns as might be used in image warping.

ATCURE Pipelined Processing Modules efficiently support the neighborhood operations and multi-image operations listed in Table II. Neighborhood operations, such as two-dimensional convolution, dilation, and erosion, typically produce one result pixel for each pixel in the source image. The result pixel is a function of the values in a small region (a neighborhood) from the source image. The location of the neighborhood in the source image used to compute each result pixel depends upon the location of the result in the image. Neighborhood operations require the definition of a template that defines the shape of the neighborhood and a set of weights that are used to combine the values of the pixels in each neighborhood to produce each result pixel. These are implemented as multiply-accumulate or add-maximum operations over the neighbors.

Each Pipelined Processing Module contains several Pipelined Processing Elements (PPEs). The PPE (Figure 13) supports linear and non-linear (morphological) neighborhood operations in the neighborhood logic unit. As image data streams into the PPE, line delays provide the temporary storage to provide up to nine pixels of data to the neighborhood logic at the beginning of each clock cycle. The neighborhood logic reconstructs 3 x 3, 9x1 or hexagonal neighborhoods for each pixel. It then applies the set of programmable weights to the pixels in the neighborhood. The weighting operation can be to add the weights to the pixel values or to multiply them by the pixel values. A single pixel result is then computed by finding the maximum of the partial results or the sum of the partial results. The weights are signed integers. A neighborhood mask is also used to disable specific pixels from the neighborhood.

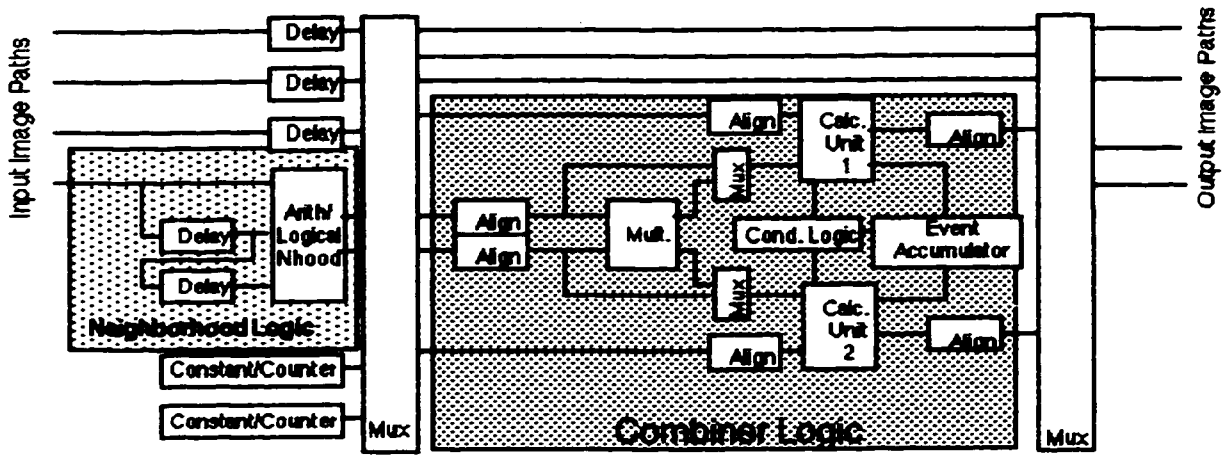

Figure 13. Block diagram of the ATCURE Pipelined Processor Element

Multi-image operations are supported in the image combiner logic. Examples of multi-image operations include addition, multiplication, and maximum. These operations are performed on corresponding pixels of different images. To perform these operations each pipelined processing element includes a multiplier, two arithmetic logic units and auxiliary functions. These support multiplication, addition, subtraction, maximum and minimum between the images. Shifters and bit mask functions support extended precision arithmetic. In addition, the results from one of the ALUs can be used to determine which of two operations are performed in the second ALU. This helps support data dependent computations.

Multiple PPEs can be programmed to implement neighborhood operations for neighborhoods larger than 3x3. Each PPE can calculate the contribution for one 3x3 neighborhood and use its image combiner to merge that result with partial results computed in another PPE. The result is that computations on neighborhoods of any size may be performed efficiently in the pipeline.

The image combiner logic also includes several counters and an accumulator. The counters can be used to generate special images in the pipeline. These include images filled with a constant and various ramp images. The accumulator can be used to

calculate global statistics such as the number of pixels that satisfy some criteria, or find the sum or maximum of the pixel values in a particular region of the image. The output of the accumulator is also used in region labeling, run-length encoding and run-length decoding.

The image-processing subsystem also includes a Feature Extractor. This module can transform an image into a histogram, feature list or coordinate list. The histogram extraction function computes the histogram of the intensities of pixels in one image. The pixels selected for contribution to the histogram are selected according to a one bit mask image. The feature and coordinate lists can include 64 bits of data per entry. A CPU on the Feature Extractor can be used to compute indirect results from the histogram or feature lists. Moment calculations, peripheral length, and similar functions can be computed using this processor.

The ATCURE Image Processing Subsystem (IPS) is a good example of how a processor can exploit the characteristics of image processing listed in Table IV on page 41. It exploits the characteristics of the control mechanisms and operations typical of this domain to provide exceptional performance.

The ATCURE also has some of the shortcomings of other Systolic Array Processors. The useful length of the pipeline is limited by the number of instructions between operations that depend on global image characteristics. It has only limited position-dependent operation capabilities. These require that operations are performed on the entire image with mask images used to select the results.

ATCURE and similar processors require that the data go into each processor in the same order that they came from the image memory or previous processing element. The pipeline approach can also be implemented so that entire images are stored between each processing element. This facilitates algorithms such as the Cooley-Tukey FFT which cannot be implemented as efficiently if only sequential access is available.

Some of the characteristics of ATCURE that could be considered for incorporation into general-purpose multi-computers include (a) hardware to support systolic operation, (b) address generators, such as those in the image memories, with multiple nested loops that support various scan patterns for sequencing through images, (c)simultaneous transfer of two or more images between pairs of processors, (d) multiple destinations when communicating image data, and (e) local reconstruction of neighborhood data from an image data stream, similar to that in the pipelined processor elements.

Arrays of Digital Signal Processors

As suggested by the framework in Figure 10 on page 42, arrays of Digital Signal Processors provide greater generality than systolic arrays while still providing more specialization than the general purpose processors often found in multiple instruction, multiple data stream (MIMD) machines. They are an attempt to provide a balance between the generality of the MIMD machines and the specialization of the other alternatives. *Digital signal processors* are optimized for vector and matrix operations that are dominated by multiply-accumulate computations. They often use a Harvard architecture in which there are separate memories for data and instructions that can be accessed simultaneously. They are optimized for signal processing and control system applications, but have also been applied with some success to image processing.

Several groups have studied the implementation of MPEG encoding algorithms on arrays of digital signal processors. At Apple Computer, Normile and Wright (1991) implemented a homogeneous array of DSP processors as a testbed to examine the interactive use of image sequence compression techniques prior to implementing special-purpose VLSI components. Each node (Figure 14) in the testbed included a DSP processor, local memory, dual port memory and an arbitration mechanism. The nodes

communicated with a host via the dual port memory and with each other via a shared memory (Figure 15).

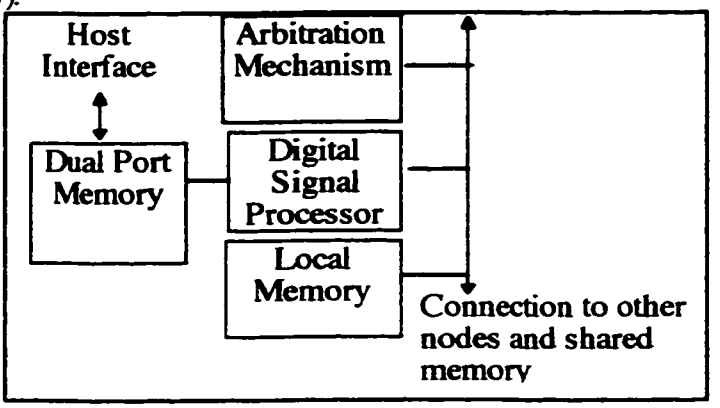

Figure 14. Node architecture from Normile and Wright (1991)

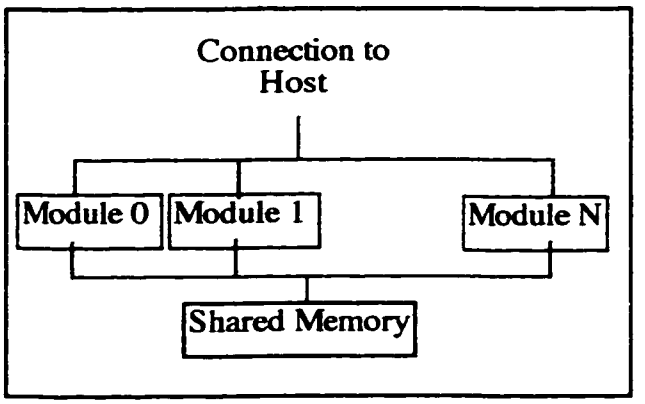

Figure 15. Parallel processor architecture from Normile and Wright (1991)

Normile and Wright studied implementations of encoders and decoders that complied with the JPEG, CCITT H.261 and MPEG-1 standards and reported results for the H.261 standard, the most mature at the time of their work. They measured the speedup and load balance achieved with up to four DSP processors. With optimized I/O software they were able to achieve 35 frames per second on 160x112 motion compensated color imagery with 4 processing elements. They achieved only 9 frames per second for 320 x 240 images.

Normile and Wright implemented data parallelism by dividing the image into stripes processed by separate processing elements. Even so, motion compensation

required data to be shared between the stripes. Their implementations used the global shared memory to provide all the processors with access to the complete image. This greatly increases bus traffic on the global bus.

They also observed that it is difficult to decompose the MPEG algorithms so each of the N processors does the complete algorithm in one Nth of the serial execution time. This is partly due to the use of temporal prediction (i.e., motion estimation) which requires a dependence between frames, making it impossible to decouple processes at the frame level.

Yamauchi et al. (1992) described a single chip Video DSP that uses a VLIW instruction architecture to control 4 processing elements. Operating at up to 25 MHz the processor, called the IDSP, has a peak performance of 300 MOPS. The IDSP includes four digital processing units (DPUs). Each DPU has a three-stage pipeline consisting of an ALU, a multiplier, and an accumulator. This configuration allows up to 12 operations per clock cycle. They describe a system that uses four of the chips to perform MPEG encoding. They reported that a CCITT H.261 (MPEG1) coding function for 64 kb/s video can be implemented to attain up to 10 frames per second at 20 MHz and 12.5 frames per second when operating at 25 MHz when compressing images of 352 by 288 pixels.

To control all the functionality within the IDSP (multiple data paths, memories, and processing elements), an innovative multi-tiered instruction fetch and decode approach was implemented. The component has a local program memory for each DPU to supplement the main program memory. Main program memory instructions can be variable length, including multiple "setup" instructions followed by a single "execute" instruction. The local program memories include two banks of sixteen 32-bit instructions used in vector operations. A main program "execute" instruction can contain a field that specifies which instruction in the local program memories of the DPUs is to be used. Thus the local program memories act to extend the instruction length of the

main program instructions. By providing a very short local program memory, the processor introduces an innovative adaptive SIMD approach in which there are separate stopping conditions and data address generation for each DPU.

The IDSP processor includes features particularly useful when executing compression algorithms. For example, the memory banks have a mode that permits more efficient execution of the area search operations used in motion estimation. In this mode, data are simultaneously read from the memory banks to the PEs and shifted within the memory banks in preparation for the calculations at the next location. This feature exploits both the regularity of the data access sequences and the spatial repetition characteristics listed in Table IV on page 41. Another somewhat specialized feature is a set of connections between the DPUs. These allow various pipeline configurations between the DPUs to facilitate butterfly operations for a DCT or FFT algorithm and the absolute difference operations for motion estimation. Using these connections, the four PEs can execute four real butterfly operations concurrently without multiple reads of the same data elements from the memory.

A complete MPEG system can be implemented using functional or data parallelism. The functional parallel approach is implemented by using one processor (component) for each of the major functions in the encoding algorithm (DCT, motion estimation, loop filtering). This allows the local program memory to suffice for that operation and keeps the data flow between processing elements at a reasonable level.

More recently, Cheng et al. (1994) proposed a video compression testbed using an array of up to 20 DSP modules. It includes features for substituting ASIC components for software modules during their development and testing. Cheng et al. modeled the DSP processor and bus performance and projected a maximum MPEG-1 encoding rate of 20 frames per second for 352 x 240 pixel images. The search range was 27 x 27 pixels, using ASICs for real-time DCT/IDCT. For MPEG-2 with larger images (1920 x 2080), they projected encoding less than 2/3 frame per second.

The Multimedia Video Processor (MVP) (Kim et al. 94, Balmer et al. 94) from Texas Instruments (TMS320C80) is an example of a programmable component that has been used for MPEG encoding. It was developed to combine concepts from graphics and digital signal processing and has specialized hardware for use in video applications but it was not developed specifically for video compression.

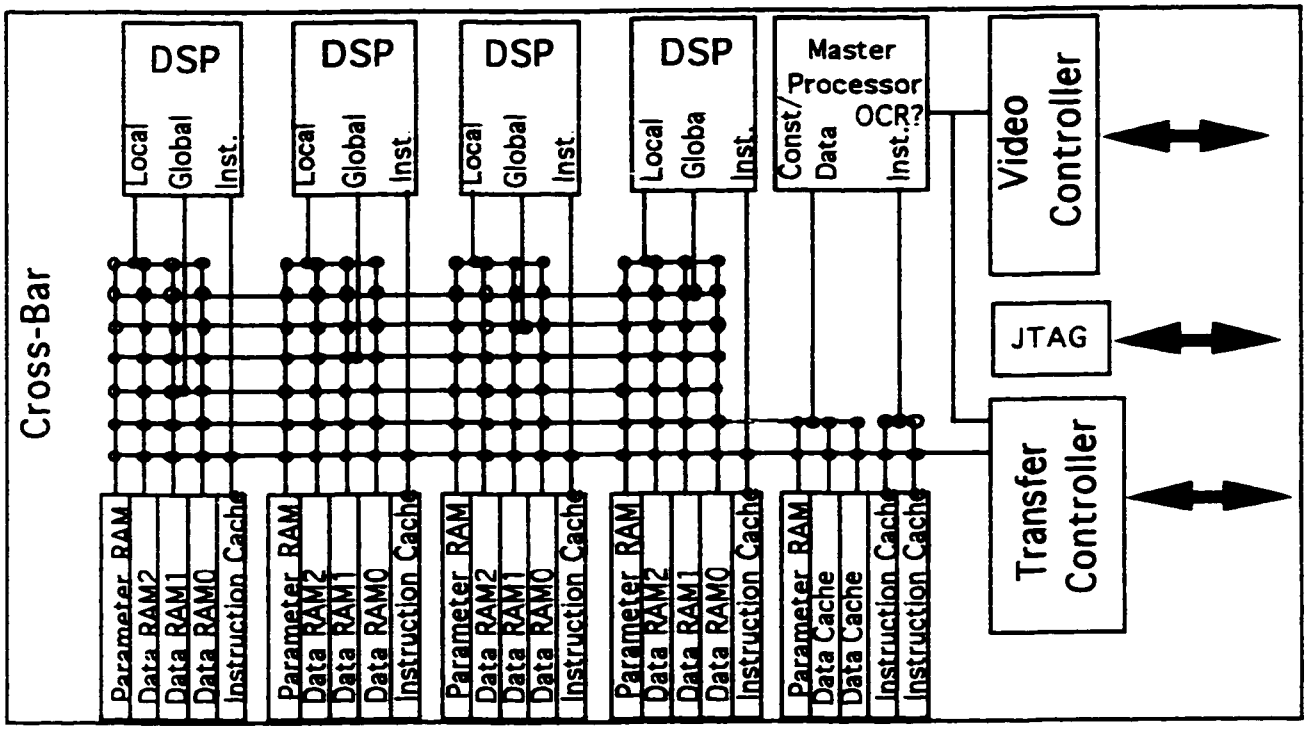

Figure 16. Texas Instruments TMS320C80 Multi-Media Video Processor

Each MVP (Figure 16) component includes four Digital Signal Processors (DSPs), a Master Processor (MP), five banks of SRAM, a Video Controller, a JTAG/Test Port, and a Transfer Controller. The Transfer Controller controls transfers between internal and external memory. It accepts, prioritizes, and executes data transfer requests from all the other processing functions. The video controller provides an interface between the MVP and image capture and/or display systems. It generates timing signals and memory addresses for external memory to control two external frame-

buffers simultaneously. The Master Processor is a general-purpose 100 MFLOPS 32-bit RISC core with an integral floating point ALU that executes the control program.

The DSPs and MP are independently programmable. The DSPs have several features that contribute to their performance when executing image-compression algorithms. They each have two memory address generators so that two memory accesses can occur simultaneously with the arithmetic operations. Each DSP includes eight octal-ported 32-bit data registers. The DSPs include an integer multiplier and an ALU. The multiplier can perform one 16 x 16 multiply or two 8x8 multiplies each clock cycle. A feature particularly important for efficient image processing is that the three-input ALU can perform one 32-bit arithmetic operation, or two 16-bit operations, or four 8-bit operations, or thirty-two 1-bit operations each clock cycle. Also important is the ability to use the third input as a mask for one or both of the other inputs.

The ALU is controlled by an 8-bit function code and arithmetic or Boolean mode selection signal. In addition, extended instructions use the content of Data Register 0 to specify the ALU function, carry-in, and conditional sign control. These extended instructions are intended to help optimize the inner loop of speed-critical operations.

The DSPs execute 64-bit instructions. These superscalar instructions provide three to 15 operations per instruction. Three sets of loop counter registers provide efficient control of the most computationally intensive inner loops commonly found when images or windows within images are used as operands.

Real-time MPEG-1 encoding of 352 x 240 pixel images using a single Texas Instruments TMS320C80 has been demonstrated (Lee et al. 1994, Kim et al. 1994, Balmer et al. 1994). However, only one field of the interlaced images was encoded. The motion compensation search size is not given.

MIMD Architectures

Multiple Instruction, Multiple Data Stream (*MIMD*) computers are composed of multiple nodes, each capable of executing an independent sequence of instructions on data that is accessed by the node independent of all other nodes (Hord 1993). Thus each node of an MIMD computer has its own instruction sequencer and data address generation capability. So, using the framework from Figure 10 on page 42, homogeneous MIMD architectures provide one of the most flexible approaches to high performance image processing discussed here. However, they are not very good at exploiting the characteristics of image processing to achieve their high performance.

The ability to have each node run its own program gives the MIMD machines their flexibility. For low-level image-processing operations, the processors may be distributed over the image (data parallelism), each processor performing similar functions on its part of the image. The processors may also be distributed over the algorithm (functional parallelism), each processor performing a different part of the algorithm. The latter approach is particularly appropriate for high-level tasks. This allows a program running on an MIMD machine to exploit some of the characteristics of image processing. For example, parallel implementations can exploit the local nature of the dependencies. Functional parallel implementations exploit the fact that the same calculations are performed on all of the data and that program sequences are very predictable.

