# Computational Redundancy in Image Processing

by

Farzad Khalvati

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

© Farzad Khalvati 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Farzad Khalvati

# Abstract

This research presents a new performance improvement technique, *window memoization*, for software and hardware implementations of local image processing algorithms. Window memoization combines the memoization techniques proposed in software and hardware with a characteristic of image data, *computational redundancy*, to improve the performance (in software) and efficiency (in hardware) of local image processing algorithms.

The computational redundancy of an image indicates the percentage of computations that can be skipped when performing a local image processing algorithm on the image. Our studies show that computational redundancy is inherited from two principal redundancies in image data: coding redundancy and interpixel redundancy. We have shown mathematically that the amount of coding and interpixel redundancy of an image has a positive effect on the computational redundancy of the image where a higher coding and interpixel redundancy leads to a higher computational redundancy. We have also demonstrated (mathematically and empirically) that the amount of coding and interpixel redundancy of an image has a positive effect on the speedup obtained for the image by window memoization in both software and hardware.

Window memoization minimizes the number of redundant computations performed on an image by identifying similar neighborhoods of pixels in the image. It uses a memory, reuse table, to store the results of previously performed computations. When a set of computations has to be performed for the first time, the computations are performed and the corresponding result is stored in the reuse table. When the same set of computations has to be performed again in the future, the previously calculated result is reused and the actual computations are skipped.

Implementing the window memoization technique in software speeds up the computations required to complete an image processing task. In software, we have developed an optimized architecture for window memoization and applied it to six image processing algorithms: Canny edge detector, morphological gradient, Kirsch edge detector, Trajkovic corner detector, median filter, and local variance. The typical speedups range from 1.2 to 7.9 with a maximum factor of 40. We have also presented a performance model to predict the speedups obtained by window memoization in software.

In hardware, we have developed an optimized architecture that embodies the window memoization technique. Our hardware design for window memoization achieves high speedups with an overhead in hardware area that is significantly less than that of conventional performance improvement techniques. As case studies in hardware, we have applied window memoization to the Kirsch edge detector and median filter. The typical and maximum speedup factors in hardware are 1.6 and 1.8, respectively, with 40% less hardware in comparison to conventional optimization techniques.

# Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Mark Aagaard, for his guidance, support, and encouragement throughout my PhD program. His tireless and detailed involvement in my research has been an invaluable asset for me. I am very grateful that he believed in me and gave me an opportunity to explore and discover my potentials in realizing new ideas.

I am deeply grateful to Professor Hamid Tizhoosh for his ongoing support in both my research work and private life. He has been a mentor and a friend to me and I am very thankful for all the encouragements and scientific advices that I have received from him.

I would like to thank the external examiner of my thesis, Professor W. James MacLean, whose constructive and thorough comments helped to further improve my dissertation.

I am grateful to Dr. Masoud Makrehchi for very fruitful discussions that helped me solve some of my research problems.

I would like to thank my parents, Sirous and Parvin Khalvati, and my sister, Behnaz, for their encouragement and support. I am very thankful to my brother, Behzad, and his wife, Maryam, for their support. Behzad has always been a good friend to me on whom I have relied so many times. I am very grateful for his generous support and presence that he offered me whenever I needed.

My many thanks go to my cousins, Reza, Mehdi, and Shahram Amirnina, Babak Kheyrizad, and Ali Khalvati who have always been very supportive of me.

I have been very fortunate to have so many friends who have always supported me and enriched my life. To name a few, I would like to thank Mark Bernhardt, Morteza Pasandideh, M. Reza Pasandideh, Nader Javaani, Reza Attarian, Rezvan Hemmesi, Elham Ashari, Rasoul Keshavarzi, Shahryar Rahnamayan, Mario Ventresca, Payvand Parvizi, Amirhossein Hajimiragha, Kourosh S. Razavi, Mehdi Kianpour, Arsen Hajian, and Shahram Yousefi.

I want to express my sincerest gratitude to my father in law, Dr. Abbass Shafaei, for inspiration, encouragement and unconditional support. Through all these years, he has always been a source of energy for me. Whenever it was a difficult time, he was always there to help. I owe him this success and I am very thankful to him. I also would like to thank Dr. Shafaei's family for their support.

I am deeply grateful to my wife, Mitra Shafaei, for all the love, support, encouragement, and energy that she has given me. This thesis would not have been possible without her countless sacrifices. Through this long journey, Mitra was my patient and loving companion and we walked the path together. Through tough times, when it was not easy to be optimistic about the outcome of my research, she always gave me hope and energy. Her constant love and support have enabled me to complete this journey.

Finally, I would like to dedicate this thesis to my son, Arta, and hope that knowledge and wisdom will be his guide all through his life.

*To my son, Arta.*

# Contents

**List of Tables** xi

**List of Figures** xiii

**1 Introduction** 1

**2 Background and Related Work** 7

2.1 Image Data Redundancy . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.1.1 Psychovisual Redundancy . . . . . . . . . . . . . . . . . . . . . 8

2.1.2 Coding Redundancy . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1.3 Interpixel Redundancy . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 Computational Redundancy . . . . . . . . . . . . . . . . . . . . . . . . 13

2.2.1 Data Locality . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.2.2 Memoization Techniques in Software . . . . . . . . . . . . . . . . 14

2.2.3 Memoization Techniques in Hardware . . . . . . . . . . . . . . . 17

2.3 Local Image Processing Algorithms . . . . . . . . . . . . . . . . . . . . 20

2.4 The Taxonomy of Local Image Processing Algorithms . . . . . . . . . . . 22

2.5 Case Study Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

2.5.1 Canny Edge Detection Algorithm . . . . . . . . . . . . . . . . . . 28

2.5.2 Morphological Gradient . . . . . . . . . . . . . . . . . . . . . . 30

2.5.3 Kirsch Edge Detection Algorithm . . . . . . . . . . . . . . . . . . . 31
2.5.4 Trajkovic Corner Detection Algorithm . . . . . . . . . . . . . . . . 33
2.5.5 Median Filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.5.6 Local Variance . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.6 Sets of Test Images . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

**3 Theory of Window Memoization 38**

3.1 Image Reusability for Local Image Processing Algorithms . . . . . . . . . 39
3.1.1 Computational Redundancy in an Image . . . . . . . . . . . . . . . 40
3.1.2 Perfect Window Memoization . . . . . . . . . . . . . . . . . . . . . 42
3.1.3 Tolerant Memoization . . . . . . . . . . . . . . . . . . . . . . . . 45
3.1.4 Reusability of an Image . . . . . . . . . . . . . . . . . . . . . . . 47
3.1.5 Ideal Reusability of an Image . . . . . . . . . . . . . . . . . . . . 50
3.1.6 Laplace Model for the Probability of Symbols . . . . . . . . . . . . 51
3.1.7 Laplace Model for Uniqueness Probability of Symbols . . . . . . . 53
3.2 Ideal Reusability of Image Data versus Coding and Interpixel Redundancy of the Image . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.2.1 Entropy of the Transformed Image versus Coding and Interpixel Redundancy of the Original Image . . . . . . . . . . . . . . . . . . 56
3.2.2 Standard Deviation of the Transformed Image versus Entropy of the Transformed Image . . . . . . . . . . . . . . . . . . . . . . . . 57
3.2.3 Ideal Reusability of an Image versus Standard Deviation of the Transformed Image . . . . . . . . . . . . . . . . . . . . . . . . . . 59
3.2.4 Summary of Ideal Reusability versus Coding and Interpixel Redundancy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.3 Hit Rate versus Coding and Interpixel Redundancy . . . . . . . . . . . . . 63
3.3.1 Reusability versus Ideal Reusability of an Image . . . . . . . . . . 64

3.3.2 Hit Rate of Perfect Window Memoization versus Reusability of an Image . . . 68
3.4 Summary . . . 72

**4 Window Memoization in Software 76**

4.1 Window Memoization Technique in Software . . . 77
4.2 Memoization Mechanism for Window Memoization in Software . . . 83
4.2.1 Generating Symbols . . . 84
4.2.2 Mapping Scheme for Memoization Mechanism . . . 86
4.2.3 Hash Function for Direct-Mapped Mapping Scheme . . . 87
4.2.4 Tolerant Memoization in Software . . . 91
4.2.5 Summary of Design Decisions for Memoization Mechanism . . . 92
4.3 Revisiting the Memoization Overhead Time Equation . . . 92
4.3.1 Memoization Overhead Time: A Linear Model . . . 93
4.3.2 Memoization Overhead Time: A Nonlinear Model . . . 96
4.3.3 Memoization Overhead Time: A Simplified Nonlinear Model . . . 96
4.4 Speedup Model Validation . . . 100
4.4.1 Model Validation for Processor 1 . . . 101
4.4.2 Model Validation for Processors 2 and 3 . . . 104
4.5 Empirical Speedup Results . . . 105
4.6 Speedup versus Coding/Interpixel Redundancy . . . 112
4.6.1 Hit Rate of Window Memoization in Software versus Computational Redundancy . . . 112
4.6.2 Speedup versus Hit Rate of Window Memoization In Software . . . 115
4.6.3 Speedup of Window Memoization in Software versus Coding/Interpixel Redundancy . . . 118
4.7 Summary . . . 122

**5 Window Memoization in Hardware** **124**

5.1 Efficiency of a Design . . . 125

5.1.1 Relative Efficiency of an Optimized Design versus a Base Design . 125

5.1.2 Relative Efficiency of Superscalar Pipeline versus Scalar Pipeline . 127

5.2 Window Memoization Technique in Hardware . . . 130

5.2.1 Architecture of Window Memoization in Hardware . . . 130

5.2.2 Speedup of Window Memoization in Hardware . . . 132

5.2.3 Sprawl of Window Memoization in Hardware . . . 134

5.3 Design Decisions for Window Memoization in Hardware . . . 135

5.3.1 Speedup . . . 136

5.3.2 Sprawl . . . 138

5.4 Parallel Reuse Tables Based on Bloom Filters . . . 140

5.4.1 Parallel Bloom Filters . . . 140

5.4.2 Parallel Reuse Tables as Parallel Bloom Filters . . . 141

5.4.3 The Probability of False Positives for Parallel Reuse Tables . . . 143

5.5 Tolerant Memoization in Hardware . . . 145

5.6 An Optimized Architecture for Window Memoization in Hardware . . . 148

5.7 Results . . . 150

5.7.1 Speedup and Accuracy Results . . . 151

5.7.2 Sprawl Results . . . 153

5.7.3 Efficiency Results . . . 155

5.8 Speedup versus Coding/Interpixel Redundancy . . . 156

5.9 Summary . . . 159

**6 Conclusion** **161**

**Appendices**

**A Notations and Terminology** **167**

**B Autocorrelation as a Measure for Interpixel Redundancy** **169**

**C Results for Window Memoization in Software** **174**

**D Results for Window Memoization in Hardware** **183**

**Bibliography** **187**

# List of Tables


2.1 Characteristics of the case study algorithms . . . . . . . . . . . . . . . . . . 27

2.2 Canny edge detector algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 30

2.3 Morphological gradient algorithm . . . . . . . . . . . . . . . . . . . . . . . . 32

2.4 Trajkovic corner detection algorithm . . . . . . . . . . . . . . . . . . . . . . 34

2.5 Median algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.1 RMSE for ideal reusability plots . . . . . . . . . . . . . . . . . . . . . . . . 61

