NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
High-Speed Multiprocessor Simulation of VBR Video Coding and Transmission through an ATM Network

by

Nabeel M. Ghuzlaan, B. Sc.
Jordan University of Science and Technology

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

March 19, 1993

© 1993 Nabeel M. Ghuzlaan
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L’auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L’auteur conserve la propriété du droit d’auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
**Subject Categories**

### THE HUMANITIES AND SOCIAL SCIENCES

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications and the Arts</td>
<td></td>
</tr>
<tr>
<td>Architecture</td>
<td>079</td>
</tr>
<tr>
<td>Art History</td>
<td>054</td>
</tr>
<tr>
<td>Cinema</td>
<td>060</td>
</tr>
<tr>
<td>Dance</td>
<td>078</td>
</tr>
<tr>
<td>Fine Arts</td>
<td>057</td>
</tr>
<tr>
<td>Information Science</td>
<td>072</td>
</tr>
<tr>
<td>Journalism</td>
<td>061</td>
</tr>
<tr>
<td>Library Science</td>
<td>069</td>
</tr>
<tr>
<td>Mass Communications</td>
<td>070</td>
</tr>
<tr>
<td>Music</td>
<td>041</td>
</tr>
<tr>
<td>Speech Communication</td>
<td>065</td>
</tr>
<tr>
<td>Theatrical</td>
<td>065</td>
</tr>
</tbody>
</table>

### EDUCATION

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>051</td>
</tr>
<tr>
<td>Administration</td>
<td>054</td>
</tr>
<tr>
<td>Adult and Continuing</td>
<td>056</td>
</tr>
<tr>
<td>Agricultural</td>
<td>057</td>
</tr>
<tr>
<td>Art</td>
<td>077</td>
</tr>
<tr>
<td>Bilingual and Multilingual</td>
<td>026</td>
</tr>
<tr>
<td>Business</td>
<td>160</td>
</tr>
<tr>
<td>Community College</td>
<td>075</td>
</tr>
<tr>
<td>Curriculum and Instruction</td>
<td>077</td>
</tr>
<tr>
<td>Early Childhood</td>
<td>058</td>
</tr>
<tr>
<td>Elementary</td>
<td>024</td>
</tr>
<tr>
<td>Finance</td>
<td>077</td>
</tr>
<tr>
<td>Guidance and Counseling</td>
<td>049</td>
</tr>
<tr>
<td>Health</td>
<td>160</td>
</tr>
<tr>
<td>Higher</td>
<td>045</td>
</tr>
<tr>
<td>History of</td>
<td>050</td>
</tr>
<tr>
<td>Home Economics</td>
<td>078</td>
</tr>
<tr>
<td>Industry</td>
<td>052</td>
</tr>
<tr>
<td>Language and Literature</td>
<td>019</td>
</tr>
<tr>
<td>Mathematics</td>
<td>040</td>
</tr>
<tr>
<td>Music</td>
<td>057</td>
</tr>
<tr>
<td>Philosophy of</td>
<td>016</td>
</tr>
<tr>
<td>Physical</td>
<td>051</td>
</tr>
</tbody>
</table>

### LANGUAGE, LITERATURE AND LINGUISTICS

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>061</td>
</tr>
<tr>
<td>Classical</td>
<td>024</td>
</tr>
<tr>
<td>Comparative</td>
<td>024</td>
</tr>
<tr>
<td>Medieval</td>
<td>022</td>
</tr>
<tr>
<td>Modern</td>
<td>021</td>
</tr>
<tr>
<td>Negro</td>
<td>026</td>
</tr>
<tr>
<td>African</td>
<td>026</td>
</tr>
<tr>
<td>American</td>
<td>051</td>
</tr>
<tr>
<td>Canadian (English)</td>
<td>052</td>
</tr>
<tr>
<td>Canadian (French)</td>
<td>053</td>
</tr>
<tr>
<td>English</td>
<td>051</td>
</tr>
<tr>
<td>German</td>
<td>051</td>
</tr>
<tr>
<td>Italian</td>
<td>032</td>
</tr>
</tbody>
</table>

### PHILOSOPHY, RELIGION AND THEOLOGY

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>042</td>
</tr>
<tr>
<td>Biblical Studies</td>
<td>021</td>
</tr>
<tr>
<td>Classical</td>
<td>049</td>
</tr>
<tr>
<td>History of Philosophy</td>
<td>022</td>
</tr>
<tr>
<td>Philosophy of</td>
<td>049</td>
</tr>
</tbody>
</table>

### SOCIAL SCIENCES

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>American Studies</td>
<td>023</td>
</tr>
<tr>
<td>Anthropology</td>
<td>024</td>
</tr>
<tr>
<td>Cultural</td>
<td>026</td>
</tr>
<tr>
<td>Business Administration</td>
<td>031</td>
</tr>
<tr>
<td>Accounting</td>
<td>022</td>
</tr>
<tr>
<td>Banking</td>
<td>070</td>
</tr>
<tr>
<td>Marketing</td>
<td>038</td>
</tr>
<tr>
<td>Canadian Studies</td>
<td>035</td>
</tr>
<tr>
<td>Economics</td>
<td>050</td>
</tr>
<tr>
<td>General</td>
<td>050</td>
</tr>
<tr>
<td>Agricultural</td>
<td>052</td>
</tr>
<tr>
<td>Commerce and Business</td>
<td>054</td>
</tr>
<tr>
<td>Finance</td>
<td>058</td>
</tr>
<tr>
<td>History</td>
<td>051</td>
</tr>
<tr>
<td>Labor</td>
<td>050</td>
</tr>
<tr>
<td>Theory</td>
<td>051</td>
</tr>
<tr>
<td>Urban and Regional Planning</td>
<td>099</td>
</tr>
<tr>
<td>Women's Studies</td>
<td>045</td>
</tr>
</tbody>
</table>

### THE SCIENCES AND ENGINEERING

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agriculture</td>
<td>047</td>
</tr>
<tr>
<td>Astronomy</td>
<td>038</td>
</tr>
<tr>
<td>Animal and Veterinary</td>
<td>057</td>
</tr>
<tr>
<td>Anatomy</td>
<td>049</td>
</tr>
<tr>
<td>Anesthesiology</td>
<td>052</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
<td>077</td>
</tr>
<tr>
<td>Biochemistry</td>
<td>051</td>
</tr>
<tr>
<td>Biology</td>
<td>036</td>
</tr>
<tr>
<td>Biophysics</td>
<td>016</td>
</tr>
<tr>
<td>Biotechnology</td>
<td>070</td>
</tr>
<tr>
<td>Body Mechanics</td>
<td>050</td>
</tr>
<tr>
<td>Botany</td>
<td>016</td>
</tr>
<tr>
<td>Carbohydrates</td>
<td>015</td>
</tr>
</tbody>
</table>

### PHYSICAL SCIENCES

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chemistry</td>
<td>067</td>
</tr>
<tr>
<td>General</td>
<td>077</td>
</tr>
<tr>
<td>Agricultural</td>
<td>052</td>
</tr>
<tr>
<td>Agricultural</td>
<td>077</td>
</tr>
<tr>
<td>Analytical Chemistry</td>
<td>092</td>
</tr>
<tr>
<td>Biomedicine</td>
<td>051</td>
</tr>
<tr>
<td>Biophysics</td>
<td>050</td>
</tr>
<tr>
<td>Biology</td>
<td>036</td>
</tr>
<tr>
<td>Biophysics</td>
<td>036</td>
</tr>
<tr>
<td>Biotechnology</td>
<td>070</td>
</tr>
<tr>
<td>Chemistry</td>
<td>050</td>
</tr>
<tr>
<td>Economic Sciences</td>
<td>033</td>
</tr>
<tr>
<td>Environmental Sciences</td>
<td>078</td>
</tr>
</tbody>
</table>

### HEALTH AND ENVIRONMENTAL SCIENCES

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environmental Sciences</td>
<td>068</td>
</tr>
<tr>
<td>Health Sciences</td>
<td>056</td>
</tr>
<tr>
<td>Audiology</td>
<td>030</td>
</tr>
<tr>
<td>Chemotherapy</td>
<td>052</td>
</tr>
<tr>
<td>Dental</td>
<td>057</td>
</tr>
<tr>
<td>Education</td>
<td>050</td>
</tr>
<tr>
<td>Hospital Management</td>
<td>079</td>
</tr>
<tr>
<td>Human Development</td>
<td>058</td>
</tr>
<tr>
<td>Immunology</td>
<td>059</td>
</tr>
<tr>
<td>Medical and Surgical</td>
<td>064</td>
</tr>
<tr>
<td>Mental Health</td>
<td>057</td>
</tr>
<tr>
<td>Nursing</td>
<td>059</td>
</tr>
<tr>
<td>Nutrition</td>
<td>070</td>
</tr>
<tr>
<td>Obstetrics and Gynecology</td>
<td>038</td>
</tr>
<tr>
<td>Ophthalmology</td>
<td>034</td>
</tr>
<tr>
<td>Pathology</td>
<td>051</td>
</tr>
<tr>
<td>Pharmacology</td>
<td>040</td>
</tr>
<tr>
<td>Physical Therapy</td>
<td>032</td>
</tr>
<tr>
<td>Public Health</td>
<td>053</td>
</tr>
<tr>
<td>Radiology</td>
<td>054</td>
</tr>
<tr>
<td>Respiration</td>
<td>057</td>
</tr>
<tr>
<td>Speech Pathology</td>
<td>060</td>
</tr>
<tr>
<td>Toxicology</td>
<td>033</td>
</tr>
<tr>
<td>Veterinary Science</td>
<td>078</td>
</tr>
<tr>
<td>Zoology</td>
<td>047</td>
</tr>
</tbody>
</table>

### PSYCHOLOGY

<table>
<thead>
<tr>
<th>Subject Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior</td>
<td>061</td>
</tr>
<tr>
<td>Clinical</td>
<td>062</td>
</tr>
<tr>
<td>Developmental</td>
<td>063</td>
</tr>
<tr>
<td>Experimental</td>
<td>064</td>
</tr>
<tr>
<td>Industrial</td>
<td>066</td>
</tr>
<tr>
<td>Personality</td>
<td>063</td>
</tr>
<tr>
<td>Physiological</td>
<td>069</td>
</tr>
<tr>
<td>Psychobiological</td>
<td>034</td>
</tr>
<tr>
<td>Psychodynamics</td>
<td>045</td>
</tr>
<tr>
<td>Social</td>
<td>045</td>
</tr>
</tbody>
</table>
The undersigned recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

“High-Speed Multiprocessor Simulation of VBR Video Coding and Transmission through an ATM Network”

Submitted by

Nabeel M. Ghuzlaan, B. Sc.

in partial fulfillment of the requirements for the degree of Master of Engineering.

Thesis Supervisor

Thesis Co-supervisor

Chair, Department of Systems and Computer Engineering

Carleton University
March 19, 1993
Abstract

Evaluation of the real-time behaviour of video coding algorithms and their robustness against ATM channel impairments normally requires computationally demanding and time-consuming simulations. The visual effect of channel impairments is even more difficult to simulate. This thesis investigates the impact of using multiprocessing architectures for these tasks. Parallel and pipelined architectures are configured using a representative multiprocessing system. A representative image coding algorithm is used to study the general behaviour of coding algorithms in multiprocessing environments. Data blocking and distribution techniques and interprocessor data transfer methods are examined and optimal methods are proposed depending on the nature of the underlying coding algorithm used. Spatial blocking data distribution methods are found to be more suitable for interframe coding in simple implementations; temporal blocking methods, for intraframe coding and more complicated implementations. Pipelined architectures are found to be better for high-level image processing operations. Star architectures are found to be optimal for less complicated processing operations and large implementations. An example system architecture design is presented based on the results of the experiments conducted. Using such architectures raises the possibility of economical, real-time simulation.
Acknowledgements

I am deeply indebted to my thesis supervisors Dr. A. R. Kaye and Dr. R. A. Goubran for their patience, help, guidance and encouragement throughout this program. Thanks to Dr. D. C. Coll, Mr. T. Pearce, Mr. F. Idris and Mr. M. Tharif for their time and help. Dr. S. A. Mahmoud’s advice and encouragement was indispensable. Help from the department staff, Darlene, Dina and Vivienne is highly appreciated. The video coding algorithm was suggested by Mr. R. Thomas. The Telecommunications Research Institute of Ontario provided financial support for this research project. Special thanks to Dr. A. U. H. Sheikh, Dr. N. Georganas, Dr. A. Abu-el-Haija and Dr. M. Irshid, without whose help I would not have had the opportunity to be in this program. Thanks to my mother, family, Sami Iqneibi and other friends for their help and encouragement. Most special thanks are due to God for His continuous care.
To Sami Iqneibi
Table of Contents

Acceptance Sheet  ii
Abstract  iii
Acknowledgements  iv
List of Tables  x
List of Figures  xi
Acronyms and Glossary  xii

1 Introduction  1
   1.1 Problem Statement  2
   1.2 Objectives  2
      1.2.1 Class of Algorithms  2
      1.2.2 Class of Architectures  3
   1.3 Motivation and Scope  3
   1.4 Approach  3
   1.5 Organization  4

2 Background  5
   2.1 Multiprocessor Architectures  5
      2.1.1 Computer Classifications  5
      2.1.2 Mesh Architectures  6
      2.1.3 Pyramid Architectures  7
      2.1.4 Hypercube Architectures  7
      2.1.5 Shared-Memory Architectures  8
      2.1.6 Systolic Arrays  8
      2.1.7 Hybrid Architectures  9
   2.2 Asynchronous Transfer Mode  9
      2.2.1 Priority Mechanisms  10
      2.2.2 Packet Video in the ATM Environment  10
   2.3 Video Compression Algorithms  10
      2.3.1 Lossless Coding  10
         2.3.1.1 Huffman Coding  11
         2.3.1.2 Lempel-Ziv Coding  11
Table of Contents

2.3.1.3 Arithmetic Coding .................................................. 12
2.3.1.4 Run-Length Coding .................................................. 13
2.3.2 Lossy Coding ......................................................... 13
  2.3.2.1 Transform Coding ............................................... 13
  2.3.2.2 Combined Transform Coding .................................... 14
  2.3.2.3 Predictive and Interpolative Coding .......................... 14
  2.3.2.4 Subband Coding ................................................. 15
  2.3.2.5 Vector Quantization ............................................ 15
  2.3.2.6 Second-Generation Compression Schemes ..................... 15
  2.3.2.7 JPEG and MPEG Standards ..................................... 16

2.4 Video Compression for ATM ............................................ 17

3 Experimental System ................................................... 18
  3.1 Simulation System .................................................... 18
    3.1.1 Compression Algorithm ........................................ 18
    3.1.2 Algorithm Implementation ..................................... 21
    3.1.3 ATM Adaptation Layer Protocol ................................. 23
    3.1.4 ATM Channel Model ............................................. 23
  3.2 Banshee Multiprocessor Board ...................................... 24
    3.2.1 The C30 Processor .............................................. 24
      3.2.1.1 Pipeline Operation ........................................ 25
      3.2.1.2 Provision for Multiprocessing ............................ 25
    3.2.2 The Banshee-Host Interface ................................... 26
      3.2.2.1 Software Interface ........................................ 26
      3.2.2.2 Hardware Interface ....................................... 26
      3.2.2.3 The Banshee-Host Communication .......................... 26
    3.2.3 The Banshee Multiprocessor Interface ......................... 27
      3.2.3.1 Interprocessor Communication ............................. 28

4 Single Processor Performance ........................................ 29
  4.1 Personal Computer Performance ................................... 30
  4.2 Single Banshee Board Performance ................................. 33

5 Multiprocessor Performance .......................................... 37
  5.1 Star Architecture .................................................. 38
# Table of Contents

5.1.1 Spatial Data Blocking ................................................................................. 39
5.1.2 Temporal Data Blocking ........................................................................... 40
5.2 Pipelined Architecture .................................................................................. 41
  5.2.1 Program-Driven Communication .............................................................. 43
  5.2.2 Program-Driven DMA Communication ..................................................... 44
  5.2.3 Interrupt-Driven DMA Communication .................................................... 45
  5.2.4 Buffered Interrupt-Driven Communication .............................................. 46
  5.2.5 Delayed, Buffered, Interrupt-Driven Communication ............................. 47
  5.2.6 Reducing Buffering Overhead .................................................................. 47
5.3 Hybrid Architecture ....................................................................................... 48
5.4 Summary of Results ....................................................................................... 50

6 Extension of the Architecture ........................................................................... 53
  6.1 MISD Systems ............................................................................................... 53
    6.1.1 Functional Segmentation ....................................................................... 54
    6.1.2 Communication Management ................................................................. 55
    6.1.3 Performance of MISD Systems ............................................................... 55
  6.2 SIMD Systems .............................................................................................. 57
    6.2.1 Data Segmentation ................................................................................ 57
    6.2.2 Synchronization and Communication .................................................... 59
    6.2.3 Performance of SIMD Systems ............................................................. 59
  6.3 Hybrid Systems ............................................................................................ 61
    6.3.1 Hybrid System Design ......................................................................... 62
    6.3.2 Performance of Hybrid Architectures ................................................... 63
  6.4 Performance Comparison ............................................................................. 65
  6.5 Example Design ......................................................................................... 66
    6.5.1 System Architecture Design ................................................................. 66
      6.5.1.1 Data Acquisition System Specifications ........................................... 67
      6.5.1.2 Data Dispensation System Specifications ........................................... 68
      6.5.1.3 Host System Specifications .............................................................. 68
  6.6 Interframe Predictive and Interpolative Coding ........................................... 69
  6.7 Colour Coding .............................................................................................. 70
  6.8 Special Considerations .................................................................................. 71
Table of Contents

6.9 Concluding Remarks .................................................. 73

7 Conclusions ............................................................... 75
  7.1 Results and Contribution .......................................... 75
  7.2 Future Work .......................................................... 77

A Visual Illustration of the Algorithm ................................. 78
  A.1 Frequency Band Division .......................................... 78
    A.1.1 Low Frequency Band Component .............................. 79
    A.1.2 Medium Frequency Band Component .......................... 79
    A.1.3 High Frequency Band Component ............................. 81
  A.2 Data Transmission Cell Loss Effects ............................ 81
    A.2.1 High-Priority Cell Loss Errors .............................. 82
    A.2.2 Low-Priority Cell Loss Errors ............................... 82

Bibliography .............................................................. 83
List of Tables

Table 2-1: Arithmetic Coding ......................................................... 12
Table 4-1: Coding Algorithm Performance on Different Systems .......... 29
Table 4-2: Personal Computer Time Cost of Subtasks ....................... 32
Table 4-3: Banshee System Time Cost of C Program Subtasks .......... 34
Table 4-4: Banshee System Time Cost of Assembly-Language Program Subtasks .... 36
Table 5-1: Comparison of CPU and DMA Transfer Schemes ............... 44
Table 5-2: Biprocessor Comparative Performance ............................. 51
List of Figures