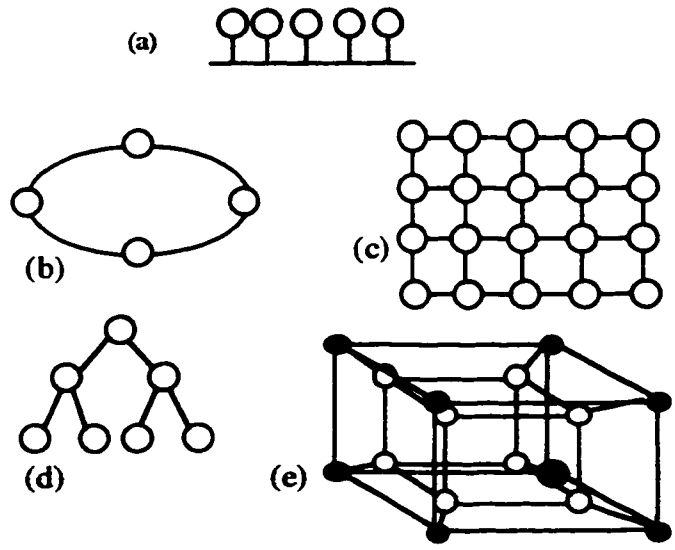

Figure 17. Duncan's MIMD interconnect topologies: (a) bus, (b) ring, (c) mesh, (d) tree, and (e) hypercube.

The important design parameters of an MIMD architecture are (1) the topology of the communications network, (2) the communication protocol or mechanism, (3) the system memory distribution, and (4) the functionality of the processing elements (Duncan 1990). Some topologies include common bus, ring, tree, hypercube, and mesh (Figure 17). The alternatives provide different tradeoffs between expandability of the system and the efficiency of the network.

System memory can be either centralized or distributed. Centralized memory (Figure 18 part a) allows access to the entire memory by each node via a common bus or switching network. In distributed memory machines, a portion of the total memory is closely linked to each node. The centralized memory model provides a simple conceptual framework but has many practical limitations. In particular, the number of nodes is typically limited to around 20 (Agarwala et al. 1995).

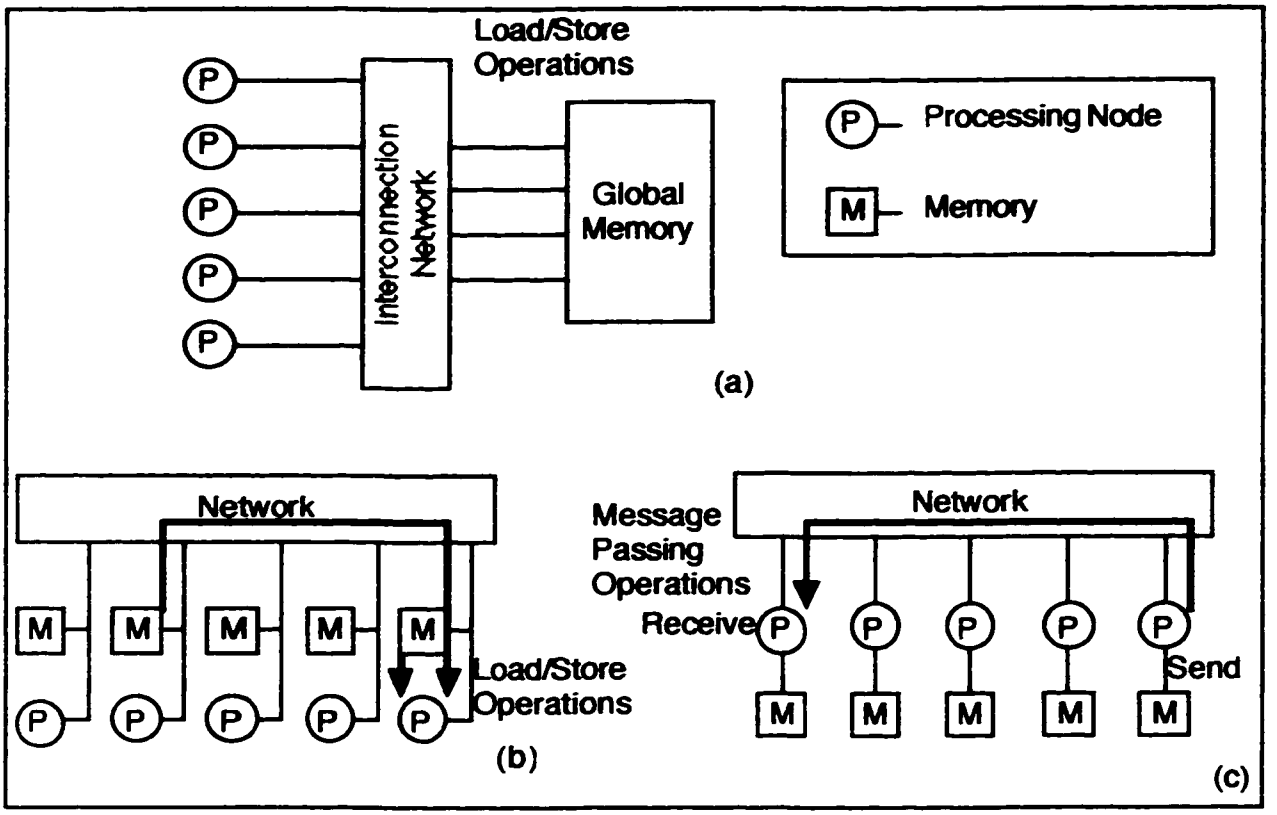

Figure 18. System memory distribution alternatives: (a) centralized, (b) distributed shared, and (c) distributed message passing.

Distributed memory machines can transfer information via message passing or shared memory. In distributed memory message passing machines (also called loosely coupled machines), data are transferred between nodes via messages sent by the processors (Figure 18 part c). This is in contrast to the distributed shared-memory approach (also called tightly coupled) in which each node can directly access data in the memory of other nodes (Figure 18 part b).

One of the key differences between the message passing and shared memory approaches is how the programs on the different nodes are synchronized. Synchronization is required whenever the program running on one node requires data produced by another node. In the message passing approach, synchronization between the parts of the program running on different nodes is inherent. The program can only proceed when the required messages are received. In contrast, on shared memory

machines, programs do not know when data is available unless they explicitly include synchronization instructions that indicate when other nodes have completed their calculations.

Although MIMD provides flexibility, it comes at the expense of size and complexity. Contrast the single bit processing elements of SIMD machines, that can have many processing elements in a single chip, with MIMD machines that typically require one or more circuit boards per node.

Although the network-topology and memory-distribution approaches are important design considerations in MIMD machines, none of the alternatives is particularly favored by the characteristics of image processing listed in Table IV on page 41. The functionality of the processing nodes can be specialized, limited or general. Nodes can be specialized for input/output of data or for image-processing transforms. The most general nodes include large amounts of local memory and mass storage. More limited nodes may have only a small amount of local memory.

Some MIMD machines have been designed specifically for image processing while others, designed for general use, have been used extensively for image processing (Edwards 1985). An advantage of the MIMD processors is their flexibility. They can be used to implement the most complex transformations. Complex operations, like position dependent templates, are readily implemented. They also take advantage of the increasing power of single processor systems, riding that technology curve. Because of their regularity and homogeneity, fault tolerance can be implemented in these processors.

For several reasons MIMD technology is not yet adequate for real-time systems, particularly those that demand a small package. For many of these systems, the number of components in each node is comparable to that of a personal computer. Another limiting factor is that, in shared memory systems, contention for memory and communication resources puts an upper limit on the number of processors that can be used effectively. As more processors are added, transfer of image data among the

processors overloads the communications network. Finally, the functionality in each processing element is not well suited for low-level image-processing operations. Because image transformations often require repeated access to all the data in the images, large percentages of the available instruction cycles are used to access memory.

Yanbin and Anastassiou (1994) studied MPEG-2 encoding by simulating a network of up to 16 workstations using UNIX sockets for communications connected by Ethernet in a LAN. They concluded that slice-based parallelism provides the best balance of communications and memory requirements.

IBM SP2 Architecture

The IBM Power Parallel Systems SP2 (Agerwala, et al. 1995) used in the research reported here can be used to demonstrate many of the advantages and disadvantages of using MIMD machines to execute image-processing application programs. Figure 19 shows the network topology of the SP2. The SP2 is a distributed memory, message passing MIMD architecture. In the SP2 each node is similar to an individual RS6000 workstation, with local memory and hard disk.

The largest SP2 systems contain 512 nodes although they typically have less than 64 nodes. The SP is part of a family of computers starting at the low end with IBM RS6000 workstations, including clusters of workstations in the middle, and ending with SP2 systems with 8 to 512 nodes. They all run the AIX operating system, IBM's version of UNIX. This allows software to be developed on a local workstation and tested on small data sets before it is ported for parallel implementations on the larger configurations for use on larger data sets.

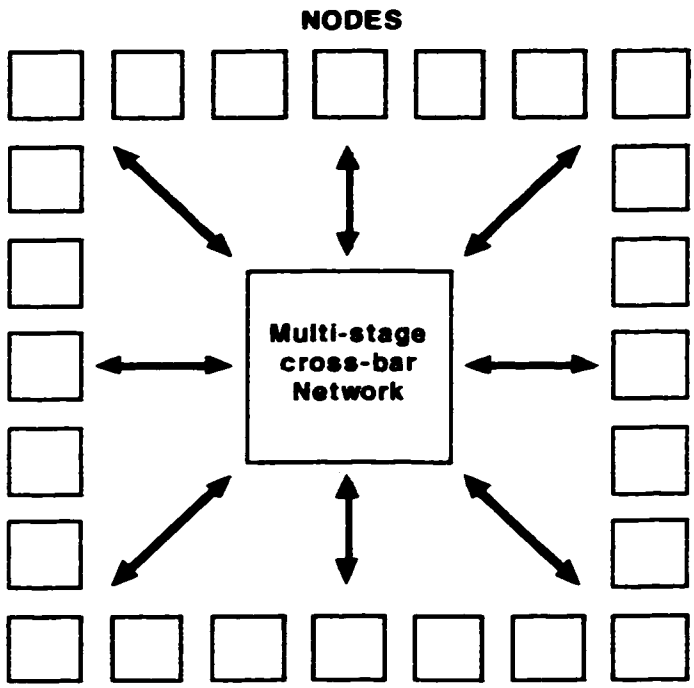

Figure 19. SP2 network topology

Topology and Communication Network of the IBM SP2

As mentioned earlier, the topology of the network in MIMD machines is a major design consideration. The SP2 implements a multi-stage packet-switched cross-bar network that allows any node to send and receive messages directly with any other node (Figure 19). The network is implemented so as to provide four alternative paths between each pair of nodes. This provides redundancy to compensate for most fault conditions. The multi-stage switch approach provides a constant bandwidth of 40 megabytes per second in each direction between nodes, irrespective of where they are in the network. This contrasts with ring, hypercube, and mesh topologies in which the bandwidth between nodes depends on the size of the network or the logical distance between the nodes.

In the SP2 a high performance switch provides a low latency (approximately 40 microseconds from node to node), high bandwidth (40 megabytes per second in each

direction) link to other nodes. The low latency is important in image-processing applications to minimize delays during synchronization between nodes.

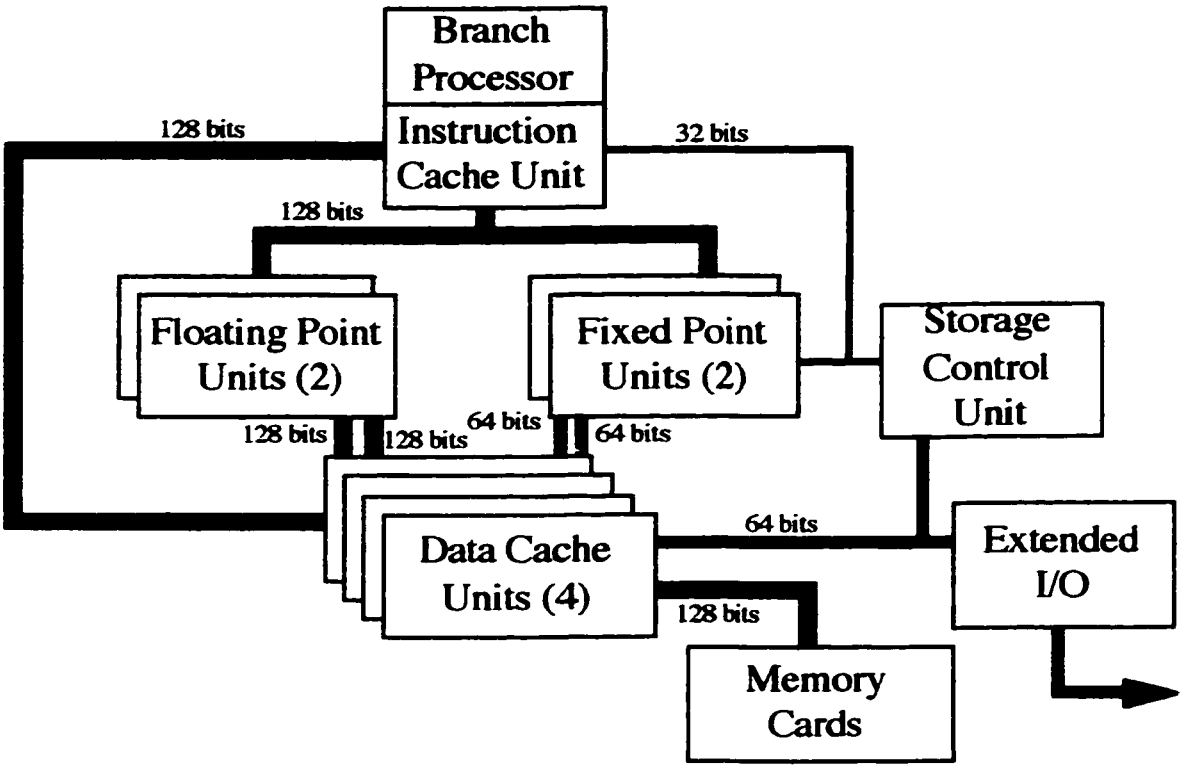

Figure 20. SP2 node CPU architecture

SP2 Processor Nodes

Each node of the SP2 includes the functionality of an RS6000 workstation, consisting of a 66.7 MHz POWER2 CPU, local disk, and an ethernet interface. Added to this is a high speed switch interface. The Power 2 CPU (Figure 20) includes two floating point units, two fixed point units, a branch controller, cache memory, and local main memory. Together they can execute up to five instructions per clock cycle.

The Power 2 implements the RS6000 instruction set architecture (ISA). This ISA is part of the same family of ISAs as the one implemented in the Power PC microprocessor family from Motorola found in the Macintosh and similar personal

computers. The instruction set is a load/store RISC architecture. That is, all memory-access instructions transfer data between memory and the registers in the CPU (Shippy, Griffith, and Braceras 1994). There are no arithmetic or logical instructions that modify memory directly.

The architecture defines thirty-two 32-bit fixed-point and thirty-two 64-bit floating-point registers. The hardware includes extra registers to implement dynamic register renaming (White and Dhawan n.d.). It also includes duplicate register files to provide the number of read ports required to feed the functional units at their maximum throughput.

As with most general-purpose processors, the Power 2 is not well suited for image processing. As mentioned in Chapter 2, the most computationally intense parts of image-processing applications typically involve low precision, fixed-point, data and operations. During these parts of the program, the floating point units of the Power 2 are left idle and the CPU falls far short of its maximum of 5 instructions per clock cycle. Furthermore, in the Power 2 the fixed-point units are used for data address calculation as well as for fixed-point operations on the data. For image processing, this further limits the rate at which fixed-point operations can be performed on the data.

Both the fixed-point and floating-point instructions can include three operands, two sources and a destination. This is in contrast to some other processors (notably those from Intel) that only have two operands, one that is a source and another that is both a source and destination. The three-operand architecture is an advantage for image processing because the operands often need to be accessed repeatedly in different combinations for neighborhood operations. The three-operand approach reduces the need to make copies of data to prevent it from being overwritten.

The floating-point ALUs can each perform dual-operation (multiply-accumulate) instructions without having to store the intermediate results in a register.

Enhanced Instruction Set Architectures

Instruction set extensions are enhancements to general-purpose processors that add instructions specifically intended to improve performance on image-processing and similar applications. These extensions perform, in one instruction, operations that would otherwise require several instructions on a general-purpose processor that does not include the extensions.

Since the computational intensity of decoding is less than that for encoding, this area is sure to be implemented first. Two manufacturers, Sun and Intel, have already enhanced their microprocessors with instructions specifically intended to accelerate decoding of compressed images. As the use of digital video grows, general-purpose processors will need to be adapted to support both encoding and decoding. Chapter 6 includes an evaluation of the use of the Intel MMX instruction set for MPEG2 encoding and identifies enhancements to general-purpose microprocessors and multiprocessors that would make them more effective in this application domain.

CHAPTER IV

EXPERIMENTAL METHODS AND TOOLS

Scope

The research reported here focused on potential improvements to computer architectures for better support of image-processing applications. The research addressed two areas of parallelism, fine-grain instruction-level and coarse-grain multi-computer-level parallelism. As explained in Chapter 1 on page 5, one of the underlying premises of this work is that it is undesirable to require an application programmer to know the specifics of the computer architecture to achieve efficient program execution; such a requirement would eliminate portability. Therefore, it was beyond the scope of this research to improve the code for more efficient execution. With two exceptions, only changes to distribute the execution among multiple processors were made. This restriction retains the character of the program as typical of image-processing applications. The exceptions are explained in Chapter VI and Chapter VII.

The research had three phases: (1) serial program analysis, (2) fine-grain execution efficiency improvements, and (3) coarse-grain implementation efficiency improvements. The serial implementation of the MPEG2 encoding program was studied to determine the program structure and to identify the most computationally intense sections. Experiments were performed to identify bottlenecks at each level while executing the sample program. Then changes of the architecture that might eliminate these bottlenecks were analyzed.

Source Code and Test Images

Source code for an MPEG encoder was acquired from the MPEG Software Simulation Group at their public access internet site (Savation 1998). The files acquired included source code for an MPEG encoder, a decoder, make files, sample command files, three test images, and standard test results. The make files are used with make, a UNIX utility, in the process of compiling and testing the programs. One of the make files had to be modified for compatibility with AIX, IBM's version of the UNIX operating system. The make files also include a test option that compresses and then decompresses the set of three test images. The test includes a comparison of the output from both the compression and decompression operations with standard results. Any discrepancies from the standard results cause a failure message to be produced.

To facilitate a more thorough timing analysis of the program, a longer sequence of images was acquired. A sequence of 150 images was acquired from the University of California at Berkeley (`ftp://ftp.ccu.edu.tu/pub3/epoch.cs.berkeley.edu/mpeg/EncodeData`).

The sequence is a video showing a flower garden, trees, and a building as seen from a moving camera. Figure 21 shows several of the images in the sequence.