4.1 Processors used in experiments . . . . . . . . . . . . . . . . . . . . . . . . . 77

4.2 Design decisions for memoization mechanism in software . . . . . . . . . . 84

4.3 Generating symbol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

4.4 Generating symbol using the overlapping windows . . . . . . . . . . . . . . 86

4.5 RMSEs (%) for linear curve fit for $t_{memo}$ for different $RT$ sizes on three processors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

4.6 RMSEs (%) for quadratic curve fit for $t_{memo}$ for different $RT$ sizes on three processors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

4.7 RMSEs (%) for simplified quadratic curve fit for $t_{memo}$ for different $RT$ sizes on three processors, using only two extreme images . . . . . . . . . . 99

4.8 RMSEs (%) for speedups on processor 1 . . . . . . . . . . . . . . . . . . . . 102

4.9 Optimal $RT_{size}$ for for processor 1 . . . . . . . . . . . . . . . . . . . . . . 104

4.10 RMSEs (%) for speedups on processors 2 and 3 . . . . . . . . . . . . . . . . 105

4.11 Optimal $RT_{size}$ for for processor 2 . . . . . . . . . . . . . . . . . . . . . . . 105

4.12 Optimal $RT_{size}$ for for processor 3 . . . . . . . . . . . . . . . . . . . . . . . 105

5.1 Design decisions for memoization mechanism in hardware . . . . . . . . . 136

5.2 Average speedups and results accuracy for in hardware . . . . . . . . . . . 152

5.3 Hardware area consumed by different components of the designs . . . . . 155

5.4 Relative efficiency for Kirsch edge detector . . . . . . . . . . . . . . . . . . . 156

5.5 Relative efficiency for median filter . . . . . . . . . . . . . . . . . . . . . . . . 156

C.1 Speedups for processor 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174

C.2 Speedups for processor 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175

C.3 Speedups for processor 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175

C.4 Accuracy of the results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175

C.5 Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for natural images 177

C.6 Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for industrial images 178

C.7 Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for medical images 179

C.8 Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for barcode images 179

C.9 Accuracy and error margin for relation $HR_{sw} \propto^{+} speedup$ for natural images 179

C.10 Accuracy and error margin for relation $HR_{sw} \propto^{+} speedup$ for industrial images . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179

C.11 Accuracy and error margin for relation $HR_{sw} \propto^{+} speedup$ for medical images 179

C.12 Accuracy and error margin for relation $HR_{sw} \propto^{+} speedup$ for barcode images . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179

# List of Figures


2.1 Image processing chain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

2.2 A $3 \times 3$ window . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

2.3 Pair of opposite points in a circular window . . . . . . . . . . . . . . . . . 33

2.4 Pairs of opposite points for a $3 \times 3$ circular window . . . . . . . . . . . . . 33

2.5 Sample images from: $row_1$: natural, $row_2$: industrial, $row_3$: medical, and $row_4$: 2-D barcode images. . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.1 Left: original images. Right: the probability density function of symbols in the original image based on $\delta$. . . . . . . . . . . . . . . . . . . . . . . . . 54

3.2 Left: original images. Right: the uniqueness probability density function of symbols in the original image based on $\delta$. . . . . . . . . . . . . . . . . . 55

3.3 Entropy versus standard deviation for the transformed images . . . . . . . 59

3.4 Ideal reusability of images versus the standard deviation of transformed images for different $\delta$ ($\delta = 0, 1, 2, 3$) . . . . . . . . . . . . . . . . . . . . . . 62

3.5 Probability density function of symbols based on $\delta$ . . . . . . . . . . . . . 66

3.6 Uniqueness probability density function of symbols based on $\delta$ . . . . . . 67

3.7 Probability density function of symbols . . . . . . . . . . . . . . . . . . . 71

3.8 Left: original images. Right: the probability density functions of image symbols. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

3.9 Computational redundancy versus coding and interpixel redundancy . . . 75

4.1 Flowchart of window memoization . . . . . . . . . . . . . . . . . . . . . . . 78

4.2 Left: $t_{key}$ for all three processors. Right: hit rate versus the reuse table size. 90

4.3 The difference in speedups of the multiplication and division methods . . 91

4.4 Average hit rate and SNR versus the number of the most significant bits used for assigning windows to symbols. Infinite SNRs have been replaced by SNR of 100. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

4.5 Linear model for $t_{memo}$ versus $HR_{sw}$ for a $16K$ entries reuse table . . . . 95

4.6 Nonlinear model for $t_{memo}$ versus $HR_{sw}$ for a $16K$ entries reuse table . . 97

4.7 Simplified nonlinear model for $t_{memo}$ versus $HR_{sw}$ for a $16K$ entries reuse table using only two extreme images . . . . . . . . . . . . . . . . . . . . 99

4.8 RMSEs for different reuse table sizes for linear, quadratic and simplified quadratic model for $t_{memo}$ versus $HR$ . . . . . . . . . . . . . . . . . . . . 100

4.9 Speedup curves for the empirical data and the model for processor 1 . . . 103

4.10 Speedup curves generated by the model for processor 2 . . . . . . . . . . . 106

4.11 Speedup curves generated by the model for processor 3 . . . . . . . . . . . 107

4.12 Average empirical speedup results . . . . . . . . . . . . . . . . . . . . . . 108

4.13 Accuracy of results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

4.14 Results for natural images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, middle: window memoization results, bottom: difference images. . . . . . . . . . . . . . . . . . . . . . 110

4.15 Results for natural images. Left to right: Corner detection, median filter, and local variance. Top: original results, middle: window memoization results, bottom: difference images. . . . . . . . . . . . . . . . . . . . . . . 111

4.16 Error margins for relation $Comp_r \propto^+ HR_{sw}$ . . . . . . . . . . . . . . . . . 115

4.17 Hit rate versus computational redundancy for natural images . . . . . . . 116

4.18 Error margins for relation $HR_{sw} \propto^+ speedup$ . . . . . . . . . . . . . . . . 118

4.19 Error margin for relation $C_r + IP_r \propto^+ speedup$ with 95% accuracy . . . . 119

4.20 Average error margins for relation $C_r + IP_r \propto^+$ *speedup* and its intermediate steps with an accuracy equal to or above 95% . . . . . . . . . . . . . 120

4.21 Speedup versus coding/interpixel redundancy for natural images run on processor 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122

5.1 Right: scalar pipeline. Left: 2-wide superscalar pipeline. . . . . . . . . . . 128

5.2 2-wide superscalar pipeline with window memoization . . . . . . . . . . . 131

5.3 Parallel reuse tables for window memoization in hardware . . . . . . . . . 143

5.4 The probability of false positives versus $RT_{width}$ . . . . . . . . . . . . . . . 145

5.5 Left: average SNR versus the number of the most significant bits used for assigning windows to symbols. Infinite SNRs have been replaced by SNR of 100. Right: speedup versus the number of the most significant bits used for assigning windows to symbols. . . . . . . . . . . . . . . . . . . . . . . . 147

5.6 An optimized architecture for 2-wide superscalar pipeline with window memoization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

5.7 Average hit rates and false positives . . . . . . . . . . . . . . . . . . . . . 151

5.8 Results for natural images for Kirsch edge detector. Top: original results, middle: window memoization results, bottom: difference images. . . . . . 153

5.9 Results for natural images for median filter. Top: original results, middle: window memoization results, bottom: difference images. . . . . . . . . . . 154

5.10 Uniqueness probability density functions . . . . . . . . . . . . . . . . . . . 158

5.11 Speedup versus coding/interpixel redundancy . . . . . . . . . . . . . . . . 160

C.1 Results for industrial images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176

C.2 Results for industrial images. Left to right: corner detection, median filter, and local variance. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176

C.3 Results for medical images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 177

C.4 Results for medical images. Left to right: corner detection, median filter, and local variance. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 177

C.5 Results for barcode images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178

C.6 Results for barcode images. Left to right: corner detection, median filter, and local variance. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178

C.7 Speedup versus coding/interpixel redundancy for industrial images run on processor 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 180

C.8 Speedup versus coding/interpixel redundancy for medical images run on processor 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181

C.9 Speedup versus coding/interpixel redundancy for barcode images run on processor 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182

D.1 Results for industrial images for Kirsch edge detector. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . 183

D.2 Results for industrial images for median filter. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . 184

D.3 Results for medical images for Kirsch edge detector. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . 184

D.4 Results for medical images for median filter. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . 185

D.5 Results for barcode images for Kirsch edge detector. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . 185

D.6 Results for barcode images for median filter. Top: original results, bottom: window memoization results. . . . . . . . . . . . . . . . . . . . . . . . . . 186

# Chapter 1

# Introduction

In recent years, as a result of advances made in digital computers and digital cameras, image processing algorithms have been used widely in applications that affect our daily lives including medical imaging (*e.g.* MRI and Ultrasound), security, navigation, multimedia, industrial inspection, and astrophysics. Many of these applications are used in soft and hard real-time systems where it is crucial to meet the performance requirements. On the other hand, image processing algorithms are usually data-intensive. This makes it both crucial and challenging to improve the performance of image processing algorithms. In soft real-time systems that require performing image processing algorithms, such as MRI, although it is not fatal not to have a high-performance system, in order to increase patient access to MRI scans, it is absolutely crucial to speed up the computations. In hard real-time applications, such as a visual based vehicle navigation system, it may be fatal not to be able to process the incoming images quickly enough.

Traditionally, software running on microprocessors and digital signal processors has provided adequate computational power to cope with the real-time image processing challenges. However, as image resolution and complexity of algorithms increase, the conventional approaches of software implementations of image processing algorithms are not able to handle these challenges. A common solution to meet the performance requirements of real-time image processing is specialized hardware implementations of the algorithms. Nevertheless, there are drawbacks to this solution. First, hardware design is much more difficult and time consuming than writing software. Thus, in situations where a long time to market cannot be tolerated, the hardware approach may not be feasible. Second,

in order to benefit from the parallel processing nature of hardware, design optimization techniques such as pipelining (scalar and/or superscalar) must be used. Implementing such optimization techniques requires a large amount of hardware area, which increases the cost of the design.