Figure 2-1: Mesh Architecture ................................................................. 6
Figure 2-2: Systolic Architecture ............................................................... 9
Figure 3-1: Illustration of Terminology ....................................................... 19
Figure 3-2: MPI Block Diagram ................................................................. 27
Figure 4-1: Comparative Performance Bar Graph ......................................... 30
Figure 5-1: System Architecture ............................................................... 37
Figure 5-2: Star Configuration ................................................................. 39
Figure 5-3: Spacial Block Distribution Scheme ............................................ 40
Figure 5-4: Temporal Block Distribution Scheme ......................................... 41
Figure 5-5: Pipeline Configuration ............................................................ 42
Figure 5-6: Piping Line Groups ................................................................. 42
Figure 5-7: Data Paths in Hybrid Architectures ........................................... 49
Figure 5-8: Overload Sharing ................................................................. 49
Figure 5-9: Biprocessor Comparative Bar Graph ......................................... 51
Figure 6-1: Video Processing Set-up .......................................................... 53
Figure 6-2: General Pipeline System ......................................................... 54
Figure 6-3: DMA and CPU Timing ............................................................ 56
Figure 6-4: Temporal Distribution ............................................................. 58
Figure 6-5: Spatial Distribution ................................................................. 58
Figure 6-6: Algorithm Arrow Diagram ....................................................... 63
Figure 6-7: Example of Hybrid Systems .................................................... 64
Figure 6-8: Pipelined and Star Performance ............................................... 65
Figure 6-9: Data Acquisition System .......................................................... 67
Figure 6-10: Data Dispensation System ...................................................... 68
Figure 6-11: VLC Decoder ........................................................................ 72
Figure A-1: Original Picture ................................................................. 78
Figure A-2: Picture Superpel Component ............................................... 79
Figure A-3: Picture Triad Component ......................................................... 80
Figure A-4: Picture High-Priority Component ............................................ 80
Figure A-5: Picture Differential Pel Component ......................................... 81
Figure A-6: Picture with Cell Loss Errors ............................................... 82
Acronyms and Glossary

AAL
ATM adaptation layer

AC
alternating current (here refers to a variable whose average is zero)

ATM
Asynchronous Transfer Mode

barrel shifter
a data shifter where combinatorial logic is used to shift the bits to achieve single cycle operation regardless of the number of shifts per operation

BISDN
Broadband Integrated Services Digital Network

CAM
content-addressable memory

CBR
constant bit rate

CPM
critical-path method

CPU
central processing unit

DC
direct current (here refers to the average of any signal)

DCT
discrete cosine transform

DMA
direct memory access

DOS
disk operating system

DPCM
differential pulse code modulation

DSP
digital-signal processing/processor
**EPROM**
erasable programmable read-only memory

**FFMR**
tree-frame-memory register

**Harvard architecture**
architecture having separate data and program memory spaces

**HDTV**
high-definition television

**high-level vision**
the image processing task that involves object recognition and interpretation

**I/O**
input/output

**intermediate-level vision**
the image processing task that involves segmenting images into objects and describing them

**Kbyte**
kilobyte (1024 8-bit bytes)

**Kword**
kiloword (1024 16-bit/32-bit words)

**low-level vision**
the image processing task that involves image acquiring, enhancing and noise suppression

**MA**
memory access

**Mbyte**
megabyte (1,048,576 bytes)

**MFLOPS**
mega floating-point operations per second

**MIMD**
multiple-instruction stream, multiple-data stream

**MIPS**
million instructions per second
Acronyms and Glossary

MISD
    multiple-instruction stream, single-data stream
MS
    Microsoft Corporation
PC
    personal computer
PE
    processing element
pel
    picture element, also pixel
PERT
    project evaluation and review technique
PLA
    programmable logic array
PU
    processing unit
RAM
    random-access memory
resource leveling
    lowering maximum resource requirements by shifting a noncritical activity between
    its maximum allowable limits
RISC
    reduced-instruction-set computer. This architecture has simple and fast basic
    instructions.
RLC
    run-length code
ROM
    read-only memory
round robin
    arranging elements in a circular queue and servicing them in that order
SBC
    subband coding
Acronyms and Glossary

SDU
service data unit

SSCI
synchronous serial communications interface

SIMD
single-instruction stream, multiple-data stream

SISD
single-instruction stream, single-data stream

SRAM
static random-access memory

TV
television

VBR
variable bit rate

VLC
variable-length code

VQ
vector quantization
Introduction

The Broadband Integrated Services Digital Network (BISDN) is gradually replacing the current wideband networks\textsuperscript{[1-2]}. One network will carry multimedia traffic of various characteristics including data, voice, audio, graphics, video and image for telephony, television, conferencing, computing and other services\textsuperscript{[3-4]}.

Asynchronous Transfer Mode (ATM) will be used for switching those services\textsuperscript{[5]}. It combines elements of circuit and packet switching\textsuperscript{[6]} and conserves bandwidth by allocating it on a statistical basis\textsuperscript{[6, 7]}. This mode of bandwidth allocation will cause the bandwidth allocated to fall short of the required peak bandwidth of bursty sources\textsuperscript{[7]} and may cause network nodes to be swamped with data and to drop cells\textsuperscript{[8]}. For real-time services, excessively delayed cells are also considered lost\textsuperscript{[9]} and retransmission cannot be considered.

Variable bit-rate (VBR) video transmission, which is more bandwidth-efficient than constant bit-rate video transmission\textsuperscript{[10]}, will suffer from the cell loss problem\textsuperscript{[5]}. What can be done, in this regard, is to devise video coding and decoding schemes that are more robust against the cell loss problem\textsuperscript{[10]}. Simulating those coding schemes in the presence of ATM cell losses is necessary to evaluate, subjectively, the effect of errors on the reconstructed image because there is no objective means of doing so.

Since a uniprocessor system is slow to perform the simulation efficiently, this thesis investigates the possibility of designing a multiprocessor system to achieve real-time simulation of the coding schemes classified in Section 1.2.1.
1.1 Problem Statement

VBR video frames undergo source coding, channel coding, channel impairments, channel decoding and source decoding before the final frames can be viewed. The amount of processing involved, especially in the compression and decompression of video images, makes it impossible for general purpose computer systems to be able to simulate the complete process in real time, especially for research purposes.

At the time this thesis commenced, a large, state-of-the-art computer system was being used for video-to-video simulation of a coding scheme alone without channel errors, resulting in a 24-hour processing time for 2 seconds of video transmission. However, this method provided the required flexibility for development of new algorithms. VLSI implementations of some video algorithms are available, but they cannot be used to investigate the efficiency and robustness of new algorithms. Furthermore, demand on picture quality and resolution is always increasing.

1.2 Objectives

An objective of this thesis is to examine the feasibility and efficiency of using multiprocessor architectures to study the performance of video coding algorithms in ATM networks. A further objective is to evaluate various processor interconnection topologies and algorithm partitioning schemes in improving video processing.

The purpose is to speed up simulation while maintaining flexibility in the system for modifying the algorithm and changing simulation parameters. The possibility of achieving real-time performance is examined. It is desirable that the system be inexpensive and require minimum technical knowledge to be used and expanded as a research tool for video coding algorithms.

1.2.1 Class of Algorithms

The class of algorithms examined lends itself well to spatial data blocking. It can process blocks of the same image frame with minimum access requirement to remote picture elements in other blocks (in other processors). Most video coding algorithms fit in this class (Section 2.3), but the class excludes interframe coding algorithms. The generalization of the results to interframe and colour coding schemes is addressed at the end of the thesis (Sections 6.6–6.7).
1.2.2 Class of Architectures

The multiprocessor systems considered in this study include boards that can share a high-speed memory communication channel with a suitable host in a host/slave configuration. These systems are programmed and controlled through the host, which provides them with power supply and input/output (I/O) services and controls their operation.

The processing power of a board is furnished by an on-board high-performance processor which has local memory and I/O peripherals on-chip or on-board. Many such boards exist, based upon processors like Motorola DSP56000 and DSP0600, Intel i860, Texas Instruments TMS320C30 and TMS320C40 families, and they can be interconnected within themselves (e.g. in a ring) in addition to their connection with the host. This class of architectures provides cost-effective, high-performance processing power; it is flexible and its use does not require much technical knowledge (see Figure 5-1 for a 2-board system example).

1.3 Motivation and Scope

This thesis aims to analyze the impact of using a multiprocessor system as the test bed of a VBR video coding and transmission algorithm on the processing time required to carry out the end-to-end frame transformation. Analysis of improvement of the processing time of the algorithm may lead to the design of a multiprocessor system that could carry out the end-to-end video processing in real time so that the resultant video images can be viewed as if they were on a television (TV) system without storage requirements.

This project is motivated by the emergence of the BISDN network. Efficient video compression, channel-coding and error-correction methods should be developed before the BISDN is in operation. These VBR video processing algorithms should increase speed and conserve bandwidth while preserving picture quality.

1.4 Approach

A simple, typical image-compression algorithm was used to carry out the source coding and decoding of the raw digital video bits. The images that were used to carry out the experiments were in 256-grey scale, every pixel being represented by one byte (octet).

First the frames were subjected to a simple subband decomposition scheme which transformed the image into three streams differing in resolution. The highest-resolution lowest-priority stream was compressed through run-length coding and then further
compressed using Huffman coding. The first two streams were merged into one stream and the resulting two streams were subjected to ATM channel impairments. Cell loss rates differed for each stream to reflect the use of cell-loss priority in the ATM network. Finally, the two streams that were received at the end of the ATM channel were used to reconstruct the destination picture. The high-priority stream was used to reconstruct the more important features of the image and then the low-priority stream was Huffman-decoded, run-length-decoded and superimposed on the low-resolution image to achieve the final image.

The above process was initially distributed between two processors, each carrying half of each image to be processed. The improvement achieved over the uniprocessor case was then projected for more processors to estimate the required number of processors to carry out the simulation in real time.

1.5 Organization

The next chapter will provide some background necessary for this study and will examine what has been done in related areas. A review of multiprocessor architectures, ATM and digital video compression is presented and a brief literature survey follows.

The third chapter addresses the problem to be solved. It provides more detailed information about the specific compression and channel models used in the simulation and presents the hardware and software system used in the experiments.

The fourth chapter presents the performance of conventional single processor systems for comparison; performances of a 386-based personal computer (PC) and a single processing board are compared.

The fifth chapter analyzes the performance of multiprocessor boards. The video processing load is distributed between two processors, and improvements on performance are analyzed.

The sixth chapter extends the performance of the multiprocessor system to cases where more processors are used to arrive at the design of the real-time simulation system. Alternative designs are suggested.

The thesis concludes with a summary of results and suggestions for future research work.
2

Background

This chapter reviews multiprocessor architectures, ATM, and video compression algorithms. The chapter concludes with a section on research done on video coding for the ATM environment.

2.1 Multiprocessor Architectures

Due to the demand on processing power, central processing unit (CPU) designs incorporate instruction decode overlap, multiple functional units, pipelining\(^{111}\) and Harvard architectures\(^{12}\) to speed up processing, but this provides limited improvements. There are several ways to gather and connect processors together to perform tasks faster and more efficiently. The optimal architecture of a multiprocessor system depends on the type of task tackled. This section gives a general review of multiprocessor architectures.

The multiprocessor architectures discussed below can be built easily using transputers, which are high-performance microprocessors, designed to facilitate interprocess and interprocessor communication\(^{13}\). Processes are the fundamental transputer software building blocks, and they can be directly implemented in hardware\(^{14}\), with the hardware management being expressed in the context of the Occam language\(^{13}\).

2.1.1 Computer Classifications

The multiprocessor system architectures are classified according to their logical and functional structure at the program levels. Different classifications were proposed by Flynn, Shore, Skillicorn, Hwang-Briggs, Erlangen, Gilo and others\(^{12, 15}\). One of the earliest and most cited schemes is Flynn’s. It consists of four taxa:
SISD: single-instruction stream, single-data stream, which is the traditional von Neumann computer.

SIMD: single-instruction stream, multiple-data stream, where the same processing is made for each processor's local data as in parallel vector or array processors.

MISD: multiple-instruction stream, single-data stream, which refers to a pipeline computer. The Hwang-Briggs modification to Flynn's classification eliminates this taxon.

MIMD: multiple-instruction stream, multiple-data stream, where each processor runs its own instructions on its local data.

Each taxon of the above spans a wide number of computers such that comparison cannot be made between them. However, it helps explain the architectures reviewed below.

2.1.2 Mesh Architectures

Mesh-connected processors were one of the first architectures used for image processing, because images map naturally onto this architecture. A typical machine consists of a large number of processors, each of which is usually connected to its four nearest neighbours, as shown in Figure 2-1. However, other connecting schemes may be used.

![Figure 2-1: Mesh Architecture](image)

Most mesh-connected machines are SIMD machines. Each processing element (PE) has its own memory and can be masked selectively using mask registers. Each PE responds in the SISD mode to controller instruction broadcast. The connections can be wrapped around in the Taurus model where top row PEs are connected to bottom row ones and the
first and last columns are connected together in a grid\textsuperscript{17}. Another variation is connecting the PEs of each row in a bus. Those enhancements do not change the basic characteristics of the model. Mesh algorithms can use simple scanning techniques and finish in their optimal time\textsuperscript{18}.

The advantage of this architecture is that images can map onto it easily, especially when the image size corresponds to the processor array size when calculations need to be made on individual pels or depend on a close neighbourhood of pels. Due to the connection architecture, this architecture is inefficient for tasks where calculations depend on distant groups of pels because communication between distant PEs is expensive and inefficient. Because this is a SIMD architecture, tasks that require more complicated processing cannot be carried on this system, especially that the PEs are simple. If the task does not map exactly to the size of the mesh, processing power will be wasted. A major drawback is the inefficiency of collecting and evaluating the results rapidly due to communication bottlenecks.

### 2.1.3 Pyramid Architectures

If the mesh is extended in the third dimension to facilitate communication between distant cells, a pyramid architecture will result. In this architecture, each level is four times the size of the level above\textsuperscript{16}. Therefore, a parent PE has four children to which it is connected in addition to its four neighbours and its parent if it is not at the top of the pyramid.

This hierarchical structure simplifies communication between distant cells, but it is more complex to build than a mesh because it involves many communication links and it requires about twice the number of PEs to process the same image resolution. All the PEs operate in the SIMD mode, running the same program. System utilization can be increased by driving each layer by a separate controller in a multiple SIMD mode. Efficient pyramid algorithms need to reduce information in two dimensions simultaneously and this, in general, requires considerable effort\textsuperscript{18}.

The computation pipeline is bottom-up causing the slowest processors in the bottom level to slow down the whole system and limit the utilization at all levels. It also requires thousands of processors to solve a problem of reasonable size. Adapting the algorithms to process problems of different sizes on the same number of processors is not always clear.

### 2.1.4 Hypercube Architectures

A hypercube is topologically an $n$-dimensional cube consisting of $2^n$ PEs where every PE
is connected to the $n$ PEs differing in only one bit position from its binary representation\textsuperscript{161}. Therefore, the longest distance between any two PEs is $n$ links.

This architecture can execute SIMD and MIMD programs. In the SIMD configuration, it is suitable for low-level and intermediate-level image processing. However, low-level and intermediate-level processing cannot occur simultaneously. In the MIMD configuration, it is suitable for intermediate-level and high-level processing. Its advantage is the efficient long-distance communication between processors. In large machines, however, communication is inefficient due to the slow communication bandwidths. A major disadvantage of the hypercube is the need to use the underlying topology for the algorithms to efficiently utilize the machine. A hypercube machine also lacks the global control.

2.1.5 Shared-Memory Architectures

Normally, shared-memory multiprocessor systems are MIMD machines\textsuperscript{161}. In addition to a large global memory, each processor has a small local memory. The processors gain access to the global memory through an interconnection network, which is usually bus-based or multistage. Due to bus access bottlenecks, bus-based systems cannot be large.

A more scalable class of shared-memory systems uses a multistage interconnection network for processor-processor or processor-memory interconnections. This configuration provides ease of programming and a uniform view of the system. This makes this architecture best suited for high-level processing. However, bottlenecks and hot spots occur due to using global memory for communication and interaction between processors and tasks. It also makes the overhead of synchronization very high. Due to the fact that accessing global memory is very slow compared to computation speed, this architecture is efficient for large-grain parallelism tasks exhibiting regular memory access patterns with little interaction. As the number of processors increase, the synchronization overhead grows and dominates the processing time, causing the return of using a larger-sized system to become negative beyond a certain size. This also limits the scalability of this class of shared-memory multiprocessors.

2.1.6 Systolic Arrays

When the algorithm can be implemented using very simple operations with a regular flow
of data and the control relying heavily on pipelining, the ideal system is a systolic array\(^\text{[116]}\).

![Systolic Architecture](image)

**Figure 2-2: Systolic Architecture**

A systolic array consists of connected processors\(^\text{[116]}\). On each machine cycle, each processor takes data from its input ports, performs the computation and passes the results and data through its output ports. The I/O only occurs at boundary cells\(^\text{[112]}\). The systolic array is a pipeline of a series of processing elements operating in parallel when the pipe is filled so they might be considered MISD\(^\text{[149]}\). The primary advantage of programmable systolic arrays is high performance for low cost\(^\text{[116, 119]}\). They are best suited for image processing. Systolic arrays can be configured in a parallel and pipeline mode (MIMD) to increase the processing efficiency\(^\text{[120]}\). The results cannot be evaluated until all data has passed through the array; therefore, it cannot adapt quickly to the characteristics of the image.

### 2.1.7 Hybrid Architectures

More flexible and partitionable system architectures were designed for more complicated image processing tasks as in computer vision where all three levels of processing need to be done simultaneously\(^\text{[116]}\). Such architectures are capable of running SIMD and MIMD programs at the same time. System configuration is under software control.

### 2.2 Asynchronous Transfer Mode

ATM has a flexible bandwidth allocation scheme which will be the transport technique for the broadband integrated services digital network, BISDN\(^\text{[17]}\). ATM is cell-based with a cell size of 53 bytes, 5 of which are the cell header and the rest is the information payload. ATM was chosen because of its capability of supporting multirate services. Its statistical multiplexing allows for the integration of services with a wide variety of characteristics. Assigning the peak rate bandwidth to calls of bursty or variable bit-rate nature decreases the network utilization considerably. On the other hand, assigning bandwidth less than the
peak rate makes cell loss inevitable. Cell loss probabilities larger than $10^{-5}$ are not acceptable for most services. Therefore, bandwidth allocation must be decided carefully.