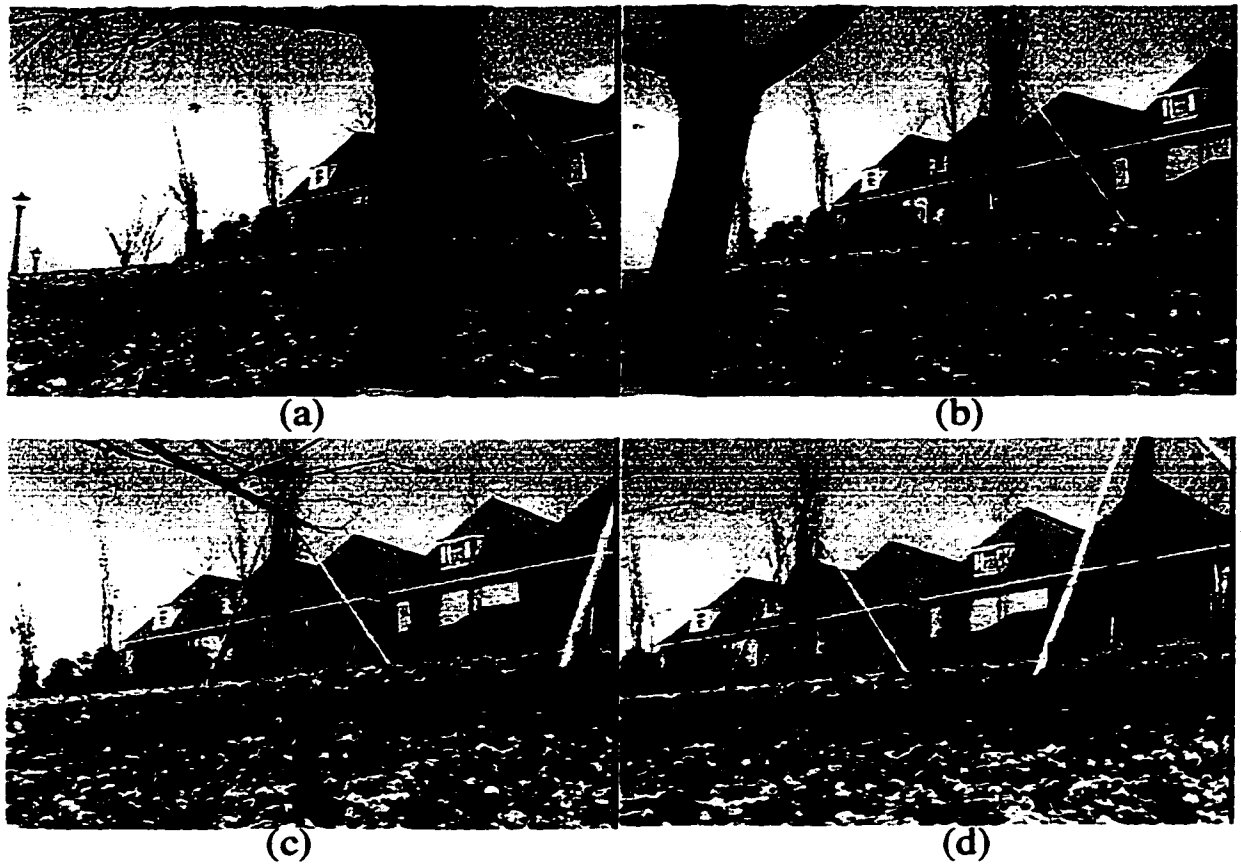

Figure 21. Images from the sflowg sequence used for program execution time measurements

Each frame is 352 by 240 pixels stored in the *.yuv format with a 4:2:2 sampling. As mentioned in Chapter 2, the 4:2:2 sampling means that, for every 2 by 2 group of pixels, 4 values are saved for the intensity (the y values), while only two u values and two v values are stored to indicate the color.

Compilation, Execution, And Timing Analysis Tools

The compiler used to produce the executable code from the source code was the xlc C compiler version 1.3 provided by IBM. The compiler includes an option to produce an assembler-language code listing. The assembler code listing includes a prediction of the number of clock cycles each instruction will require to execute. The clock cycle prediction is specific for the CPU chosen as the target machine for execution,

in this case the Power2 CPU. The timing listing is indicative of the minimum number of clock cycles that would be required to execute a sequence of instructions. It does not account for delays caused by cache misses, or context switches required by multi-tasking of the CPU.

The programs were run under AIX version 3.2, which is IBM's version of UNIX. A standard UNIX utility, prof, was used for the initial analysis of the execution times of the program. When a program is compiled with the appropriate compiler options, the compiler inserts a small number of instructions into each subroutine that are used to collect data about the execution of the program. These instructions cause the program to record data that can later be processed using the UNIX prof utility. This utility produces a listing for each routine in the program that includes a list of all the subroutines called by the routine. For each subroutine called, the list includes the number of times it was called by the routine and the total time spent executing the subroutine as a result of the calls from the higher level routine. This information can be used to determine which subroutines account for the largest share of the time required to execute the program, and which higher level routines are the most responsible for calling the most time consuming subroutines.

Parallelization Tools

The parallel programming tool called Linda (Gelernter 1988; Carriero and Gelernter 1989) was used to implement several alternative parallel implementations that would run on multiple nodes of the IBM SP2 computer. Linda is one of several tools available to enable using a collection of computers for parallel processing. Other examples include PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) (Snir, Otto, Huss-Lederman, Walker and Dongarra 1996). These tools each provide a mechanism for running programs on multiple compute nodes of varied architectures.

They can be used on computers such as the SP2 that have a dedicated network between the nodes or on clusters of computers connected via a shared network such as the internet.

It should be noted that these tools do not provide automatic parallelization of serial programs. There are tools for preprocessing C programs to produce code that will run in parallel on a shared-memory multi-computer. There is also High Performance Fortran (HPF) that converts Fortran source code into object code that can run on parallel processors if the programmer puts structured comments into the source code to guide the compiler. Similar tools, however, are not yet available for producing object code from C that will run in parallel on distributed-memory multi-computers.

Linda was chosen from the tools that are available because of its ease of use, portability, and ready availability at the site the research was performed. Linda comes in two versions, one for use with FORTRAN programs and another for use with C programs. The C programming version was used since that is the language of the original MPEG program.

Linda augments the C language with a few simple constructs that permit programs to be parallelized. A key concept used by Linda is that of an associative memory space, called tuple-space, shared by all nodes executing the program. Linda implements a virtual associative memory. In the virtual associative memory each entry, called a tuple, consists of one or more fields. Each field can contain a single numeric value, an array of values, an alpha-numeric string, or a data structure. The memory can hold any number of tuples and each tuple can have up to 16 fields. Thus, at the programmer's discretion, there could be a single tuple for each image, or a tuple could contain the list of parameters that are the input for a subroutine.

There are four basic instructions that Linda adds to the C language: out, eval, in, and rd. The *out* instruction places a tuple in the tuple-space. For example, the instruction

```
out ("ball", 32, i, f(i))
```

creates a tuple with four items. The first is the string "ball," the second is the integer 32, the third is the value of the variable i, and the fourth is the value returned by the function f when passed the variable i.

The *eval* instruction is used to create parallel processes. Each tuple created with an eval instruction implicitly creates a parallel process to evaluate each argument. The operation creates the parallel process and then returns. Thus the instruction

```
eval ("ball", 32, i, f(i))
```

creates a tuple with three entries for the first three arguments, and a process to evaluate f(i). On the IBM SP2, execution of the parallel process to evaluate f(i) is assigned to one of the nodes in the pool of nodes assigned to the job. Selection of which node in the pool is done automatically.

The *in* and *read* instructions are the key to the associative memory character of tuple-space. They each retrieve a tuple that matches a specified pattern or template. The templates consist of a sequence of typed fields, some of which hold values, and others of which are place holders.

For example, the instruction

rd ("ball", ?x, 10, ?y)

would find a tuple that has four components where the first component is the character sequence "ball," the third component has the value 10, and the other two components match the types of x and y. Once a match is found, the local variables x and y are assigned the values in the second and fourth entries in the tuple. If there are several tuples that match the template, then any one of them may be used to assign the values for x and y. Once the data is read the in instruction removes the tuple from tuple-space while the read instruction leaves the tuple in tuple-space. If no match is found, the process waits until a matching tuple is created.

The use of standard UNIX utilities for gathering timing information and the use of Linda for the parallelization tool provided readily available tools for executing the

research. Chapters V through VIII provide details on how these tools were used in the experiments.

CHAPTER V

SERIAL PROGRAM ANALYSIS

As mentioned in Chapter 4, the first phase of the research was to analyze the original source code to compare the control and data structures with typical image-processing programs. Analysis of the serial code found that the program is representative of image-processing programs in that it includes many of the common image-processing operations included in Table II on page 20. As is common in computationally intensive image-processing applications, it also was found that most of the execution time for the serial code--over 80%--is consumed while performing neighborhood operations, in this case the code required for motion estimation (see Figure 8, page 34).

Serial Code Structure Analysis

The documentation for the benchmark software indicates it is compatible with both K&R and ANSI C. Figure 22 is a block diagram of the original source code.

The program initialization routine reads a parameter file that indicates:

1. the names of the files containing the image sequence,
2. the name and path to a file for storing collected statistics and diagnostic outputs,
3. the file names containing any non-default quantization or transform pattern matrixes,
4. the pattern of I-, P-, and B-frames for encoding,
5. bit rate control parameters (e.g., target buffer size, target bit rate),
6. image format parameters (e.g., width, height, interlacing), and
7. MPEG2 features to use (e.g., profile, level, and concealment motion vectors).

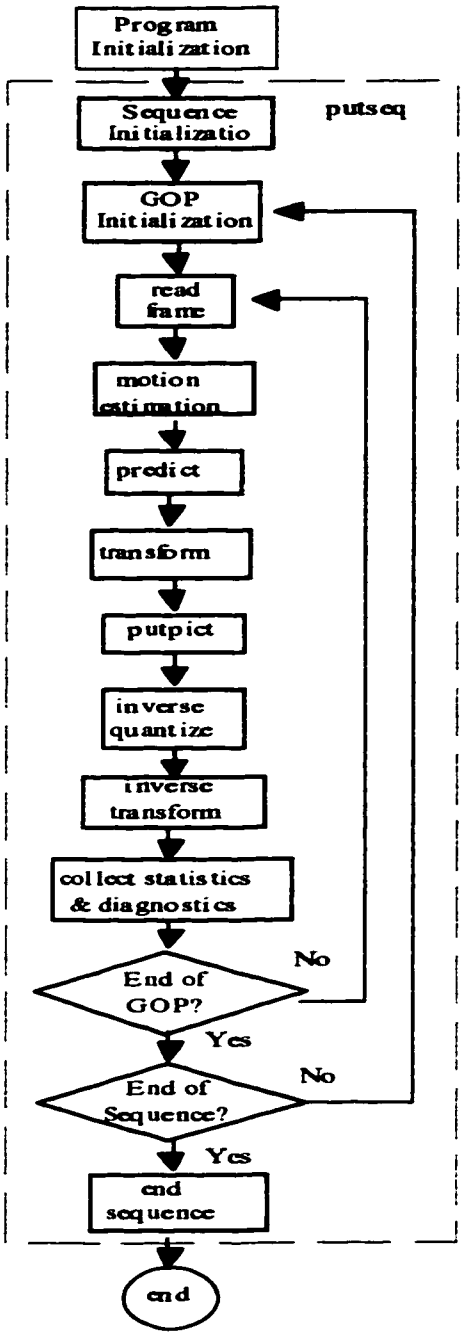

Figure 22. Structure of MPEG2 encoding program

Frame type	progressive
Frame size	352 by 240 pixels
Frames per GOP	15
Ratio of P to B frames	1:2
Frame rate	30 frames/sec.
Frames in test sequence	150
Target bit steam bit rate	5 M bits/sec.

Table V. Key parameters used in encoding experiments

Some of the key parameters used in the experiments are shown in Table V. After reading the parameter file, the program initialization routine checks the legality of all the parameters, reads the specified transform matrixes, and then executes initialization routines needed by the discrete cosine transform routines and the routine that stores the output stream. Then the sequence initialization routine transmits the data sent at the start of a sequence as indicated in the section on MPEG2 Header Information in Chapter 2, page 36.

The bulk of the program execution is controlled by a subroutine called putseq. It calls a sequence of subroutines grouped as indicated in Figure 22. The GOP initialization routine is executed once for each group of pictures. The readframe routine retrieves one image file and crops or extends the frame to match the output format specified in the parameter file. This is similar to other image-processing applications, including guidance and factory automation, in which a new frame must be acquired repeatedly. In this MPEG encoding program, the readframe routine also may convert the image format if the input image is not in a yuv array format (e.g., from RGB).

The first frame in each GOP is an I-Frame. Subsequent frames in each GOP will be P- or B-Frames in a ratio defined by the user in the parameter file. The motion estimation routine is called for each frame, although for I-frames its only function is to store flags indicating that each macroblock is encoded without use of a reference image. Motion estimation is performed for each macroblock of each P- and B-frame. Matching operations, such as these, are often found in tracking programs and pattern recognition applications. The control structures, memory access pattern, and calculations performed by the motion estimation routine are similar to the neighborhood operations found in many image-processing applications.

The motion estimation routine searches the reference images for the region that best matches each macroblock in the frame currently being encoded. The output of the motion estimation routine is a set of vectors, one for each macroblock. Each vector indicates the offset, in pixel coordinates, between the macroblock position in the current image and the location of the best match in the reference image.

The search is done first in whole pixel steps. In this coarse search, a spiral pattern is followed out from the center of each macroblock. At each location the absolute values of the differences between the current frame and the reference frame(s) are added as indicated in Eq. 4. Only the values in the luminance, y, part of the image are used in this search. The search ends once the edges of the search window are reached. If the option is selected, the motion estimation routine may refine the search to half pixel quanta. In this case, once the course location of the best match is found, a 3x3 half pixel interpolation search is performed around the course resolution solution to produce the final motion estimation vectors.

$$S = \left\{ (i,j) \middle| \min_{\substack{imax < i < imax \\ jmax < j < jmax}} \sum_{l=0}^{h} \sum_{k=0}^{16} \left| p(x-l,y-k) - q(x-i-l,y-j-k) \right|, \forall (x-i-l,y-j-k) \ni P, \right\} \qquad (4)$$

This function is similar to other common image-processing operations such as convolution, dilation, and erosion. In each case the function involves an operation between corresponding pixels at each location followed by an operation that accumulates a neighborhood result. In convolution the corresponding pixels are multiplied and the results from each location are added to create a neighborhood result. In dilation and erosion, the corresponding pixels are subtracted and the maximum or minimum, respectively, is found for the neighborhood. In the MPEG matching operation, the absolute value of the difference between corresponding pixels is computed and then the sum of these calculations produces the neighborhood result.

This calculation requires one subtraction, one absolute value, and one addition for up to 256 pixels for each location within the window. The number of pixels in a standard aspect-ratio image is $O(n^2)$ where n is the length or width of the image. Typically, the length and width of the search window also will increase in proportion to n so that the number of calculations to determine the motion of each macroblock is $O(n^2)$. Since the total number of macroblocks is proportional to the number of pixels in the image, the total number of calculations is $O(n^4)$. Since no fast algorithms have been identified for motion estimation, and since the sizes of video images continue to increase, this part of the application program will be very computationally intensive.

Once motion estimation is completed for all the macroblocks, the predict subroutine computes a difference image for all three channels of the image. The difference between each block in the current frame and a block sized region in the reference frame that is located at the position indicated by the motion vector is computed as indicated in Eq. (5). This computation is similar to the other multi-image operations listed in Table II in that it computes the difference between two images within corresponding windows.

$$d(i,j) = p(x-i, y-j) - q(x-i-m, y-j-n) : 0 < i < 8, 0 < j < 8 \quad (5)$$

The next step is performed by the transform routine. It computes the two-dimensional forward discrete-cosine transform of each block in the difference image produced by the predict subroutine. This operation has a memory access pattern and calculations similar to other whole-image transformations such as the FFT.

The putpict subroutine performs several important functions. It quantizes the values for each block, reorders the two-dimensional data from each block into a one-dimensional sequence, performs run length encoding, and then puts the run length encoded data into the output stream. Quantization reduces the number of bits used to store each coefficient of the discrete cosine transform output. This is the only irreversible step in the encoding, that is, information is lost in this process. The quantization levels are determined partly by comparing a running estimate of the average bit rate for the GOP with a goal bit rate. Thus, if the number of bits required to encode the previous macroblocks is above the goal, then the quantization used on the current block is made larger to reduce the number of bits required to encode it. The impact of this characteristic on the parallelizability of the program will be discussed further in the section on parallel implementation efficiency improvements later in this chapter.

The inverse quantize and inverse transform subroutines shown in Figure 22 are used to generate the reference images needed to compress subsequent frames. As mentioned previously, these images are the same as would be produced by an MPEG decoder. Because quantization causes a loss of information, the reconstructed images differ slightly from the original images. However the decoder never actually has the original images. It must use reconstructed images as the reference frames. Therefore the encoder also must use the reconstructed images as references so that the errors do not accumulate and cause additional distortion in the reconstructed images.

The final subroutine called by the putseq routine is to collect statistics for diagnostic purposes. It is not essential to the MPEG encoding process.

This analysis confirms that the MPEG encoding program used in this research can be considered representative of image-processing programs. It includes many of the neighborhood operations, multi-image operations, and whole image operations listed in Table II. Therefore, improvements to computer designs suggested by analysis of the execution of this program can be predicted to provide improved performance for many other image-processing application programs.

Serial Code Timing Analysis

Consistent with good parallelization practices, the serial implementation was studied to find the most computationally intense sections. The code was imported and compiled with alternative compiler optimization options selected. Table VI and Table VIII summarize the time spent in each of the seven major sections of the putseq section of the program without any optimization while processing 150, 352x240 pixel, frames of the pflowg image sequence. It took nearly 24 minutes (1,437 seconds) to compress the 150 frames for an average of nearly 8 seconds per frame. This is nearly 200 times slower than the 33 ms that is available to process real-time video at 30 frames per second. The last column in Table VI shows that 81.5 percent of the execution time was for motion estimation and another 12.03 percent of the execution time was for the discrete cosine transform routines, transform and itransform. The bottom row of Table VIII indicates that the P-frames take the longest to compress. This is caused by the larger search areas for motion estimation which is due, partly, to the longer times between the P-frames and their associated reference frames--either another P-frame or an I-frame (see Figure 7).