The main goal of this research is to develop an innovative optimization technique, *window memoization*, which improves real-time image processing. Window memoization improves the performance (in software) and efficiency (in hardware) of local image processing algorithms. Local processing algorithms that deal with local features in images constitute a large portion of low-level and mid-level image processing algorithms. In software, the goal is to speed up the calculations required to complete an image processing task. This will reduce the cost of embedded systems by using microprocessors instead of specialized hardware. Employing the window memoization technique in software increases the ability of embedded processors to meet the performance requirements of real-time image processing systems and hence, in many cases, it eliminates the need to migrate to hardware-based systems. In hardware, the objective is to speed up the calculations with significantly less hardware area than conventional performance improvement methods.

In this work, we investigate the underlying characteristics of images that make window memoization an effective optimization. We implement window memoization in software and hardware. In software, the typical speedups range from 1.2 to 7.9 with a maximum factor of 40. In hardware, the typical and maximum speedup factors are 1.6 and 1.8, respectively, with less than 20% extra hardware area.

The main initiative behind this research is to reduce the amount of computation that an image processing algorithm must perform. The underlying basis of reduction is to remove the *redundant computations*; the computations that are not necessary to perform in order to complete an image processing task. Analogous to image compression algorithms that exploit data redundancy to reduce the size of images, computational redundancy can be used to reduce the amount of computation and hence, to improve the performance of image processing algorithms. In software, removing the computational redundancy of an image processing task speeds up the whole task. In hardware, doing so decreases the amount of hardware area needed to speed up the calculations in comparison to conventional optimization techniques.

Window memoization uses a reuse table to store the results of previously performed computations. When a set of computations has to be performed for the first time, the computations are performed and the corresponding result is stored in the reuse table. When the same set of computations has to be performed again in the future, the previously calculated result is reused and the actual computations are skipped.

Image data has two principal types of redundancy: coding redundancy and interpixel redundancy. Coding redundancy is due to the fact that images are usually represented using more bits per pixel than is actually needed. Interpixel redundancy is a result of the correlation among neighboring pixels. Our studies show that the coding and interpixel redundancy of an image result in having similar neighborhoods (or windows) of pixels in the image. For a given algorithm, similar windows in an image produce the same results and thus, they are reusable. The percentage of similar windows in an image indicates the computational redundancy of the image.

We have shown that the amount of coding and interpixel redundancy of an image has a positive relationship with the computational redundancy of the image, which means that an image with a higher coding and interpixel redundancy will have a higher computational redundancy. In doing so, we have defined a few intermediate steps and concepts: *perfect window memoization*, the *hit rate* of perfect window memoization, the *reusability of an image* and the *ideal reusability of an image*. Perfect window memoization is a high level processing model, which gives an upper-bound for the potential performance gain achieved by window memoization in software and hardware. The hit rate of perfect window memoization for an image is the percentage of windows in the image that match a symbol in the reuse table. Perfect window memoization gives the maximum hit rate that the software and hardware implementations of window memoization can possibly achieve. The reusability of an image gives the average number of unnecessary sets of computations per each set of necessary computations in the image. In other words, the reusability of an image gives the average number of redundant sets of computations per each reuse table cell. The ideal reusability of an image gives the percentage of redundant sets of computations in the image ignoring the first time misses (*i.e.* compulsory misses). In order to show that the coding and interpixel redundancy of an image have a positive relationship with the computational redundancy of the image we have shown that there is a positive

relationship between each of the following intermediate steps: coding/interpixel redundancy and ideal reusability, ideal reusability and reusability, reusability and the hit rate of perfect window memoization, and finally, the hit rate of perfect window memoization and computational redundancy.

We have implemented window memoization in both software and hardware. In software, we have presented an architecture that draws on concepts from hash tables in software and caches in hardware. We have outlined a set of design decisions to arrive at an optimal configuration of the architecture. We have also presented a performance model to predict the speedup obtained by window memoization in software and assist in choosing the optimal reuse table sizes, which yield maximum speedups. In software, we have applied the technique to six different case study algorithms, implemented in $C$, and demonstrated that window memoization yields significant speedups across different images and processors. The typical speedups range from 1.2 to 7.9 with a maximum factor of 40. Finally, we have shown that the computational redundancy of an image has a positive relationship with the speedup obtained for the image by window memoization in software. This means that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in software.

In hardware, we have presented an optimized architecture for window memoization, which improves the performance of local image processing algorithms with less cost (*i.e.* hardware area) in comparison to conventional performance improvement techniques. Similar to window memoization in software, we have explored different design decisions in order to achieve high speedups with less cost. In contrast to software, which uses a 1-level regular reuse table, in order to increase the efficiency of the design in hardware, we have used a 2-level reuse table that is based on parallel Bloom filters. We have applied the window memoization technique to two case study algorithms and implemented them at the register-transfer-level using VHDL. For typical images, on average our technique improves the performance by 58% with less than 20% extra hardware area. Finally, we have shown that the reusability of an image has a positive relationship with the speedup obtained for the image by window memoization in hardware. This means that the coding and interpixel redundancy of an image has a positive relationship with the speedup

obtained for the image by window memoization in hardware.

We have shown that the coding and interpixel redundancy of an image are the root causes of having similar windows across the image. The window memoization technique improves the performance of image processing in both software and hardware by identifying similar windows of pixels across the image and eliminating the computations that are not necessary. We have shown that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained by window memoization in software and hardware. This is a simple, yet revealing concept to use in practice. Images can be categorized based on their potential performance gain in software and hardware by only their coding and interpixel redundancy, with no need to actually implement the window memoization technique. Coding and interpixel redundancy are fundamental characteristics of images and hence, they are independent of any implementation artifact (*e.g.* identifying similar windows in an image using a reuse table and a mapping scheme). Therefore, the relationship between the coding/interpixel redundancy and the performance improvement obtained by window memoization can be used as a useful tool in analyzing images from the performance perspective in the early stages of designing an optimization technique.

The contributions of this research include:

- define a quantitative measure for computational redundancy of image.
- show mathematically that the computational redundancy has been inherited from two principal redundancies in image data: coding and interpixel redundancy.
- introduce a high level processing model, perfect window memoization, which gives an upper-bound for the performance gain in software and hardware achieved by exploiting the computational redundancy of an image.
- present an optimized architecture for the software implementation of window memoization, typical speedups: 1.2 to 7.9.
- present a model for speedup of the software implementation of window memoization.
- show that the coding and interpixel redundancy of an image have a positive relationship with the speedup for the image obtained by window memoization in

software.

- present an optimized architecture for the hardware implementation of window memoization, which yields significant speedups with much less hardware area than conventional digital design techniques, typical efficiencies: 1.4.

- show that the coding and interpixel redundancy of an image has a positive relationship with the speedup for the image obtained by window memoization in hardware.

The outline of the remainder of this thesis is as follows. In chapter 2, we present a background review and related work on redundancy in image data, computational redundancy in computer programs, memoization techniques proposed in software and hardware, and local image processing algorithms. In this chapter, we also present a taxonomy of local image processing algorithms as well as an overview of six different local image processing algorithms, which are used as case studies for this research.

In chapter 3, the underlying roots of our performance improvement technique, window memoization, is presented. We show mathematically that the amount of coding and interpixel redundancy of an image has a positive relationship with the computational redundancy of the image.

In chapter 4, we present the implementation of window memoization in software. We also present a model that predicts the speedup of all images in a data set with minimum required information. In addition, we show that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in software.

In chapter 5, we present an optimized architecture for the hardware implementation of the window memoization technique. Moreover, we show that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in hardware.

Chapter 6 presents the conclusion. The thesis also includes four appendices. Appendix A illustrates the notations and terminology used throughout the thesis. Appendix B elaborates on using autocorrelation as a measure for interpixel redundancy of image data. Appendices C and D present the results for window memoization in software and hardware, respectively.

# Chapter 2

# Background and Related Work

The background for our research includes three different topics: redundancy in image data, memoization techniques proposed in software and hardware, and local image processing algorithms. Our research shows that the redundancy in image data leads to computational redundancy in local image processing algorithms (chapter 3). We present architectures in both software and hardware to exploit the computational redundancy to improve performance and efficiency (chapters 4 and 5). In this chapter, we present a background review and related work on each of the aforementioned topics. In section 2.1, a background review on different types of redundancy in image data is given. Section 2.2 presents a background review on computational redundancy in computer programs. It also presents related work on different memoization techniques proposed in software and hardware. In section 2.3, a brief introduction to local image processing algorithms is given. We present a taxonomy of local image processing algorithms in section 2.4. In section 2.5, we present an overview of six different local image processing algorithms, which are used as case studies for this research. Finally, in section 2.6, we present the sets of images that we use in our experiments throughout this thesis.

## 2.1 Image Data Redundancy

Image processing exploits three types of redundancy in image data: *psychovisual redundancy*, *coding redundancy*, and *interpixel or spatial redundancy* [23]. Psychovisual redundancy indicates that some information in the image is irrelevant and therefore, it is

ignored by the human vision system. Coding redundancy is due to the fact that images are usually represented using higher number of bits per pixel than is actually needed. Interpixel redundancy is due to the correlation among neighboring pixels. These characteristics of image data have been identified and exploited in the image compression research field where the main goal is to reduce the amount of data required to store, transmit, and/or represent an image. Nevertheless, the redundancy in image data has not been considered as a potential source for computational redundancy. Therefore, there is no explicit previous research on the relationship of the data redundancy and the computational redundancy of image data. In the following sections, we present an overview of the three types of image data redundancy.

### 2.1.1 Psychovisual Redundancy

Psychovisual redundancy in an image exists because human vision does not weigh all the information coming from the image equally. Some parts of the image are considered to contain more important information than the other parts. The less important information is considered to be redundant since it is ignored by the human vision system. From the image compression perspective, a region of an image, which has psychovisual redundancy can be either omitted or represented using much less number of bits per pixel [23].

### 2.1.2 Coding Redundancy

An image is said to contain coding redundancy if the number of bits per pixel that is required to represent the image is higher than is necessary [23]. To remove coding redundancy in an image, the gray levels of the image are encoded in a way such that a higher number of bits are assigned to less probable gray levels and vice versa. This is in contrast to natural binary code where the gray levels are encoded with a fixed length code. An image with equal probabilities of gray levels (*i.e.* flat histogram) has the least coding redundancy (*i.e.* 0 bits/pixel). The other extreme case is a white image where all pixels have the same gray level. In such a case, the coding redundancy is maximum (*i.e.* 8 bits/pixel) since the image does not carry any information.