2.2.1 Priority Mechanisms

The quality of service required by services varies widely. It depends mainly on the cell transfer delay and the cell loss probability\(^{[2]}\). The main factor in cell delay is the queueing delay which varies statistically depending mainly on the link's load. Cell loss caused mainly by buffer overflow depends on the network load and the size of the buffers.

There is one priority control bit in an ATM header although more priority levels may be needed for different classes of services\(^{[4, 7, 21, 22, 23]}\).

2.2.2 Packet Video in the ATM Environment

ATM provides a flexible environment for VBR video\(^{[3, 24]}\). The variability of video bit rate is expected to reduce delay, keep video quality constant, improve transmission efficiency\(^{[2]}\) and enable users to control picture quality. However, this requires solutions to cell loss, high-speed multiplexing and appropriate protocols to support VBR video transmission.

2.3 Video Compression Algorithms

Digital image data is compressed to save channel bandwidth. Compression removes different portions of the spatial and temporal redundancy exhibited in digital video. Layered video coding schemes arrange picture information in a pyramid according to their importance\(^{[4]}\). Bit-plane separation schemes assign the top of the pyramid to the most significant bit plane. Frequency-domain separation schemes, like subband coding, separate the image information into frequency bands; they are suitable for low loss rates because of the non-linear feature of frequency bands. Some schemes combine both methods to degrade gracefully in low and high loss rates. Some compression methods are completely reversible without losing any of the original data and hence are called 'lossless' coding techniques. Others remove insignificant details of the original image and hence are called 'lossy' coding techniques. Following is a brief review of some video compression methods.

2.3.1 Lossless Coding

Lossless coding preserves all the information of the coded data. These coding schemes
have low compression ratios and are usually used to code high priority information or motion vectors after lossy coding\footnote{\textsuperscript{125}}.

### 2.3.1.1 Huffman Coding

This scheme uses variable-length codes. When an alphabet of $n$ symbols is coded, the intuitive idea is to assign short code words to the most probable symbols\footnote{\textsuperscript{125}}. For an alphabet of $n$ symbols associated with an information source, Huffman coding algorithm guarantees a uniquely decodable code with the minimum average number of bits per symbol, $R$, with

$$H \leq R \leq H + 1, \text{ where } H = \sum_{i=1}^{n} p_i \log_2 p_i,$$

$p_i$ is the probability of symbol $S_i$, $\sum_{i=1}^{n} p_i = 1$. $H$ is the entropy of the information source.

The average code length varies depending on the symbol probability distribution. Huffman coding is highly efficient in situations where the probability distribution of adjacent pels is not uniform. A Huffman code can be generated systematically and easily with a computer program. The ideal decoding scheme is a code tree\textsuperscript{126} since this code abides by the prefix condition\textsuperscript{1}. Partitioning the incoming data rate among multiprocessors cannot guarantee fair partitioning of the coding processing load because of the entropy variation.

### 2.3.1.2 Lempel-Ziv Coding

This scheme applies variable-rate coding to a “standard scan” of a picture\textsuperscript{127}. An information lossless encoder of finite order is used to encode the largest number of distinct strings that constitute the input data stream\textsuperscript{128}. Initially, the code table is empty. Strings that are not in the code table are added when first encountered in the input stream, increasing the size of the code table as encoding proceeds. Each new string to be added is a codeword that already exists in the table plus a symbol, such that the new string does not yet exist in the table. This new table entry is the next input string. In doing so, the algorithm adapts to the input stream; however, the compression ratio is low before

\footnote{1 No codeword is a prefix of another}
sufficient adaptation has occurred. After the restricted code table is full, the algorithm stops to adapt. The algorithm is optimal for infinite input streams (with an infinite code table size). A modified version of this coding scheme is used in compressing computer data files.

2.3.1.3 Arithmetic Coding

This coding technique represents information at least as compact as the Huffman method. Arithmetic coding dispenses with the restriction of having to assign an integral number of bits to each symbol. The scheme works by assigning subintervals to symbols. A symbol’s subinterval is equal in length to its probability. A message is represented by a real number in the total interval in the form

\[ \text{Message}_0 = IS_0 + p_0 (IS_1 + p_1 (IS_2 + \ldots)) , \]

where \( IS_i \) is the interval start of symbol \( S_i \), \( p_i \) is the probability of symbol \( S_i \), the index \( i \) denotes the symbols’ order in the message, not the code table (see below for illustration).

In decoding, the first symbol is the one in whose subinterval the real number lies. Then the first symbol subinterval is scaled up to one to decode the next symbol in the same fashion:

\[ \text{Message}_1 = \frac{\text{Message}_0 - IS_0}{p_0} , \]

where \( \text{Message}_0 \) denotes the original message and \( \text{Message}_1 \) denotes the original message after decoding the first symbol. This procedure is repeated until the entire message is decoded.

Below is a table showing the symbols, probabilities, and subintervals:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>Subinterval</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.2</td>
<td>[0, 0.2)</td>
</tr>
<tr>
<td>b</td>
<td>0.1</td>
<td>[0.2, 0.3)</td>
</tr>
<tr>
<td>c</td>
<td>0.4</td>
<td>[0.3, 0.7)</td>
</tr>
<tr>
<td>d</td>
<td>0.3</td>
<td>[0.7, 1)</td>
</tr>
</tbody>
</table>

1 The popular compress program used on UNIX machines uses a Welch modified version of Lempel-Ziv algorithm, hence the file name extension LZW.
Consider the symbol probability distribution in Table 2-1. The message "bad" can be represented by any number in the interval $[0.214, 0.220)$. Suppose the real number $0.215$ was transmitted. Because it lies in the subinterval $[0.2, 0.3)$, the symbol "b" is recognized, then the number, $0.215 - 0.2 = 0.015$ is scaled up to $0.15$ (by dividing it by $p("b")=0.1$), which lies in the "a" subinterval. Then the number, $0.15$, is scaled up to $0.75$ (by dividing it by $p("a")=0.2$) which lies in the "d" subinterval. In this fashion, the scheme can remove the sharp boundaries between codes to achieve higher compression than Huffman and Lempel-Ziv methods. Usually, it is easier to implement and faster than the Huffman scheme. This scheme is being standardized in the Joint Binary Image Group (JBIG) standard\textsuperscript{13}.

2.3.1.4 Run-Length Coding

This simple method exploits pel-to-pel correlation\textsuperscript{13}. Considerable reduction occurs when, instead of transmitting a long run of the same symbol, only the run length and the symbol are transmitted. With increased high-frequency content regions, runs are very short and it may require more bits to transmit the run parameters than to transmit the original pels. Bit plane images can be run-length-coded. This scheme can be extended to a second dimension by area coding, where a 2-dimensional region consists of the same pel value, but this does not occur often. This scheme can be easily programmed and is computationally inexpensive, especially in the decode stage.

2.3.2 Lossy Coding

Lossy coding methods are not completely reversible; distortion and loss of fidelity may result in the coded image. The distortion and lack of minor details caused by this class of coding are intended to be transparent, or very closely so, to the practical destination of the image data.

2.3.2.1 Transform Coding

Transforms are used to decorrelate image data\textsuperscript{13}. The aim is to compact the image energy into a minimum number of statistically independent coefficients concentrated to a minimum region. The optimal transform for this is the computationally expensive Karhunen-Loève transform (KLT). This transform is often approximated by fast sinusoidal unitary transforms such as Discrete Fourier transform (DFT), Discrete Cosine
transform (DCT), Discrete Sine transform (DST), etc. DCT is generally considered to be the closest approximation to KLT. Due to their computational efficiency, nonsinusoidal orthogonal unitary transforms are used for video compression; examples are Walsh-Hamard and Haar transforms. Series expansion models are also used, such as singular value decomposition, which is superior in compression performance but prohibitively expensive to be used in real-time applications.

Interframe coding schemes can reduce the data rate considerably. Conditional-replenishment coding is such a predictive scheme, which transmits the difference between the actual and predicted frames. Motion compensation can be used in predictive coding schemes. A number of coding schemes is based upon interframe motion compensation and intraframe transform coding.

2.3.2.2 Combined Transform Coding

Dividing images into blocks and transforming them independently in transform coding leaves the correlation between different blocks high. Besides affecting compression performance, this creates a "blocking effect" across the block boundaries, especially at low bit rates\[^{11}\]. This blocking effect is the main limitation of transform-based low bit-rate coding. Various methods were used to reduce this effect. Smoothing the block edges may cause the loss of high-frequency information. Other methods have various disadvantages; therefore, a combined transform scheme was suggested.

The combined transform coding scheme divides the image into two image planes: the upper image plane (UIP), which contains the most important information, and the lower image plane (LIP). The UIP, the motion vectors and direct-current (DC) coefficients of the LIP are losslessly coded and are assigned the highest priority. The transform coefficients of the LIP form a pyramid in the frequency plane. Cells are dropped starting from the bottom of the pyramid in case of congestion so that performance degrades gracefully as the loss rate increases.

2.3.2.3 Predictive and Interpolative Coding

Due to the high correlation between pels in a local region, a pel might be predicted from knowledge of past pels or interpolatively inferred from past and future pels\[^{31}\]. Transmission of a starting value or periodical transmission of actual pel values, to avoid large errors, might be required. A compromise is made between the accuracy of the
predictor and its complexity. This compromise solution is to use a linear predictor and transmit a starting value and prediction error estimation.

Interpolative coding may, for example, transmit every other line of the image and use interpolation to estimate the missing lines. Computationally complex interpolative systems can reconstruct a band-limited signal exactly, but due to their complexity, simpler methods have been considered.

2.3.2.4 Subband Coding

This scheme separates the video signal into subbands of different frequencies. Each subband is then encoded separately. This allows for higher compression because interframe motion can be estimated more efficiently for subbands than for full bands. Furthermore, it can be combined with other coding techniques. SBC/DPCM (subband coding/differential pulse code modulation), SBC/DCT, SBC/VQ (vector quantization) and other combinations have been developed. VBR SBC schemes have the robustness and compression performance appropriate for packet video\textsuperscript{13}.

2.3.2.5 Vector Quantization

Vector quantization schemes can remove interpel correlation, statistical dependence and some of the geometric redundancy\textsuperscript{13}. Huge processing is involved in establishing and searching code books that are used by the scheme. This makes it not ready yet to be used for real-time VBR transmission.

2.3.2.6 Second-Generation Compression Schemes

Second-generation compression schemes employ geometric and global redundancy in natural scenes. Such schemes use computer vision, image segmentation and the understanding and characteristics of the human visual system, mostly operating on images with simple geometric patterns.

The fractal scheme\textsuperscript{134} breaks digital images into segments using color separation, edge detection and other techniques. Using the collage theorem, segments are looked up in a library of codes used to reproduce corresponding fractals. The appropriate codes replace the original segments, achieving compression ratios in the order of 10,000 or more, but the ratio is much lower for natural images. Images can be reproduced to any desired accuracy. However, this scheme is computationally very expensive\textsuperscript{1}. IFSIS\textsuperscript{134}, a special hardware
device, achieved a decoding performance of several frames per second. This scheme is still not suitable for real-time applications.

2.3.2.7 JPEG and MPEG Standards

The JPEG standard was developed by the Joint Photographic Experts Group. The MPEG standard was developed by the Moving Pictures Experts Group—an ISO committee[55]. JPEG is a general-purpose standard designed for continuous-tone, still-image applications[56] as in videotex and still-image telephony. MPEG is a generic standard designed for video signals at about 1.5 Mb/s, audio signals at 64, 128 and 192 kb/s and synchronizing and multiplexing multiple compressed video and audio streams. Draft documents[57] show that the MPEG 1 VBR peak bit rate is 1.5 Mbps and the MPEG2 VBR peak bit rate will be 10 Mbps, both with a Type 2 AAL. Broadband video recommendations are to be approved in 1994. MPEG2 is expected in 1994 for digital VTR, digital disc and cable TV. MPEG3 is expected to be approved in a few years. Its peak bit rate will be 40 Mbps and will be used for HDTV systems.

MPEG1 codes frames in three types: intra-coded (I) frames, predictive-coded (P) frames and bidirectionally predictive-coded (B) frames[58]. I-frames are encoded individually with intraframe coding methods. P-frames are coded using motion compensated prediction from past I or P-frames. B-frames are coded using motion compensated prediction from past and/or future I or P-frames. A group of frames begins with an I-frame and ends with the P-frame just before the next I-frame and a sequence of frames is a series of those groups of frames. This scheme is capable of forward and backward, normal and fast play and skip, pause, and high-resolution still. Blocks are zigzag-scanned, DCT-transformed and the coefficients variable-length-coded (VLC-coded). For colour video, luminance (Y), red chroma (CR) and blue chroma (CB) matrices are constructed. Motion information is based on 16-by-16 blocks and is compressed via VLC to be transformed along with spatial information. Many other unapproved proposed algorithms are being developed by different companies.

JPEG can also be used for video, achieving compression ratios in the vicinity of 25:1 with good picture quality. MPEG1 achieves 2.5 times the compression ratio achieved by JPEG[55]; however, it is much more complicated. JPEG is symmetrical so it can be encoded

---

1. Processing a complex colour image on Masscomp 5600 (a workstation based upon 2 68020 processors) took 100 hours to encode and 30 minutes to decode.
and decoded on the same system, but MPEG1 is asymmetrical and another system is required to carry out the inverse half of the cycle. Therefore, it is suited to asymmetrical applications where end user products need only decompress data which have been already compressed by the source. A single-chip, high-performance processor has been built which could carry out MPEG1 decompression in real time.\textsuperscript{151}

2.4 Video Compression for ATM

Video transmission will be a major service of ATM\textsuperscript{109} since there is a high demand for video conferencing and telephony, TV and high-definition television (HDTV), etc.\textsuperscript{100} The cell-loss characteristics of ATM\textsuperscript{100} may degrade video quality considerably\textsuperscript{131}, especially since video services are much more sensitive to cell loss than voice telephony\textsuperscript{109}. The need to compress HDTV by a ratio over four\textsuperscript{404}, and encode it in VBR, probably using interframe compression, will make cell loss a serious drawback\textsuperscript{100}. Much research has been done to investigate cell loss protection and recovery for video transmission in the ATM environment\textsuperscript{3}. Many research papers address VBR video coding for packet switching networks and ATM.

Verbiest \textit{et al.}\textsuperscript{91} discussed the impact of ATM on video coding. Ghanbari\textsuperscript{91} discusses general aspects of video packetizing. Lee \textit{et al.}\textsuperscript{106} analyzed video packet loss in ATM networks. Morrison\textsuperscript{411} describes a hardware implementation of a variable bit-rate extended standard fixed-rate video coder for ATM. Zhang \textit{et al.}\textsuperscript{131} surveyed methods to compensate for cell loss in VBR video transmission and elaborated on different layered source coding schemes. Verbiest and Pinnood\textsuperscript{51} presented a DPCM-based VBR video codec for ATM with cell loss. Takase \textit{et al.}\textsuperscript{140} discussed coding and functional requirements of VBR HDTV in ATM networks. Kishino \textit{et al.}\textsuperscript{121} presented a two-priority layered VBR video coding scheme to compensate for cell loss. Darragh and Baker\textsuperscript{142} presented a fixed-distortion subband image coding scheme for packet-switched networks. Kinoshita and Nakahashi\textsuperscript{134} described a two-layer DCT-based codec for HDTV, using block interleaving and data transposition with cell loss detection.
In this thesis, a representative scheme of VBR video coding and transmission is used to evaluate the possibility of applying multiprocessor simulation to the general class of VBR coding algorithms. The following sections explain the software representing the VBR coding and transmission scheme and the hardware system used to carry out the required simulation processing.

3.1 Simulation System

The simulated system includes the coding algorithm, a basic part of the ATM adaptation layer, and a simplified model of the ATM channel followed by decoding. The compression algorithm used in the experiments exemplifies behaviour exhibited by various coding algorithms in having a certain pattern of data dependencies and irregularities in subtask and frame coding processing demand. Single frames are used in the initial set of experiments, and the results are expanded later to more general video processing schemes. The source and destination of image data is the memory of the host system.

3.1.1 Compression Algorithm

The video compression algorithm used in the simulation is a layered video coding scheme which separates the feature plane into three layers. The three layers in descending order of feature importance are the superpel, triad and pel layers. In this section, width is the image width in pels, M and N are the height and width of a superpel in pels, respectively. Image
width is divisible by superpel width. \( N \). Figure 3-1 illustrates this nomenclature.

![Figure 3-1: Illustration of Terminology](image)

For an image matrix, a superpel (here, the average of 18 pels) is calculated as follows:

\[
\text{superpel}_k = \frac{1}{MN} \sum_{j = N \text{mod} \left( k, \frac{\text{width}}{N} \right)} \sum_{i = M \text{div} \left( k, \frac{\text{width}}{N} \right)} \text{pel}_{ij},
\]

where \( \text{div}(a, b) \) denotes the quotient of the integer division of \( a \) by \( b \), and \( \text{mod}(a, b) \) denotes the remainder of dividing \( a \) by \( b \); \( a, b \) are integers. The above expression simply means that the horizontal index of the pel at the top left corner of the superpel is incremented by \( N \) with every new superpel. It is reset and the vertical index of that pel is incremented by \( M \) when the superpel index is divisible by the number of superpels per line.

In the algorithm implemented in this work, \( M = 3, N = 6 \). The above expression evaluates for superpel_{width/6} as:
\[ \frac{1}{3 \times 6} \sum_{j = 6 \mod \left( \frac{\text{width}}{6}, \frac{\text{width}}{6} \right)}^5 \sum_{i = 3 \div \left( \frac{\text{width}}{6}, \frac{\text{width}}{6} \right)}^5 \text{pel}_{ij} = \frac{1}{18} \sum_{j = 0}^{5} \sum_{i = 3}^{5} \text{pel}_{ij}. \]

A triad in the upper row of a line group (where \( \text{mod}(\text{div}(l, \frac{\text{width}}{2}), 2) = 0 \)) is calculated by:

\[
\text{triad}_l = \frac{1}{3} (\text{pel}_{i, j} + \text{pel}_{i, j+1} + \text{pel}_{i+1, j}) - \text{superpel}_{\text{mod}(\text{div}(l, 3), \frac{\text{width}}{6}) + \frac{\text{width}}{6} - \text{div}(l, \text{width})}
\]

where \( i = 3 \div (l, \text{width}) \) and \( j = 2 \mod (l, \frac{\text{width}}{2}) \). The first term in the superpel index above accounts for the horizontal location of the superpel and the second term accounts for the vertical location of the superpel. The superpel is subtracted from the triad to achieve the alternating current (AC) value of the triad. A triad in the lower row of a line group (where \( \text{mod}(\text{div}(l, \frac{\text{width}}{2}), 2) = 1 \)) is calculated by:

\[
\text{triad}_l = \frac{1}{3} (\text{pel}_{i, j} + \text{pel}_{i, j+1} + \text{pel}_{i-1, j+1}) - \text{superpel}_{\text{mod}(\text{div}(l, 3), \frac{\text{width}}{6}) + \frac{\text{width}}{6} - \text{div}(l, \text{width})}
\]

where \( i = 3 \div (l, \text{width}) + 2 \), and \( j = 2 \mod (l, \frac{\text{width}}{2}) \).