	I-Frame		P-Frame		B-Frame		Total	
	number in sequence	%	number in sequence	%	number in sequence	%	number in sequence	%
	11	7.33	40	26.67	99	66.00	150	100.00
Execution Time	msec	%	msec	%	msec	%	msec	%
readframe	214	0.01	707	0.05	1,733	0.12	2,653	0.18
motion estimation	361	0.03	438,003	30.48	733,538	51.00	1,171,901	81.50
predict	215	0.01	1,535	0.11	6,882	0.48	8,632	0.60
transform	11,136	0.77	40,287	2.80	99,967	6.96	151,389	10.53
putpict	6,297	0.44	21,437	1.49	36,990	2.57	64,724	4.50
iquant	1,240	0.09	4,511	0.31	10,154	0.71	15,905	1.11
itransform	1,670	0.12	5,915	0.41	13,953	0.97	21,538	1.50
total for frame type	21,151	1.47	512,484	35.66	903,367	62.86	1,437,003	100.00

Table VI. Unoptimized-implementation execution time for each subroutine by frame encoding type

	I-Frames		P-Frames		B-Frames		Total	
	number in sequence	%	number in sequence	%	number in sequence	%	number in sequence	%
	11	7.33	40	26.67	99	66.00	150	100.00
Execution Time	msec	%	msec	%	msec	%	msec	%
readframe	263	0.05	758	0.15	1,638	0.32	2,659	0.51
motion estimation	112	0.02	174,126	33.53	268,230	51.65	442,468	85.20
predict	36	0.01	454	0.09	2,251	0.43	2,741	0.53
transform	2,357	0.43	8,547	1.55	21,181	3.84	32,086	5.82
putpict	2,793	0.54	9,104	1.75	15,477	2.98	27,374	5.27
iquant	371	0.07	1,296	0.25	2,814	0.54	4,481	0.86
itransform	553	0.11	2,094	0.40	4,681	0.90	7,328	1.41
total frame time	6,500	1.25	196,417	37.82	316,382	60.92	519,300	100.00

Table VII. Execution time breakdown for optimized serial code processing 150 frames

	Time (msec.) Total/Average			
	I-Frame	P-Frame	B-Frame	Average
read frame	19.4	17.7	17.5	18.2
motion estimation	32.8	10,950.0	7,409.5	6,130.8
predict	19.5	38.4	69.5	42.5
transform	1,012.4	1,007.1	1,009.6	1,009.7
putpict	572.4	535.9	373.6	494.0
inverse quantize	112.7	112.7	102.6	109.4
inverse transform	151.8	147.9	140.9	146.9
Total	1,922.8	12,812.1	9,124.9	7,953.3

Table VIII. Execution time per frame for unoptimized serial implementation

	Average Time (msec.)			
Program Section	I-Frame	P-Frame	B-Frame	Average
read frame	23.9	18.9	16.5	17.7
motion estimation	10.2	4,535.1	2,709.3	2,949.8
predict	3.3	11.3	22.7	18.3
transform	214.2	213.6	213.9	213.8
putpict	253.9	272.6	156.3	182.5
inverse quantize	33.7	32.4	28.4	29.9
inverse transform	50.2	52.4	47.3	48.9
Total	590.9	4,910.4	3,195.8	3,462.0

Table IX. Average time to process one frame of each type using optimized serial implementation

To find the best choices for the compiler options a variety of combinations were studied. IBM's xl C compiler provides several different options that can improve a programs performance. The combination of options found most effective was with -O3 -qarch=pwr2 -Q+dist1. The -O3 option instructs the compiler to use aggressive optimization. The -qarch=pwr2 tells the compiler that the code will be run on a Power2 processor, and to therefore, use techniques that take advantage of that architecture. The

-Q+dist1 option tells the compiler to copy the code for the dist1 routine into any routine that calls it, rather than using a subroutine call to execute the routine.

When this combination of compiler optimization options is used, the total execution time to process the 150 image sequence drops to less than 9 minutes (519.3 seconds). Table VII and Table IX show the details of the timing when these compiler optimization alternatives were selected. While this is more than twice as fast as the unoptimized code, the average is still over 3 seconds per frame--over 100 times too slow for real-time video-sequence compression.

When compiler optimization is used, the percentage of the execution time spent doing motion estimation increased from 81.5% to 85.2% of the total execution time. This shows that the compiler was able to optimize this routine less effectively than the other routines. On the other hand, the percentage of execution time spent in the forward discrete cosine transform routine dropped from 10.53 to 5.82%, showing that the compiler was more effective in reducing the execution time for this routine than other routines.

CHAPTER VI

FINE-GRAIN (INSTRUCTION-LEVEL) EXECUTION EFFICIENCY IMPROVEMENTS

As mentioned in Chapter IV, the second phase of the research was to identify potential fine-grained execution efficiency improvements. This study demonstrated that the template operation that consumes most of the execution time for this image-processing application is dominated by the use of fixed-point-arithmetic resources. Significant improvement in execution time can be accomplished with compiler improvements, additional integer units, more efficient conversion from integer to floating point formats, or floating point units capable of executing integer operations.

Two experiments were performed that lead to these conclusions. Briefly, the first experiment studied the compiler's ability to exploit the instruction set of a processor. This was made possible by the observation that the compiler was not using one of the relevant instructions available in the Power2 instruction set. To simulate an improved compiler the code was modified slightly to force the compiler to use the available instruction. The change was made in a way that did not change the order of calculations and so that the results were identical to the original program.

The second potential improvement tested was addition of more integer functional units. To simulate the availability of additional integer functional units, the code was modified slightly so that the floating-point functional units would be used in some of the calculations. The details of the experiments are presented in the following sections.

Instruction Level Timing Analysis

Recompiling and executing the program with the profiling option turned on showed that almost all the time required for motion estimation is to execute a subroutine called dist1. Figure 23 shows the C code for the inner loop of dist1. This routine calculates the sum of the absolute differences between the target macroblock and a macroblock-sized region in the intensity channel of the reference image. It is executed once for each of the offsets between the reference and new image that are checked to find the minimum difference. The inner loop is executed 8 times for each call if the image sequence contains interlaced frames and 16 times if it contains progressive frames.

```
int dist1(p1, p2, lx,  h, distlim)
int lx, h, distlim
unsigned char *p1, *p2
{
      int s, j;

       s = 0;
      for (j=0; j<h; j++)
      {
            if ((v=p1[0] - p2[0]) < 0) v = -v; s+=v;
            if((v=p1[1] - p2[1]) < 0) v = -v; s+=v;
            .
            .
            .
            if ((v=p1[15] - p2[15]) < 0) v = -v; s+=v;

            if (s >= distlim) break;

            p1 += lx;
            p2 += lx;
      }
      return s;
}
```

Figure 23. dist1 code for calculating absolute difference of image regions

The types of images and other parameters included in the sample parameter files caused the dist1 routine used for motion estimation to be called 121 times for each macroblock in a P-frame and 58 times for each macroblock in a B-frame. For the 352 by

240 pixel images in the sample sequence the dist1 routine was called approximately 40,000 times per P-frame and 20,000 times per B-frame.

In the dist1 code, the inputs p1 and p2 are pointers to pixels in the reference frame and the new frame, respectively, lx is the number of pixels in a row, h is the number of rows to process and distlim is the previously found minimum. Note that, even though this code is computing an absolute value, there are no absolute value statements in the C source code. This provides greater portability since, although it is usually available as a library subroutine, neither K&R nor ANSI C include this function. This demonstrates the emphasis often placed on the portability of image-processing application source code. The emphasis on portability is an important factor when considering the advantages of computer hardware features that improve image-processing program execution times. Specialized features that require programmer knowledge of the specific computer executing a program will be ignored if portability is important to the developer.

The essential assembler language code generated for computing the first pixel's contribution is shown in Table X. This sequence exploits the fact that v is a temporary variable that does not need to be stored. The first two instructions fetch the corresponding pixels from the reference and new images, respectively. They also increment the pointers p1 and p2 by lx. The third instruction computes the difference between the pixels and sets Condition Register 0. The fourth instruction is a branch if the result was not negative. If the result was negative, then the code generated by the compiler performs the subtraction with the operands reversed. Finally, the resulting absolute value of the differences is added to the running sum. This set of instructions must be repeated 16 times for each of the lines in the block. Analysis of the assembler code indicated that the 83 instruction loop would execute in 48 clock cycles. To check that this actually occurs in execution, profiling information was collected while processing 30 frames of the pflowg image sequence. The profiles indicate the dist1 subroutine is executed 9,327,168 times and required 110.5 seconds to process this

segment of the sequence. Accounting for the 66 MHz clock rate of the SP2 processors used for these experiments, this implies an average of 48.64 clock cycles per loop which is close to that predicted.

C Source Code		
if(v=p1[0]-p2[0]<0) v=-v; s+=v; p1+=lx; p2+=lx;		
Compiler Generated Assembler Code		
inst	operands	comment
lbzux	gr17,gr19=(*)uchar(gr19,gr21,0)	load p1[0]; p1+=lx;
lbzux	gr15,gr22=(*)uchar(gr22,gr21,0)	load p2[0]; p2+=lx;
sf	gr30,cr0=gr17,gr15	v = p1[0]-p2[0];
bc	CL.58,cr0,0x1/lt ,	if (v<0)
sf	gr30=gr15,gr17	v = p2[0]-p1[0];
	CL.58	
a	gr30=gr30,gr3	s = v + s;

Table X. Assembler code from one line of inner loop of original source code

Data parallelism can be used to exploit instruction level (fine-grained) parallelism in this set of instructions as illustrated in Table XI. In this table the first column is the clock cycle while the remaining columns represent each of the six instructions used to process each pair of pixels. Thus the second column corresponds to the first load instruction in the assembler code shown in Table X. Similarly, the third column in Table XI corresponds to the subtract instruction in Table X.

The numbers in the body of the table correspond to the index into the p1 and p2 arrays for the data used by the instruction issued during that clock cycle. The Power 2 CPU can issue two instruction per clock cycle and there are only two integer arithmetic-logic units (ALU) to calculate effective addresses. The zeros in the second row indicate

that the instructions to load p1(0) and p2(0) into registers from memory are issued during the first clock cycle. Since only one load can actually be executed in each clock cycle, it takes two clock cycles to complete the two instructions. Therefore, the compiler could organize the assembler code so that the subtract instruction that computes p1(0)-p2(0) is issued during the third clock cycle, as indicated in the third column under sf. This would allow the two instructions for loading p0(1) and p2(1) to be issued during the second clock cycle as indicated in the table. During the third clock cycle, while one integer ALU is used to subtract p1(0) from p2(0), the other could be used to calculate the effective address for loading p0(2) as indicated. The conditional branch instruction, bc, cannot be executed until two clock cycles after the subtraction upon whose result its direction depends. Therefore, the bc instruction that depends on the result of the subtraction p1(0)-p0(0) need not be issued until clock cycle 5. The compiler could, therefore, arrange the assembler code so that, during clock cycle 4, the value for p1(2) is loaded and the subtraction, p1(1)-p0(2), is calculated.

The Power 2 CPU allows the next instruction after a branch to be issued at the same time as the branch instruction. If the branch is taken, the results of the calculation in that instruction are discarded. In Table XI this occurs in clock cycle 5 which shows the subtraction instruction, sf, for calculating p0(0)-p1(0) being issued at the same time as the branch instruction. If the result of the subtraction during clock cycle 3 yields a negative value for p1(0)-p0(0), then the result of the subtraction in clock cycle 5 is saved in the registers. If the result in clock cycle 3 is non-negative, then the results of the subtraction in clock cycle 5 are discarded.

clock	lbz	lbz	sf	bc	sf	a
1	0	0				
2	1	1				
3	2		0			
4		2	1			
5				0	0	
6				1	1	
7			2			0
8	3					1
9		3				
10				2	2	
11	4		3			
12		4				2
13				3	3	
14	5		4			
15		5				3
16				4	4	
17	6		5			
18		6				4
19				5	5	
20	7		6			
21		7				5
22				6	6	
23	8		7			
24		8				6
25				7	7	
26	9		8			
27		9				7
28				8	8	
29	10		9			
30		10				8
31				9	9	
32	11		10			
33		11				9
34				10	10	
35	12		11			
36		12				10
37				11	11	
38	13		12			
39		13				11
40				12	12	
41	14		13			
42		14				12
43				13	13	
44	15		14			
45		15				13
46				14	14	
47			15			
48						14
49				15	15	
50						
51						15

Table XI. Essential operations of inner loop of dist1 for original code

The clock count in the first column assumes that the branches are never taken. This is a worst-case assumption. A more reasonable assumption would be that half the branches would be taken. That is, half the intensities in each block of the new image are less than the intensities in the corresponding pixel of each location of the search window in the reference image. If this assumption is true, then 45 clock cycles would be the average required to execute this sequence.

The mathematical operations performed are one subtraction, one absolute value, and one addition per pair of pixels for a total of 48 mathematical operations per pass through the loop. Thus, this implementation would average slightly over 1 mathematical operation per clock cycle.

However, the compiler is not this efficient. It generates code that, for some lines, temporarily stores and then retrieves the intermediate results. The number of instructions per loop iteration is 106-130 (depending on the number of branches taken) requiring 98-119 clock cycles to execute. The average will be 120 instructions executed and 108.5 clock cycles if half the intensities in each block of the new image are less than the intensities in the corresponding pixel of each location of the search window in the reference image.

Impact of Compiler Exploitation of Instruction Set Architecture

The instruction set of the RS6000 processor used in the SP2 includes both floating-point and fixed-point absolute-value instructions. The assembler language output of the compiler did not use this instruction regardless of which optimization options were selected. It also did not recognize that the variable v is superfluous.

To determine the improvement that could be achieved by using the more efficient machine instruction, the 16 lines of source code were rewritten to use the absolute value

instruction as shown in Figure 24. This experiment illustrates the impact that the instruction set architecture can have on the execution speed.

```
int dist1(p1, p2, lx,  h, distlim)
int lx, h, distlim
unsigned char *p1, *p2
{
 int s, j;

s = 0;
for (j=0; j<h; j++)
 {
s += __abs(p1[0] - p2[0])
     + __abs(p1[1] - p2[1])
     .
     .
     .
     + __abs(p1[15] - p2[15]);

if (s >= distlim) break;

p1 += lx;
p2 += lx;
  }
return s;
}
```

Figure 24. dist1 code for calculating absolute difference of image regions

The double underscore before the abs subroutine name in Figure 24 tells the compiler to put the subroutine in-line rather than as a called library routine. Since the library routine is just one instruction, the assembler code produced includes the absolute value instruction without using a subroutine call.

The essential assembler language code generated for computing the first pixel's contribution is shown in Table XII. As in the previous assembler language code, the first two instructions load the corresponding pixels from the reference and new images, respectively. They also increment the pointers p1 and p2 by lx, the length of one row in the images. The third instruction computes the difference between the pixels. Contrary to the previous implementation, the fourth instruction computes the absolute value of the

difference. No branch instruction is needed. The last instruction adds the result to the running sum. As before, this set of instructions must be repeated 16 times for each of the lines in a block.

C Source Code		
s += abs(p1[0] - p2[0]); p1+= lx; p2+= lx;		
Compiler Generated Assembler Code		
inst.	operands	comments
lbzux	gr6,gr4=(*)uchar(gr4,gr5,0)	gr6=p2[0]; gr4=*p2[0]
lbzux	gr0,gr7=(*)uchar(gr7,gr5,0)	gr0=p1[0]; gr7=*p1[0]
sf	gr0=gr0,gr6	gr0=p1[0]-p2[0]
abs	gr10=gr0	gr10=|p1[0]-p2[0]|
a	gr6=gr10,gr6	s(gr6)=|p1[0]-p2[0]|+s

Table XII. Assembler code for one line of inner loop of source code in Figure 24

An efficient execution of this sequence is shown in Table XIII. As before, each column represents one of the five instructions used to process each pixel pair. The clock count is in the first column. The 42 clocks required should be compared with the estimate of 45 clocks for the average execution of the idealized implementation for the original code. This is only a 6.67% improvement. As before, the number of mathematical operations is 48 so the average number of mathematical operations per clock is 48/42=1.14.

clock	ld p1[•]	ld p2[•]	sf	abs	a
1	0	0			
2	1	1			
3	2		0		
4		2	1		
5			2	0	
6	3			1	
7				2	0
8	4	3			
9		4			1
10	5		3		
11			4		2
12		5		3	
13	6			4	
14			5		3
15	7	6			
16		7		5	
17	8		6		
18			7		5
19		8		6	
20	9			7	
21			8		6
22		9			7
23	10			8	
24		10	9		
25	11				8
26			10	9	
27	12	11			
28				10	9
29		12	11		
30	13				10
31			12	11	
32	14	13			
33				12	11
34		14	13		
35	15				12
36			14	13	
37		15			
38				14	13
39			15		
40					14
41				15	
42					15

Table XIII. Essential operations of inner loop of dist1 using __abs instruction

	I-Frame		P-Frame		B-Frame		Total	
	number in sequence	%	number in sequence	%	number in sequence	%	number in sequence	%
	11	7.33	99	66.00	40	26.67	150	100.00
Execution Time	msec	%	msec	%	msec	%	msec	%
readframe	339	0.10	2896	0.85	1,486	0.44	4,721	1.38
motion estimation	138	0.04	165,625	48.53	96,775	28.35	262,539	76.92
predict	60	0.02	2,207	0.65	477	0.14	2,743	0.80
transform	2,373	0.70	20,420	6.28	8,630	2.43	32,424	9.50
putpict	2,731	0.80	15,186	4.45	9,202	2.70	27,119	7.95
iquant	357	0.10	2,728	0.80	1,266	0.37	4,351	1.27
itransform	552	0.16	4,666	1.37	2,053	0.60	7,271	2.13
total for frame type	6,562	1.92	214,825	62.9	119,926	35.14	341,313	100.00

Table XIV. Execution time breakdown for serial implementation that uses abs instruction

The performance of the compiler with the code in Figure 24 is much more efficient than its performance with the original code in Figure 23. The number of instructions per loop iteration is 83, requiring only 46 clocks. Thus the number of instructions per clock is 1.8, nearly twice the rate for the original code. More important, the number of mathematical operations per clock goes up to 1.14 from 0.44 mathematical operations per clock using the original code.

Table XIV and Table XV summarize the time spent in each of the seven major sections of the putseq section of the program while processing 150 frames. The total time to process the 150 frame sequence when the abs instruction was used was 341 seconds as compared to 519 seconds when the original code compiled with optimizations was used.

Program Section	Average Time (msec.)			
	I-Frame	P-Frame	B-Frame	Average
read frame	30.8	37.1	29.3	31.5
motion estimation	12.6	2,419.4	1,673.0	1,750.3
predict	5.4	11.9	22.3	18.3
transform	215.6	215.6	216.3	216.1
putpict	248.2	230.0	153.4	180.8
inverse quantize	32.5	31.6	27.6	29.0
inverse transform	50.2	51.3	47.1	48.5
Total	596.6	2,998.2	2,170.0	2,275.4

Table XV. Average times, by frame type, for executing serial implementation that uses abs instruction

Impact of Additional Functional Units

Another possible way to increase the computational capacity of a computer is to increase the number of functional units. The Power 2 architecture includes two integer ALUs and two floating-point ALUs. With these, a Power 2 CPU can simultaneously execute up to four different instructions. The experiments described in the previous sections show the level of performance that can be achieved when the two integer units are used simultaneously and some of the implications of including the abs instruction in the instruction set architecture. Further improvements in the execution time could be achieved if additional functional units could be used. The architecture could be modified by including additional integer units. This would increase the performance of the Power 2 CPUs for integer intensive applications such as image processing. Another possible modification would be to make it easy to use the floating point units for integer operations.

To explore the impact these changes might produce, an attempt was made to modify the code so that the floating point units would be used during execution of the motion estimation subroutine. There are two ways that this can be done. One approach would convert both the images referenced by the dist1 subroutine to floating point format. This approach, although possibly very effective, was considered impractical because it would have required extensive changes throughout the entire program. Another approach was found that only required changes within the dist1 subroutine. This was done by making the conversion from integer to floating point format after the difference between the pairs of pixels is computed. This approach is shown in Figure 25. The difference between this code and that in Figure 24 is the use of floating-point instructions when computing the absolute values.

```
int dist1(p1, p2, lx,  h, distlim)
int lx, h, distlim
unsigned char *p1, *p2
{
 int s, j;
float v;

s = 0;
for (j=0; j<h; j++)
 {
   v += __fabs(p1[0] - p2[0])
        + __fabs(p1[1] - p2[1])
        .
        .
        .
        + __fabs(p1[15] - p2[15]);
   s += v;
   if (s >= distlim) break;

   p1 += lx;
   p2 += lx;
  }
return s;
}
```

Figure 25. dist1 code modified to use floating point absolute value and addition instructions

Table XVI shows the essential assembler language code generated for computing the first pixel's contribution. The differences between this assembler code and that in Table XII result from the use of the floating-point instructions in the source code. As in the previous examples, the first two instructions load the corresponding pixels from the reference and new images while simultaneously incrementing the pointers p1 and p2 by lx. The third instruction computes the difference between the pixels. The next three instructions are required to convert the single precision integer data to double precision floating point format. First the integer is stored in memory (and cache). The previous location in memory has been filled previously with the high order word needed to represent an unnormalized integer in floating point format. The double word is then loaded and zero is added to it to normalize the result. This cumbersome method of converting from an integer to a floating-point number is a serious concern for

applications such as image processing that are integer-operation-intensive. The last two instructions in the sequence take the absolute value and add the result to the partial sum. The result is that there are still five instructions that use the fixed point units. So, even though the absolute value and partial sum use the floating point units, the total execution time cannot be reduced. This is borne out in the measured execution times shown in Table XVII and Table XVIII.