The coding redundancy of an image is measured based on the *entropy* of the image. Entropy is the average information of an image per pixel, which is calculated as [23]:

$$H = -\sum_{i=0}^{GL-1} p_i \times log_2 p_i \tag{2.1}$$

where $GL$ and $p_i$ are the number of gray levels and the probability of occurrence of gray level $i$ in the image, respectively. For an image with $GL$ gray levels, the sum of the average coding redundancy per pixel ($C_r$) and average information per pixel (entropy $H$) is a constant value [52]:

$$C_r + H = log_2(GL) \tag{2.2}$$

In other words, for an image with a given number of gray levels ($GL$), the entropy of the image ($H$) determines the coding redundancy of the image ($C_r$).

$$C_r = log_2(GL) - H \tag{2.3}$$

For an image with 256 gray levels we will have: $C_r = 8 - H$.

Coding redundancy has been exploited in several image compression algorithms to reduce the size of the image. Huffman coding, Golomb coding, and arithmetic coding are among the common lossless compression algorithms that benefit from coding redundancy in images [23].

### 2.1.3 Interpixel Redundancy

When measuring the coding redundancy of an image, it is assumed that the pixels in the image are uncorrelated. However, in real-world images, the neighboring pixels are correlated because they usually belong to one object or background with similar gray levels. The correlations among image pixels, which result from the structural or geometrical relationships between the objects in the image lead to interpixel redundancy [23]. For many pixels, much of information that the pixel carries is redundant and it can be predicted with a reasonably high accuracy from the values of its neighboring pixels. There are many lossless image compression algorithms that take advantage of interpixel redundancy in images including run-length coding, difference coding, lossless predictive coding, LZW coding, and vector coding. Lossy predictive coding is an example for lossy image compression algorithms, which exploit interpixel redundancy [23] [16].

Interpixel redundancy has also been exploited in image segmentation where groups of pixels are assigned to different classes, indicating objects or different types of surfaces in the image. Two approaches that use interpixel redundancy in this field are spatial stochastic model and Markov model. In these approaches, the image is classified based on the assumption that there is a local dependency of a pixel on its neighbors [56] [25] [30].

To measure interpixel redundancy, three different methods are usually used: mapping transforms, Markov model, and autocorrelation. In the followings, each of the methods is described.

#### Interpixel Redundancy Based on Mapping Transforms

As mentioned earlier, interpixel redundancy exists because much of information carried by each pixel is redundant with the information carried by the pixels around it. *Mapping transforms* convert an image into a new format where the interpixel redundancy of data has been eliminated [23]. Each pixel in the transformed image contains the information that is carried solely by the corresponding single pixel in the original image.

In mapping transforms, the interpixel redundancy of a pixel is calculated with respect to a window of pixels that appears around the pixel in the image. Thus, the size of the window plays a role in defining the interpixel redundancy of an image. For example, the information that a pixel shares with a $3 \times 3$ window of pixels around it is different than that shared with a $5 \times 5$ window. The other issue in extracting the interpixel redundancy is the amount of weight that is given to the interpixel redundancy between a pixel and each neighboring pixel. The general form of a mapping transform, which extracts the information carried solely by the central pixels of windows is:

$$Img^{icp}(x,y) = \sum_{i=-m_1}^{m_1} \sum_{j=-m_2}^{m_2} \alpha_{ij} Img(x-i, y-j) \tag{2.4}$$

In the equation above, *icp* stands for *information at central pixel*, $(2m_1+1)\times(2m_2+1)$ is the size of the windows, based on which the interpixel redundancy is calculated, and $\alpha_{ij}$ determines the weights given to the interpixel redundancy between a pixel and each neighboring pixel.

Once the mapping has been applied to an image, the entropy of the transformed image ($H^{icp}$) will measure the amount of information stored in the transformed image ($Img^{icp}$), which is usually much less than that stored in the original image ($H^{orig}$). The decrease in entropy reflects the removal of interpixel redundancy ($IP_r$) with respect to the transform used ($icp$) to generate the transformed image ($Img^{icp}$) [23]. Therefore, the interpixel redundancy of an image can be calculated as:

$$IP_r = H^{orig} - H^{icp} \tag{2.5}$$

**Interpixel Redundancy Based on Markov Model**

Markov model assumes that the probability that a pixel at a location has a certain gray level is a function of the gray levels of some number of neighboring pixels [16]. The number of neighboring pixels is the order of the Markov source. When pixels in an image are assumed to be independent random variables, in fact it is assumed that the image is a Markov source of $0^{th}$ order. This is the case when coding redundancy is calculated. A $k^{th}$ order Markov source considers each window of $k$ pixels in an image as a *k-D gray-level vector*. If the $k$ neighboring pixels share information with each other, then the entropy of the $k^{th}$ order Markov source will be less than that of the $0^{th}$ order Markov source, which is the original image. Similar to mapping transforms, the decrease in entropy is considered to reflect the removal of interpixel redundancy among the pixels of windows with $k$ pixel size:

$$IP_r = H^{orig} - H^{k} \tag{2.6}$$

In equation 2.6, $IP_r$ is the interpixel redundancy, $H^{orig}$ and $H^k$ are the entropies of the original image and the $k^{th}$ order Markov source of the image, respectively. Similar to mapping transforms, the interpixel redundancy calculated based on Markov model depends on the size of windows of pixels, based on which gray-level vectors are built.

### Interpixel Redundancy Based on Autocorrelation

Autocorrelation is another measure used for interpixel redundancy of images. It is usually used to measure the interpixel redundancy between two adjacent pixels. Nevertheless, it can be generalized to measure the interpixel redundancy among a pixel and its neighboring pixels.

Autocorrelation measures the correlation of an image with the shifted version of the image in time or space [23]. For our discussion, we consider the correlation of image with itself in space. Equation 2.7 measures the autocorrelation of an image with itself that has been shifted by $n$ pixels. Note that the shift can be in any of eight directions (north, south, east, west, northwest, northeast, southwest, southeast).

$$AC(n) = \frac{E[(Img - \mu)(Img_n - \mu)]}{E[(Img - \mu)^2]} \tag{2.7}$$

In the equation above, $\mu$ is the mean of the original image ($Img$). $Img_n$ is equal to $Img$ shifted by $n$ pixels. *E[Img]* is the *expected value* of random variable *Img*, which is defined as:

$$E[Img] = \int_{-\infty}^{+\infty} xp(x)dx \tag{2.8}$$

where *p(x)* is the PDF of random variable *Img*. The denominator of equation 2.7 is in fact the variance of the original image, $\sigma^2$:

$$\sigma^2 = E[(Img - \mu)^2] = \int_{-\infty}^{+\infty} (x - \mu)^2 p(x)dx \tag{2.9}$$

*AC(1)* ($n = 1$) measures the correlation between the original image and the image shifted by one pixel. In other words, *AC(1)* indicates the interpixel redundancy between adjacent pixels of the image: the closer the gray levels of adjacent pixels, the higher *AC(1)*:

$$AC(1) = \frac{E[(Img_i - \mu)(Img_{i+1} - \mu)]}{\sigma^2} \tag{2.10}$$

In our discussions in chapter 3, it will be required to measure the coding and interpixel redundancy of images. As mentioned in section 2.1.2, the coding redundancy of an image is calculated based on the entropy of the image. Interpixel redundancy, however,

can be calculated using different measures (mapping transform, Markov model, and autocorrelation). As it will be discussed in chapter 3, both coding redundancy and interpixel redundancy of an image affect the computational redundancy of the image. Therefore, both coding and interpixel redundancies must be taken into account when dealing with computational redundancy. We have discussed in appendix B that for our purpose, autocorrelation is not a suitable measure for interpixel redundancy because there is no direct link between autocorrelation and the coding redundancy of an image. This makes it impossible to consider both redundancies using autocorrelation.

In chapter 3, we will show that the coding and interpixel redundancy of an image have a positive relationship with the computational redundancy of the image. In chapters 4 and 5, we will demonstrate that the computational redundancy of an image leads to performance improvement. Our goal is to develop a method, which makes it easy to categorize images based on their expected performance improvement without having information about the actual performance improvements. As it will be discussed in chapter 3, this is achieved by using a mapping transform as a measure to calculate interpixel redundancy. Beside the fact that mapping transform is a common approach to calculate interpixel redundancy in image processing, calculating interpixel redundancy using Markov model is a computationally intensive method. For example, for a Markov model of $9^{th}$ order (*i.e.* windows of $3 \times 3$ pixels), the Markov model requires to calculate 9-D vector histograms, with $2^{72}$ possible gray-level vectors or bins. For equivalent size windows, the mapping transform requires to calculate histograms of transformed images with only 511 possible bins.

## 2.2 Computational Redundancy

Computational redundancy has long been used in memoization techniques proposed for computer programming and processor design where in some cases the image processing algorithms have been used as test cases. However, the proposed memoization techniques are usually generic methods, which do not concentrate on any particular class of input data or algorithms. Therefore, the computational redundancy of image data has not been directly studied and exploited in designing the memoization methods. Moreover, the memoization techniques proposed in hardware are mostly for processor design. In

contrast, our memoization technique in hardware targets the hardware implementations of local image processing algorithms. In this section, we present a background review on locality of data in computer programs. We also present the related work on memoization techniques proposed in software and hardware.

### 2.2.1 Data Locality

The notion of *data locality* is used in both memory hierarchy design and optimizations of computations in software and hardware. Nevertheless, what is meant by data locality in these two contexts are usually different. In cache hierarchy design, data locality implies the high possibility of referencing the same resource in future, which can be used to reduce the memory access time [49]. From computational perspective, data locality is defined as the possibility of repeatedly encountering previously seen data, upon which the same calculation is to be performed. This type of data locality leads to computational redundancy. *Redundant computations* are operations that repeatedly perform the same function because they repeatedly see the same operands [46]. In both cases, data locality is caused by many factors among which data redundancy is the main reason. Data redundancy which means unnecessary reappearances of data is due to the fact that many programs use data that has little differences [34]. In our discussions, the "data locality" that we refer to is the one that causes computational redundancy. Computational redundancy has been exploited to optimize the computations by memoization techniques in both software and hardware. In the following sections, related work on memoization techniques proposed in software and hardware is presented.

### 2.2.2 Memoization Techniques in Software

*Memoization* is a performance improvement technique, which exploits the locality of data to speed up the calculations in a computer program. It removes the computational redundancy by storing the results of previous subcomputations in a reuse memory and reusing them in the future. Assuming that the overhead time for storing/loading the results to/from memory is small, reusing the previously computed results and skipping the actual subcomputations improves performance. In other words, memoization improves performance in exchange for increased memory usage.

**Memoization versus Dynamic Programming**

There are two approaches for the general method of memoization in software: top-down and bottom-up. The top-down approach is also called memoization while the bottom-up approach is called *dynamic programming* [10].