The AC value of each pel is evaluated by subtracting the sum of the superpel and triad from the original pel. Next is the thresholding of the AC pels. In this part, each pel less in absolute value than a prescribed threshold is considered zero. Run-length coding is then used to decorrelate close pels in the same line group. The output of the run-length coding (RLC) stage is entropy-coded (VLC-coded) using the Huffman scheme. The VLC stage cost depends highly upon the entropy value of the pels.
The lossy ATM channel is simulated by a geometrically distributed random function. The function generates the intervals between each lost cell and the next one.

The decode program segment reverses the encoding scheme. The resulting image differs from the original due to the quantization error in thresholding differential pels and to lost cells in the three feature bands.

3.1.2 Algorithm Implementation

The first part of the algorithm averages the pels in three bands of resolution, the sum of which effectively reconstructs the original image. A superpel is the average of 18 pels. A triad is the average of 3 pels; 6 triads constitute a superpel. This can be used to save calculations by first calculating the triads and then averaging the triads in each superpel to achieve the superpel value. This task can be achieved by calculating the sum of every group of 3 pels that forms a triad and storing them. Then the sum of the triads that form a superpel is calculated and (effectively\(^1\)) divided by 18. The triad averages can then be evaluated by (effectively) dividing the triad sums by three. Next, final pel values are achieved by subtracting superpel values from them. Finally, superpels are subtracted from triad averages to form the final triad values.

For a superpel (18 pels), the operations (additions, subtractions, multiplications/divisions) and memory accesses (MA) involved in the aforementioned task are:

\[
\begin{align*}
6(2\text{ADD}+4\text{MA}) & \quad \text{(trial sum)} \\
+5\text{ADD}+7\text{MA}+1\text{DIV} & \quad \text{(superpel average)} \\
+6(2\text{MA}+1\text{DIV}) & \quad \text{(trial average)} \\
+6(3\text{SUB}+7\text{MA}) & \quad \text{(final AC pels)} \\
+6\text{SUB}+13\text{MA} & \quad \text{(final AC triads)}
\end{align*}
\]

That averages to 2.28 add/sub, 5.44 MA and 0.39 div per pel.

The next task is to disassemble the superpels and triads into ATM-like data cells. This task mainly transfers superpels and triads to a stream buffer, interleaved by cell identification words (overhead of 10% of the data size). This involves 15.4 MA per superpel or 0.86 MA per pel.

Pels smaller in absolute value than a specific threshold are substituted with zeros. This

---

1. They can be multiplied by the reciprocal of the divisor since the divisor is constant when fast floating-point multiplication is supported by hardware. Other processors may have hardware division instructions.
task involves 2 MA and 1–2 comparisons per pel (due to the two destinations of comparison, caused by using an absolute value for the threshold). The resulting data is compressed using run-length coding.

Run-length coding (RLC) is followed by variable-length Huffman coding (VLC). To prevent error (or loss) propagation in the cells that follow a lost cell, each cell must be coded individually. Therefore, a VLC length counter is initialized at the beginning of the RLC coding of every cell to follow the progress of the equivalent Huffman length of the data encoded in the cell. When a byte is coded, the VLC counter is incremented by the VLC code length of the byte. An address field is prepended at the beginning of every cell to specify the location of the information content of the cell in the image frame. This method ensures that every cell is loaded with data according to its capacity and makes decoding a cell independent of any other cell. A homogeneous run must be longer than three pelts. This compression method achieves a compression percentage of about 20% for the image samples used in the experiments reported in Chapter 5. Therefore, each pel requires an average 3 add/sub (the address, the VLC counter, and the RLC counter) 4.8 MA (1 for the source; 0.8, destination; 1, preceding pel; 1, address; and 1, length), and 1 comparison. This cost is statistically variable.

Huffman VLC is applied to the result of the preceding stage. Every byte is replaced by its VLC code without any calculations because the necessary arrangement was made in the RLC stage. The result of this stage is a sequence of ATM-like cells ready for the ATM channel stage. On average, an 8-bit byte is compressed into 5 bits. This involves an average of 5 register shifts and 3.02 MA per pel. This average comes from 0.8 MA for each of the source, code, and length; and 0.62 MA for the destination. (The factors come from the respective average compression ratios.)

In the channel loss simulation routine, for every lost cell, a geometrically distributed random number is generated. Generating the random number mainly involves generating a uniformly distributed random number, calculating a logarithm and 2 divisions. The probability of cell loss is very low (lower than $10^{-6}$ for the low-priority stream) such that the computational cost of this routine can be neglected. The loss probability that was used in the actual simulations was much higher (about 0.02) so that generated errors could be detected. The average cost is much less than 1 multiplication per pel.

The decoding stage starts with VLC decoding. Assuming an average of 5 bits per pel, decoding one symbol (pel) using an optimal method requires 5 register shifts and 5.4 MA
(0.6 for reading the symbol; 0.8 for each shift; 0.8 for writing it). Practically speaking, this stage requires 4-6 times the time required by VLC encoding.

RLC decoding is less computationally expensive than RLC encoding. Decoding a pel approximately costs 2 MA.

Finally, the image is reconstructed from the three bands. This stage requires 2 additions and 2.4 MA per pel (0.06 for superpel; 0.33, triad; 1, pel read; 1, pel write).

The operations total is about 7.3 add/sub, 0.4 div/mul, 25.9 MA, 2.5 comparisons and 10 register shifts per picture element. Execution overhead includes accessing instructions, calculating array indices, branching, etc. The efficiency of execution depends on the architectural design and features of the specific processor used to carry out the algorithm.

3.1.3 ATM Adaptation Layer Protocol

The ATM adaptation layer supports information-transfer protocols not based on ATM\textsuperscript{144}. Its purpose is to assemble and disassemble bits, frames and data to and from ATM cells in a fashion suitable for the application. It provides and maintains timing relationships of real-time applications, handles cell loss, forward error correction and other attributes not carried out or preserved by ATM, and provides segmentation and reassembly of ATM adaptation layer (AAL) service data units (SDUs) in and out of ATM cells\textsuperscript{145}. These functions include ones which are common to all services and others which vary depending on the service in question. To support these classes of service, AAL protocols are required.

In this thesis, the algorithm does not involve the cell header except in assuming two priority levels. The algorithm also leaves 4 bytes of the information payload field to be used by the ATM adaptation layer protocols. The control scheme used by the algorithm uses two bytes of every high-priority cell payload for serial identification (ID) numbering and four bytes of every low priority cell payload for the same purpose. Once a specific AAL protocol is decided, the assumption used can be changed accordingly.

3.1.4 ATM Channel Model

The ATM channel is represented by a geometrically distributed time sequence between lost cells. The cell loss rate used for the high-priority stream differs from that used for the low-priority one. The algorithm uses cell loss rates much higher than the real case in order to enhance the observation of typical results of errors with the limited image data sample used in the experiments.
3.2 Banshee Multiprocessor Board

The Banshee board is based upon Texas Instruments TMS320C30 floating-point digital-signal processor. It has a static random-access memory (SRAM) and dual-port memory that can be addressed readily by both the Banshee processor and the host processor. A multiprocessor interface (MPI) daughter board is used to interface the Banshee board to other Banshees via dual-port memory and common buses. These are explained in the following sections.

3.2.1 The C30 Processor

The information in this section is based on TMS320C30 User’s Guide[46]. The TMS320C30 is a CMOS 32-bit floating-point digital-signal processor. It achieves high performance by implementing in hardware what general purpose processors implement in software or microcode, achieving a 60-ns single-cycle execution time. The processor can perform parallel multiplication and ALU operations on integer or floating-point data in a single cycle. The C30 possesses a general-purpose register file, program cache, dedicated auxiliary register arithmetic units, internal dual-access memories, one direct-memory-access (DMA) channel supporting concurrent I/O, and a short machine cycle time.

The 60-ns execution cycle corresponds to 16.7 million instructions per second (MIPS), and the ability to perform two floating-point operations within a single cycle makes the floating-point data processing peak 33.3 mega floating-point operations per second (MFLOPS). The processor’s two address generators with eight auxiliary registers and two auxiliary arithmetic units make it possible to address two data words at the same time. The on-chip memory supports this dual-access feature.

The on-chip memory consists of 6 kilowords (Kwords) divided into a 4-Kword dual-access ROM block and two 1-Kword dual-access random-access memory (RAM) blocks. The program cache is 64 words controllable by software to be cleared, enabled or frozen. The data bus is 32 bits wide and the address bus is 24 bits wide. Although the addressable data element is a 32-bit word, the processor has eight 40-bit wide floating-point registers and a supporting multiplier data bus in addition to the 32-bit integer-data processing capabilities. An improvement over most previous digital-signal processors is the capability of single-instruction and block repeats with zero overhead. The processor has conditional instructions for calls, returns and loads.
The C30 processor enhances general-purpose applications by having a large address space, a multiprocessor interface, internally and externally generated wait states, two external interface ports, two timers, two serial ports, and a multiple interrupt structure. To facilitate multiprocessing, the C30 has two external interface ports and interlocked instructions.

3.2.1.1 Pipeline Operation

The C30 pipeline structure includes five units: Fetch Unit (F), Decode Unit (D), Read Unit (R), Execute Unit (E) and DMA Channel (DMA). A basic instruction has four levels: fetch, decode, read and execute. The perfect overlap occurs when the four units are each busy with a different instruction. A pipeline conflict occurs when one unit finishes while the next unit is still busy; in this case, it waits until the next level is ready. Because the DMA, which has the lowest priority, has its own data and address buses, conflicts with the CPU can be minimized or eliminated by suitable data structuring.

Pipeline conflicts can be grouped into three main categories: branch conflicts, which involve most operations that read or modify the PC; register conflicts, which involve delays occurring when reading or writing registers used for address generation; and memory conflicts, which occur when the internal units of the processor compete for memory resources. Causes of memory conflicts include multicycle memory access and interlocked load instructions. Multicycle memory accesses are caused by the gap between the CPU and memory bandwidths. The interlocked load may be delayed by another processor's locking memory.

3.2.1.2 Provision for Multiprocessing

Processors in a multiprocessor system may share global memory. In order for the global memory to be shared by processors in a coherent manner, arbitration or handshaking is necessary. This requirement of arbitration is the purpose of the interlocked operations. The C30 has five interlocked operations which provide synchronization mechanisms through the use of external signals, guaranteeing the integrity of the communication and resulting in a high-speed operation. The operations are integer and floating-point load and store, and interlocked signal. When a processor executes an interlocked loading instruction, other processors are denied interlocked memory access until the interlock is cleared (by an interlocked store operation) by the same processor. Interlocked signal operations can
synchronize processors by using hardware signals to idle until interlocked signal operations are executed by the other processors.

3.2.2 The Banshee-Host Interface

The information in this section is based upon *Banshee System Instruction Manual*[^1]. The Banshee Mother Board is a full-length AT-compatible board.

3.2.2.1 Software Interface

The Banshee board is accompanied by a shell program, a debugger, a program loader, a C compiler, a C30 assembler and signal-processing supporting libraries. The shell is an extension of DOS to execute Banshee programs from the DOS prompt. A symbolic debugger is included to interactively debug programs from the PC screen. A loader is supplied to load Banshee target programs to be executed in the background. SPOX, a real-time signal processing operating system, provides a device-independent I/O and digital-signal processing (DSP) functions to the system.

3.2.2.2 Hardware Interface

A Banshee board can be configured on either of two I/O base addresses. Each of those base addresses can take up to 16 boards making a maximum of 32 boards to be hosted on one host. This is made possible by decoding the next more significant 4 bits of the I/O address bus that the host computer does not decode.

The host computer’s processor address space interfaces to the Banshee memory through a 16-Kbyte (4-Kword) memory window. By disabling and enabling the Banshee memory accessed by the host through this window, any number of Banshee systems can share the same memory window address as long as only one board is active at any one time.

3.2.2.3 The Banshee-Host Communication

Communication and data transfer between the host and the Banshee system can be carried out in several different ways.

The host can directly write to the Banshee memory through the 16-Kbyte memory window of the host. In this method, the Banshee processor must be in a HOLD state. This method is useful when transferring large blocks of data to justify holding the Banshee processor.
Through the same memory window, the host can access the dual-port memory while the Banshee processor is active. The dual-port memory is 8 Kbytes; therefore, it is useful for smaller blocks of data, especially as it is slower than the Banshee static memory.

In the former two methods, a signalling protocol must be used between the two processors since the memory cannot set any flags or activate any interrupts. In those two methods, the host DMA can be used to do the transfer, as is also the case in the following two methods.

A rather slow way of block communication is to set up a channel between the host DMA and the Banshee processor. This way, interrupts can be used to relieve both processors from data transfer but the Banshee processor reads one word at a time.

The last method is the program I/O data exchange, where the host writes to a 4-byte I/O register that can be read as a word by the Banshee processor. Interrupts can be generated on either or both of the processors. This method is useful for signalling the arrival of block data or the start or end of a program task; however, it is very slow for large data transfers.

3.2.3 The Banshee Multiprocessor Interface

The Banshee Multi-Processor Interface (MPI) daughterboard connects multiple Banshee boards in a ring structure providing communications between each processor and its left and right neighbours. This interface provides multiprocessor synchronization, a common clock, a common hold and a DT-connection. The DT-connection provides a parallel data port and an RS-422 buffered serial port for the motherboard processor.

![MPI Block Diagram](image)

The daughterboard has a 2-Kword dual-port memory buffer between processors with interprocessor memory allocation flags and interprocessor message status flags with interrupt capability. It also provides a common master clock, common SYNC, and host-controlled common processor HOLD for all processors. It provides the host with the status
of SYNC and HOLD signals. It has a Multi-Processor Interface interrupt controller and a status port.

The interface is connected to the expansion bus of the motherboard and provides three ports; two to connect to the RIGHT and LEFT MPI boards. The third port provides an external serial interface through a D-type connector on the back of the motherboard and two 16-bit bidirectional parallel data ports through two 26-pin connectors for the DT-connection. The dual-port memory on the MPI is mapped into both the memory and I/O address spaces of the motherboard processor. The MPI has an interrupt control port accessible by the host.

The MPI interrupts and external ports can be configured by a Banshee processor write to the control register. Status can be read from the same register. Another control register is available to the host to control and monitor the common hold request, the common hold bus and synchronization signals.

3.2.3.1 Interprocessor Communication

A number of Banshee systems can be configured in a ring using the dual-port memory. Data can be passed from any system to its LEFT or RIGHT neighbour directly through the shared memory. Longer hops can be made around the ring or by the host’s bridging the gap.

The dual-port memory requires one wait cycle from the Banshee processor, more if two processors attempt to access the same word simultaneously. It can be accessed as I/O or memory. When it is accessed as I/O it can set or clear flags and generate interrupts, but it is one cycle slower for read. This memory is a fast communication channel taking two cycles to read and three cycles to write, so the peak bandwidth is one word per five cycles, which comes to 13.2 megabytes (Mbytes) per second, provided that only one processor at a time accesses the memory to avoid memory data conflicts. There is a mechanism for software access organization – a flag is used for each LEFT or RIGHT channel to show whether memory is allocated free; however, access will not be physically denied if memory is allocated to the other processor. Nonetheless, physical access denial can be implemented using interlocked instructions.
4

Single Processor Performance

This chapter first presents performances of several single processor systems in order to establish comparative speeds for this application. The coding algorithm was implemented with a C program which was run on a Sun 4/75, a 33-MHz 386/387-based PC, a 33-MHz 486-based PC and Banshee board machines. The 386 and 486 processors were run in the Real Mode.

The program processed a 144-by-96 8-bit pel frame (depicted in Figure A-1). Table 4-1 and Figure 4-1 only reflect processing times; file loading and data transfer times between the host and the processing elements are not included.

Table 4-1: Coding Algorithm Performance on Different Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Operating System</th>
<th>Compiler</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun 4/75</td>
<td>UNIX</td>
<td>GNU CC V. 2.3.3</td>
<td>0.54</td>
</tr>
<tr>
<td>386/387-PC</td>
<td>MS-DOS</td>
<td>Borland C++ V. 2.0</td>
<td>1.50</td>
</tr>
<tr>
<td>486-PC</td>
<td>MS-DOS</td>
<td>Borland C++ V. 2.0</td>
<td>0.80</td>
</tr>
<tr>
<td>Banshee</td>
<td>SPOX/AShell/MS-DOS</td>
<td>TI C V. 4.10</td>
<td>0.34 †</td>
</tr>
<tr>
<td>Banshee</td>
<td>SPOX‡</td>
<td>TI C V. 4.10</td>
<td>0.35 †</td>
</tr>
<tr>
<td>Banshee</td>
<td>none (Assembly Language)‡</td>
<td>TI Assembler V. 4.10</td>
<td>0.20</td>
</tr>
</tbody>
</table>

†. With less compiler optimization, processing time may reach 0.90s
‡. This mode can be used for multiprocessing.

In the remainder of this chapter, examination is made of the performance of two single processor systems running the algorithm explained earlier (in Section 3.1.2). The
computational cost of the routines of the algorithm achieved below pertains to the completely optimized implementation of the algorithm. In high-level language implementations, there is additional overhead on top of the algorithmic computations involved.

![Comparative Performance Bar Graph](image)

**Figure 4-1: Comparative Performance Bar Graph**

4.1 Personal Computer Performance

In this section, brief examination is made of running the image coding and decoding algorithm presented in Section 3.1.2 on a personal computer based upon Intel 80386 or 80486 microprocessor.

The Intel 386 processor is a 32-bit microprocessor[^48], with 8 32-bit general-purpose registers, although the registers do not work identically. It has two modes of operation: Real Mode and Protected Mode. In the Real Mode, it operates as a very fast 8086. In this experimental work, it was operated in the Real Mode. In this mode, the maximum contiguous memory segment size is fixed at 64 Kbytes. For data transfer, the shortest time unit is a bus state which is one processor cycle in duration. A complete data transfer requires a bus cycle which is composed of two or more bus states. Simple data transfers range from 2 to 4 cycles, and up to 7 cycles if simple arithmetic is involved.