C Source Code		
v += fabs(p1[0] - p2[0]); p1+= lx; p2+= lx;		
Compiler Generated Assembler Code		
inst.	**operands**	**comments**
lbzux	gr0,gr7=(*)uchar(gr7,gr5,0)	gr0=p1[0]; gr7=*p1[0]
lbzux	gr6,gr4=(*)uchar(gr4,gr5,0)	gr6=p2[0]; gr4=*p2[0]
sf	gr0=gr0,gr6	gr0=p1[0]-p2[0]
st	#MX_CONVU6(gr1,-60)=gr0	store single precision of floating point
lfd	fp0=#MX_CONVU6(gr1,-64)	retrieve double
fa	fp1=fp0,fp3,fcr	normalize by adding 0
fabs	fp1=fp0,fp3,fcr	fp1=|p1[0]-p2[0]|
fa	fp1=fp5,fp1,fcr	v(fp1)=|p1[0]-p2[0]|+v

Table XVI. Essential assembler language instructions for inner loop of source code in Figure 25

	I-Frame		P-Frame		B-Frame		Total	
	number in sequence	%	number in sequence	%	number in sequence	%	number in sequence	%
	11	7.33	40	26.67	99	66.00	150	100.00
Execution Time	msec	%	msec	%	msec	%	msec	%
readframe	131	0.04	781	0.24	1,908	0.59	2,819	0.87
motion estimation	112	0.03	66,351	20.42	182,015	56.03	248,478	76.48
predict	93	0.03	150	0.05	1,400	0.43	1,643	0.51
transform	2,361	0.73	8,565	2.64	21,342	6.56	32,267	9.93
putpict	2,346	0.72	7,967	2.45	17,351	5.34	27,665	8.52
iquant	352	0.11	1,266	0.39	2,990	0.92	4,608	1.42
itransform	542	0.17	1,961	0.60	4,755	1.46	7,258	2.23
total for frame type	5,948	1.83	87,075	26.80	231,853	71.37	324,877	100.00

Table XVII. Execution time breakdown for serial implementation using floating point absolute value instruction

Program Section	Average Time (msec.)			
	I-Frame	P-Frame	B-Frame	Average
read frame	4.0	3.9	3.9	3.9
motion estimation	10.2	1,584.1	1,811.0	1,618.4
predict	3.0	3.7	13.7	10.2
transform	224.7	214.6	215.2	215.0
putpict	206.6	196.9	175.3	183.4
inverse quantize	32.1	31.6	30.6	31.0
inverse transform	48.8	48.7	48.0	48.3
Total	520.5	2084.4	2,298.6	2,111.1

Table XVIII. Execution time breakdown for serial implementation optimization using floating point absolute value (fabs) instruction

Figure 26 summarizes the four groups of timing measurements. The average time required by each of the major sections of the program to processes a frame is shown. The numerical values for the two sections that require the most time (motion estimation and predict) are included at the top of their respective bars.

Recall that the only changes to the mathematical operations in the source code were those inserted to force the compiler to use different assembler language instructions while computing the absolute values of the differences between pixels. The only other changes were to eliminate diagnostic data collection.

The experiments summarized in Figure 26 show that

1) even the best implementation is too slow by a factor of nearly 50 to support real-time compression,

2) the impact of compiler optimization combined with changes to guide the compiler to use the abs instruction provides most of the speedup over the unoptimized version

3) the compiler was able to achieve a maximum of 1.8 instructions per clock, of these 1.44 instructions per clock were mathematical operations

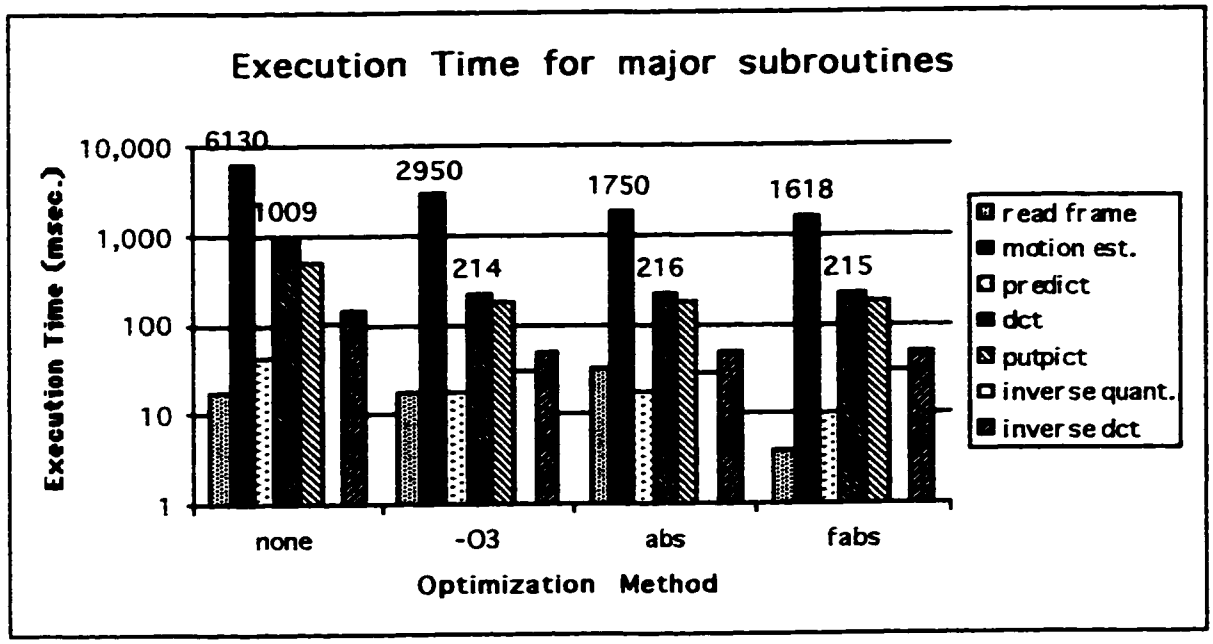

Figure 26. Execution time improvements for alternative optimization methods

CHAPTER VII

COARSE-GRAIN PARALLELIZATION AND EFFICIENCY IMPROVEMENTS

Chapter III describes the approaches used in specialized image processors to provide exceptional throughput while performing image transformation operations. Some of the characteristics found in the ATCURE image processing subsystem that could be incorporated into general-purpose multi-computers include (a) hardware to support systolic operation, (b) sophisticated image-oriented address generators, (c) simultaneous transfer of two or more images between pairs of processors, (d) multiple destinations when communicating image data, and (e) local reconstruction of neighborhood data from an image data stream. Part of the hypothesis of this dissertation is that the most significant bottlenecks encountered when executing image-processing application programs could be addressed by incorporating one or more of these features.

As mentioned in Chapter IV, the third phase of the research reported here tested this hypothesis by studying coarse-grained parallelization implementations to identify potential improvements in the communications structure of the SP2 that would have a significant impact on execution times. The research showed that the MPEG application software used in this study includes many of the characteristics of image-processing applications that make them difficult to distribute among multiple nodes of a distributed memory multi-computer. The study identified the factors that limit the achievable improvement in execution time when the program is executed using multiple nodes. These factors include (1) the timing (e.g., synchronization) requirements of the parallel implementation, (2) the algorithms implemented by the original source code, and (3) the performance of the communication system and system software of the computer.

The most significant limit to the throughput is that some code could not be parallelized due to dependencies that force serial execution. The next most significant factor limiting the throughput of the parallel implementation, after the serial code execution limits, is the overhead required for communication. When small messages are used the overhead is dominated by the multiple levels of software required to initiate and transfer each message. For longer messages, the overhead is dominated by the time to move non-contiguous blocks of data into a single message. This suggests that specialized address generators to support the direct memory transfer of image data to the communication channels would have the greatest impact on overall parallel program execution times.

Parallel Program Structure Analysis

The first step in testing the hypothesis was to identify an effective way to parallelize the benchmark MPEG2 program. Many approaches can be considered for a parallel implementation of the program. Each of these alternative approaches would have a different impact on the latency and throughput of the implementation. Here *latency* refers to the time from when a frame arrives until it is completely encoded. *Throughput* refers to the number of frames per second that can be encoded by the system. Using the analogy of a pipeline, the throughput is the rate at which material flows past one point (such as the beginning or end) of the pipeline, while latency is the time it takes for material to pass from the beginning to the end of the pipeline. A premise of the research reported here was that the applications are real-time broadcast television and video conferencing. These applications require that the latency be minimized (e.g., under 0.1 second) and that the throughput be 30 frames per second.

Another premise of the research reported here is that general-purpose processors should be improved so they can execute typical image-processing applications faster. It

has already been demonstrated that the MPEG code used as a benchmark includes control and data structures that are similar to many image-processing application programs. From these considerations came a constraint that the research reported here would be conducted without changing the algorithms used by the benchmark program. The architecture enhancements sought are those that would improve execution time for the code as it exists.

Two major techniques for parallel implementations are the functional- and data-parallel approaches. Coarse-grained functional parallelism would assign consecutive subroutines (or groups of subroutines) in the main loop of the program to different nodes of the computer. Each node would perform its functions and then pass the results to the next node for further processing. Thus, from Figure 22 on page 77, the routines in putpict could each be assigned to a different node. If more parallelism is needed, then each of these routines could be divided into parts for execution by different nodes.

Data-parallel approaches divide the data between nodes so that they each process a part of the data. Each node performs all operations on its part of the data and the partial results from all nodes are combined to form the complete result. Thus, in a perfectly data parallel approach each node would execute all of the subroutines called by putpict but only on the part of the data assigned to that node. The data assigned to each node could be as much as an entire group-of-pictures (GOP) or as little as a small part of a frame.

Coarse-grained functional parallelism cannot be used with the MPEG software used in this study to meet the throughput and latency requirements of the applications due to the concentration of the computational task in one small subroutine. The timing analysis for the serial code described earlier in this chapter shows that about 75% of the execution time is consumed by one subroutine, dist1. Coarse-grained functional parallelism would assign this subroutine to a single node. The minimum time to execute the program would then be limited by the time for that node to receive all of its inputs, execute the dist1 routine on them, and transmit the results. This would take at least as

long as it takes to perform this function when the entire program is executed using a single node. Since the total time used to perform dist1 on a single node is over 75% of the total execution time, coarse-grained functional parallelism can, at most, produce a speed up by a factor of 1/0.75 or 1.33. This is far from the factor of 60 needed to go from the serial execution time of about two seconds per frame to 30 frames per second. Hence, coarse-grained functional parallelization cannot meet the requirements and was, therefore, not attempted.

Given the current performance of the nodes in an SP2, the latency and throughput requirements limited which data-parallel approaches can be used. For example, if a group-of-pictures was assigned to each node, then the minimum latency would be approximately the time required for one node to process one GOP. Similarly, if a single frame was assigned to each node, then the minimum latency would be the time for one node to process one frame. The timing experiments performed using the serial code indicate that it takes over 2 seconds to encode a single frame using just one node. This would be unacceptable for broadcast video and video conferencing applications. So the data-parallel approaches must be limited to those that divide each frame into smaller parts that are each processed by a different node.

The timing constraints dictate the use of a data-parallel approach that assigns only part of a frame to each node but the need to minimize communication suggests the need to break the frame into a minimum number of regions. A speedup of at least 60 over the single node execution time is required to reduce execution time from 2 seconds per frame to 30 frames per second. The parallelization scheme must therefore allow for at least 60 nodes to work together to achieve real-time encoding.

The structure of the original source code limits how a parallelizing compiler can distribute the program among multiple nodes of a distributed-memory multi-computer like the IBM SP2. The data structures and program code for the subroutines called by putseq determine how the program can be divided among 60 or more nodes. Within

several of the major subroutines there are loops that sequence through each of the macroblocks in a frame. Each of these subroutines does some analysis or transform of all of the macroblocks in a frame and then returns. This structure suggests a parallel program implementation in which each node performs the transforms of all of the subroutines called by putpict but only on a subset of the macroblocks in an image.

Although previous work had assigned one macroblock to each node, the approach taken in this research allowed the number of nodes available to vary. This permitted analysis of the scalability of the computer architecture both for handling larger images and for using additional nodes with the same image size. Figure 27 shows the resulting MPEG program structure. This implementation should be compared to the MPEG encoding block diagram in Figure 8 on page 34.

The parallel implementation block diagram assumes there is a single master node and multiple slave nodes. Conceptually, a group of macroblocks is assigned to each slave node. At the top of Figure 27 (step 1) the master node reads the next frame and distributes the appropriate parts to each of the slave nodes.

In step 2, each slave node performs the first operation, motion estimation, for the macroblocks assigned to it. Motion estimation derives motion vectors for each macroblock. The motion vectors point to the regions in the reference frames that most closely match each macroblock in the current frame. The motion estimation subroutine uses the intensity channels from the current frame and from reference frames. The reference frames are stored in each node's local frame memory. The local frame memory contains the appropriate regions of the results from the encoding of previous frames. For P-frames, there is one reference frame; for B-frames, there are two reference frames. For each reference frame, the motion estimation subroutine uses both the intensity channel of the source image and the reconstructed image for that frame.

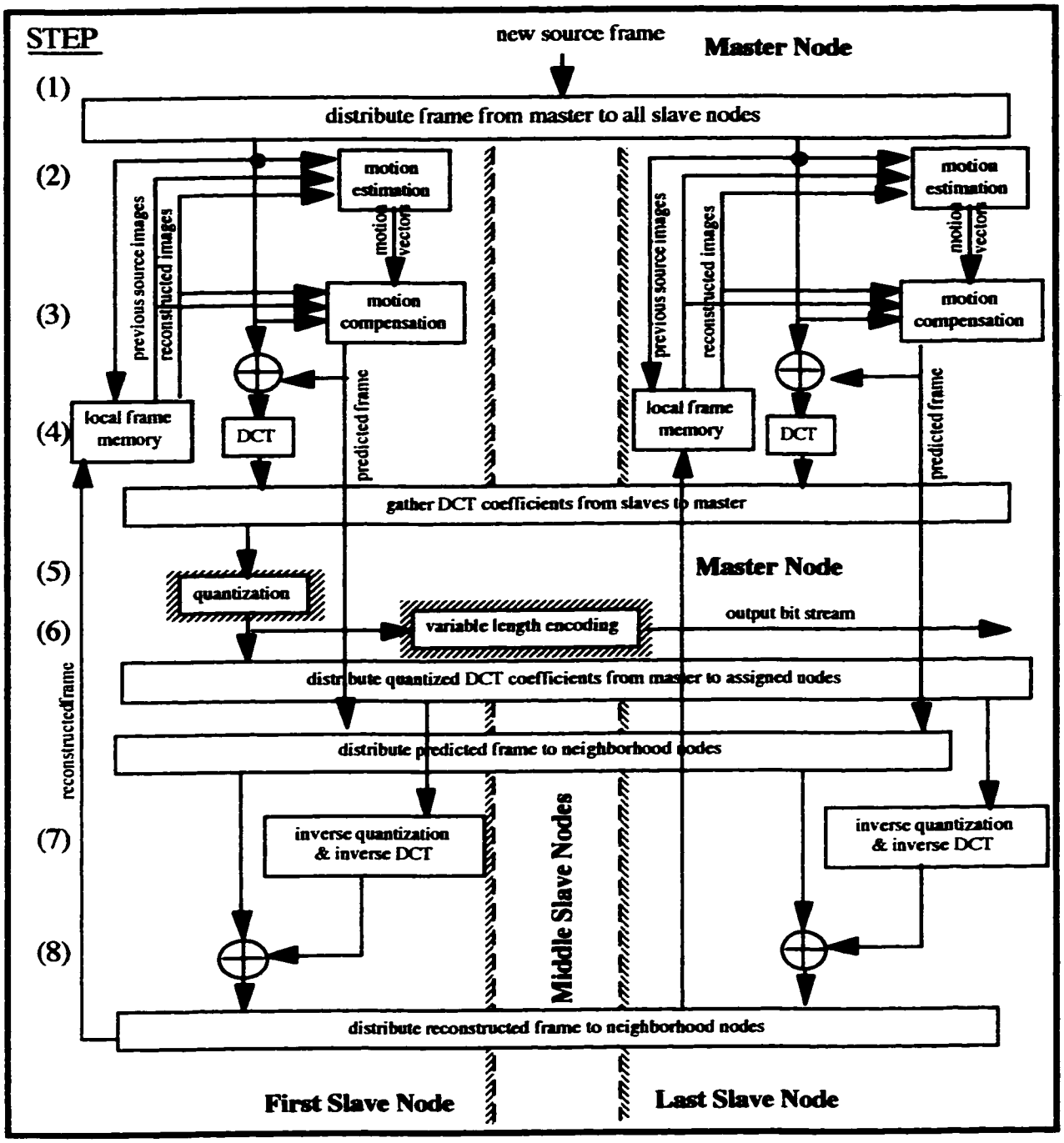

Figure 27. Parallel implementation block diagram

Once the motion has been estimated for each macroblock assigned to a node, motion compensation is performed (step 3). Motion compensation creates a new image, the predicted frame, that replaces each macroblock with pixel values derived from the matching regions in the reference images. The difference between the original frame and

the predicted frame is then broken into 8 x 8 pixel regions called blocks. The discrete cosine transform (DCT) is then performed on each block (step 4).

The DCT coefficients for each block must be sent to the master node for quantization. Quantization (step 5 in Figure 27) must be performed serially because the quantization threshold for each macroblock depends upon the average bit rate achieved for all previous blocks. The next step (6) is variable length encoding and assembly of the results into the output bit stream. Assembly of the data into the output bit stream is inherently a serial function. Also, the original source code performs quantization, variable length encoding, and bit stream assembly in one routine called putpict. Therefore, it is best to do all of these in the master node to minimize communication.

The quantized blocks must be decoded to create the reference frames for future use. This requires distribution of the quantized DCT coefficients to the appropriate nodes. Each node then can perform inverse quantization and inverse DCT for its assigned macroblocks (step 7). The result is added to the predicted frame to produce a new reference frame (step 8). This reference frame is the image that would be created after a receiver decoded the output bit stream.

Parallel Program Implementation Technique

The program structure shown in Figure 27 was used to test the hypothesis that innovations developed for specialized image processors can be used to improve performance of general-purpose computers when executing image-processing programs. As mentioned earlier, for the purposes of the research, the IBM SP2 was chosen as a representative general-purpose parallel computer. Theoretically, the IBM SP2 provides an almost ideal connection topology. Any node can communicate directly with any other node. There are enough paths through the network of switches to prevent communication between one pair of nodes from blocking communication between another pair of nodes.

The bandwidth of the communication network scales linearly with the number of nodes and the latency of the network is only microseconds. These features suggested that the SP2 would be an excellent testbed for testing the hypothesis.