In the top-down approach (memoization), first, those subcomputations of a program that can benefit from memoization are identified. Afterward, the memoization mechanism is integrated into the original program to store and reuse the result of one or multiple subcomputations. Based on the dependency of the subcomputations, reusing a result could lead to skipping one or multiple subcomputations. Top-down memoization does not transform the original program into a new code. It maintains the flow of the program unchanged and only some subcomputations are replaced with the reuse of memory. Both identifying the subcomputations that benefit from memoization and integrating the memoization mechanism into the program can be done either manually or automatically.

In the bottom-up approach (dynamic programming), it is assumed that an optimal solution to a problem contains optimal solutions to the subproblems of the original problem. Moreover, the optimal solution to the whole problem is achieved by the optimal solutions to the subproblems, used in a bottom-up manner. Another element of dynamic programming is the fact that same subproblems are encountered in a program frequently. Such subproblems are called *overlapping subproblems*. Dynamic programming uses memoization to avoid repeating redundant subcomputations. Therefore, it is considered a special case of memoization.

In the top-down memoization, based on the input, it is decided whether to perform a subcomputation or reuse it. Therefore, not all possible subcomputations are executed. Only those are performed that are necessary. This is not the case in the bottom-up approach (*i.e.* dynamic programming) where each subcomputation is executed at least once regardless of whether the subcomputation is necessary. In practice, usually a portion of subcomputations are needed, in which cases memoization can be more efficient than dynamic programming. On the other hand, dynamic programming can benefit from the regular pattern of accesses to reuse memory to reduce the memoization overhead time and the required memory size.

In the following, the related work on top-down memoization techniques proposed in

software is presented.

### Related Work on Memoization Techniques in Software

The concept of memoization was originally introduced by Michie [38]. He used *memo-functions* to memoize the result of each function to avoid redundant function calls. Bird [2] proposed a memoization technique called *exact tabulation*, which eliminates the recursion in recursive programs. Exact tabulation is suitable for recursive algorithms whose dependency graphs do not have a uniform property. Exact tabulation requires large reuse memory since no entry is evicted from the memory once it is stored. Hughes [28] introduced the notion of *lazy memo-functions* by relaxing the requirement for equality of subcomputations; rather than require that the equal subcomputations have identical arguments, the subcomputations are considered equal if their arguments are stored in the same memory location. In other words, the equality evaluation of subcomputations are based on the addresses where they are stored in the reuse memory. This eliminates the need for comparing complex data structures and leads to a more efficient memoization mechanism. The technique also claims that it reduces the size of required reuse memory by discarding old arguments and results.

Pugh and Teitelbaum [44] applied memoization to incremental computations. Incremental computations is a technique used to efficiently update the result of a computation only when the input is changed. They introduced the notion of stable decomposition, meaning that a problem is decomposed into common sub-problems. The reported speedups for a case study (an incremental theorem proving) is in the range of 4-6. In another work, instead of previously used eviction methods (LRU and FIFO), Pugh [43] proposed a new method, which is based on the number of hits an entry receives and the estimated amount of time required to recompute it. It was reported that the proposed eviction strategy reduce the overhead time incurred by memoization. Mayfield [37] proposed an automated memoization technique that avoids the redundant calculations both within a function and across the invocations. The technique provides the user with methods to decide which functions to reuse and automatically integrate the memoization mechanism to the program.

In embedded software, the memoization techniques are more interesting because they

both improve the performance and reduce power consumption. Ding [15] proposed a computation reuse technique for programs run on handheld devices. The technique uses profiling tools to identify segments of the program that are executed more frequently. Afterward, the memoization mechanism is integrated to the program in order to reuse the results of the selected segments. In addition, an *IF-merging* technique is used to reduce the number of branches by merging $IF$ statements with identical $IF$ conditions. For computation reuse, a typical speedup of 1.37 with 22% reduction in power consumption were reported. For IF-merging, a typical speedup and power saving of 1.06 and 6% were reported, respectively.

Wang [55] proposed a systematic methodology to identify and eliminate redundancies in embedded software programs. Similar to the work done by Ding [15], a profiling tool is used to identify the regions of the program that have high probability of performing redundant computations. Afterward, various caching strategies are explored to achieve maximum energy savings. Finally, the program is modified to include the code that implements the memoization mechanism and performs reuse memory lookups and updates. The proposed technique yields speedups up to 1.45 with up to 47% reduction in power consumption.

### 2.2.3 Memoization Techniques in Hardware

Despite the innovations made in modern microprocessors design, their performance is essentially limited by two program characteristics: *control flow limit* and *data flow limit*. Control flow limit of a program can cause control hazards, which arise from the speculative execution of instructions. Data flow limit, on the other hand, is caused by data hazards, which are due to un-handled data dependency between consecutive instructions. The goal of memoization techniques proposed in hardware is to improve the performance of pipelined designs beyond the data flow limit.

There are three different classes of data dependencies in microprocessor design [49]:

- Anti-dependency or Write After Read (WAR) dependency
- Output dependency or Write After Write (WAW) dependency
- True dependency or Read After Write (RAW) dependency

Anti-dependency is caused by writing to a register before allowing the previous value of the register to be read. Output dependency is caused by writing to a register before allowing the previous write operation on the register to take place. These two data dependencies (anti-dependency and output dependency), which are called false dependencies can be resolved by dynamically renaming the destination operand to a unique location. The proposed techniques that are able to remove false data dependencies efficiently have been implemented in processor design for the last four decades. Nevertheless, true data dependency, which is the critical path between a consumer instruction and its source operands determines the data flow limit in processors. In other words, the data flow limit occurs when the unavailability of an instruction's operands halts the instruction.

The performance of a pipelined design is indicated by *throughput*, which is measured in terms of the number of parcels of data entering the pipeline per clock cycle. In the absence of data flow limit, the throughput of an n-wide superscalar pipeline is $n$ parcels of data per clock cycle. Data flow limit in a superscalar pipeline causes the pipeline to stall, reducing the throughput below $n$ parcels of data per clock cycle. The memoization techniques in hardware introduce additional instruction-level parallelism by reusing the result of an instruction without executing it. The techniques produce the result of an instruction, with which the consequent instructions have true data dependencies as soon as possible and hence reduce the number of cycles that pipeline has to stall. Reducing the number of pipeline stalls will increase throughput beyond the data flow limit. To reuse an instruction, it is necessary to verify that its future result is going to be the same as the previous one, in which case the result can be reused. This is done by recording the instruction and its operands in a reuse memory. Similar to memoization techniques in software, memoization techniques in hardware take advantage of the locality of data fed into the in-flight instructions.

### Related Work on Memoization Techniques in Hardware

Richardson [46] proposed to embed a memoization technique in processor architecture to look up the results of a set of targeted operations (multiply, division and square root) that are redundant. Different SPEC benchmarks used for simulation showed various speedups ranging from 1.04 up to 1.48 with up to $16K$ entry direct mapped reuse memory.

Sodani and Sohi [51] proposed a microarchitectural technique, instruction reuse, which reduces the number of instructions that have to be executed dynamically. The idea behind instruction reuse is to use value locality of instructions and operands to produce the result of an instruction as soon as it is fetched — without ever executing the instruction. This reduces the number of cycles that the pipeline has to stall for data dependencies and hence increases the throughput of the pipeline. Producing the instructions results as soon as they are fetched eliminates the true data dependency between instructions and therefore, it breaks the data flow limit of the program. In instruction reuse, three schemes have been proposed to control the reuse memory. The first scheme checks operand values of each instruction to verify if they are present in the reuse memory. The second scheme tracks the operands names and the third one follows the data dependencies among the instructions. Because the reuse memory access can be pipelined, despite its size, it is unlikely to be a part of the critical path of the design. For a 1024 entry reuse memory, the reported speedups for different benchmarks range from 1.03 up to 1.43.

Citron *et al.* [7] extended Sodani's work and proposed a technique that enables executing multi-cycle operations in a single cycle by adding a *MEMO-TABLE* to each computation unit in microprocessor architecture. The multi-media benchmarks were chosen as test cases because of expected high value locality of data. The proposed scheme is different than instruction reuse bcause it only records multi-cycle instructions in MEMO-TABLEs. In addition, the instructions are stored in MEMO-TABLEs based on their types, rather than their addresses. For the selected multi-media benchmark with a 32 entries, 4-way set associative MEMO-TABLEs, the maximum speedup was 1.22.

In another work, Citron and Feitelson [8] proposed adding two new instructions to the instructions set of processor, which look up and update a generic MEMO-TABLE (function reuse). These two instructions make it possible to reuse multi-cycle mathematical and trigonometric functions, most of which are not included in the instructions set of most microprocessors. It was concluded that for applications that use many functions of these two types, the proposed approach yields better results in comparison to instruction reuse. The best result was achieved for a combination of both instruction and function reuse, which gave the average speedup of 1.14 with 5 MEMO-TABLEs of 256 entries and 4-way set associativity.

Kavi and Chen [31] studied the possibility of improving performance by reusing the results from previous function invocations. It was concluded that there is a great potential for exploiting function reuse and also functions with fewer arguments have higher probability of reuse. Huang and Lilja [26] applied value reuse to blocks rather than single instructions. This technique, which is called block reuse, showed a speedup between 1.01 and 1.14 with 2048 entries block history buffer for a selected benchmark.

In a recent study, Citron and Feitelson [9] showed that because of long latency memory access in new processors, the memoization concept is useful only for multi-cycle instructions. In other words, it is useless to reuse single-cycle instruction.

Although the simulation results for different proposed reuse architectures show significant performance improvement, none of these techniques has been implemented in a real design yet. The reasons are that implementing these techniques will require significant modifications to existing control and datapath circuitry in microprocessors and that designers in industry are not convinced yet that the price they have to pay for design modifications will be returned by the gain in performance [49].

## 2.3 Local Image Processing Algorithms

In this section, a brief overview of local image processing algorithms is given. From the perspective of the mechanics of computations, the image processing algorithms can be categorized into two major categories: spatial domain algorithms and transform-domain algorithms. Spatial domain algorithms deal with image pixels directly, while transform-domain algorithms work with the result of an image transform such as Fourier transform or Wavelet transform. Each category contains different subcategories as shown below:

- Spatial domain
  - Point Processing: mask operations performed on a pixel
  - Local Processing: mask operations performed on a local window of pixels
  - Global Processing: mask operations performed on the whole image
- Transform-domain

- Fourier Transform: transforms the image into the frequency domain; the image is represented as the sum of sines and cosines of different frequencies, each weighted with a differen coefficient [23]
- Discrete Cosine Transform: is similar to Fourier transform but only uses cosines function to represent the image [23].
- Wavelet Transform: transforms the image into a 2-dimensional domain of scale and space. The image is represented as the sum of some functions that are not necessarily sinusoidal [23].