When dealing with huge blocks of data, a particular architectural point of the processor is disadvantageous, namely, the limited size memory segment. Blocks of data
larger than 64 Kbytes must be partitioned among different segments or into extended or auxiliary memory and be switched back and forth. When data involved is larger than 640 Kbytes, use of extended or expanded memory or auxiliary memory is required. Extended and expanded memories are managed by software interfaces\textsuperscript{[40]}, which makes it definitely slower than the processor's main memory. New compilers are available which run in the Protected Mode.

An instruction cache can be of benefit for speeding up a program if the code is written to use it efficiently by localizing code and using as few branches as possible. In general, programs can be executed more efficiently by optimization techniques usually integrated in optimizing compilers. Some of those techniques involve expression simplification and alias disambiguation\textsuperscript{[40]}. Data flow optimization, control flow and loop optimization and in-line expansion of some code can be of a great effect on the program efficiency. Most often, expanding the program on larger memory can make it execute more efficiently.

The compression algorithm starts by calculating the superpels and triads, which are averages calculated by dividing the sum by an integer. This part is not computationally expensive; it takes about 10\% of the total processing time; Table 4-2 shows the details. There are six triads and one superpel to be calculated for each group of eighteen pels. Calculating the superpels can be optimized by coding them in Assembly language rather than using a general shift-and-subtract division algorithm. Calculating the triads can be sped up by using functional arrays if necessary, which means indexing the quotients by the nominator values. In some special cases where the denominators are powers of two, divisions map directly into register right shifts. Then the calculated results are disassembled into ATM cells. This part is fairly fast and simple. It can be sped up only by coding it in Assembly language. The pels are quantized by truncating low values to zero. A look-up table can be easily used to speed up this operation by saving branches on comparisons.

The run-length coding used in this algorithm is more complicated than normal. Here the RLC coding is done so that the Huffman coding of the result will efficiently fit into ATM cells (see Section 3.1.2). Therefore, as the code is built, the Huffman code length is updated as every code byte is produced. It can be coded in Assembly language to achieve

\textsuperscript{1} A technique that determines when two pointer expressions cannot point to the same location to allow the compiler to freely optimize them.
better performance.

<table>
<thead>
<tr>
<th>Subtask</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculate superpels, triads and pels</td>
<td>0.2</td>
</tr>
<tr>
<td>disassemble superpels triads to cells</td>
<td>0.1</td>
</tr>
<tr>
<td>quantize pels</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>run-length-encode</td>
<td>0.1</td>
</tr>
<tr>
<td>Huffman-encode</td>
<td>0.1</td>
</tr>
<tr>
<td>introduce channel loss</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>unpack cells to bytes</td>
<td>0.1</td>
</tr>
<tr>
<td>Huffman-decode</td>
<td>0.7</td>
</tr>
<tr>
<td>run-length-decode</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>build result pels</td>
<td>0.1</td>
</tr>
<tr>
<td>total</td>
<td>1.5</td>
</tr>
</tbody>
</table>

The resulting run-length code from above is Huffman-coded. The coding is simple: codes are read from a coding array, their lengths are read from another array and the codes are shifted into a register according to the code length parameter. The routine can be optimized by coding it in Assembly language.

The channel simulator routine takes the generated cells, the high-priority data from the disassembler routine above and the low-priority data from the Huffman coding routine and passes it through a geometrically distributed cell-loss function. The cells, pointed to by the random function, are replaced by zeros. However, any other action of the algorithm can be taken, such as using the data in the spatially or temporally previous line. Each value of the random cell loss function involves one call to each of: a uniform random function; a logarithm function; a floor function; a division and a multiplication. This random sequence can be generated beforehand and can be read from an array to save time.

Next, the high priority components, the superpels and triads, are disassembled from the cells into arrays. This is a simple routine that mainly involves shifting and masking. It
can be easily implemented in Assembly language.

The Huffman code in the low-priority component is decoded. This is the most computationally expensive routine in the program; it consumes approximately 47% of the total processing time. The Huffman codes are searched linearly in a decoding table, in which the codes are sorted in descending order of frequency. The search can be done in the binary-search method to save about 30% of the time. However, a better search method can be designed to take into consideration the relative frequency of the codes, since the upper half of the code table is considerably more likely to have the desired value than the lower half. This is usually the case in most table partitions. Since this is a critically expensive part of the code, it is best to be implemented in Assembly language, especially since it mainly involves searching and then shifting the code out of the code register, according to its length, from another table, and storing the symbol in its respective location.

The run-length decoding is computationally very inexpensive. It only involves reading run-length counts from the code array and transferring data accordingly to another array. Coding it in Assembly language can make it faster, but produces little improvement.

The last stage is adding the differential superpels, triads and pels together to generate the resulting frame, which involves two additions per pel. Three loops are required for this task. Temporary variables can be used to save recalculation of array indices.

4.2 Single Banshee Board Performance

There are some program optimization techniques specific to the C30 processor because of its special architecture. The C30 has a large register file that can be used for storing frequently-used pointers and variables to exploit the relative speed of accessing registers. Register variable tracking avoids reloading registers if they are to be reused soon. Cost-based register allocation, autoincrement addressing, repeat blocks, parallel and conditional instructions and delayed branches are techniques used by the optimizing compiler. However, the programmer can help the optimizer, and make it more efficient, by avoiding complicated programming styles that decrease the efficiency of compiler optimization.

The C30 has 2 Kwords of fast internal RAM that can be used as scratch processing areas and variables, providing higher performance than off-chip memory. It also has an instruction cache that can be used efficiently by good design of the program (localizing
code and minimizing branches).

<table>
<thead>
<tr>
<th>Subtask</th>
<th>Time (s) (without optimization)</th>
<th>Time (s) (with optimization)</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpack words into bytes</td>
<td>0.023</td>
<td>0.007</td>
</tr>
<tr>
<td>calculate superpels, triads and pels</td>
<td>0.090</td>
<td>0.023</td>
</tr>
<tr>
<td>disassemble superpels and triads to cells</td>
<td>0.011</td>
<td>0.006</td>
</tr>
<tr>
<td>quantize pels</td>
<td>0.060</td>
<td>0.040</td>
</tr>
<tr>
<td>run-length-encode</td>
<td>0.110</td>
<td>0.053</td>
</tr>
<tr>
<td>Huffman-encode</td>
<td>0.075</td>
<td>0.025</td>
</tr>
<tr>
<td>introduce channel loss</td>
<td>&lt;0.0005</td>
<td>&lt;0.0005</td>
</tr>
<tr>
<td>unpack cells to bytes</td>
<td>0.103</td>
<td>0.003</td>
</tr>
<tr>
<td>Huffman-decode</td>
<td>0.343</td>
<td>0.149</td>
</tr>
<tr>
<td>run-length-decode</td>
<td>0.028</td>
<td>0.014</td>
</tr>
<tr>
<td>building pels</td>
<td>0.030</td>
<td>0.012</td>
</tr>
<tr>
<td>packing pels into words</td>
<td>0.017</td>
<td>0.011</td>
</tr>
<tr>
<td>total</td>
<td>0.890</td>
<td>0.343</td>
</tr>
</tbody>
</table>

Calculating the superpels and triads accounts for approximately 11% of the total processing time; details are in Table 4-3. Analysis here is done on the unoptimized version to be compared with the unoptimized Assembly Language version in Table 4-4. Division was implemented by transforming division (by a constant) into multiplication by the reciprocal of the constant divisor. In processors where fast multiplication is not provided, division can be optimized by coding it in specific Assembly language division routines instead of using less-efficient general shift-and-subtract division algorithms. Calculating the triads can be sped up by using functional arrays (where quotients are indexed via numerator values). The availability of floating-point arithmetic on this processor made it efficient at transforming integers to floating-point values and using unicycle floating-point multiplication to calculate the quotient. The floating-point quotient is then transformed
back to an integer. The two transformations and one multiplication require three execution cycles—faster than any integer-division routine.

Next, the calculated results are disassembled into ATM cells. This simple task can only be sped up by coding it in Assembly language.

The pel's are quantized by truncating low values to zero. A look-up table can be used to speed up pel quantization by saving branches on comparisons. This reduces its execution time substantially.

The run-length coding can be programmed in Assembly language, achieving better performance by minimizing the effect of branches that consume most of this task time.

The resulting run-length code from the routine above is Huffman-coded. The coding routine is simple, but it involves reading codes and lengths from two arrays and shifting the codes into a code register according to their length. The routine can be optimized by coding it in Assembly language.

Each value of the geometric random cell-loss function involves one call to each of: a uniform random number generator; a logarithm function; a floor function; a division and a multiplication. This random sequence can be generated beforehand and can be read from an array to save time. Using floating-point operations reduces the processing cost of the logarithm function. All calculations are made in floating-point arithmetic and the final result is transformed into an integer value.

The high priority components, the superpels and triads, are then disassembled from the cells into arrays. This is a simple routine that mainly involves shifting and masking. It can be easily implemented in Assembly language.

Decoding the Huffman code is the most expensive task, taking about 40% of the total processing time. It can be improved by the same methods suggested in Section 4.1.

The run-length decoding only involves reading counts from the array and transferring data accordingly to another array. Coding it in Assembly language can make it faster, but it is not a time-consuming task.

Building the result frame from the superpels, triads and pel's is identical to the case in the previous section. The main computational cost is address arithmetics.

The temporary arrays for keeping superpels, triads, pel's, and run-length code while calculating can be allocated in the internal memory since writing to external memory takes double the time of writing to internal memory.
In the multiprocessing experiments reported in the next chapter, an Assembly-language version of the above program was used because of its relative ease and accuracy of interaction with the hardware and timing measurement. Because of the overhead of high-level languages mentioned earlier, the assembly-language program is over four times faster than its C counterpart; see Table 4-4 for details. It, too, can be further optimized, in both coding and algorithms of individual tasks.

Table 4-4: Banshee System Time Cost of Assembly-Language Program Subtasks

<table>
<thead>
<tr>
<th>Subtask</th>
<th>Processing Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpack words into bytes</td>
<td>3.8</td>
</tr>
<tr>
<td>calculate superpels</td>
<td>2.2</td>
</tr>
<tr>
<td>calculate triads</td>
<td>8.4</td>
</tr>
<tr>
<td>disassemble superpels and triads to cells</td>
<td>1.5</td>
</tr>
<tr>
<td>quantize pels</td>
<td>12.3</td>
</tr>
<tr>
<td>run-length-encode</td>
<td>24.1</td>
</tr>
<tr>
<td>Huffman-encode</td>
<td>15.8</td>
</tr>
<tr>
<td>introduce channel loss</td>
<td>0.1</td>
</tr>
<tr>
<td>unpack cells to bytes</td>
<td>2.4</td>
</tr>
<tr>
<td>Huffman-decode</td>
<td>110.9</td>
</tr>
<tr>
<td>run-length-decode</td>
<td>5.0</td>
</tr>
<tr>
<td>adding superpels to pels</td>
<td>5.2</td>
</tr>
<tr>
<td>adding triads to pels</td>
<td>4.5</td>
</tr>
<tr>
<td>packing pels into words</td>
<td>3.6</td>
</tr>
<tr>
<td>total</td>
<td>199.8</td>
</tr>
</tbody>
</table>
5

Multiprocessor Performance

This chapter addresses the processing of the algorithm described in Section 3.1.2 on two of the Banshee boards described in Section 3.2. The processors are connected in a ring and each is also connected to the host computer as illustrated in Figure 5-1. The host computer can address all the memory address space of any of the PEs and each PE shares a mail box memory with its neighbour[47].

![System Architecture]

**Figure 5-1: System Architecture**

As documented in Table 4-1, the program execution time is 0.20 seconds when it is run on a single Banshee system board. The overhead added to this processing time by the PC:
data transfers to and from the board's local memory makes the total time 0.209 s. Ideally using two Banshee system boards in parallel should perform the processing task in half the time required by a single board. However, achievement of this performance is impeded by the inefficiency and overhead imposed by splitting the processing unfairly between processors, as well as the time required for interprocessor communication. In effect, the overhead caused by loading data to the Banshee memory is constant, thereby reduces the multiprocessing gain. This overhead becomes more pronounced as the number of processors increases.

The program timing dissected for a single processor requires 0.0808 s for the top half of the image, and 0.1197 s for its lower half. The difference is due to the difference in image complexity between the two parts of the picture. The program stages, for a complete image, consume 0.0684 s for simulating the coding and transmission stage (from source encoding to simulating the channel loss), and 0.1317 s for simulating the reception and decoding stage (the reverse process of achieving the result frame). This makes fair load sharing per frame not readily possible. The following sections discuss various methods for load assignment because no general solution of the load balance problem has been achieved.

5.1 Star Architecture

In the star architecture, the processing elements operate in a SIMD\(^1\) mode with the host processor operating as the controller. However, it is not a tight SIMD since each PE has its own instruction stream stored in its local memory. Nonetheless, the host processor distributes data to the PEs, signals them to start and collects the results. In this mode, the PEs are completely controlled by the host processor because it provides each of them with I/O services and operation control.

In the experiments reported on in this section, each PE executed the coding algorithm explained in Section 3.1.2. Data is loaded into the PE local memory and then the PE is signalled to start processing. When the PE completes its task, it signals the host processor to download the results. The processed images' resolution is 144-by-96 8-bit pels (see Appendix A).

\(^1\) In a true SIMD system, instructions are broadcasted by a controller and are executed by all processors at the same time. Here, processors run the same program, but asynchronously, each under its own control.
The following sections evaluate spatial and temporal data distribution schemes.

![Diagram showing spatial data distribution](image)

Grey connections not used

**Figure 5.2: Star Configuration**

### 5.1.1 Spatial Data Blocking

In this mode, each incoming frame is partitioned into a number of blocks. The data blocks are then distributed to the PEs. The spatial location of a block within a frame decides which PE to be assigned to. Depending on the coding algorithm, a PE may have to process blocks of the same spatial areas in each frame, such as in interframe coding schemes. In the experiments conducted, the image frame was split into two blocks.

The absence of data dependencies among blocks allows images to map naturally to this architecture and permits PEs to process blocks in parallel with no interprocessor communication overhead. In this architecture, the overall system performance is bounded by the slowest PE. In the experiments conducted on this configuration, one PE was about 50\%\(^1\) slower than the other because of the relative complexity of its load share. The necessity of holding the PEs idle while data was transferred in and out of their local static memory increased the overall processing time to 0.126s (from 0.1197s). This overhead comprises loading the two data blocks to the boards and unloading only the second block.

---

\(^1\) It was mentioned, at the beginning of this chapter, that the first half of the image required 0.0808s, whereas the second half required 0.1197s processing time.
from the board. The total overhead time amounts to about 0.75\(^1\) of the 0.009-s overhead of the uniprocessor case, which is the expected performance of this configuration.

![Spatial Block Distribution Scheme](image)

**Figure 5-3: Spatial Block Distribution Scheme**

This configuration required about 59.8\% of the processing time and about 63.0\% of the total turn-around time of the single processor mode.

### 5.1.2 Temporal Data Blocking

In this configuration, integral frames are assigned to PEs. For the intraframe coding schemes, this architecture is often more suitable than the previous one. In this architecture, the processing load is, on the average, shared fairly between the processors because the temporal redundancy maps into similar computational complexities of subsequent frames. This configuration is optimal for intraframe coding schemes.

Due to the fair load distribution in this case, the processing speed was twice that of the uniprocessor case, requiring 0.10s per frame. The data transfer time overhead raised it to 0.107s. Theoretically, it should have been 0.1045s, but in this experimental case loading the two frames is overhead, whereas the unloading of only a single frame represents overhead. This makes the overhead time 0.75 of the uniprocessor case, rather than 0.50 of it (as in Section 5.1.1). Data transfer time was slightly less than that of the previous case because half the memory switching time was saved by transferring a complete frame at

---

1 The first block was already unloaded since the first processor finished well before the second
once rather than a half-frame block.

![Temporal Block Distribution Scheme](image)

**Figure 5-4: Temporal Block Distribution Scheme**

This configuration took 50.1% of the processing time, and 51.2% of the total turnaround time of the uniprocessor mode.

### 5.2 Pipelined Architecture

This architecture operates in the MISP mode. The data stream enters the system at the start of the pipeline. A PE processes the data stream by executing its functional part of the program and passing the partial results to the next PE. At the end of the pipeline, the resulting data is collected by the host processor. Figure 5-5 illustrates data flow paths between processors.

This architecture is suitable for cases where the complete task divides into a number of approximately equally complex, simpler subtasks. In these cases, each PE program can be optimized for one subtask. This configuration allows for both temporal and spatial data dependencies. In this configuration, approximately all the data needs to be passed a number of times equal to the number of units in the pipeline. Transferring data in parallel with processing can convert the pipelining overhead to time delay. In that case, the maximum data transfer rate is the upper bound for system throughput.

The aforementioned algorithm was divided into two parts: the encoding part and the decoding part. The following sections investigate various methods of interprocessor
communication to reduce the pipeline overhead time.

Figure 5-5: Pipeline Configuration

Figure 5-6: Piping Line Groups

The data size involved in the pipeline is 3860 (32-bit) words. Each word requires 3 (60-ns) cycles to be written to the I/O buffer or read from there. Added to that, is 1 cycle to read a word from the processor local memory and 2 cycles to write it there. This amounts to 4 cycles per word to transfer out, or 5 cycles to transfer in. However, using the C30 processor's busses in parallel can reduce this to 3 cycles per word in either direction. That amounts to 1390 µs for piping the complete data frame, reading and writing. The total transfer overhead was measured to be 1.0 ms for piping out and 1.3 ms for piping in. The
net overhead, which is slightly less than 900 μs, can be slightly reduced but cannot be eliminated in any communication scheme. However, the actual transfer time may be considerably reduced or eliminated. A more serious problem than the communication overhead is idle waiting for buffer access.

5.2.1 Program-Driven Communication

The interprocessor communication method used in this case is direct reading and writing to shared mailbox memory under program control. A processor writes the desired data to the mailbox memory, and the next processor reads that data from there. The first processor cannot write the next data block until the next processor reads the previous one.

This method forces the two processors to be synchronized in the sense that a processor stays idle until the previous processor finishes the data block it is working on, and if it finishes its task, it cannot pass it along until the next processor finishes processing its previous data block. This adds waiting overhead on top of the overhead of data passing. The load distribution ratio was about 1:2\(^1\); therefore, the first processor had to always wait for the next processor to read the previous data batch.

This method required 0.1349 s. Thus, it is slower than the parallel spatial data blocking method\(^2\) by 12.6%. It took about 67.4% of processing time and pipeline delay\(^3\) as the reference single processor mode.

The above configuration divided the processing load in a 1:2 ratio, with the second processor being the bottleneck. When the load was redistributed by programming the first processor to perform parts of the Huffman decoding originally intended for the second processor, a 1:1\(^2\) load sharing ratio was achieved, with a very slight additional arrangement cost of 1% (1 ms) of the total time of each processor. The total processing time was reduced to 0.1102 s—55% of the uniprocessor case.