To create an implementation of the program with the structure shown in Figure 27 the MPEG program was modified by inserting Linda `eval` instructions to launch multiple processes, each of which would execute on a different node of the SP2. The putseq subroutine was modified to provide conditional execution of the subroutines and data transfer processes depending on whether the code is executed on the master or a slave node. Linda `out`, `rd` and `in` instructions were added to the code to transfer data between the processes. Finally, each subroutine had to be modified so that each node only processes the data assigned to it. The Linda profile option was used with the display program ParaGraph to visualize the communication patterns and timing of program execution. The Linda documentation indicates that collection of the profile data adds less than one percent to the execution time, mostly at initiation and completion of communication activities.

Once the MPEG program had been modified for data-parallel execution, experiments were performed to determine optimal approaches for two remaining issues, (1) the method used to determine which macroblocks are processed by each node, and (2) the message sizes and structures used for communication between the nodes.

Parallel Implementation Task Distribution And Control

Selecting the method used to assign which macroblocks are processed by each node is analogous to the load balancing problem studied extensively in Computer Science. In this case a task could be to perform one of the operations shown in Figure 27 on a particular macroblock. On distributed-memory multi-computers, one important consideration is how to distribute the tasks so as to minimize the delays due to

communications. For some computers the communication topology has a particular data distribution pattern that minimizes communication delay time. For example, on a two-dimensional mesh or hypercube, data assignments are usually chosen so that adjacent regions of the images are on nodes with direct connections between them. This is not a problem for the SP2 because any node can communicate directly with any other node through the switching network. On the SP2 each parallel program is assigned a pool of nodes the size of which is part of the job submission request. Execution of a Linda `eval` statement in the program causes the operating system to spawn a new process on one of the nodes assigned to the job. The eval statement can include a set of variables that are passed to the new process. One of the variables can be an index used to uniquely identify the process.

Parallel Program Control

The simplest implementation tested provided dynamic assignment of macroblocks to the nodes. The form of dynamic load balancing is often used to assign tasks between nodes during parallel program execution. In *dynamic load balancing* the assignment of tasks to nodes is not pre-determined. Instead, some mechanism is used to assign tasks to each node based upon the overall progress as the parallel program executes. Dynamic load balancing can be used to improve the flexibility of a program so that it can run easily without special code to adapt to the number of available nodes or the size of the data set to be processed.

To implement the dynamic assignment, a token tuple was passed between the nodes with a count indicating the next macroblock to be processed. When a node completed processing a macroblock it would read the token from tuple-space, update the counter, and then process the macroblock with the index indicated by the token. This approach was inefficient because (1) each node must have the input data required to

process all macroblocks since it could be assigned any one, and (2) handling the token becomes a bottleneck when there are many nodes.

To overcome this bottleneck a deterministic formula was used to assign tasks to each worker. The deterministic formula assigns a contiguous slice of the image to each node. The number of macroblocks in the slice is determined by the number of macroblocks in the image and the number of nodes available for execution of the program. With this formula each node could calculate the indexes of the macroblocks assigned to it for processing.

Parallel Implementation Communication Protocols

There are 5 communication points in the parallel implementation block diagram shown in Figure 27 on page 110. The first communication point is to distribute each new image to the nodes. The second and third communication points are (1) to transmit the DCT coefficients computed in each node back to the master for quantization, and (2) to transfer the quantized DCT coefficients from the master back to the nodes. In the fourth and fifth communication points of the parallel program block diagram, intermediate results for neighboring regions must be transferred between nodes twice. First, the predicted frame and then the reconstructed frame must be transferred.

Three distinct communication patterns, illustrated in Figure 28, are required to implement the five communication points just described. The communication patterns included (A) transfer of frame data from the master to slaves, (B) transfer of DCT coefficients between the master and the slaves, and (C) transfer of frame data between slaves. These three communication patterns correspond to three major challenges often encountered when developing a parallel program structure for image-processing applications. The three challenges are: (1) the need to distribute a part of each new frame to each node, (2) the need to transfer intermediate results from surrounding regions to

complete processing of each region, and (3) the dependence upon global results part way through the program. How well a computer system design addresses these three concerns can have a major impact on the performance of the system.

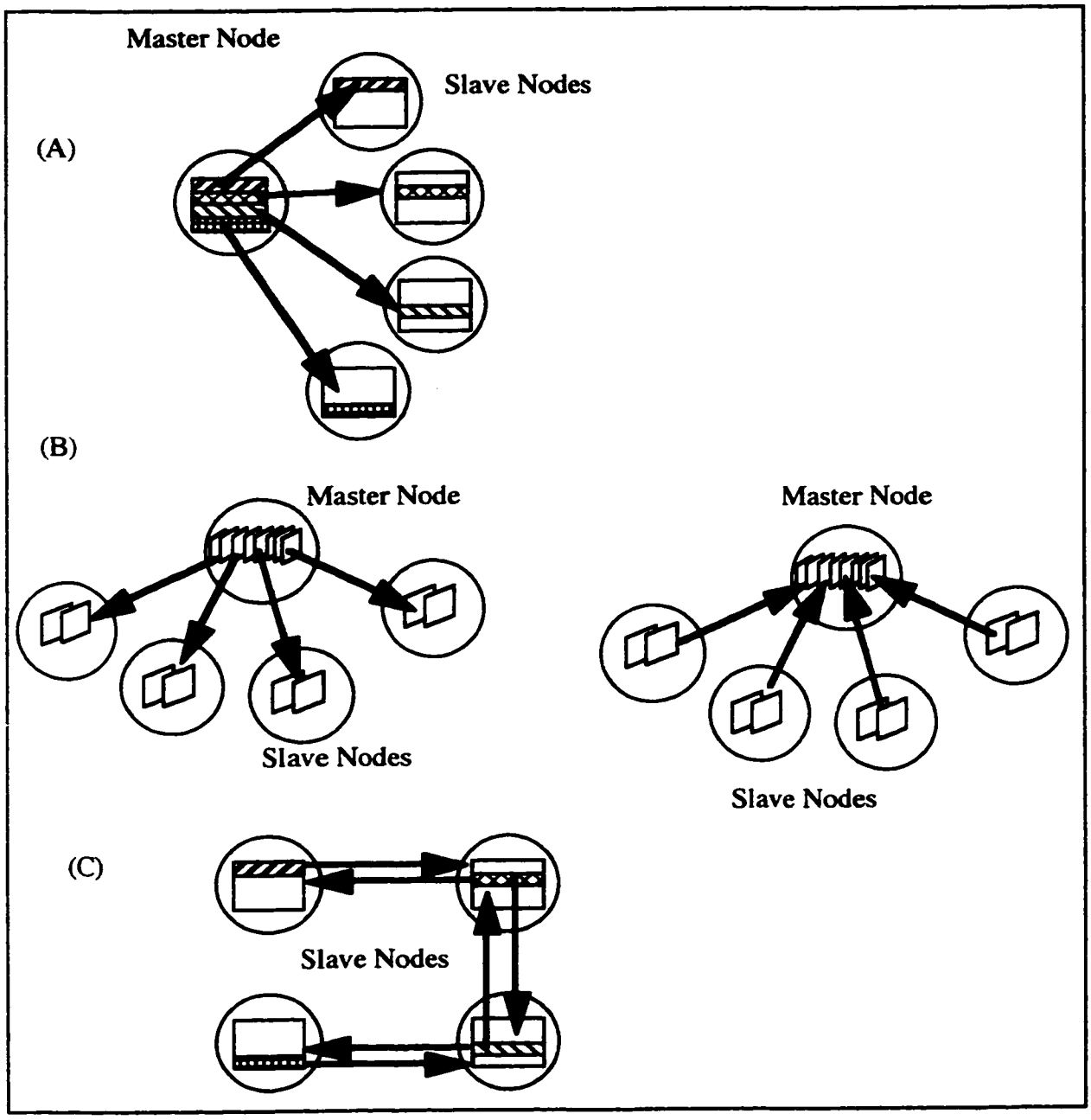

Figure 28. Communication patterns for parallel execution

The first communication pattern (A) was used to distribute each new source frame to the workers. The minimum amount of data required by each node is that required for motion estimation. The motion estimation routine searches for a best match between each macroblock in the current image and a macroblock sized region in reference images. The reference images are previously encoded frames. The size of the search region is specified in the parameter file mentioned in Chapter II. For the experiments reported here, the search window sizes were those provided in the sample parameter files acquired with the software. The largest search window, used to encode P-frames, was 11 by 11 pixels.

To test the hypothesis, it was first necessary to determine the minimum communication time achievable with the existing SP2 hardware and software and to compare this with the required communication rates. Each node needs access to the pixel data for the macroblocks assigned to it plus a region around each of these macroblocks. There are 126,720 bytes per frame in the video sequence used in the experiments. At 30 frames per second the master must transmit an average of 3.8 megabytes per second to distribute the appropriate parts of the source frames to the slave nodes.

Several techniques were tested that only transmitted the required parts of each new image to the nodes. Three distribution schemes were used: (1) sending all the information from the master to each worker, (2) sending only the information needed by each worker from the master, and (3) sending all the information to all the workers using a message tree.

The most efficient method to distribute the source image data was found to be when the entire image is transmitted to all of the nodes. This result was counter-intuitive and requires some explanation. On the surface, it seems more efficient to transfer only the required data to each node. For N nodes, only one-Nth of the data needs to be transferred to each node so the total time to distribute the data should be approximately the same time as transferring one frame from one node to another.

However, there are two factors that complicate the situation. First, there is a significant overhead associated with each message. Part of the reason for this is that, when the image is divided into regions, the data is not consecutive. To transfer the data it must be put into a single contiguous message buffer which requires extra data movement and record keeping. This is because the communication in the SP2 is implemented using a direct memory access (DMA) controller. A DMA controller is specialized hardware normally used for moving data between memory and other devices. In the IBM SP2 the DMA controller is controlled by system software and used to pass messages between the memory of a node and the communication network. The DMA controller in the SP2 can efficiently move data from one contiguous block of memory. If the data is not merged into one message buffer then extra messages are required. Second, each node requires data from the area around the macroblocks assigned to it. This increases record keeping and the amount of data to be transmitted. At the extreme, where each macroblock is assigned to a different node, each pixel must be transmitted from the master to as many as nine nodes.

Experimental results found that it is more efficient to transmit the entire image (which is stored in one contiguous block of memory) to all of the nodes using a tree structure. The message tree works as shown in Figure 29. The master sends all the data to the first worker. The master and the first worker then send the data to the second and third workers respectively. Then each of the four nodes that have the data sends it to one of the fourth-through-seventh nodes. This pattern repeats until all the nodes have received the required data. The time to perform this type of data transfer is proportional to the log of the number of nodes.

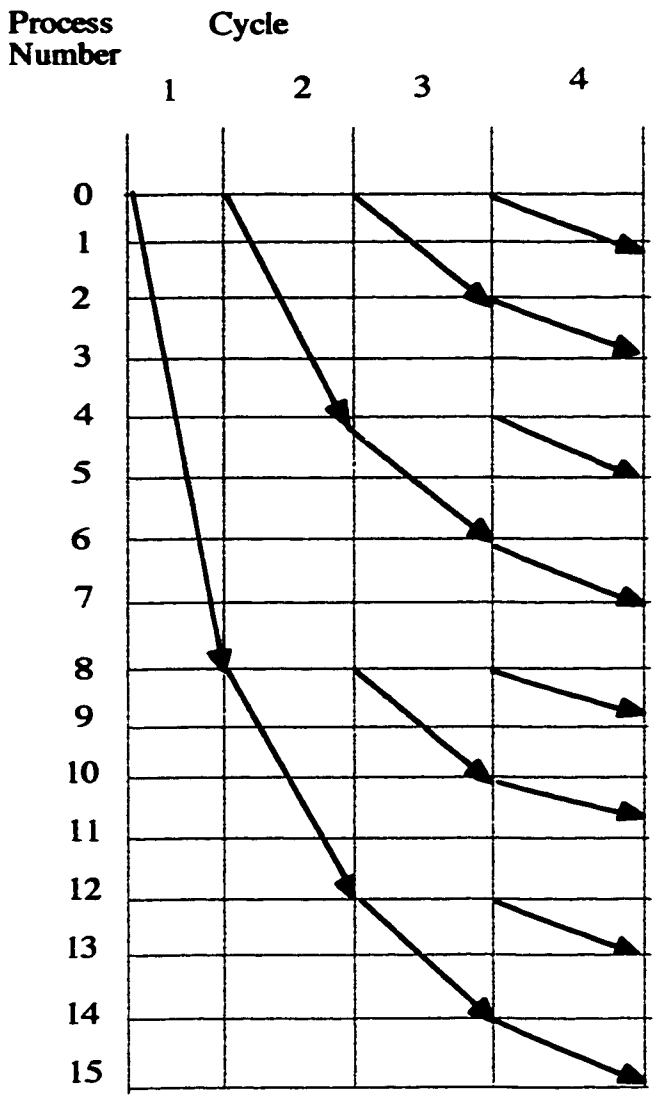

Figure 29. Message tree used to distribute source images to 16 nodes

The second communication pattern (B in Figure 28), and the simplest to implement, was for transfer of the DCT coefficients. There are the same number of DCT coefficients as there are pixels in the image. However, unlike the image data, the DCT coefficients for each macroblock are stored in a contiguous block of memory. This allows the transfer of each macroblock to be easily accomplished with a single message.

To perform these transfers in real time the master must receive and transmit an average 3.8 megabytes per second to perform each of these tasks. To implement this using Linda, the source node places the coefficients for each macroblock along with the

coordinates of transfer the macroblock in a tuple in tuple-space. The destination node then reads the macroblock from tuple-space. This mechanism was found to be efficient only when the destination of each tuple is explicit. This was accomplished by placing an integer in the tuple representing the process that was to receive the data. Execution of this type of data transfer is independent of the number of nodes since the time for the master node to receive or send all of the data is the limiting factor. These transfers involve the master either receiving or sending all of the messages and the number of messages is independent of the number of nodes used.

One challenge in coarse-grained data parallelization of image-processing applications is that the data and operations assigned to each node are rarely completely independent. That is, each node requires intermediate results from other nodes before it can complete its calculations. This problem occurs in the motion estimation and predict routines. The third communication pattern (C in Figure 28) is used to transfer intermediate results from each node to those nodes processing the adjacent macroblocks. If 60 nodes are used to achieve the speedup required then, for the 352 by 240 pixel images used in this study, each node would be assigned 5 or 6 macroblocks. The intermediate results for each macroblock would need to be transmitted to those nodes responsible for its neighbors. The worst case would require that a node transmit and receive data for 20 macroblocks each for the predicted frame and the reconstructed frame. At 384 bytes per macroblock and 30 frames per second this would require an average bandwidth for each node of 921,600 bytes per second.

To implement this data transfer the program was modified so each node determines which parts of the image it needs to send to each other node. For each macroblock assigned to a node, the program determines whether each of the adjacent macroblocks are assigned to a different node. If so, it creates a tuple containing the image data for the macroblock it has and sends it to the node responsible for that adjacent macroblock. To construct the tuple, the image data for the 16 by 16 pixel region is first

moved into a structure similar to that used to store the DCT coefficients for each macroblock. This structure places the image data in one contiguous block of memory. The macroblocks are then put into tuple-space, one macroblock per tuple, with an entry indicating the destination node. Once each node places its messages into tuple-space it calculates which macroblocks it should retrieve and reads them from tuple-space. The time to perform this type of data transfer is nearly constant.

Table XIX summarizes the bandwidth required just to transfer image data. It does not include other communication to transfer control information and synchronization. The communication channels of the SP2 can transfer 40 megabytes per second suggesting that a perfectly parallel implementation could achieve real-time execution. As will be shown in the next section, this rate was not achieved.

	Pattern from Figure 28	Bandwidth required
Distribute source frames from master to slaves	A	3.8 Mb/s
Gather DCT coefficients and distribute quantized DCT coefficients	B	7.6 Mb/s
Node to node transfers	C	1.8 Mb/s
Total		13.2 Mb/s

Table XIX. Minimum required communication bandwidth for real-time execution

Limits Due To Algorithms Used In The Original Source Code

Even when there is adequate bandwidth between the nodes, there are often factors that limit the speedup that can be achieved using data parallel approaches. The extent of the parallelization and subsequent execution-time reduction using data-parallel approaches is limited by a dependence on global results in the quantization algorithm. Analysis of the serial-code performance leads to a conclusion that coarse-grained functional parallelization cannot improve execution time enough to achieve real-time

encoding. It is not surprising that the MPEG encoding program used as a benchmark in this research has characteristics that limit how much of the program can be executed in parallel using data-parallel approaches. Amdahl (1967) first pointed out that the speed up that can be achieved using parallel processing is limited by the time required to execute those parts of a program that must be run serially. In image-processing applications, this problem is most acute when there is an image transformation that depends upon a global result. For example, this often occurs when a global threshold used to separate background from objects of interest depends upon analysis of the histogram of intensities from an entire frame. The potential impact of computer architecture innovations upon the decrease in execution time using data-parallel approaches is limited for programs implementing algorithms with these characteristics.

In this MPEG program a similar problem occurs in the putpict routine where there is a threshold based upon the prior compression ratio attained. The average bit rate achieved from compression of previous images and from previous parts of the current image is used to determine the quantization threshold for the next macroblock. The quantization threshold has a direct impact upon the compression ratio, so this feedback loop acts as a control structure to achieve the desired bit rate.

The result is that parallelization cannot fully achieve the throughput and latency requirements. The putpict subroutine must be assigned to a single node. The average execution time for this routine on a single node is about 0.2 seconds. This execution time leads to a lower bound on the latency of 0.2 seconds and an upper bound on the throughput of about 5 frames per second.

Clearly, minor changes to the algorithm would result in code that could be parallelized more completely. If the algorithm is changed so that the quantization threshold is updated based upon the bit rate achieved through the previous frame instead of through the previous macroblock, then the quantization of all the blocks in a frame could be performed in parallel. There are no apparent reasons why a viewer would

perceive the difference. However, this change would violate the constraints placed upon the research (i.e., that the algorithm could not be changed). Furthermore, it would change the results produced by the program and would, therefore, hinder use of the tests supplied with the source code to verify that the program is executing properly. The research, therefore, focused upon what could be learned without changing the algorithms used in the putpict routine.

Limits Due to Communication Hardware and Software

The communication topology is a fundamental system design decision for multi-computer systems. Two other important design considerations are (1) the interface between the nodes and the communication network, and (2) the features of the communication software.

It is instructive to compare the predicted time to transfer data based on bandwidth considerations with the actual times required. Table XX and Figure 30 summarize execution times achieved when 8, 16, and 24 nodes were used.