In this research, our focus is spatial domain algorithms. To perform an image processing task in spatial domain, usually a set of image processing algorithms are used in a chain. A chain contains different stages and each stage performs a set of calculations on a pixel (point processing), a local window of pixels (local processing) or the whole image (global processing) at each iteration.

Local processing algorithms that mainly deal with extracting local features in image (*e.g.* edges, corners, blobs) are increasingly used in many image processing applications such as texture recognition, scene classification, and robot navigation. The reason for popularity of these algorithms is that using local features of an image overcomes the need for high level algorithms where a semantic-level understanding of the image is required [54]. The main drawback for local algorithms is that they are usually computationally expensive; a set of calculations must be repeated all over the image for numerous times.

The local algorithms use a small neighborhood of a pixel in an image to process and produce a new gray level for the image in the location of the pixel. The size of local windows can vary based on the algorithm but for most algorithms the local windows contain 9 pixels ($3 \times 3$ pixels) or 25 pixels ($5 \times 5$ pixels). A local processing algorithm applies a set of operations, which is called *the mask operations set* ($f$), to each window of pixels ($w_{ij}$) in the image to produce the *response ($r_{ij}$)* of the window.

$$r_{ij} = f(w_{ij}) \tag{2.11}$$

As equation 2.11 indicates, in local processing algorithms, the response of each local window $r_{ij}$ only depends on the pixels in the local window $w_{ij}$.

## 2.4 The Taxonomy of Local Image Processing Algorithms

From the perspective of functionality, the image processing algorithms can be categorized into intermediate steps of a processing chain as shown in figure 2.1 [17]. At each stage, series of calculations are performed on the input image to reach the desired goal (*e.g.* enhance the quality of image, find the objects in the image). In the following, each stage will be described briefly and then the local processing algorithms that belong to the stage will be listed. It should be noted that the algorithms/filters that are designed in frequency domain are usually transformed back to spatial domain in order to be implemented in software or hardware [23]. Therefore, our taxonomy of local image processing algorithms will also include such frequency domain algorithms.

Figure 2.1: Image processing chain

- Pre-Processing: the first stage in the image processing chain. It usually falls into two categories: image restoration and image enhancement. Usually, the algorithms of image restoration and image enhancement are local processing.
  - Image Restoration: recovers an image that has been degraded by noise, by using a priori knowledge about the degradation process [23] [52] [22] [13]:
    * Mean Filters: reduces the noise in an image by removing the detail in the image and blurring the image:
      1. Arithmetic mean filter
      2. Geometric mean filter
      3. Averaging with limited data
      4. Harmonic mean filter
      5. Contraharmonic mean filter
      6. Averaging according to inverse gradient
      7. Averaging using rotating masks

* Order-Statistics Filters: the responses of the filters are based on the order of the pixels in the local neighborhoods:
  1. Median filter
  2. Max and min filter
  3. Midpoint filter
  4. Alpha-trimmed mean filter
* Image Smoothing Using Frequency Domain Lowpass Filters:
  1. Butterworth lowpass filter
  2. Gaussian lowpass filter
  3. Wiener lowpass filter
* Image Smoothing Using Frequency Domain Selective Filters:
  1. Butterworth bandreject filter
  2. Gaussian bandreject filter
  3. Butterworth bandpass filter
  4. Gaussian bandpass filter
  5. Butterworth notch filter
  6. Gaussian notch filter
* Adaptive Filters: takes into account the statistical characteristics (*e.g.* variance, mean) of the local neighborhood of pixels:
  1. Adaptive noise reduction filter
  2. Adaptive median filter
  3. Adaptive frost filter
  4. Sticks filter
  5. Kuwahara Filter

– Image Enhancement: magnifies specific features in the image [23] [52]:
  * Gradient operators: detect edges and/or sharpen the image by calculating the gradient of a neighborhood of pixels:
    1. Roberts operator
    2. Prewitt operator
    3. Laplace operator

4. Sobel operator
5. Kirsch operator
6. Canny edge detector

* Unsharp masking: sharpens images by subtracting an unsharp version of an image from the original image

- Feature Extraction: reduces the amount of data in image in order to extract a desirable information about the content of the image
  - Rapid Transform: is a shift-invariant transform, which locates a pattern in the image [18]
  - Corner Detection: detects the intersection of lines in image [39] [24] [50] [53] [52]:
    1. Moravec corner detection
    2. Harris & Stephens/Plessey corner detection
    3. Level curve curvature corner detection
    4. Wang and Brady corner detection algorithm
    5. SUSAN corner detector
    6. Trajkovic and Hedley corner detector
    7. FAST feature detector
    8. Facet model
  - Blob Detection: detects areas in the image that are brighter or darker than their surroundings. In other words, blobs are areas whose brightness is above or below a threshold.
    1. Laplacian of Gaussian (LoG)
    2. Difference of Gaussians approach
    3. Determinant of the Hessian
    4. Maximally stable extremal regions (MSER)
  - Ridge Detection: for an $n$ variable function, the ridges are a set of curves whose points are local maxima in $n-1$ dimensions. Ridges are used to capture the interior of elongated objects in the image.

- Scale-Invariant Feature Transform (SIFT): detects local features in an image. The result is invariant to change in scale and illumination [36].

- Local Segmentation: partitions the sub-images in isolation into regions such that the representation of the image is changed into a more meaningful form [52] [47] [48]:

  1. Locally-Adaptive Thresholding: adapts the threshold according to the local characteristics of image
     (a) Surface fitting thresholding
     (b) Kriging method
     (c) Center-surround method
  2. Line Finding Operators: detect lines in image
  3. Non-Maximal Suppression: suppresses multiple edge responses in the neighborhood of single boundaries.
  4. Edge Linking: identifies and links the edges that belong to an object.
  5. Segmentation based on Spatial Entropy: segments image based on the entropy of the foreground and background regions, and the cross-entropy between the original and binarized image.
  6. Block Truncation Coding: operates on small neighborhoods of pixels (blocks) in image that are non-overlapping. Each block then is divided into two regions of foreground and background.
  7. Filtering Using Explicit Local Segmentation (FUELS): a more generalized version of block truncation coding where overlapping neighborhoods are used and each sub-image is divided into more than two regions.

- Object Recognition: recognizes objects in image and estimates the positions and orientations of the recognized objects [23]:

  - Template matching: recognizes an object in image by matching sub-images of the image and the target subject

- Morphological Algorithms: are usually used to enhance an image and/or extract a certain information from an image and they can be seen as preprocessing and/or

feature extraction algorithms. However, these algorithms are usually categorized as a separate class of algorithms, which uses the set theory as its language [23] [52]. Therefore, we put them in a separate category in the taxonomy of image processing algorithms:

1. Dilation: grows image regions by combining two sets of pixels using vector addition
2. Erosion: shrinks image regions by combining two sets of pixels using vector subtraction
3. Hit-or-Miss transformation: finds a match or miss-match between an image and a window of pixels
4. Opening: eliminates particular details in image that are smaller than a specific size (structuring element), using erosion followed by dilation
5. Closing: connects the object that are close, using dilation followed by erosion
6. Top-Hat Transformation: segments the object in image that differ from background, using the residue of opening compared to the original image
7. Sequential Thinning: thins the lines down to one pixel wide using local windows (structuring elements)
8. Morphological Gradient: calculates the gradient of image based on dilation and erosion
9. Morphological Grayscale Reconstruction

The last step in the image processing chain (figure 2.1) is *image understanding*, which is the highest processing level in computer vision. Image understanding algorithms usually process the whole image or a large subimage at each iteration, which is considered to be a global processing algorithm, rather than local. Thus, we have not included image understanding algorithms in the taxonomy.

## 2.5 Case Study Algorithms

In this section, we present six local image processing algorithms that we have chosen as case studies for our experiments. In choosing the case study algorithms, the followings

have been considered:

- belong to different classes of local image processing algorithms
- use different local window sizes
- have different complexities
- have binary and gray level outputs

The following shows the case study algorithms and the classes they belong to:

- Canny edge detector: Preprocessing, image enhancement. Segmentation, Non-maximal suppression
- Morphological gradient: Mathematical morphology algorithms.
- Kirsch edge detector: Preprocessing, Image enhancement
- Trajkovic corner detector: Feature extraction, Corner detection
- Median filter : Preprocessing, Image restoration
- Local Variance: Preprocessing, Image restoration

Table 2.1: Characteristics of the case study algorithms

| Algorithm/ Characteristic | Window Size | Complexity (ms) | Output |
|---|---|---|---|
| Canny Edge Detector | $3 \times 3$ | 64 | Binary |
| Morphological Gradient | $3 \times 3$ | 49 | Binary |
| Kirsch Edge Detector | $3 \times 3$ | 37 | Binary |
| Corner Detector | $3 \times 3$ | 315 | Binary |
| Median Filter | $3 \times 3$ | 36 | Gray Level |
| Local Variance | $5 \times 5$ | 2045 | Gray Level |

Table 2.1 lists the characteristics of the case study algorithms including the window size, the complexity of the algorithms and whether the output is binary or gray level. For complexity, we measure the time that it takes to run each algorithm on a mid-range

processor (Intel(R) XEON(TM), CPU: 1.80GHz, cache size: 512KB). It is seen that the algorithms use different window sizes ($3 \times 3$ and $5 \times 5$), cover a wide range of complexities ($36ms$ to $2045ms$), and generate both binary and gray-level outputs.

### 2.5.1 Canny Edge Detection Algorithm

The Canny edge detection algorithm assumes that there is a step edge, which is subject to white Gaussian noise. The algorithm uses a filter (*i.e.* derivative of a Gaussian function $\nabla G(x, y)$) to obtain an image that has enhanced edges, even in the presence of noise [52]. Gaussian function is a smoothing filter that removes any quick changes in the image. It is given by:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.12}$$

where $\sigma$ is the scale factor.