Waiting for buffer access accounts for most of the overhead time. Without waiting for

1. Recall that the first stage of processing takes 0.0684 s and the second stage takes 0.1317 s.
2. This method was selected for comparison due to its flexibility to lend itself to more coding schemes.
3. The host data transfer time was not included here because it represents a constant delay independent of the pipeline scheme.
4. Accurately, the ratio that was achieved was 988:1030, bringing the lower boundary for processing to 0.1030 s.
butler access, the subprograms require 99.9 ms and 104.1 ms, respectively. Waiting time for
butler access, excluding data transfer time, was 2.1 ms and 6.2 ms, respectively. Actual data
transfer time was 1.06 ms and 0.93 ms, respectively. Total execution time was 102.0 s and
110.2 s, respectively.

Table 5-1: Comparison of CPU and DMA Transfer Schemes

<table>
<thead>
<tr>
<th>Aspect</th>
<th>CPU Transfer</th>
<th>DMA Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE1 Idle Waiting</td>
<td>2139</td>
<td>2463</td>
</tr>
<tr>
<td>PE2 Idle Waiting</td>
<td>6201</td>
<td>6042</td>
</tr>
<tr>
<td>PE1 Total* Communication</td>
<td>3198</td>
<td>3468</td>
</tr>
<tr>
<td>PE2 Total* Communication</td>
<td>7128</td>
<td>7419</td>
</tr>
<tr>
<td>PE1 Minimum* Waiting</td>
<td>84</td>
<td>84</td>
</tr>
<tr>
<td>PE2 Minimum* Waiting</td>
<td>105</td>
<td>105</td>
</tr>
<tr>
<td>PE1 Minimum* Communication</td>
<td>984</td>
<td>1092</td>
</tr>
<tr>
<td>PE2 Minimum* Communication</td>
<td>1014</td>
<td>1488</td>
</tr>
<tr>
<td>PE1 Execution Lower Bound</td>
<td>99855</td>
<td>99966</td>
</tr>
<tr>
<td>PE2 Execution Lower Bound</td>
<td>104142</td>
<td>104616</td>
</tr>
</tbody>
</table>

\* Times are in μs.
\* Total refers to normal processing times, minimum refers to cases where a
processor is outputting/inputting data as fast as the other processor can handle,
such that there is no idle time.

5.2.2 Program-Driven DMA Communication

In this experiment, the direct memory access method is used to pass data instead of the
CPU. The CPU stays idle while the DMA transfers data because of the data
synchronization problem described above (Section 5.2.1). The DMA controller is
optimized for memory and I/O data transfers, but in the case of RISC (reduced-
instruction-set computer) processors, there is often no improvement in performance if the
CPU idles while the DMA is working. The DMA requires 3 cycles to transfer a word,
which is equal to that which the CPU requires for the source and destination used in the
program. In this experiment, there was no noticeable improvement on the results in the
previous section. Table 5-1 compares the two methods.

Although using DMA in this sense does not improve on the previous method, it provides useful information to be used in subsequent sections.

Table 5-1 shows that, even when PE1's share is less than PE2's, sometimes PE2 waits idly for PE1 to finish a data batch. Part of this waiting is due to the latency of PE1 to start piping data out. This is generally the case for a pipeline system.

5.2.3 Interrupt-Driven DMA Communication

In previous pipeline configurations, the second processor, PE2, was the bottleneck of the pipeline due to the relative complexity of its load. Relieving the processors of the communication overhead would not make a very significant improvement on performance (because it does not modify the processing load assignment), but it will make solving the more important problem (idle waiting) easier later (Sections 5.2.4 5.2.5). The first processor's (PE1) communication overhead can be reduced to only initializing the DMA, leaving actual data transfer to the DMA. This configuration increases the idle waiting time of the first processor from 2.14 ms to 2.67 ms, because it now proceeds faster and must wait longer for the second processor. However, it also reduces the overall communication time from 3.20 ms to 2.75 ms because the processor does not perform the actual data transfer. The second processor (PE2) is sped up by 0.2 ms which is the overall speed-up of the pipeline. Making the same modification (using interrupts to drive the DMA) on the receiving end speeds up both processors by also cutting down on the idle waiting. It increases the idle waiting of the second processor from 6.0 ms to 6.2 ms but reduces the communication time from 6.9 ms to 6.3 ms, achieving a total speed-up of 0.33 ms. Modification of both programs reduced the overhead from 10.2 ms to 9.67 ms, which is an improvement of 5.2%. This corresponds to 54.8% of the uniprocessor execution time.

The first CPU data transfer overhead can be minimized if the processor does not wait idly at all. This can be done by programming the CPU to initialize the DMA to start the transfer, resume its processing and leave the rest to an interrupt service routine activated at the end of the DMA transfer. This interrupt notifies the next processor of the arrival of data and releases the shared data to be read by the other processor. However, even in this case, the DMA transfers rarely conflict with CPU memory accesses. This memory contention adds a slight proportional cost to the DMA constant incremental cost. This cost is a cycle for several word transfers.
At the receiving end, data arrival activates a data reception interrupt service routine which starts the DMA transfer process. At the end of the DMA transfer, another interrupt occurs to release the shared communication buffer memory and to flag that the data is then inside the processor's local memory. However, this configuration may cause buffer overrun if buffered transfer is not supported. This is addressed in the following section.

5.2.4 Buffered Interrupt-Driven Communication

In the previous experiments, the synchronization between processors, to maintain data integrity, caused idle waiting for communication buffer access. A method for independent I/O is required to break this synchronization and allow processors to work independently and, therefore, efficiently. Independent I/O can be achieved by implementing a two-pointer circular buffer at the receiving end. In this configuration, the transmitting processor sets the DMA to make the data transfer. When the DMA completes the transfer, it interrupts the processor to release the shared mailbox memory.

At the receiving end, an interrupt service routine is triggered by data arrival. This interrupt routine sets the DMA to transfer the data to the circular buffer in the PE local memory. When the transfer is complete, the DMA interrupts the receiving processor to release the shared memory.

The interrupt part of the program (at the receiving PE) inserts data to the buffer at the end-of-buffer queue pointer and the sequential part of the program reads data at the beginning-of-buffer queue pointer. When the beginning of the queue coincides with the end of the queue, the queue is empty. In the empty queue case, the program idles until an interrupt occurs. In this case (buffer underflow), the program leaves the busses for the DMA and transfer occurs at maximum speed.

The CPU transfer version of this configuration is slower by a time period depending on the transferred data block size. This is because the DMA setup takes a constant time regardless of the data block size, whereas the CPU-access transfer method consumes time in proportion to the size of the data transferred.

This setup relieves both processors of most of the communication overhead. However, in the case of buffer underflow, the program at the receiving end must wait idly. The remaining overhead (which is not eliminated) is proportional to the frequency of the transfers, not to their sizes. There is new overhead introduced at the receiving end by the buffer management software. Data is transferred twice, once from the shared memory to
the buffer (under interrupt control) and again from the buffer to the data arrays (under sequential program control). This increases the overall overhead by 0.6% and makes it slower than the unbuffered scheme above (Section 5.2.3). This problem is addressed in the next section.

This configuration achieved a processing time of 56% of the single processor case 12% slower than the ideal.

5.2.5 Delayed, Buffered, Interrupt-Driven Communication

Most of the communication overhead incurred in the previous cases was caused by idle waiting due to buffer underflow. Buffer underflow can be avoided if the transmitting processor starts a sufficient amount of time before the receiving one. In this case, when processing of the frame at the receiving processor commences, a sufficiently sized part of the data has already been received. The underflow of the buffer can be guaranteed never to happen if the processing load is fairly distributed and this is usually possible since parts of the processing can be moved from one processor to the other, especially in the case of computationally cost-constant algorithms. Once the steady state is reached, the system speed becomes the average speed of the slowest processing unit.

If the case of buffer overflow is likely to happen, another circular buffer can be implemented at the transmitting end. Then, the transmitting buffer status can be checked each time the shared memory is read. However, buffer overflow or underflow can be avoided by increasing the size of the receiving buffer (if possible) and by distributing the load reasonably among the processors. This also saves the additional management overhead required by implementing another buffer.

This implementation achieved a processing time slower by 3.7% than the ideal case of dual processors (103.7 ms), i.e., 51.8% of the single processor case.

5.2.6 Reducing Buffering Overhead

The overhead involved in the preceding section consists of two parts: the DMA management overhead and the circular buffer management overhead. DMA management overhead is proportional to the number of transfer batches per frame but almost independent of the data batch sizes. Circular buffer management is proportional to the size of data involved (it also includes a very small component which is proportional to the frequency of transfers). This scheme is the fastest of the pipelining configurations
examined in this chapter.

DMA management cost can be decreased by reducing the number of data batches, which means increasing the batch size. This configuration requires larger memory to hold data buffers. Buffer management cost can be minimized by transferring data directly to the circular buffer without passing through the shared memory stage. This is only possible if the circular buffer can be allocated in the shared memory, which requires a larger size of dual-port memory. Memory contention does not happen when the two processors access the shared buffer simultaneously because one processor writes at the end of the buffer and the other reads at the beginning of the buffer.

A limitation of this configuration is encountered in the case of a full, small or almost full buffer. The buffer transmitting processor waits until the receiving one reads its data and updates the buffer pointer, despite the possibility of its writing data while the other processor is reading (writing is slower than or as fast as reading). This can be solved by the reading processor updating the pointer directly before it starts reading. A fair statistical share of the processing load and a sufficient size of the shared memory buffer can achieve optimal pipeline efficiency.

Reading a word from the shared memory and writing it to the local circular buffer memory in parallel takes 3 cycles and transferring it to the data arrays takes 2 additional cycles; hence, every word costs 5 cycles besides the incremental cost of setting up for the transfer. However, if the circular buffer is allocated in the shared memory, it takes only 3 cycles per word to transfer data from there to the data arrays, thereby saving 2 cycles per word. For the frame size used in the experiments, that amounts to 0.4 ms in addition to the incremental cost saved. This setup also allows for the overlap of shared buffer reading and writing, which makes the peak throughput of the buffer approach 0.33 words per cycle (or about 176 Mbps).

These methods can bring the processing time down to about 51.2% of the single processor case.

5.3 Hybrid Architecture

It is possible to combine processors in a general mode where data is transferred between the processing elements themselves and between the processing elements and the host in both directions.

This general configuration requires an increase in the program modularity, additional
interprocessor communication and protocol complexity. The additional complexity results in a shorter processing time. Depending on the type of algorithm, such a configuration may be impossible to implement or be complex beyond being beneficial or feasible.

![Diagram of data paths in hybrid architectures](image)

**Figure 5-7: Data Paths in Hybrid Architectures**

However, this configuration can be implemented easily in the cases of low-level processing algorithms that do not require a temporal history of data and do not have spatial dependencies. In higher level processing algorithms, transfer of the excess load to another processor may require transfer of much larger-size temporal and spatial data and may cost more time than performing processing on the same processor.

![Diagram of overload sharing](image)

**Figure 5-8: Overload Sharing**

The algorithm investigated in this chapter is low-level. In this case, at the termination of processing the local data, a processor signals its neighbour. The neighbouring processor checks whether there is sufficient data to be shared. If so, it sends the data to the other
processor which makes the processing and returns the data to its original source. Finally, the host collects the data from the processors.

This implementation incurs an additional constant cost of the slow processor to transfer the data and receive the results, but this cost is only incurred if the remaining data takes more time to process. On the receiving (faster) end, the processor requires the same time to receive and then transmit the data back; however, it stays idle at least part of the reception time. If the faster processor waits idle in any case then it is justifiable to incur that cost.

Assuming the same algorithm with the distributed ratio of 2:3 (4:6), the remaining data at the completion time of the first processor will be 20% of the total data. The remaining 20% must be shared equally between the two processors because the data transmission between processors is so fast that it can be ignored. This part of the algorithm is similar to processing this 10% part in pipelining except that the second processor performs all the processing—double its share of a pipelined 10% portion—which is equivalent to 20% of a pipelined frame processing. Since this 10% load is piped twice (the second time it returns to its origin), its transmission time is also equivalent to that of a pipelined 20% load. Hence, the total processing time is 80% of the star processing time (Section 5.1.2) plus 20% of the pipeline processing time (Sections 5.2.1–5.2.3, depending on the data transfer method used). The additional overhead to the ideal star case (Section 5.1.2) is 20% of the pipeline overhead provided that the load ratio variability is less than the overhead ratio (i.e., this overhead time is greater than the processing time difference between the two processors including run-time redistribution). Because of the receiving processor’s idle wait, this case cannot be partially compared to the buffered communication cases, so the processing time will be about 51% \((0.8 \times 50\% + 0.2 \times 55\%)\) of the single processor mode. Including the data transfer time brings it to 52.1% of the reference case. This configuration is faster than the pipeline configurations discussed in Sections 5.2.1–5.2.5 but its complexity is noticeably higher.

Other combined architectures require more than two processors. This is investigated in Chapter 6.

5.4 Summary of Results

Table 5-2 and the bar graph in Figure 5-9 summarize the results of this chapter, showing the processing time of the different two-processor configurations as a percentage of the
single processor case.

![Figure 5-9: Biprocessor Comparative Bar Graph](image)

**Table 5-2: Biprocessor Comparative Performance**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Percentage $^{i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Architecture, Spatial Segmentation</td>
<td>63.9</td>
</tr>
<tr>
<td>Star Architecture, Temporal Segmentation</td>
<td>51.2</td>
</tr>
<tr>
<td>Hybrid Architecture (Overload Sharing)</td>
<td>52.1</td>
</tr>
<tr>
<td>Pipelined Architecture, Program Transfer</td>
<td>55.0</td>
</tr>
<tr>
<td>Pipelined Architecture, DMA Transfer</td>
<td>55.0</td>
</tr>
<tr>
<td>Pipelined, DMA, Interrupts</td>
<td>54.8</td>
</tr>
<tr>
<td>Pipelined, DMA, Interrupts, Buffered</td>
<td>56.0</td>
</tr>
<tr>
<td>Pipelined, DMA, Interrupts, Buffered, Delayed</td>
<td>51.8</td>
</tr>
<tr>
<td>Pipelined, DMA, Interrupts, Buffered, Delayed, Overhead Reduction</td>
<td>51.2</td>
</tr>
</tbody>
</table>

*i: Percentages of single processor processing time*

The difference in performance between the two star configurations is due to the load imbalance in the spatial segmentation mode. In these configurations and in the hybrid configuration, the reported time includes data transfer between the host and the processing elements because this time depends on the number of processors and because the host capacity affects the performance peak of the system. In the hybrid architecture, the
processing load is divided evenly between the processors, but there is an overhead additional to that in the temporal star case because of data transfer between the two PEs, hence the difference in performance.

Using DMA transfer instead of CPU transfer did not improve on the performance of pipelined architectures but made other configurations possible. Buffering alone degraded performance but made isolation of the PEs possible. That improved the performance, permitting reduction of the buffering overhead.
Extension of the Architecture

In Chapter 5, experimental results on multiprocessing were presented. In this chapter, the significance of these results is discussed. They are then generalized to cases involving larger numbers of processing elements. Architectures are discussed which keep data in one single stream, which divide data in parallel and, which combine both.

Figure 6-1 depicts the default architecture of a video processing system. The simplest architecture of the system includes a single processing element. The fixed-rate input stream carrying the raw video data is processed sequentially to produce the fixed-rate output stream that carries the resulting video stream. To increase the video system throughput, data can be split into multiple streams among processing elements or processing elements can be cascaded to process the same data stream.

![Diagram](image)

**Figure 6-1: Video Processing Setup**

6.1 MISD Systems

Processing the incoming video data sequentially in a single stream does not require a mechanism for disassembling and reassembling data streams. Pipeline systems have the ability to perform any level of processing in every processing element because of the
availability of the required data in their local memory. However, a pipeline system is required to pipe data equivalent to the entire original input stream from every processing element to the next. Therefore, the peak performance of the system cannot exceed the pipeline transfer bandwidth. Upon reaching the processing peak permitted by the communication capacity of the pipeline between processors, insertion of additional processors in the system may degrade performance. It is, therefore, beneficial to maximize effective data transfer across links by the design of functional segmentation. A complete pipeline system operating at its data transfer peak is analogous to a single processor system where pipelining cannot be employed to enhance system performance.

![Diagram of General Pipeline System]

**Figure 6-2: General Pipeline System**

### 6.1.1 Functional Segmentation

Often, ideal functional partitioning is not achievable due to the computational cost distribution of program functional segments. A functional segment that requires much processing can be further segmented. This often requires repeating the functional segmentation of the entire algorithm.

Achieving optimal functional segmentation becomes more complicated as the number of processing elements increases. An alternative method (to the logical functional segmentation) is partitioning a given task between two or more successive units by making each unit carry out the task on a portion of the data stream. This technique was used in the experimental pipeline system at the end of Section 5.2.1 and thereafter.

The variability of processing-demand distribution among frames alternates the bottlenecks among processing elements, increasing the difficulty of achieving an optimal functional partition. This problem was encountered in Section 5.2.2 and was solved by partitioning the processing functions according to their statistical computational cost. This behaviour of the program reduces system utilization.
Modifying the pipeline size necessitates redesigning the program segmentation unless it is done to meet the processing demands of the current partitioning.

6.1.2 Communication Management

The communication overhead is the main disadvantage of the pipeline architecture, making it slower than a multiple-data stream parallel architecture. Amongst the various methods of achieving efficient communication, investigated in Sections 5.2.1-5.2.5, the best performance was achieved by employing DMA transfers to an elastic isolation buffer in Section 5.2.5. The communication capacity was reduced by the fact that the shared dual-port memory generates a single wait state, as well as the bus design, which required two additional cycles for writing. The use of no-wait-state memory (25-ns access time) and faster buses (single-cycle read/write) triples the pipeline bandwidth.

If each processing element has only one DMA channel, it can only transfer data at one port (input or output) in parallel with processing. This does not impose a limit on the system if DMA channels are used to either read or write data in all units. It is, in this case, better to use DMA channels for inputting data and to leave outputting data for the CPU as it produces results. When the CPU writes to communication memory, no overhead is incurred if no wait states are generated because the CPU can write data directly to the communication memory instead of storing it in local memory (as is the case if the DMA were to transfer the results out).

Using larger data blocks for DMA transfers decreases the frequency of DMA block transfers and, therefore, considerably reduces the CPU time required to service DMA initialization and interrupts.