	Average Execution Time Per Frame (msec)					
Number of nodes	distribute	front end	putpict	2nd distribute	back end	Total
8	80	661	204	0.6	53	998.6
16	95	367	198	0.6	36	696.6
24	120	252	204	0.5	30	606.5

Table XX. Parallel implementation execution times

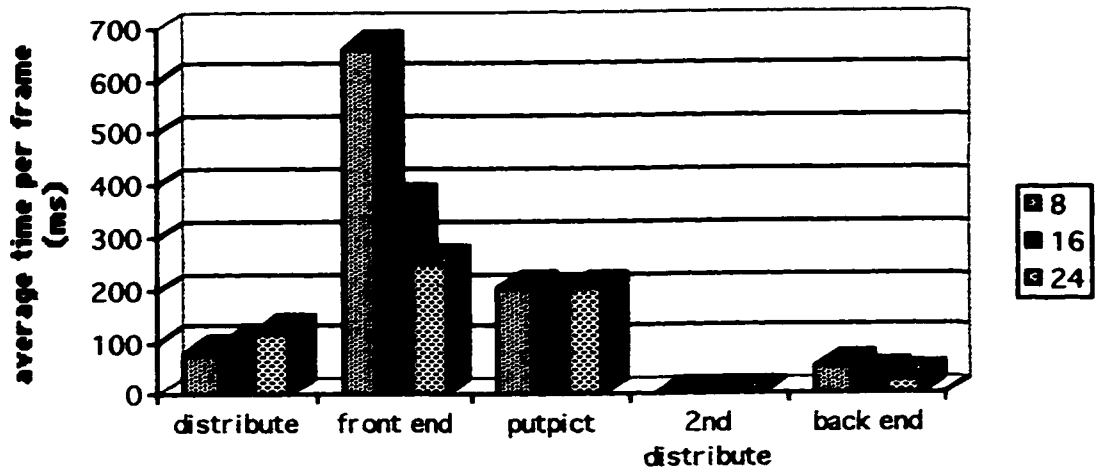

Figure 30. Parallel implementation execution times versus number of nodes used

The average execution time per frame was measured while compressing the full 150 frame pflowg sequence. The measurements were made by inserting timer software at critical points in the program. Referring to Figure 27 on page 110, the first time, distribute, includes the time from after an image was read from disk until it had been distributed to all the nodes. The second time, front end, includes steps 2 and 3, motion estimation and the discrete cosine transform. The third time, putpict, includes steps 5 and 6, quantization and variable length encoding. The fourth time, 2nd distribute, is the time to redistribute the quantized DCT coefficients from the master to their assigned nodes. The fifth time, back end, is for steps 7 and 8, inverse quantization, inverse DCT and formation of the reconstructed frame.

Note that for N processors, the time to distribute the image data is O(logN), the front end (which includes motion estimation) is O(1/N) and that the putpict time is almost constant. These observations are consistent with the timing relationships predicted previously. Also, note that the time for the second distribution is insignificant, even though it involves the same amount of communication as the first one. This is because the communication can overlap with the operations of the back end.

Evaluation of Hypothesized Architecture Improvements

Using the experimental results it is now possible to evaluate the impact of each of the potential improvements suggested in the hypothesis on page 105. The suggested improvements were (a) hardware to support systolic operation, (b) sophisticated image-oriented address generators, (c) simultaneous transfer of two or more images between pairs of processors, (d) multiple destinations when communicating image data, and (e) local reconstruction of neighborhood data from an image data stream.

a) Hardware to support systolic operation -- Systolic operation is a form of functional parallelism. The serial code analysis concluded that functional parallelism would not provide a significant improvement in execution times because the majority of program execution time is used by one subroutine, dist1. This situation, where much of the execution time is spent in a subroutine that does neighborhood operations, is common in image-processing applications. Therefore, this part of the hypothesis is not supported by the analysis.

b) Image oriented address generators -- The experiments suggest that this hypothesis is true. On the SP2 and similar MIMD machines, the transmission of parts of an image is most efficient if the data are in consecutive locations in memory. However, data parallel processing of images requires distribution of rectangular regions within the image to each of the nodes. To accommodate this, ATCURE includes a multi-level loop counting mechanism to automatically retrieve data from a region in an image. More complicated patterns can also be implemented. A hardware improvement such as that in ATCURE -- to make it easier to transmit rectangular regions in a multi-dimensional array -- would have eliminated the need to use the image distribution tree illustrated in Figure 29 on page 118.

c) Simultaneous transmission of multiple images between a pair of nodes -- There were no situations in the parallel implementation of this MPEG 2 encoding

program that required simultaneous transmission of multiple images between a particular pair of nodes. Therefore, this hypothesis was not supported by this research.

d) Multiple destinations for communications -- Distribution of data to multiple destinations occurs in three parts of the parallel implementation. The first is the distribution of the source frames to the slave nodes. The ability to designate multiple destinations would reduce the communication time. Note, also, that this enhancement would provide the same benefit for distribution of the source frames as the image oriented address generators discussed previously. Both could eliminate the need to use a distribution tree.

The second and third possible uses for multiple destinations are for the distribution of the predicted frame and the reconstructed frame to the nodes responsible for adjacent macroblocks. However, the timing data suggests that this is not a significant part of the required time for communication.

Therefore, this hypothesis is supported with the understanding that it is redundant with part (b) of the hypothesis.

e) Local reconstruction of neighborhoods from image streams -- Local reconstruction of neighborhood data is a benefit for systolic execution of image-processing applications. Since analysis of this code indicated that systolic processing was inappropriate, this part of the hypothesis is not supported.

CHAPTER VIII

SIMD ISA EXTENSIONS FOR IMAGE PROCESSING ON GENERAL PURPOSE COMPUTERS

The Subword Parallelism Approach

As previously stated, this research is partly based on the assumption that abstractions developed for image-processing languages can be applied to improve general-purpose computers so they execute image-processing applications in less time. Recent microprocessor developments provide an opportunity to evaluate this part of the hypothesis in a way that could not be done on the SP2. These recent enhancements seek to exploit the fact that many image-processing functions operate on low precision data. Most images are stored as one byte per pixel per color channel. As a result many operations can be performed using 8-bit or 16-bit arithmetic. However, until recently the trend in microprocessor technology was to provide ever wider data paths, registers, and functional units. This leaves much of the computational power of a conventional general-purpose microprocessor idle when performing image processing. One method to exploit the idle computational capability is to execute each instruction on several elements of the data simultaneously. The approach is sometimes called sub-word parallelism.

Subword parallelism is an SIMD approach that uses each data path, register, and functional unit to fetch, store, or operate upon several lower precision data elements simultaneously. With this approach a 64-bit register can store eight 8-bit, four 16-bit, or two 32-bit data elements. Similarly, the functional units are modified so that they execute each instruction on all, or some, of the data elements in a register simultaneously.

Thus, a 64-bit arithmetic unit can simultaneously perform the same instruction on eight, four, or two data elements of 8-bits, 16-bits, or 32-bits respectively. A 128-bit register or functional unit could hold or operate upon twice as many data elements of each precision. This approach also exploits another common characteristic of image-processing operations; the same sequence of instructions is often performed on many locations in the image and the computations for each location are independent of the results from other locations.

Several microprocessors that incorporate SIMD instructions have already been introduced. Intel introduced the MMX extensions for the Pentium and the P6 microprocessors. The MAX-2, VIS, and AltiVec SIMD instructions were introduced respectively in the HP-PA by Hewlett-Packard, the Ultrasparc, and the PowerPC by Motorola. Each of these instruction sets represents a different solution to a difficult set of design decisions. They were each developed based upon studies of the types of operations performed in multi-media, signal processing, and/or graphics applications.

Table XXI summarizes the instructions included in each Instruction Set Architecture (ISA) extension. Explanation of some of these instructions is necessary. One unusual feature of the ISA extensions is the saturation options. Saturation occurs when the result of an arithmetic operation exceeds a limit. When this occurs the result is forced to the limit. If signed arithmetic is used then the limits can be the largest positive integer and the most negative integer that can be represented by the number of bits in the data. For twos-compliment 8-bit data this would be +127 and -128. When unsigned arithmetic is used the limits are the largest positive integer and zero. For 8-bit data the largest positive integer would be 255. When saturation is not used it is up to the programmer to be sure that overflow does not occur. If it does occur, the results are computed as in modulo arithmetic, that is, the results wrap around from the largest to the smallest number.

Instruction Type	MMX	VIS	MAX-2
Processor Family	x86, Pentium, P6	UltraSparc	HP-PA
Addition	8, 16, 32 bits signed; 8, 16 bits signed saturate; 8, 16 bits unsigned saturate;	16, 32 bits signed	16 bit signed, signed saturated, unsigned saturated
subtraction	8, 16, 32 bits signed wrap; 8, 16 bits signed saturated; 8, 16 bits unsigned saturated;	16, 32 signed wrap	16 bit modulo, signed-saturated, unsigned-saturated
Shift	16, 32 bits logical left or right; 16, 32 bits arithmetic left		16 bit right, logical or arithmetic; 16 bit left 0 fill; align 128->64 bits
Bitwise Logic	64 bit and, and not, or, xor	not, and, or, xor, nand	
Constants		ones, zeros	
Compare	8, 16, 32 bit =, >	16, 32 =, !=, <=, >=	
Specialized	Multiply-Accum 16*16 + 16*16->32 bits	8 -> 64 bit sum of abs diff.	16 bit shift & add with & without saturation; 16 bit average;
Multiply	16*16 -> 16 bits high or low	8*16->16 high or low; 16*16->16 using 2 instructions; 16*16->32 using 2 instructions	
Shuffle (Unpack)	8, 16, 32 bits from high- or low-32 bits of sources	8 bit from low-32 bits of sources	16 bit from left or right of 32 bits; permute
Expand		8->16 bits	
Pack	8, 16 bits from 16, 32 signed-saturated or unsigned-saturated	32->8 with shift 32->16	
Store	32 bits low to integer register or memory; 64 bits to memory	partial, 8, 16, 32 bits with edge mask generation	
load	32 bits low from integer register or memory; 64 bits from memory aligned	64 bits unaligned with 2 instructions; 8*64 load w/o cache load non-blocking	cache hint for no reuse; prefetch
address operations		align, 8,16, 32 bit 3D array	
Move	64 bits	64 bits	align 128->64; extract; deposit;

Table XXI. Subword instruction sets

Another group of unusual instructions is the set associated with moving data between registers. These include the shuffle, pack and unpack instructions. These are used to convert data to different levels of precision, to combine the data from two registers into the subwords of a single register, or to separate the data from the subwords of one register into two registers.

In addition, the VIS ISA extension includes specialized load and store instructions to improve access times for data that is not aligned on boundaries consistent with the register data width. The specialized load and store instructions reduce the number of instructions required to move data between memory and the registers when the data location in memory is not at an address that is divisible by the register width.

Effectiveness of MMX ISA Extensions On MPEG2 Program Execution

A close look at the effectiveness of the MMX ISA extension set on MPEG 2 encoding demonstrates the strengths and weaknesses of the subword parallel SIMD approaches. In some circumstances, the subword parallel approach could have an eight-fold speedup for image-processing operations on 8-bit data. However, a detailed analysis shows less than half of this speedup can be achieved without completely rewriting the code.

In Chapter 5 we showed that execution of the dist1 subroutine requires about 75% of the execution time of the MPEG 2 software used for this study. Eq. (6), using the notation of Hatfield, Miner, and Wilkes (1991), summarizes the calculations.

$$\text{function dist1}\left(p_0[\mathbf{X},\mathbf{Y}], p_1(\mathbf{X},\mathbf{Y}), (x_0,y_0), (x_1,y_1)\right) \rightarrow \text{dst} : \mathbf{Z}^2 \times \mathbf{Z}^2 \times \mathbf{Z}^2 \times \mathbf{Z}^2 \rightarrow \mathbf{Z} \equiv$$
$$dst = \sum_{j=0}^{15 \text{ w(s min)}} \left(s = \sum_{i=0}^{15} \left| p_0(x_0+i, y_0+j) - p_1(x_1+i, y_1+j) \right| \right) \qquad (6)$$

The function dist1 is called once for each location in the reference image that is to be compared to the current macroblock in the source image. The pointers p_0 and p_1 point to the beginning of the reference and source frames respectively. The values (x_0, y_0) are the indexes for the starting location in the reference frame of the region that is to be compared with the source frame macroblock. The values (x_1, y_1) are the indexes for the macroblock in the source image. The value s is the sum of the absolute value of the differences of the 16 pixels in one row. The value dst is the sum of absolute differences

between two 16 by 16 pixel regions. The outer sum continues until either the previous minimum, min, has been exceeded or the full 16 by 16 summation has been completed. The indexes i and j are used to increment relative to the starting location for each region.

Note that there are two stages to the mathematical operations of this function. First, there are the operations that occur independently for each pixel location. These are the subtraction and the absolute value. The second stage is called a reduction operation. The sum of the results from the first stage produces a single result that summarizes the information from all of the pixels.

Table XXII shows one possible implementation using the MMX instruction set.

inst.	type	Instruction	Comment
		ra and rb point to 1st pixel of p0 and p1 respectively	at least 2 instructions
		mm7=0	mm7 contains zero used as a constant
		mm6=0x1111	mm6 contains four 16-bit values of 1
load data			
1	P2	lsi gpr1, (ra), 16	load 16 bytes (unaligned) into general purpose registers 1 & 2
2	mmx	movq mm0, gpr1	move 1st 8 bytes of p0 to mmx register 0
3	mmx	movq mm1, gpr2	move 2nd 8 bytes of p0 to mmx register 1
4	mmx	movq mm2, (rb)	load 1st 8 bytes of p1 to mmx register 2
5	mmx	movq mm3, (rb+8)	load 2nd 8 bytes of p1 to mmx register 3
compute absolute values of differences			
6	mmx	movq mm4 mm0	copy 1st 8 pixels of p0 to mmx register 4
7	mmx	psubusb mm4, mm2	subtract one way with saturation
8	mmx	psubusb mm2, mm0	subtract the other way
9	mmx	por mm2, mm4	or results to get absolute value of differences for 1st 8 pixels
10	mmx	movq mm4, mm1	copy 2nd 8 pixels of p0 to mmx register 4
11	mmx	psubusb mm4, mm3	subtract one way with saturation
12	mmx	psubusb mm3 mm1	subtract the other way
13	mmx	por mm3, mm4	or results to get absolute value of differences for 2nd 8 pixels
sum results			
14	mmx	movq mm0,mm2	copy 1st 8 partial results
15	mmx	punpckhbw mm0, mm7	unpack first 4 result bytes to words
16	mmx	punpcklbw mm2, mm7	unpack 2nd 4 result bytes to words
17	mmx	movq mm1, mm3	copy 2nd 8 pixels of results
18	mmx	punpckhbw mm1, mm7	unpack 3rd 4 result bytes to words
19	mmx	punpcklbw mm3, mm7	unpack 4th 4 result bytes to words
20	mmx	padduw mm0, mm2	add 1st and 2nd 4 words of results
21	mmx	padduw mm1, mm3	add 3rd & 4th group of results
22	mmx	padduw mm0, mm1	add to get 4 partial results
23	mmx	pmuaddwd mm0, mm4	multiply by 1 and add pairs
24	mmx	packssdw mm0, mm4	pack double words to words
25	mmx	pmuaddwd mm0, mm4	multiply by 1 and add pairs to get final result

Table XXII. dist1 implementation using MMX instructions

Several observations are important. First, it takes only 8 instructions to simultaneously compute the absolute value of the differences at the 16 pixel locations. The MMX instructions provide eight-fold parallelism for this calculation.

Second, it takes 12 instructions to sum the 16 pixels in a row. Currently, the most efficient instruction to perform the required summation is the multiply-accumulate instruction used above. It multiplies four pairs of 16-bit numbers stored in two different MMX registers and then adds two pairs of 32-bit products to produce two 32-bit sums. The instruction could be efficient for computing the product of two complex numbers but it is inefficient for summing the values in one register. Thus, while the instruction set extension provides full use of the parallelism while computing the difference and absolute value of the corresponding pixels, there is no such parallelism possible for the summation operation.

To perform a summation within a register, the MMX instruction set requires multiple shift and add operations. Thus, only about half of the instructions for the dist1 routine implemented using MMX actually use the full subword parallelism. The net result is that it takes 25 instructions using the MMX extensions as compared to 96 Power 2 instructions to perform the same calculation.

Currently the documentation for the MMX extension recommends that code such as that used in the dist1 routine be rewritten to exploit the SIMD ISA extensions more efficiently. The original source code computes the sums row-wise. The eight-fold improvement can be approached only by computing the summation column-wise, before adding between columns. This reduces the number of times that the sum of the values in a register must be computed. That is, the subtraction and absolute value operations in the inner loop can be performed eight at a time. The summation operation must take place column-wise before the horizontal summation occurs.

There are two problems with this solution. First, it is not mathematically identical to the original code because the original code checks the current partial sum against the

previous minimum after each summation across the row. Second, reorganizing the computations from row-wise to column-wise may not be readily accomplished with automatic compilation from the serial code. It is likely that a programmer would have to rewrite the source code to take achieve much of the expected speedup with MMX instructions.

Improved SIMD Extensions For Image Processing Applications

This example highlights some weaknesses found in all of the subword parallelism ISA extensions. Specialized SIMD architectures can be very efficient for image processing. However, efficient use of the specialized architecture requires computer code adapted for that processor. A more general theoretical foundation is required for the choice of instructions implemented in the ISA extensions. The abstractions and theoretical developments in image algebra described in Chapter II can provide that foundation. It is important to repeat that not published articles were found to suggest that the theoretical foundations of image algebra were considered in the development of the sub-word parallel instruction sets.

The AFATL image algebra treats images as a distinct data type and includes image-image and image-template operations. As explained in Chapter II, the image-image operations involve pairs of corresponding pixels from two images. The arithmetic operations performed on each pair of pixels are common arithmetic functions such as addition or multiplication. In contrast, the template operations work on neighborhoods. They include a pixel by pixel operation within a small region of the image followed by a reduction operation. This is repeated at each location in the image. For example, in convolution, the sum of products is computed. The pixel by pixel operation is multiplication and the reduction operation is addition. The other two template operations

that are part of the minimum set defined by the image algebra are max of sums and max of products.

Image algebra suggests features required for efficient image-processing application execution. The image-image operations are supported by individual instructions in the subword parallel ISA extensions. In each of the ISA extensions, the pixel by pixel operations can be performed in parallel. However, the image-template operations are not well supported. In particular, they are not efficient at performing the reduction operations, such as summation or maximum, on the contents of a single register. In addition, the extensions assume that data access is aligned on boundaries consistent with the length of the registers. However, the image-template operations, by their very nature, require non-aligned image data access as they access a neighborhood centered around each pixel of the image. So non-aligned access is critical for one of the fundamental operations that is the most computationally intensive part of many image-processing application programs.

The need for these additional instructions is a key insight provided by this work. The required instructions are the sum of the subwords in a register and the maximum of the subwords in a register. It should be noted that there are other common reduction operation used in median filtering and rank-order filters. An instruction that sorts the data in the subwords of a register could be used for all of the logical reduction operations -- maximum, minimum, median, or any other rank-order filter.

It is now possible to estimate the impact of an improved SIMD instruction set extension on the execution of the dist1 subroutine in particular and to extend that evaluation to the potential impact on the execution of the entire MPEG2 program. If the MMX instruction set included a more robust way to load unaligned data and to sum the values in a register, it could execute the code in the dist1 subroutine more efficiently. If a sum instruction were included in the SIMD instruction set then it would take only 3 instructions to compute the sum of the sixteen values left in registers mm2 and mm3 after

instruction 13 in Table XXII. Even if the sum instruction took two clocks to execute, it would only take 17 clock cycles to compute the sum of the absolute differences in a row of 16 pixels.