The mathematic of Canny edge detector is as follows. First, the derivative of the Gaussian filter $\nabla(G(x, y))$ is convolved with the input image $I$:

$$J(x, y) = \nabla(G(x, y)) * I(x, y) \tag{2.13}$$

The derivative must be taken in the direction of edge. However, the orientation of the edge is still unknown. So assuming that the orientation is $\mathbf{n}$, it is obtained:

$$J(x, y) = \nabla(G(x, y)) * I(x, y) = \frac{\partial G(x, y)}{\partial n} * I(x, y) \tag{2.14}$$

$\mathbf{n}$ can be estimated as follows:

$$n = \frac{\nabla(G(x, y)) * I(x, y)}{|\nabla(G(x, y)) * I(x, y)|} = n_x i + n_y j \tag{2.15}$$

By having the direction of the edge, the location of the edge can be calculated. The location of the edge is where the second derivative of the Gaussian filter convolved with the input image is zero:

$$\frac{\partial \frac{\partial G(x,y)}{\partial n}}{\partial n} * I(x, y) = \frac{\partial^2 G(x, y)}{\partial n^2} * I(x, y) = 0 \tag{2.16}$$

In order to calculate equation 2.16, we begin with the first derivative of Gaussian filter in the direction of $\mathbf{n}$:

$$\frac{\partial G(x,y)}{\partial n} = \frac{\partial G(x,y)}{\partial x}\frac{n_x}{|n|} + \frac{\partial G(x,y)}{\partial y}\frac{n_y}{|n|} \tag{2.17}$$

where:

$$n = \sqrt{n_x^2 + n_y^2} \tag{2.18}$$

The second derivative of Gaussian in the direction of **n** is calculated as follows:

$$\frac{\partial^2 G(x,y)}{\partial n^2} = [\frac{\partial^2 G(x,y)}{\partial x^2}\frac{n_x}{|n|} + \frac{\partial^2 G(x,y)}{\partial x \partial y}\frac{n_y}{|n|}]\frac{n_x}{|n|} + [\frac{\partial^2 G(x,y)}{\partial y^2}\frac{n_y}{|n|} + \frac{\partial^2 G(x,y)}{\partial x \partial y}\frac{n_x}{|n|}]\frac{n_y}{|n|} \tag{2.19}$$

which gives:

$$\frac{\partial^2 G(x,y)}{\partial n^2} = \frac{\partial^2 G(x,y)}{\partial x^2}\frac{n_x^2}{|n|^2} + 2\frac{\partial^2 G(x,y)}{\partial x \partial y}\frac{n_x n_y}{|n|^2} + \frac{\partial^2 G(x,y)}{\partial y^2}\frac{n_y^2}{|n|^2} \tag{2.20}$$

Equation 2.20 [1] shows that in order to find the location of an edge, we can analytically calculate $G_{xx}(x,y)$, $G_{yy}(x,y)$ and $G_{xy}(x,y)$ and convolve the results with the image $I$. If the second derivative of Gaussian function convolved with the input image is zero in a location, then that is the location of a candidate edge. Otherwise, the edge is a false edge and therefore, it is eliminated.

After determining the location of edges and suppressing false edges, the edge magnitudes are computed using:

$$M(x,y) = |\frac{\partial G(x,y)}{\partial n} * I(x,y)| \tag{2.21}$$

Substituting equation 2.17 into equation 2.21 yields:

$$\begin{aligned} M(x,y) &= |\frac{n_x}{|n|}\frac{\partial G(x,y)}{\partial x} * I(x,y) + \frac{n_y}{|n|}\frac{\partial G(x,y)}{\partial y} * I(x,y)| \\ &= |\frac{n_x}{|n|}J_x(x,y)| + \frac{n_y}{|n|}J_y(x,y)| \\ &= \sqrt{J_x^2(x,y) + J_y^2(x,y)} \end{aligned} \tag{2.22}$$

which gives:

$$M(x,y) = |\nabla(G(x,y)) * I(x,y)| \tag{2.23}$$

[1] Equation 2.20 is a specific form of “steerable filters” [21] that constitute a class of filters in which a filter of arbitrary orientation is built based on a linear combination of filters with fixed orientations.

The magnitudes of edge responses are compared against a threshold interval [low threshold, high threshold]. Those pixels with edge magnitudes above the high threshold are considered as definite edges and those below low threshold are regarded as non-edges. The rest that are between low and high thresholds are set to be as candidate edges, which will be defined as edge or non-edge with a hysteresis. Those candidate edges that are connected to a definite edge are considered as definite edges as well. Otherwise, they will be removed from the edge map. The Canny edge detection algorithm is listed in table 2.2 [52].

Table 2.2: Canny edge detector algorithm

1. Input image $I$
2. For all windows in image $I$, convolve the windows with the mask of Gaussian derivative in $x$ direction $\frac{\partial G(x,y)}{\partial x}$ to produce $J_x = G_x(x,y) * I(x,y)$.
3. For all windows in image $I$, convolve the windows with the mask of Gaussian derivative in $y$ direction $\frac{\partial G(x,y)}{\partial y}$ to produce $J_y = G_y(x,y) * I(x,y)$.
4. For all windows in image $I$, calculate the second derivative of Gaussian $\frac{\partial^2 G(x,y)}{\partial n^2}$, using equation 2.20.
5. For all windows in image $I$, calculate the magnitudes of edge responses as: $M(x,y) = \sqrt{J_x(x,y)^2 + J_y(x,y)^2}$.
6. If $\frac{\partial^2 G(x,y)}{\partial n^2} = 0$, set edge map $E(x,y)$ to the magnitude $M(x,y)$, otherwise, set it to 0
7. For all locations in $E(x,y)$:
   a. If $E(x,y) < threshold_{low}$, no edge. Set edge map $E(x,y)$ to 0.
   b. If $E(x,y) > threshold_{high}$, a definite edge. Set edge map $E(x,y)$ to 2.
   c. If $threshold_{low} < E(x,y) < threshold_{high}$, a candidate edge. Set edge map $E(x,y)$ to 1.
5. Use hysteresis to determine whether the candidate edges are definite edges.

For our experiments, we implement the Canny edge detector using $3 \times 3$ local windows.

### 2.5.2 Morphological Gradient

The morphological gradient algorithm enhances the edges and suppresses the homogeneous areas in the image. This effect is similar to what a derivative causes and hence,

it is called "gradient". The result image can be thresholded to obtain the edge map of the image. The algorithm includes two basic mathematical operators of gray-scale morphology: *Dilation* (grows image regions) and *Erosion* (shrinks image regions). These two operators are fundamental building blocks for gray-scale morphology, based on which many morphological algorithms have been developed. Usually, the non-flat (non-uniform) structuring elements that are used for morphological algorithms are decomposed down to $3 \times 3$ bases to improve the performance [41] [5].

Let $I(x, y)$ and $se(x, y)$ be input image function and structuring element function, respectively where $I : Z^2 \rightarrow Z$ and $se : Z^2 \rightarrow Z$. Gray-scale dilation, denoted by $I \oplus se$ is defined as [23]:

$$(I \oplus se)(x, y) = max\{I(x - s, y - t) + se(s, t)|(x - s), (y - t) \in D_I; (s, t) \in D_{se}\} \quad (2.24)$$

Gray-scale erosion, denoted by $I \ominus se$ is defined as [23]:

$$(I \ominus se)(x, y) = min\{I(x + s, y + t) - se(s, t)|(x + s), (y + t) \in D_I; (s, t) \in D_{se}\} \quad (2.25)$$

where $D_I$ and $D_{se}$ are the domains of $I$ and $se$, respectively. The morphological gradient is computed as:

$$g(x, y) = (I \oplus se)(x, y) - (I \ominus se)(x, y) \quad (2.26)$$

For our experiments, we implement the morphological gradient using non-flat $3 \times 3$ structuring elements. The morphological gradient algorithm is listed in table 2.3 [23].

### 2.5.3 Kirsch Edge Detection Algorithm

The Kirsch edge detector [32] uses a contrast function to calculate the magnitude of the gradient of the image at an arbitrary location g$(x, y)$ as shown in the equation below:

$$g_i(x, y) = 5 \times (p_i + p_{i+1} + p_{i+2}) - 3 \times (p_{i+3} + p_{i+4} + ... + p_{i+7}) \quad (2.27)$$

Table 2.3: Morphological gradient algorithm

---

1. Input image $I$
2. For all windows in image $I$, calculate the sums of each pixel in the window and its corresponding value in the structuring element $se$ to produce $w_{sum}$.
3. Find the maximum value of the $w_{sum}$, which is $(I \oplus se)(x, y)$.
4. For all windows in image $I$, from each pixel in the window, subtract its corresponding value in the structuring element $se$ to produce $w_{sub}$.
5. Find the minimum value of the $w_{sub}$, which is $(I \ominus se)(x, y)$.
6. For all windows in image $I$, calculate the gradient as: $g(x, y) = (I \oplus se)(x, y) - (I \ominus se)(x, y)$.
7. For all windows in image $I$, compare the gradient $g(x, y)$ against a threshold to detect edges.

---

| $p_0$ | $p_1$ | $p_2$ |
|---|---|---|
| $p_7$ | $p_c$ | $p_3$ |
| $p_6$ | $p_5$ | $p_4$ |

Figure 2.2: A $3 \times 3$ window

where the subscripts are evaluated modulo 8. As shown in figure 2.2, $p_c$ is the central pixel in the window and $p_0$, $p_i$,...,$p_7$ are the neighboring pixels.

The gradient $g_i(x, y)$ in equation 2.27 is calculated for all eight locations of neighboring pixels: $i = 0, 1, 2, ..., 7$. The magnitude at location $(x, y)$ is the maximum value of the eight calculated gradient:

$$g(x, y) \quad = \quad max(g_0(x, y), g_1(x, y), g_2(x, y), ..., g_7(x, y)) \tag{2.28}$$

The algorithm performs convolutions of $3 \times 3$ windows with 8 masks, calculating 8 gradients for 8 different edge orientations [52]. The pixel in the center of the window is identified as an edge if the maximum of the 8 gradients is greater than a threshold.

### 2.5.4 Trajkovic Corner Detection Algorithm

The Trajkovic corner detection algorithm detects corners based on the characteristic of corners that the intensity change of image must be high in all mutually orthogonal directions [53]. It computes the corner response function as the minimum intensity change in all possible directions. The algorithm considers a circular window with all the lines that pass through the nucleus of the circle and intersect the circle in two opposite points of $P$ and $P'$ (Figure 2.3).

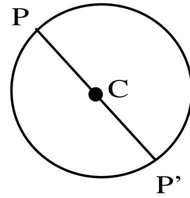



Figure 2.3: Pair of opposite points in a circular window

The corner response function $R_C$ is defined as :

$$R_C = min((f_P - f_C)^2 + (f_{P'} - f_C)^2) \tag{2.29}$$

where $C$ is the central point and $f_P$ represents the image intensity at point $P$. In practice, a discrete approximation of the circular window is used. For example, for a $3 \times 3$ circular window, there will be 4 pairs of opposite points (*i.e.* (A,A'), (B,B'), (D,D'), and (E,E')), as seen in figure 2.4.

| D | B | E |
|---|---|---|
| A' | C | A |
| E' | B' | D' |

Figure 2.4: Pairs of opposite points for a $3 \times 3$ circular window

In order to avoid false corner response, which is caused by a strong edge with a direction different to the ones examined, an interpixel approximation is used, which approximates the corner response function for any pair of points that is located between the known points. Moreover, to reduce the cost of computation, a lower resolution of

the image is used for detecting the corners. The Trajkovic corner detection algorithm is summarized in table 2.4 [42]. The algorithm uses windows of $3 \times 3$ pixels.