6.1.3 Performance of MISD Systems

It was shown, in Section 5.2.5, that the interrupt-driven DMA block-transfer method with an elastic isolation buffer was the least costly communication method, enabling a processing element to either save resulting data in the memory of the next processor with no additional cost, or save it where a constant-cost DMA block transfer can perform the transfer. Data arrival interrupts the receiving processing element to initiate the DMA transfer into local buffer memory. In the experiments carried out in Sections 5.2.1-5.2.5, the approximate computational cost was 500 cycles per word per processor, the program segmentation cost (due to splitting logical segments) was about 1% of the computational
cost, and the communication cost was about 4 cycles per word.

![Diagram](image)

Figure 6-3: DMA and CPU Timing

Ideally, increasing the number of processors would decrease the computational cost per frame per processor proportionally. The segmentation cost per processor would also be expected to decrease. However, the communication cost per processor is almost independent of the number of processors, but dependent on the size of data passed across communication links, which can be slightly controlled by delineating segments where the least data needs to be passed. The optimal delineation is very difficult to achieve and is also limited in effect, as it causes interrupt service routines, for initiating and concluding DMA transfers, to increase gradually as the number of processors increases. Figure 6-3 illustrates the correspondence between CPU and DMA activities as the number of processors increases to saturation; timing is not to scale. Only DMA idle time and CPU processing time are decreased and the CPU overhead remains constant (per frame).

An assumption is made that the data source can operate at data rates as high as the pipeline can support. When the DMA initialization time matches the conclusion time of the previous block transfer, as in the peak case of Figure 6-3, the system throughput becomes bounded by the I/O capacity of the pipeline. A further increase of the number of processors cannot ameliorate the system performance (unless the segment delineations are moved to optimal positions). Before this condition happens, the speed-up achieved by using \( n \) identical processors can be calculated by

\[
S_n = \frac{nt_P}{t_P + t_s + n t_{I/O}},
\]

where \( t_P \) is the processing time required by a single processor, \( t_s \) is the overhead incurred by segmenting the program, and \( t_{I/O} \) is the processor time spent on I/O. If all processors
take the same time to execute then \( t_p^* = t_p \); otherwise \( t_p^* = n t_{pnm} \), where \( t_{pnm} \) is the estimated execution time of the slowest processor. This is valid as long as \( t_{DMAm} \leq t_{pnm} \), where \( t_{DMAm} \) is the maximum time required by any DMA block transfer. In Section 5.2.5, the approximate values of those parameters were \( t_{DMA} = 0.5 \) ms, \( t_v = 2 \) ms, \( t_p = 204.7 \) ms. The above speed-up equation is valid since \( t_{DMAm} \) is about 1 ms. Therefore, speed-up achieved by using 10 processors, calculated by the above equation, is 9.45 and the total overhead is 5.5% (cf. Section 6.2.3).

6.2 SIMD Systems

A multiple data-stream configuration is the optimal architecture when a means of data disassembling and reassembling is used. It is also the only method to enhance the performance of a saturated pipeline system (through hybridization).

6.2.1 Data Segmentation

Data is segmented among processing elements in a spatial or temporal manner. Spatial data segmentation schemes are suitable for algorithms that operate on closely localized data elements. Temporal data segmentation is required by algorithms that operate on distributed data elements, especially in high-level vision and image processing. In distributing data among processors, spatial data segmentation schemes require higher switching speeds (when using multiplexing devices) because each frame is distributed among many processing elements, whereas temporal data segmentation schemes require a frame or more to be assigned to each processor. Therefore, for efficient utilization of a multiprocessor system, there must be a simple ratio between the number of blocks per frame and the number of processors in the system.

Temporal data segmentation schemes that assign single frames to processors are inefficient for interframe coding schemes. For this reason, frames must be distributed to processors in \( n \)-frame batches such that as one processor processes \( n \) frames, the next processor processes the next \( n \) frames. Sequences of frames must be stored in the multiprocessor system controller (host) memory or the local memory of individual processing elements. Assuming \( N \) processors, this segmentation scheme requires additional storage of \( nN \) frames waiting for processing. Distributing this storage media
among processing elements requires every processing element to accommodate \( n \) additional frames in its local memory. This scheme nonetheless can be employed by both intraframe and interframe coding schemes.

![Temporal Distribution](image)

Figure 6-4: Temporal Distribution

![Spatial Distribution](image)

Figure 6-5: Spatial Distribution

In a spatial data segmentation scheme, the algorithm imposes a limit on the number of blocks a frame can be partitioned into. When the number of processors exceeds the number of blocks, more than one frame is required to load all processors. However, in interframe coding schemes, successive blocks corresponding to the same spatial area must be processed by the same processor. This requirement introduces temporal gaps in distributing frames among processors after the last block of a frame has been assigned, otherwise processors would require past image blocks, which are in the memory modules of other processing elements. Transferring those blocks between processors would require significant overhead. For large numbers of processors, spatial data segmentation is
analogous to temporal data segmentation for groups of processors. Each time a new frame sequence is distributed to a processor group, a base frame is transmitted because (due to the unavailability of the previous frame) a predictive frame cannot be encoded.

6.2.2 Synchronization and Communication

A video data source outputs a single stream of constant-rate digital video data, and a video monitor accepts a single stream of the same class. Therefore, a host system is required to carry out data segmentation and reassembly. Data segmentation and reassembly devices can be employed by the host or used independently to carry out this task. Synchronization of processing elements is required because input and output data streams are synchronous.

Data transfer between processing elements and the external environment can be performed via memory access or I/O schemes. The selection of the communication method affects the hardware configuration of the system. Memory-access communication schemes require the host system to provide processing elements with I/O services. However, because of the limitation of the host I/O capacity, I/O communication methods require multiplexing and demultiplexing devices to provide the required I/O services.

The complexity of multiplexing and demultiplexing devices varies according to the data segmentation scheme employed. Segmenting frames into rectangular blocks requires more sophisticated devices. Distributing integral frames among processing elements only requires switching I/O channels synchronously at a relatively low frequency without stress on computation. I/O data acquisition and dispensation devices require processing elements to be tightly synchronized to provide high-speed synchronous data at their output ports in a precise, timely fashion. This is a very strict timing condition which requires the presence of two high-speed serial ports in every processing element that can be serviced independently of processing.

It is assumed here that a host system is used for I/O services. This host system includes necessary high-speed serial data acquisition and dispensation devices instead of distributing those devices among all processing elements. If the I/O load on the host system is very high, input serial ports may be attached to processing elements to relieve the host of half of the I/O load. However, output must be handled exclusively by the host because of its strict timing requirements.

6.2.3 Performance of SIMD Systems

Section 5.1.2 established that the best data segmentation scheme for a multiple data stream
system is the temporal segmentation method, which essentially provides processors with
the same load share. This method is easily scaled to any number of processors.

In this architecture, the host I/O overhead limits the speed-up return of processors.
Increasing the number of processing elements occupies the former host's idle time with I/O
exchanges with processors and external I/O devices, as long as external data rates are
increased to utilize available processing power.

This scheme does not have program segmentation overhead because the same
uniprocessor program is run on all processors. The choice of the I/O exchange method
between the host and processors does not affect the relative speed-up return because all
processors share the same timing distribution. However, the distribution of idle times
imposed by the I/O scheme affects the absolute system speed and the upper bound of
system performance.

The availability of data acquisition and dispensation boards on the host system limits
the host task to transferring data from the acquisition board memory to a processing
element and then transferring the resulting data to the data dispensation board memory.
Using dual-port memory in processing elements and I/O devices provides high-efficiency
data transfers.

At the start of loading, \( n^2 - n \) I/O slots are lost. Then idle time is only lost due to
statistical deviation of the real execution times from the expected execution time. The
effect of this statistical deviation can be minimized by employing buffered I/O. As long as
the idle time of the host is sufficient for the insertion of an additional processor\(^1\), the system
speed-up is lower bounded by \( t_{pm} / n \), where \( t_{pm} \) is the maximum execution time required
by any processor.

Consider the experiment of Section 5.1.2. The uniprocessor overhead \( t_{I/O} \) is 9 ms; the
uniprocessor execution time \( t_p \) is 100 ms. Assuming that \( t_{pm} = 101 \) ms, the speed-up
factor achieved by employing 10 processors can be calculated by

\[
S_n = \frac{nt_p}{t_{pm}} = S_{10} = \frac{10 \times 100}{101} = 9.9.
\]

---

\(^1\) That is to say, host idle time between loading the last PE and unloading the first PE is larger than the time
required to load a PE. If this condition does not hold, PEs must carry out their own input services.
with 17% total overhead (cf. Section 6.1.3). It is required that \( nt_{141} \leq t_{pm} \). This condition is valid up to \( n = 11 \). The bottleneck then is the host, unlike that of the pipeline case where the bottleneck was the pipeline link data transfer capacity. For \( n > 11 \),

\[
S_{n > 11} = \frac{t_{pm}}{t_{l/O}} = \frac{101}{q} = 11.22.
\]

### 6.3 Hybrid Systems

Configurations that combine single and multiple data paths can be exploited for video processing. The architecture of such systems depends largely upon the underlying coding algorithm. Many such configurations employ shared memory or shared bus schemes. That requires arbitration and limits scalability. In this section, systems constituted from units that pass data from port to port are considered. Those systems operate in the MIMD mode.

The advantage of hybrid configurations is that some operations of video coding lend themselves nicely to SIMD processing and others fit well in MISD schemes. By using a combination of both architectures, a pipeline system can be established where appropriate functional segments of the algorithm can be distributed in parallel in a SIMD mode. One method of designing a hybrid system originates from modifying a pipeline system. Hybrid systems can also be designed using arrow network diagrams.

Data is transferred through serial or parallel I/O ports because it cannot be written directly to other processors’ memory without implementing bus windows, which is inefficient [51]. The system can be configured such that any processing element receives all the data it needs to perform its task from the previous stage to eliminate additional communication overhead.

Synchronization is required to ensure that no processing element is overrun with data. This can be handled by exchanging ready signals between a processor and the previous processing stage. Each stage of the pipeline is limited in performance by the slowest processor and the overall system performance is limited by the slowest pipeline stage. If the data rate is relatively high, two processing elements at either end of the system can be used exclusively to handle I/O between other processing stages and the external environment.

Due to system complexity, expanding a hybrid system requires major changes in system software and hardware. To utilize hardware expansion efficiently, system software
requires considerable modifications to keep processing loads shared fairly among processors.

6.3.1 Hybrid System Design

A hybrid system can be designed using a pipeline prototype. A functional division of the algorithm explained in Section 3.1.2 can be as follows: (execution times in ms are in parentheses):

2. Quantization, RLC and VLC of band 3, including packing into ATM cells (52.2).
3. Disassembling the bands into two streams of ATM high priority cells (1.5).
4. Channel simulation (0.1).
5. VLC and RLC decoding of band 3 (115.9).
6. Building the complete image (15.7).

The timing shows that items 3 and 4 can be combined with item 2 (or 5). Item 2 should be split among three processors, and item 5 should be split among 7 processors. This results in the following partition:

- **Band array calculation (14.4),**
- **Quantization, RLC, VLC, Disassembling, Channel Simulation (17.9 x 3),**
- **VLC, RLC decoding (16.6 x 7),**
- **Building the complete image (15.7).**

This requires 12 processors to achieve a processing rate of 55.9 frames per second. The bottleneck is item 2 and the fastest processing is in item 1. Band 3 does not require disassembling because it is done in the VLC coding stage. Therefore, disassembly can be moved to the first (or third) stage resulting in 15.9 at stage one and 17.4 x 3 at stage 2, corresponding to 57.5 frames per second, which in turn corresponds to an efficiency of 96% (the optimal rate is 60 frames per second) which is better than that of the pipeline system (94% efficiency for 12 processors). Increasing the number of processors by one (at the bottleneck stage) achieves a processing rate of 60.2 and an efficiency of 93% (very close to that of the pipeline architecture, despite imposing the additional processor without high computational demand). The performance of this architecture, in general cases, is inferior to that of the temporal distribution star architecture due to the simplicity of its
design which does not provide for fair processing load segmentation and does not require acquisition and dispensation devices. This architecture has no segmentation cost and is suitable for cases where the processing elements are different such that certain segments of the program can be optimally performed by special purpose devices.

Arrow network diagrams\textsuperscript{[52]} can be used to represent the interdependencies and precedence relationships among the tasks of an algorithm. Then, PERF-CPM (project evaluation and review technique-critical-path method) can be used to schedule the program. Every task is represented by an arrow which starts at the end of the tasks that must precede it and ends at the beginning of the tasks that depend on it. In Figure 6-6.a, tasks ABF can be combined in one task AF with the sum of their individual lengths to be the combined task length. Task groups CEG and CDF can also be combined in the same manner, resulting in Figure 6-6.b.

![Figure 6-6: Algorithm Arrow Diagram](image)

A processor or a number of processors is assigned to every arrow (task). If noticeable floats are present in noncritical paths, resource leveling can be used to assign processors to tasks by relocating processors of noncritical paths to share critical tasks so that the number of processors is kept minimal at all points in time and the algorithm is finished in the shortest time possible.

This design method is efficient in the cases where floats are small. This approach is also useful in determining critical tasks in complicated algorithms to estimate the required resources in a general hybrid system.

6.3.2 Performance of Hybrid Architectures

In Section 5.3, it was shown that a hybrid system was superior to the pipeline systems discussed previously. This was due to the flexibility of run-time load distribution. The implementation of this property in a general hybrid system, by interconnecting processors
of the same stage, significantly complicates programming and increases hardware complexity and cost.

Because the desired system is a research tool, system configuration should lend itself to extension and topological changes. Ensuring that data is transferred between first-degree neighbours diminishes transfer costs but, more importantly, requires that topology be tailored according to data dependencies. Expansion of such a system requires redesigning the topology according to the program and data segmentation that conforms to the new number of processors.

![Figure 6-7: Example of Hybrid Systems](image)

Consider expanding the sample system, a part of which is depicted in Figure 6-7. Assume that the DCT stage is more computationintensive than its two neighbouring stages, but the configuration, displayed in Figure 6-7, is optimal for the available number of processors. To estimate the speed-up achieved by adding a processor in the DCT stage, the subsystem constituted from the DCT stage and the two neighbouring stages is considered (PE 1 through PE 6). The additional communication cost incurred by this expansion consists mainly of the software initialization of the additional connection added to each of PE 1 and PE 6. If the communication protocol transmits data blocks intended for the DCT stage on a block-by-block basis, there will be no additional communication cost because the change will only involve a modification of destination addresses. In the case of significant overhead, the performance of the preceding stage is reduced (probably creating a new bottleneck). The speed-up gain of expanding the system is the difference between the new slowest stage throughput and the former DCT stage throughput.

Assume PE 7 is the system bottleneck and it is desired to share its processing load with a processor to be inserted at its output port. In this case, the subsystem to be considered is the longest single processor pipeline that includes PE 7. If rearranging tasks among processors involves dividing the program in a fashion that conflicts with its logical segments, then new segmentation overhead is incurred in addition to the additional I/O
overhead. In most cases, segmentation overhead can be neglected and I/O overhead can be assumed constant for all processors. Therefore, the total load shared by the processors is

\[ t_{\text{total}} = t_{\text{original}} + n t_{\text{I/O}} \]

In the best case, the new execution time of the subsystem will be less than the ideal by \( t_{\text{I/O}} (n + 1) \), otherwise

\[ t_{n+1} = \frac{t_{\text{original}} + (n + 1) t_{\text{I/O}}}{n + 1} \]

The overall system throughput is limited by the new bottleneck stage.

### 6.4 Performance Comparison

Star performance increases almost linearly as the number of processors increases. It saturates as the host bus becomes overloaded with data transfer between the host and the

![Speed-up vs Number of Processors](image)

**Figure 6-8: Pipelined and Star Performance**

PEs. Figure 6-8 compares between speed-up of star and pipelined architectures. (Graph generated from formulae in Sections 6.1.3 and 6.2.3.)

The rate of pipelined architecture speed-up decreases as the number of processors is increased. Processing tasks become smaller while the I/O overhead stays almost constant and gradually dominates processor times. The performance peak is also bounded by the communication bandwidth of the pipeline. The performance of hybrid systems shows no general tendency, but it sometimes exceeds that of the pipelined architecture and
sometimes is below it, depending considerably on the functional distribution of the algorithm tasks.

6.5 Example Design

This section investigates the real time implementation of the algorithm explained in Section 3.1.2, using Banshee boards. It is desired to design a system that can provide real-time processing of 30 frames per second with a frame size of 288-by-192 8-bit pels (256-grey levels). The process starts with the original frame (similar to that in Figure A-1) and ends with the result frame that only differs from the original by the effects of coding and channel errors (similar to that in Figure A-6).

The performance of the algorithm in question depends considerably upon the nature of the images being processed. Worst-case images can take more than double the time required by the image of Figure A-1. To account for such computationally intensive images, a frame slot time of 20 ms per frame is selected instead of the average 33-ms frame time slot (of the 30-frame-per-second rate).

This algorithm, which is desired to be implemented, achieves the performance of 0.2 seconds processing time per 144-by-96 frame on a single Banshee processor; the processing time increases proportionally with image size.

To achieve the required real-time performance of 0.02 seconds per 144-by-96 frame, 10 times the (effective) processing power of a Banshee board is required. Therefore, the effective processing power of 40 Banshee processors is required for the real-time processing of the frame in question (assuming the same image complexity for this frame size).

6.5.1 System Architecture Design

In Section 6.5 above, it was shown that the effective processing power of 40 Banshee processors is required for real-time processing of 288-by-192 256-grey scale video frames. In Sections 5.1.2 and 6.2.3, it was demonstrated that the most efficient data distribution scheme divided data between processors in integral frames (achieving linear efficient speed-up with less than 1% overhead within host I/O limits). In Section 6.2.2, it was suggested that the host system provide I/O services to the processing elements.

1 For an optimal choice, extensive simulations must be carried out.
The incoming data bit rate is:

$$8 \text{ bits/pel} \times 288 \text{ pels/line} \times 192 \text{ lines/frame} \times 30 \text{ frames/s} = 13.3 \text{ Mbps}.$$ 

This data rate must be accommodated at the input and output ports of the host system. The input subsystem must have the ability to receive this high-speed data and store it in dual-port memory for later retrieval by the host system. The output subsystem must carry out the inverse of this operation by reading dual-port memory written by the host and transmitting resulting information in the same data rate it had been received from the video source.

### 6.5.1.1 Data Acquisition System Specifications

Figure 6-10 depicts the block diagram of a data acquisition system. The Synchronous Serial Communications Interface (SSCI) receives serial input data which is then read by the CPU of the device and written into the dual-port memory. The internal CPU executes a program stored in the device EPROM. When it writes a frame to its local memory a bit associated with the corresponding memory block in the Free Frame Memory Register (FFMR) is set; when the host reads a frame from memory, the corresponding flag in

![SSCI Block Diagram](image)

**Figure 6-9: Data Acquisition System**

FFMR is reset. If the host attempts to read an empty memory block, wait states are generated until the block is written by the device CPU (the memory ready signal is ANDed with the corresponding FFMR flag). Attempting to write to a full block generates waiting states for the device CPU and if data is not read by the host promptly, the SSCI may be overrun. If data overrun occurs, the host is notified (for example, to retransmit a previous frame).