Table XXIII shows one possible implementation if the Power2 processor included an SIMD instruction set that included the MMX instructions, instructions similar to the load unaligned instructions available in VIS, and reduction instructions such as sum and sort.

inst.	type	Instruction	Comment
		set ra and rb to 1st pixel of p0 and p1 respectively	at least 2 instructions
		mm7=0	mm7 contains zero used as a constant
		mm6=0x1111	mm6 contains four 16-bit values of 1
load data			
1	P2	lsi gpr1, (ra), 16	load 16 bytes (unaligned) into general-purpose registers 1 & 2
2	mmx	movq mm0, gpr1	move 1st 8 bytes of p0 to mmx register 0
3	mmx	movq mm1, gpr2	move 2nd 8 bytes of p0 to mmx register 1
4	mmx	movq mm2, (rb)	load 1st 8 bytes of p1 to mmx register 2
5	mmx	movq mm3, (rb+8)	load 2nd 8 bytes of p1 to mmx register 3
compute absolute values of differences			
6	mmx	movq mm4 mm0	copy 1st 8 pixels of p0 to mmx register 4
7	mmx	psubusb mm4, mm2	subtract one way with saturation
8	mmx	psubusb mm2, mm0	subtract the other way
9	mmx	por mm2, mm4	or results to get absolute value of differences for 1st 8 pixels
19	mmx	movq mm4, mm1	copy 2nd 8 pixels of p0 to mmx register 4
11	mmx	psubusb mm4, mm3	subtract one way with saturation
12	mmx	psubusb mm3 mm1	subtract the other way
13	mmx	por mm3, mm4	or results to get absolute value of differences for 2nd 8 pixels
sum results			
14	new	sum mm2	sum 1st 8 partial results
15	new	sum mm3	sum 2nd 8 partial results
16	mmx	paddw mm2, mm3	sum partial results to get total

Table XXIII. dist1 implementation using improved SIMD instruction set

The impact on the dist1 implementation of adding reduction operations to the ISA extensions is that the number of instructions drops to 16. This is compared with 25 instructions using the MMX instruction set and 96 instructions using only the Power2 instruction set. Thus, the enhanced SIMD ISA instruction set can provide a speedup by a factor of six over the standard Power2 instruction set.

Impact of Subword Parallel ISA Extensions on MPEG2 Execution Times

The potential impact of subword parallel ISA extensions on execution times for image-processing applications can now be estimated using the MPEG2 image sequence encoding program as an example. Table XXIV shows estimated execution time improvements for three alternative enhancements to the Power 2 processors used for this study.

	66 MHz Power 2 CPU	Power 2 with ISA extensions	500 MHz Power2 CPU	500 MHz Power 2 with ISA extensions
read frame	32	32	4	4
motion estimation	1750	292	231	39
predict	18	18	2	2
transform	216	72	29	10
putpict	181	181	24	24
inverse quantize	29	29	4	4
inverse transform	48	16	6	2
Total	2274	581	300	83

Table XXIV. Potential execution times (msec per frame) for alternative CPU enhancements

The first column of the table shows the average execution time for processing one frame using a single node of the SP2 used in the experiments. The best implementation using the Power2 instruction set required, on the average, 2.275 seconds per frame, 77% of which was spent in the dist1 routine. To achieve real-time encoding of these 240 x 352 pixel images using 66 MHz Power 2 CPUs would require 69 processors assuming perfect speedup. The second column shows the improvements if the proposed enhanced MMX instruction set were added to the Power 2 CPUs in the SP2. It is assumed that the enhanced instructions would be as effective at improving the execution times for the DCT transforms as it was shown to be for the motion estimation execution times. In this

case the average execution time per frame on a single node drops to 581 ms. With just this improvement, real-time encoding on an SP2 could be achieved with 18 nodes.

The third column shows the improvement if only the clock rate of the processors on the nodes were improved, without any instruction set changes. With a 500 MHz CPU the average execution time could be projected to be 300 ms per frame. This improvement alone would bring the number of nodes required for real-time execution down to ten. Finally, if both the clock speed increases and the instruction set is improved, the execution time is projected to reduce to 83 ms per frame. At that rate only three nodes would be needed to perform real-time encoding of the test sequence.

CHAPTER IX

CONCLUSIONS

The hypothesis advanced by the research reported in this dissertation is that abstractions developed for image-processing languages and special-purpose image-processing computers can be applied to improve general-purpose computers. The specific abstractions considered were those of the AFATL image algebra developed by Ritter, et al. (1987). These include the concept of an image as a fundamental data type, and the definition of a minimum set of image-image and image-template operations.

Both experimental and analytical methods were used to evaluate the hypothesis. Experiments were performed using a specific MPEG2 image sequence encoding program running on an IBM SP2 computer. The MPEG2 software was shown to have computational characteristics that are representative of a wide variety of image-processing application programs. The IBM SP2 has characteristics that are representative of high-end distributed-memory message-passing multi-computers. Analytical methods were used to project the impact of changes to the instruction set of the microprocessor in each node and changes to the communication network between the nodes.

Three approaches to increased functionality within a single node were used to evaluate the hypothesis. These include modification of the integer functional units, modification of floating point functional units, and subword parallelism. Measurements from execution of the program on a single node showed that nearly 75% of the computational time was in a single subroutine that performs an image-template operation. It was argued that this type of operation is often the most computationally intensive part of image-processing application programs. Therefore, improvements to a computer's

ability to perform this type of operation are the most beneficial to improving image-processing application-program execution times. It was shown that execution times for image-template operations are substantially limited by the ability to perform integer operations with relatively low precision (8 or 16 bits). It was shown that the Power2 microprocessor used in the IBM SP2 could be improved by increasing the number of integer functional units or providing floating point units that could also perform integer operations. Subword parallel instruction set extensions were also found to provide much of the functionality required for parallel execution of the fundamental functions required for the image algebra operations. However, two significant shortcomings were identified in the current subword parallel instruction sets. First, the image-template operations require non-aligned access to memory and this type of access is not well supported. Second, the image-template operations include a reduction operation, that is, a calculation whose operands are some or all of the contents of one of the SIMD registers. The ability to compute the sum or to find the maximum of the content in one of the registers would have a significant impact. In addition, the ability to efficiently sort the contents of one of the SIMD registers would improve performance on some very common image-processing functions.

Five features that support the image algebra abstractions, and that were previously incorporated into special-purpose image processors, were used to evaluate the implications of the hypothesis on the communication structure of general-purpose distributed memory multi-computers: (a) hardware to support systolic operation, (b) sophisticated image-oriented address generators, (c) simultaneous transfer of two or more images between pairs of processors, (d) multiple destinations when communicating image data, and (e) local reconstruction of neighborhood data from an image data stream.

Evaluation of these alternatives was accomplished by analyzing measurements made from the execution of the program on a single node and then on multiple nodes. Only two of the five hardware features suggested were projected to have a significant

impact on the execution times. These were image oriented address generators, and multiple destinations for communication. Either of these could significantly reduce the time required to distribute each new frame to the nodes that are assigned to process each part of the data. Without specialized hardware, sending the regions of each image assigned to each node is very time consuming. Either each region is sent a row at a time, requiring a high number of messages, or each rectangular region must be reorganized into a single message buffer, requiring extra memory accesses and record keeping. A third approach, using a message tree to distribute the whole image to each worker, has the drawback that the communication time increases with the number of nodes used. Image oriented address generators would allow more efficient access to the rectangular parts of an image that have to be transmitted to each node. The need to reorganize the data in memory would be eliminated if the DMA controllers used in the communication hardware had the ability to move rectangular regions in a 2 dimensional array as a single message. Alternatively, the problem would be eliminated if the entire image could be sent to all of the receiving nodes at once.

Several other significant conclusions can be drawn from the experiments. It is important to acknowledge the complexity required to create the parallel version of the program. First, the right granularity of parallelization had to be selected to achieve compatibility with the throughput and latency requirements of the application. Then, five different subroutines (motion estimation, predict, transform, inverse quantize, and inverse transform) had to be modified so that they all use the same distribution of the data amongst the nodes. The record keeping and insertion of data transfers between nodes was far from trivial. Simple schemes are inefficient. Complex schemes are prone to overlook transfer of needed data. Furthermore, careful attention and timing studies are required to make sure the overhead for data transfer do not overwhelm the program execution time.

Another observation is that the coarse-grain parallelism is limited by the program structure, as it includes a significant section that can not be executed in parallel without algorithmic changes. Therefore, real-time execution will also require improved fine grained parallelism or clock rate improvements. In addition, this research has significant implications for future developments in SIMD instruction sets for microprocessors and for the communications functions of MIMD computers. Future work should proceed with more detailed simulations of the proposed improvements to determine the extent to which they achieve the reduction in speed suggested by the analysis reported here.

The conclusions developed from this research prove the hypothesis that abstractions developed for image algebra and special-purpose image processors can be applied to improve general-purpose computers. The improvements suggested by the theory are applicable to many image-processing applications and can be incorporated into many different types of general-purpose computers.

The research reported here suggests there are additional insights that future research might derive from additional evaluation of the theoretical and practical foundation. For example, translation-dependent and data-dependent template operations included in the AFATL image algebra are important for implementation of some transforms such as the Fourier transform. Research similar to that reported here but using an application that requires translation-dependent templates might identify other features critical to improving future general-purpose computers. In addition, many theorems have been derived using the image algebra. Future research should explore the application of these concepts to determine whether they suggest additional enhancements that are generally applicable to further improve the execution times of image-processing applications programs.

REFERENCES

REFERENCES

Adams, Charlotte. 1991. Chasing the Elusive Gigaflop In a Soup Can. *Military and Aerospace Electronics*, February.

Agerwala, T., J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias, and M. Snir. 1995. SP2 System Architecture. *IBM Systems Journal* 34, no. 2 - Scaleable Parallel Computing, Reprint Order No. G321-5563, http://www-1.almaden.ibm.com/journal/sj/agerw/agerw.html

Akiyama, T., H. Aono, K. Aoki, K.W. Ler, B. Wilson, T. Araki, T. Morishige, H. Takeno, A. Sato, S. Nakatani, and T. Senoh. 1994 MPEG2 Video Codec Using Image Compression DSP. *IEEE Transactions on Consumer Electronics*. 40, no. 3 (August): 466-472

Almasi, George S., and Allan Gottlieb. 1989. *Highly Parallel Computing.* Redwood City, California: Benjamin/Cummings.

Alsford J., P. Dewar, J. Illingworth, J. Kittler, J. Lewis, K. Paler, P. Wilde and W. Thomas. 1985. CRS Image Processing Systems with VLSI Modules. In *Image Processing System Architectures.* ed. Josef Kittler and Michael J.B. Duff, 49-81. New York: John Wiley & Sons.

Amdahl, G.M. 1967. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings* held in Atlantic City, NJ, 18-20 April 1967, 483-485. Reston, Virginia: AFIPS Press.

Araki, et al., 1992. The Architecture of a Vector Digital Signal Processor for Video Coding. *T. Proc. of ICASSP 92*. vol. 5 V-681, Mar.

Araki, T., et al., 1994. Video DSP Architecture for MPEG2 CODEC. *Proc. of ICASSP-4* April.

Baer, Jean-Loup. 1980. *Computer Systems Architectures*. Rockville, Maryland: Computer Science Press.

Ballard, Dana H., and Christopher M. Brown. 1982. *Computer Vision.* Englewood Cliffs, New Jersey: Prentice-Hall.

Balmer, K., N. Ing-Simmons, P. Moyse, I. Robertson, J. Keay, M. Hammes, E. Oakland, R. Simpson, G. Barr, and D. Roskell. 1994. A Single Chip Multimedia Video Processor. In *Proceedings of the IEEE 1994 Custom Integrated Circuits Conference.* 91-94.

Barr, Avron and Edward A. Feigenbaum, eds. 1982. *The Handbook of Artificial Intelligence.* Los Altos: William Daufman, Inc.

Carrara, Walter G., Ron S. Goodman, and Ronald M. Majewski. 1995. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms.* Boston: Artech House.

Carriero, Nicholas and David Gelernter. 1989. Linda in Context. *Communications of the ACM* 32, no. 4 (April): 444-458.

Cheng, Ching-Min, Chien-Hsing Wu, Soo-Chang Pei, Hungwen Li, and Bor-Shenn Jeng. 1994. High Speed Video Compression Testbed. *IEEE Transactions on Consumer Electronics* 40, no. 3: 538-547.

Duin, Robert P.W. and Frans A. Gerritsen. 1985. The Delft Image Analysis Laboratory: A Multi-user Facility for Research, Development and Education of Image Analysis Methods. In *Image Processing System Architectures,* ed. Josef Kittler and Michael J.B. Duff, 127-152. New York: John Wiley & Sons.

Duncan, Ralph. 1990. A Survey of Parallel Computer Architectures. *IEEE Computer*. 23, no. 2 (February): 5-16

Edwards, M.D. 1985. A Review of MIMD Architectures for Image Processing. In *Image Processing System Architectures,* ed. Josef Kittler and Michael J.B. Duff, 127-152. New York: John Wiley & Sons.

Flynn, Michael J. 1966. Very High-Speed Computing Systems. *Proceedings of the IEEE* 54 (December):1901-9.

Flynn, Michael J. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* C-21 (September): 948-59.

Fontain, T.J. 1985. A Review of SIMD Architectures. In *Image Processing System Architectures,* ed. Josef Kittler and Michael J.B. Duff, 3-22. New York: John Wiley & Sons.

Gelernter, David. 1988. Getting the Job Done. *Byte* 13, no. 12 (Nov): 301-310.

Hatfield, Donald, Richard A. Miner and C. Thomas Wilkes. 1991. *A Mathematical Expression Language for Imaging Applications.* Online Document available from http://www.wildfire.com/~miner/thesis/journal/journal.html; Internet; accessed 22 February 1998.

Hord, R. Michael. 1993. *Parallel Supercomputing In MIMD Architectures.* Boca Raton: CRC Press.

International Organisation for Standardisation and the International Electrotechnical Commission. 1994. *Information Technology-Generic Coding of Moving Pictures and Associated Audio Recommendation H.262 ISO/IED 13818-2.* ISO/IEC, 1994

ISO. 1992. *Digital Compression and Coding of Continuous-Tone Still Images: ISO/IEC Draft International Standard 10918-1.* January 10, 1992

ISO. 1993. *MPEG2, Coding of Moving Pictures and Associated Audio.* ISO/IEC JTC/SC29, CD 13818-2. November 1993.

ISSCC. 1994. A Video DSP with a Macro-block-Level-Pipeline and a SIMD type Vector-Pipeline Architecture for MPEG2 CODEC. *Digest of Technical Papers, ISSCC94.* Feb. 1994 [ ]

Jurgen, Ronald K. 1992. Digital Video. *IEEE Spectrum,* March, 24-30.

Kim, D., J. Young, S. Milton, H.J. Kim and Y. Kim. 1994. A Real-Time MPEG Encoder Using a Programmable Processor. *IEEE Transactions on Consumer Electronics.* 40, no. 2 (May): 161-170.

Lee, Woobin, Jeremiah Golston, Robert J. Gove, and Yongmin Kim. 1994. Real-Time MPEG Video Codec on a Single-chip Multiprocessor. In *Digital Video Compression On Personal Computers: Algorithms and Technologies: Proc. SPIE 2187,* ed. Arturo A. Rodriguez, 32-42. Bellingham, Washington: SPIE-The International Society for Optical Engineering.

Lougheed, Robert M. 1985. A High Speed Recirculating Neighborhood Processing Architecture. In *Architectures and Algorithms for Digital Image Processing II: Proc. SPIE Vol. 534,* ed. Francis J. Corbett 22-33. Bellingham, Washington: SPIE-The International Society for Optical Engineering.

Normile, James, Dan Wright. 1991. Image Compression Using Coarse Grain Parallel Processing. In *1991 International Conference on Acoustics, Speech, and Signal Processing: Speech Processing 2: VLSI Underwater Signal Processing* held in Toronto, Canada, May 14-17, 1991, by the Institute of Electrical and Electronics Engineers, Signal Processing Society, 1121-24. **IEEE**.

Pass, Stephen. 1985. The GRID Parallel Computer System. In *Image Processing System Architectures*, ed. Josef Kittler and Michael J.B. Duff, 23-35. New York: John Wiley & Sons.

Rich, Elaine. 1983. *Artificial Intelligence.* New York: McGraw-Hill Book Co.

Ritter, Gerhard X., and Paul D. Gader. 1987. Image Algebra Techniques for Parallel Image Processing. *Journal of Parallel And Distributed Computing.* 4: 7-44.

Ritter, G.X., and J.N. Wilson. 1987. Image Algebra In A Nutshell. *First International Conference on Computer Vision* held in London. IEEE.

Salinger, Jeremy. 1995. ATCURE: A Heterogeneous Processor for Image Recognition. In Proceedings of the ICASSP 1995 IEEE. Get complete citation from Ackenhusen

Salinger, Jeremy, and John Ackenhusen. 1993. Heterogeneous computer architecture for embedded real-time automatic target recognition. *ADPA National Avionics Symposium*, November

Salinger, Jeremy, and R. Michael Hord. 1995. ATCURE: Heterogeneous computer architecture for real-time image information analysis. In *Proceedings of the 23rd AIPR Workshop: Image and Information Systems: Applications and Opportunities SPIE 2368* held in Washington, DC 12-14 October 1994, 229-234. Bellingham, Washington: Society of Photo-Optical Instrumentation Engineers.

Salinger, Jeremy. 1994. Heterogenous Computer Architecture for Embedded Real-Time Image Interpretation. In *Proceedings of the Society of Photo-Optical Instrumentation Engineers (SPIE) International Symposium on Optical Engineering and Photonics in Aersopace and Remote Sensing Vol. 1957.* Bellingham, Washington: Society of Photo-Optical Instrumentation Engineers.

Savation, Tristan. 1998. *MPEG Pointers and Resources.* MpegTV. available from http://www.mpeg.org; Internet; last accessed April, 1998

Schmitt, Lorenz A., and Stephen S. Wilson. 1988. The AIS-5000 Parallel Processor. *IEEE Trans. PAMI* 10, no. 3 (May): 320-330.

Shippy, D.J., T.W. Griffith, and Gerodie Braceras. 1994. *POWER2 Fixed-Point, Data Cache, and Storage Control Units.* [on-line document] IBM. last accessed 4 May 1998. available from http://www.rs6000.ibm.com/resource/technology/fxu.html

Siegel, Shep. 1985. VME Building Blocks Make Image Processing a Snap. *Digital Design*, October, 48-52.

Siewiorek, Daniel P., C. Gordon Bell, and Allen Newell. 1982. *Computer Structures: Principles and Examples.* New York: McGraw-Hill.

Snir, Marc, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1996. *MPI: The Complete Reference.* Cambridge: MIT Press.

Unger, S.H. 1958. A Computer Oriented Towards Spatial Problems. *Proceedings of the IRE* 46:1744-50.

Webb, Jon A. and Takeo Kanade. 1986. Vision on a Systolic Array Machine. In *Evaluation of Multicomputers for Image Processing,* ed. L. Uhr, K. Preston, Jr., L. Levialdi, and M.J.B. Duff 181-201. Orlando: Academic Press

White, Steven W., and Sudhir Dhawan. n.d. *POWER2: Next Generation of the RISC System/6000 Family.* available from http://ibm.tc.cornell.edu/ibm/pps/power/power2/index.html; Internet; accessed 7 August 1998.

Yamauchi, Hironori, Yutaka Tashiro, Toshihiro Minami, and Yutaka Suzuki. 1992. Architecture and Implementation of a Highly Parallel Single-Chip Video DSP. *IEEE Trans. Circuits and Systems for Video Technology* 2, no. 2 (June): 207-220.

Yanbin, Yu, and Dimitris Anastassiou. 1994. Software Implementation of MPEG-II video Encoding Using Socket Programming in LAN. In *Digital Video Compression on Personal Computers; Algorithms and Technologies, SPIE Vol. 2187,* edited by A. Rodriguez, 229-240. Bellingham, Washington: Society of Photo-Optical Instrumentation Engineers.