The input device CPU is required to write 30 frames per second to its local memory.
This rate is equal to 415 kwords per second. Since the CPU is dedicated to this task, it can achieve that by executing a tight loop of reading the SSCI and writing the RAM (unless data overrun occurs). This I/O task can be handled by a 16-MHz 386 processor. RAM is required to have any access time below 120 nanoseconds. The SSCI is required to handle 13.3 Mbps.

6.5.1.2 Data Dispensation System Specifications

The host reads video frames from the input subsystem and writes them to processing elements in a round robin fashion (analogous to that in Figure 6-4). Each processing element must be read directly before it is loaded with its next frame. The host must carry this out synchronously with high accuracy. Synchronization can be achieved by employing an interrupt mechanism between the host and the input system, but it is better that the system be driven by an internal timer to avoid interrupt conflicts between the input and output devices. Due to the criticality of the output system task and the inability of the host to overcome synchronization errors that may happen there, the output system must generate interrupts to keep the host in synchronism. An interrupt generator (IG) requests host service when a frame memory block is read. The I/O requirements of the output device are the dual of those of the input device; therefore, the specifications of the CPU, SSCI, and RAM in both systems are identical.

6.5.1.3 Host System Specifications

It is necessary for the host to carry out the required I/O transfers for a frame in less than a frame time slot—33 ms. Board and memory bank switching overhead requires about 3 ms. This leaves 30 ms for reading a frame twice: once from the input subsystem and the other
from the processing element memory, and writing it twice: once to the processing element and the other to the output system. The memory accesses involved (per frame) are 55300 32-bit words (the frame size is 13824 words) at an effective timing of less than 540 ns per word (to transfer a frame in 30 ms). If an Intel 386 is the controller of the host, then words can be moved in 4 cycles per word (read and write together, i.e., with an average access time of 2 cycles per word, using repeat move string word instructions, REP MOVSW). Therefore, 3.33 Mcycles per second are dedicated to data transfer (of 30 frames). The processor is mainly required to service 30 interrupts per second, in addition to transferring data (and system peripheral services). Servicing interrupts is inexpensive because most of the interrupt time is spent on data transfer (which was accounted for, above). Hence, a 16-MHz Intel-386-based system is capable of handling the PE and I/O services.

Each I/O subsystem is required to accommodate a minimum of two frames, one to be serially received or transmitted and the other to be accessed by the host. Increasing the available frame buffer size enables the system to avoid synchronization errors.

The processing power required by the system can be furnished by 40 C30 processors. The efficiency of the temporal distribution scheme used here is very high (>99%). The effect of the 1% overhead is overcome by the caution that was exerted in selecting the frame time slot to be 20 ms. The processors are configured in the star architecture with no interconnections required between them.

6.6 Interframe Predictive and Interpolative Coding

In interframe predictive coding, the previous frame is required for a processor to encode predictive and motion-compensated information. To achieve this task using the architecture developed in the preceding section, a processing element is required to receive the previous frame, in addition to the frame to be encoded. Both frames can be read from the input subsystem and written to the processing element. This doubles the number of memory transfers required from the host processor. However, the processor that has been selected for the host is capable of handling this increase in I/O demand. The data buffer of the input subsystem is required to accommodate an additional frame. Depending on the coding scheme, this implementation may require additional processing for the previous frame to be suitable for use as reference.

Another data distribution method can keep the I/O demand from the host processor as it was in the previous section. If the coding scheme uses periodic replenishment, this
distribution method will not incur any reference frame overhead (see Figure 6-4). The host loads each processing element with a group of frames which starts in a frame that will be used for replenishment and ends with the frame before the next replenishment frame. After the last processing element is loaded, the host reads result frames from the first processing element and loads a frame from input for every frame it passes to output. The idle time of a processing element while it is loaded with a group of frames is negligible in comparison with the time required to process them. Interpolative coding schemes can be performed in the same manner. This method saves most of the processing overhead required by the reference frame. However, it requires larger data buffers in the processing elements.

6.7 Colour Coding

Generally, the human eye distinguishes about 32 grey levels but it is capable of distinguishing several hundred discrete color levels\(^{[53]}\). The algorithm, described in Section 3.1.2, used a threshold of 5 to quantize pels, resulting in 24 effective grey-levels (\(L = 256/(5 + (-5) + 1)\)). The MPEG standard uses three arrays for coloured video coding: one for luminance and two for the R and B colour information. The three arrays undergo the same coding operations. In this method, coloured images demand three times the processing power required by monochrome images. It is possible to employ lower resolution for the colour components than that of the luminance component without severely affecting picture quality; this is because luminance contrast partially masks subjective detection of colour sharpness decrease at transition boundaries between colours\(^{[34]}\) (and because of the nature of most images).

Assuming colour resolution of quarter the luminance resolution results in the total of the colour arrays being equal to half the luminance matrix. One colour array can be appended to the other and the same coding scheme can be run on the resulting colour matrix. The version of the coding scheme processing colour information is scaled down by half in the width dimension.

The same coding scheme can process the colour matrix if it was equal in dimensions to the luminance matrix. This selection of colour matrix dimensions makes the luminance and colour matrices interchangeable as far as processing is concerned. It also provides better picture quality. This ratio of chrominance to luminance was used in the HDTV prototype system proposed by Lei and Sun.\(^{[39]}\)

The first method of colour coding demands 150% of the processing power required by
which provides better picture quality and no program modification, demands 200% of the monochrome processing power to provide the same frame output rate.

6.8 Special Considerations

In the previous experiments conducted in Chapter 5, the most important feature of the TMS320C30 processor was rarely used, namely, its high performance in floating-point processing. Experiments that do not use this feature efficiently can be carried out using cheaper fixed-point processors. A problem with implementing those experiments using a TMS320C25 fixed-point processor is the limitation of its memory space (128K 16 bit words divided in half between data and program memory spaces, in Harvard architecture, which is another limitation). Limited stack depth and limited processor synchronization mechanisms are other obstructions to conducting the experiments efficiently.

Special devices can be included in the system to carry out time-consuming tasks that require special-purpose design to be handled efficiently. In the experiments carried out in Chapter 5, the most time-consuming task was Huffman VLC decoding (56% of the total time) because it consisted mainly of searching, which requires frequent branches, unless content-addressable memory (CAM) is used.

When the first codeword is decoded by matching the first bits of the input stream to the codeword table, the codelength of the word is looked up in a codelength table to be used in shifting the decoded bits out to determine the start of the next word. CAM, or programmable logic array (PLA), can be used for codeword matching tables, but a PLA is faster and cheaper. However, if it is desired to modify code words, then using CAM is more flexible. If it is desired to decode multiple streams that are multiplexed into the decoder (such as in decoding the luminance and the two chrominance components, each using a distinct decode table), layers of PLAs can be paged in and out according to the incoming stream (under software control).

Barrel shifters are used to achieve the highest performance. When the output of the input barrel shifter is identified at the input of the PLA, the decoded word and the codelength are outputted in the same cycle. The codelength is available to shift the input barrel shifter and decode the next word in a single cycle. When the accumulated codelength exceeds the longest codeword length, a new codeword is read from input to replace the
Figure 6-11[26] depicts the block diagram of the VLC coder. Barrel shifter 1 acts with register $D_2$ as a 4-bit accumulator of codeword length (by rotating a 1 bit round the 16 bit locations$^1$) to control barrel shifter 0 and to generate the read request signal.$^2$

In applications where one word per cycle decoding speed is not required, ordinary shifters and adders can be used to replace barrel shifter 0 and barrel shifter 1, respectively. In that case, the maximum number of cycles required to decode a word is equal to the maximum codeword length.

To incorporate this decoder in the design presented in Section 6.5.1, it must be modified slightly (besides replacing the barrel shifters). The read line (framed in the figure) $^1$ This bit is loaded on reset in $D_2$ in the bit location that causes no shift in Barrel Shifter 1.

$^2$ When the 1-bit is in the last (n-1st) location, the input to the NOR gate is zero, which results in an active input to the AND gate that activates the Read signal when the Ready control is enabled (outside reset).
must be disconnected so that D0 may be addressed and controlled directly by the board processor. The decoder can be addressed using four address locations on the system bus for D0 (data write), Output (data read), Read (status read), and the fourth for reset (control write). Since each register group (data read/write and status/control) consists of one read-only register and one write-only register, each group can be accessed at one address location with the Read/Write control signal acting as a register select input.

For research purposes, it is better to use CAM instead of PLAs (especially since top speed is not a requirement). Assuming 20 60-ns cycles decoding time per symbol, a 288-by-192 frame consumes 0.07 seconds, saving 51% of the total processing time of the algorithm. The remaining processing demand left for the PE processors is halved, saving 20 processors (replacing them with 20 VLC decoders). Achieving single-cycle decode time saves 4 additional PEs. The cost of 4 PEs must be compared with the cost of upgrading the VLC decoders and losing the flexibility of CAM in order to decide on the optimal choice.

Other parts of the algorithm can be replaced by hardware implementations, especially after their feasibility is proven, so that effective analysis of the rest of the algorithm can be conducted. It is specially useful to implement more general tasks in this fashion so that the implementations can be used if design modifications of the system are made.

Other coding schemes may benefit from a multitude of readily available chips. The DCT transform is a time-consuming task that is used in many coding algorithms. VLSI implementations are proposed for this transform and can be used to carry out simulation of coding algorithms efficiently.

Another instance of using efficient devices to implement time-consuming tasks is the use of floating-point processors. Systems, in which a small proportion of operations are floating-point, can be configured mostly with fixed-point processors together with a limited number of floating-point processors.

6.9 Concluding Remarks

The system architecture proposed in this chapter is not the most effective architecture to carry out every single video coding algorithm. Tailoring the architecture for any narrow class of coding algorithms would cause loss of generality and flexibility as a research tool.

In order to simulate algorithms that use a long history of frame data, it may be necessary to increase the system memory size. Certain algorithms would run most
efficiently in real applications using CAM or other specific devices and architectures. These algorithms can be simulated using the proposed architecture, but they will not perform as fast as they would using their specific implementation architectures. That is why a multiprocessor system is proposed: to compensate for not having the optimum, but highly specialized, hardware architecture while maintaining the required flexibility.
Conclusions

This thesis addressed the use of DSP processors to speed-up simulation of video coding algorithms and ATM channels. Experiments studied the basic aspects of using multiprocessing for this class of image and video processing techniques.

7.1 Results and Contribution

The research showed the promise of using DSP processors to tackle real-time processor-bound tasks. DSP processors are even more efficient in this regard than most high-performance, general-purpose processors. Example techniques were proposed to increase the efficiency of processing using single or multiple processors. Digital video processing using multiprocessor architectures proved to be very efficient and effectively lent itself to data segmentation for multiprocessing.

The coding algorithm used exhibited the problems that occur in most multiprocessor systems, such as the irregularity of processing demand per frame, the irregularity of processing demands for blocks within the same frame, and the irregularity of processing demand of the stages of the algorithm for the same block. Solutions were implemented to diminish the effects of those problems.

In SIMD systems, the main problem was the load imbalance among processors. The suggested solution was to distribute data in integral frames, exploiting the interframe temporal redundancy to distribute the load fairly among processors. The scalability requires minimum modification to the hardware and software of the system. It may be limited by the I/O bandwidth of the host system. This requires changing or upgrading the host processors and the I/O devices. This configuration is found to be more efficient than pipelining, with a performance close to the ideal and an approximately linear speed-up, due to the lack of interprocessor communication among the processing elements. Where
interprocessor communication is required among PEs, the techniques developed for pipelining can be used. The difference from the ideal speed-up is due to the incomplete load balance. As the number of processors increases, imbalance, on the average, increases because of the decrease in temporal redundancy upon which the efficiency of the temporal distribution is based; however, the efficiency (based upon the average frame processing time) is lower bounded by that based upon the most demanding frame.

The main problem faced in the MlSD systems is the data pipelining overhead time control. Due to this overhead, pipelining systems are less responsive to an addition of processors to the system. As the number of processors increases, the load share per processor decreases but the I/O overhead essentially remains constant, causing I/O to dominate the processors' time. System throughput is limited by the I/O bandwidth of the pipeline links. This limit can be extended by segmenting programs across the points where the least data is required to be passed to the next processing elements (although the number of PEs may need to be increased for adjusting the program delineation). However, software modifications are required whenever the system is extended. Thus, various methods of data transfer across pipeline links were evaluated. Larger DMA data block sizes exhibited more efficiency due to the need for interprocessor data passing being lowered and the I/O set-up overhead time being minimized. However, using larger block sizes requires that DMA operation be synchronized with program operation to keep utilizing the communication link for a maximum amount of time, without requiring huge memory space. However, the data transfer set-up may cause a frame period lag between every two neighbouring processing elements. However, pipeline architectures can process data at any level without the need for new communication overhead or special data distribution arrangements.

Different data transfer methods were studied and the efficiency of shared memory data paths was observed. This study showed the advantage of using DMA and interrupt methods in decreasing communication overhead and performing I/O and processing in parallel. It also emphasized the benefits of buffered I/O.

Hybrid architectures were considered and shown to be very efficient for small system sizes, but as the system size grows, hardware and software complexity increases exponentially and they become difficult to manage and program. Using heterogeneous devices and systems to share the processing load was discussed.

Finally, a typical system was designed that can carry out real time video processing. The design approach was generalized to interframe and colour coding schemes with finite frame history requirement as is generally the case.
The approach used in this thesis can be applied to many real-time systems where high processing power is required. It may also be used to speed-up virtual time systems. If interprocessor communication is required, it can be performed using the methods developed for pipelining, even in SIMD systems.

7.2 Future Work

The ATM environment is still under research, especially VBR video transmission. There is a high level of interaction between VBR video coding algorithms and the ATM environment specifications, especially those of the adaptation layers, which are also still under research.

Since the target is to implement inexpensive, flexible video codecs to be used in households and offices, considering the economics of the hardware implementation of those devices may affect the underlying algorithms and adaptation layers. In any case, research should be done on VLSI implementation of real-time video coding systems. Some codecs have actually been proposed and VLSI architectures have been proposed as showing promise for this scheme. More research on this has to be done to arrive at inexpensive products.

Current research papers usually discuss a particular video coding scheme and attempt to prove that it provides high quality images and high compression ratios. The future implementations may include multisystem products that automatically allow for changing coding systems such that the broadcasting industry may use the coding system that suits the specific program being transmitted. Trying to integrate such coding systems may affect coding algorithm development, especially as there are some common routines between most coding schemes. Also, the user may chose to change the required video quality, thus requiring the end product to be more flexible.

More complicated multiprocessor DSP architectures are becoming easily implemented by the advent of DSP processors tailored for multiprocessing such as TI TMS320C40. Such processors not only make the implementation of conventional multiprocessor configurations easier and more efficient, but they also provide for new configurations. Hardware support of concurrent interprocessor communication, while processing is going on, changes parameters affecting the efficiency and performance of certain architectures. Therefore, researching these architectures may open up new aspects of multiprocessing, image processing and signal processing.
Visual Illustration of the Algorithm

A.1 Frequency Band Division

This appendix gives visual illustrations of the image coding algorithm that was used throughout the experimental part of this work. The algorithm filters the image into three bands, the sum of which is the complete image.

The video film used in the illustrations was taken by a video camera in a car moving to the right. Two frames of the film were used to produce the results in this thesis. Other
pictures were used to verify the representative nature of the sample used, but those pictures were not used for producing any results. Figure A-1 depicts the original image so that it can be compared with images that resulted from filtering. The picture resolution of the original illustrations is 360-by-240 of 256-level grey scale.

A.1.1 Low Frequency Band Component

The first component of an image consists of the superpels, which are 6-by-3-pel rectangles. Figure A-2 depicts the result of passing the original image through a superpel band filter.

![Figure A-2: Picture Superpel Component](image)

A.1.2 Medium Frequency Band Component

After the superpels are subtracted from the original image, it is further passed through a filter that calculates the triads which are 3-pel elements. The array of triads is appended to the array of superpels and the compound array is disassembled into ATM cells and is then transmitted in the high-priority stream. Figure A-3 depicts the absolute values of the triads. Blank areas correspond to zero values and darker areas correspond to areas of higher intensity. (The pels are inverted: the dark is light and the light is dark because the
original ones have small values that would generate a picture too dark for pels to be discerned.)

Figure A-3: Picture Triad Component

Figure A-4: Picture High-Priority Component

Figure A-4 depicts the picture reconstructed from the superpels and triads (i.e., from
DC triads). The picture is a collection of 3-pel sets.

A.1.3 High Frequency Band Component

After the superpels and triads are subtracted from the original image, the remaining full resolution information corresponding to the fine details of the picture is encoded and transmitted in the low priority stream. Figure A-5 depicts the finer details of the picture in absolute value. (This picture is inverted.) Before transmission, this information is threshold-clipped, run-length-encoded and Huffman-encoded into ATM cells.

Figure A-5: Picture Differential Pel Component

A.2 Data Transmission Cell Loss Effects

Of the transmission errors, cell loss is the error which affects the picture quality the most. Lost cells of a frame can be replaced by cells from the same coordinates of the previous frame or by cells calculated according to a cell compensation algorithm. Lost cells in the following illustrations are assumed to be replaced by zero value data.

In the actual algorithm, lost cells were selected by a geometric random distribution function. High-priority and low-priority cells have different adjustable error rates. In the following illustrations, cell loss errors are generated deterministically for illustration purposes.
A.2.1 High-Priority Cell Loss Errors

High-priority cells carry superpel and triad information. Each cell carries 40 information octets (bytes). The first cell in a line group carries 40 superpels that represent most of the information in a rectangular area of 240-by-3 pels.

![Figure A-6: Picture with Cell Loss Errors](image)

Figure A-6 shows the picture with four missing cells. The dark line across the picture is caused by losing a cell of superpels. Another high priority cell was lost, but the information load of that cell belongs to the two raster lines at which the solid arrow points. Because the cell contained triad data, its loss is hardly detectable. Losing many triad cells results in areas of the frame with only the superpel component as in Figure A-2.

A.2.2 Low-Priority Cell Loss Errors

Cell loss errors in low-priority, line details are much less detectable than those in coarser details. The dashed arrow in Figure A-6 points to the line where two low priority cells were lost. The distortion introduced by the loss of those two cells is unlikely to be detected using monitors of small to medium sizes, especially because of the frame's timely updating. Losing larger sequences of low-priority cells makes areas of the frame look like in Figure A-4.
Bibliography


83


END
15-1 2-93
FIN