Table 2.4: Trajkovic corner detection algorithm

1. Input original image $I$ and low resolution image $J$
2. For all windows centered at (x,y) in the low resolution image $J$, calculate the cornerness measure: $R(x,y) = min(r_A, r_B, r_D, r_E)$ where

   $r_A = (f_A - f_C)^2 + (f_{A'} - f_C)^2$

   $r_B = (f_B - f_C)^2 + (f_{B'} - f_C)^2$

   $r_D = (f_D - f_C)^2 + (f_{D'} - f_C)^2$

   $r_E = (f_E - f_C)^2 + (f_{E'} - f_C)^2$
3. If $R(x,y) > threshold_1$ then mark the pixel at (x,y) as a corner candidate in the cornerness map $C(x,y)$.
4. For all windows centered at (x,y) in the low resolution image $J$, calculate the interpixel approximation cornerness measure $R_{interpixel}$:

   a. If for all i=1,2,3,4 either $B_i \geq 0$ or $A_i + B_i \leq 0$ then $R_{interpixel}(x,y) = R(x,y)$

   b. if for all i=1,2,3,4 $B_i < 0$ and $A_i + B_i > 0$ then $R_{interpixel}(x,y) = min(r_i - \frac{B_i^2}{A_i})$

   where

   $r_1 = r_A$, $r_2 = r_B$, $r_3 = r_D$, and $r_4 = r_E$

   $B_1 = (f_B - f_A)(f_A - f_C) + (f_{B'} - f_{A'})(f_{A'} - f_C)$

   $B_2 = (f_D - f_B)(f_B - f_C) + (f_{D'} - f_{B'})(f_{B'} - f_C)$

   $B_3 = (f_E - f_D)(f_E - f_C) + (f_{E'} - f_{D'}(f_{E'} - f_C)$

   $B_4 = (f_{A'} - f_E)(f_E - f_C) + (f_A - f'_E)(f_{E'} - f_C)$

   $A_1 = r_B - r_A - 2B_1$

   $A_2 = r_D - r_B - 2B_2$

   $A_3 = r_E - r_D - 2B_3$

   $A_4 = r_A - r_E - 2B_4$
5. If $R_{interpixel}(x,y) < threshold_2$ then the pixel at (x,y) is not a corner, otherwise set C(x,y) to $R_{interpixel}(x,y)$.
6. Perform non-maximal suppression on C(x,y) to find local maxima. The final non-zero values on C(x,y) indicate the locations of corners in the original image.

### 2.5.5 Median Filter

Median filter removes noise (*e.g.* impulse noise) by smoothing the image while it reduces the blurring of edges [52]. First, it sorts the pixels of a neighborhood and then it replaces the central pixel of the window with the median value of the pixels in that neighborhood. Performing a sort on pixels of each neighborhood can be very expensive. A more efficient algorithm, proposed by Huang *et al.* [27] benefits from the fact that the windows of pixels in an image overlap. In other words, at each step, the current window loses the leftmost column and replaces it with a new right column. Thus, for a window of $m$ rows and $n$ columns, $mn - 2m$ pixels remain unchanged and do not need to be re-sorted. The algorithm is listed in table 2.5 [27] [52]. For our experiments, we implement the median filter using $3 \times 3$ windows of pixels.

### 2.5.6 Local Variance

This algorithm calculates the local variances of $5 \times 5$ windows of pixels in the image. In order to produce a gray-level image as the output, the algorithm compares the local variance of each window against a threshold; if the local variance is lower than the threshold, the response of the window is the average value of all 25 pixels in the windows. Otherwise, the response is the central pixel of the window.

## 2.6 Sets of Test Images

In this section, we present the sets of test images that we use for our experiments throughout this thesis. Our data sets contain four different sets of images as listed below:

1. Natural Images: 40 natural images of $512 \times 512$ pixels
2. Industrial Images: 8 industrial images of $512 \times 512$ pixels including text classification, quality control, and cell imaging.
3. Medical Images: 30 ultrasound images of $280 \times 400$ pixels for prostate [33] and breast cancer [29]

Table 2.5: Median algorithm

1. Input image $I$
2. set $th = \frac{mn}{2}$
3. Sort the first window in the new row and generate a histogram $H$ of the pixels in the window. Determine the median value of pixels in the window, $med$, and calculate $lt_med$: the number of pixels with intensity less than or equal to $med$.
4. For each pixel $p$ in the leftmost column of the window with the intensity of $p_g$, perform:
   a. $H[p_g] = H[p_g] - 1$
   b. If $p_g < med$ then $lt_med = lt_med - 1$
5. Move the window one column right: For each pixel $p$ in the rightmost column of the window with the intensity of $p_g$, perform:
   a. $H[p_g] = H[p_g] + 1$
   b. If $p_g < med$ then $lt_med = lt_med + 1$
6. If $lt_med > th$ then go to 7, else repeat:
   a. $lt_med = lt_med + H[med]$
   b. $med = med + 1$
   until $lt_med \geq th$. Go to 8.
7. Repeat:
   a. $med = med - 1$
   b. $lt_med = lt_med - H[med]$
   until $lt_med \leq th$.
8. If the window is not the last window in the row, go to 4.
9. If the window is not the last window in the image, go to 3.

4. Barcode Images: 50 images of $480 \times 640$ pixels of a 2-dimensional barcode used in an industrial image recognition system [11]

Figure 2.5 shows three sample images from each set of images.

Figure 2.5: Sample images from: $row_1$: natural, $row_2$: industrial, $row_3$: medical, and $row_4$: 2-D barcode images.

# Chapter 3

# Theory of Window Memoization

In this chapter, the underlying roots of our performance improvement technique, window memoization, are presented. Window memoization benefits from a characteristic of image data, computational redundancy, to improve the performance of local image processing algorithms. Our goal in this chapter is to show mathematically that the sum of the coding and interpixel redundancy of an image has a positive relationship with the computational redundancy of the image: that is, higher coding and interpixel redundancy in an image leads to higher computational redundancy in the image. This enables us to categorize images based on their potential performance improvement obtained by window memoization in software and hardware without actually implementing the window memoization technique. In chapters 4 and 5, we will present the implementations of window memoization in software and hardware, respectively.

The coding redundancy of an image indicates that each pixel is represented with more bits than is necessary. Interpixel redundancy reveals that neighboring pixels are similar and correlated. Our studies show that these two types of redundancy lead to similar windows of pixels across the image. In local image processing algorithms, the presence of similar windows across the image leads to computational redundancy, which can be exploited to improve performance.

In section 3.1, we introduce the notions of *computational redundancy* in an image, *perfect window memoization*, and the idea of *tolerant memoization*. In order to show that the coding and interpixel redundancy of an image ($C_r + IP_r$) have a positive relationship

with the computational redundancy of the image ($Comp_r$), a few intermediate notions are also introduced in section 3.1 including the *ideal reusability* ($R_{ideal}$) and *reusability* ($R$) of image, and the *hit rate* of perfect window memoization ($HR_{pc}$). Throughout this chapter, we show mathematically that the following chain of relationships holds:

$$(C_r + IP_r) \propto^+ R_{ideal} \propto^+ R \propto^+ HR_{pc} \propto^+ Comp_r \tag{3.1}$$

To show that the whole chain holds, first, in section 3.1, we show that the last step of the chain holds: the hit rate of perfect window memoization for an image has a positive relationship with the computational redundancy of the image ($HR_{pc} \propto^+ Comp_r$). Afterward, we begin from the first step of the chain. In section 3.2, it is shown that the amount of coding and interpixel redundancy of an image has a positive relationship with the ideal reusability of the image ($(C_r + IP_r) \propto^+ R_{ideal}$). In section 3.3, first, we show that the ideal reusability of an image has a positive relationship with the reusability of the image ($R_{ideal} \propto^+ R$). Afterward, it is demonstrated that the reusability of an image has a positive relationship with the hit rate of perfect window memoization for the image ($R \propto^+ HR_{pc}$). Putting all the above mentioned intermediate steps together, it is concluded that the amount of coding and interpixel redundancy of an image has a positive relationship with the computational redundancy of the image.

$$(C_r + IP_r) \propto^+ Comp_r \tag{3.2}$$

## 3.1 Image Reusability for Local Image Processing Algorithms

In this section, first, we define computational redundancy of an image as a measure that indicates the percentage of unnecessary computation sets performed on the image. We also define a high level model, perfect window memoization, which gives the upper-bound for performance improvement obtained by exploiting the computational redundancy of an image. We show that the hit rate of perfect window memoization for an image has a

positive relationship with the computational redundancy of the image ($HR_{pc} \propto^{+} Comp_r$). Afterward, we present the idea of tolerant memoization, which allows one to reuse the response of similar but not necessarily identical windows of pixels in an image. We also define two measures: reusability and ideal reusability of an image. These measures help to build intermediate steps in order to complete our goal, which is to show that the amount of coding and interpixel redundancy of an image has a positive relationship with the computational redundancy of the image.

### 3.1.1 Computational Redundancy in an Image

The windows of pixels are essential parts of designing or implementing local processing algorithms. When dealing with the windows of pixels in an image as the building blocks of the image, it is more convenient to consider the windows of pixels as a higher dimension gray levels or *gray-level vectors*. Gray-level vectors, which we call *symbols*, are defined based on the size of local windows. For windows of $m \times m$ pixels, an $m^2$-dimensional symbol represents all the windows in the image whose corresponding pixels are identical. In other words, a symbol represents all identical windows in the image. A window $win$ belongs to (or matches) a symbol $sym$ if each pixel in $win$ is equal to the corresponding pixel in $sym$:

$$\forall pix \in win, \forall pix' \in sym,\ pix = pix' \implies win \in sym \tag{3.3}$$

where $pix$ and $pix'$ are corresponding pixels of window $win$ and symbol $sym$, respectively.

For local windows of 2 pixels, the possible number of 2-D symbols will be $256 \times 256 = 65536$, assuming that the pixels in the original image have 256 gray levels. In general, for local windows of $m \times m$ pixels, the possible number of $m^2$-D symbols will be $GL^{m^2}$ where $GL$ is the number of gray levels in the original image. Symbols are equal to normal 1-D gray levels when the size of local windows are one pixel.

Ignoring the geometry of objects in an image, for a given local window size, the image can be characterized by the probability of occurrences of symbols in the image. Assume that a discrete random variable, $s_i$ in the interval $[0, s)$, represents the symbols of an image. The probability of occurrence of each symbol ($s_i$) in the image is:

$$P(s_i) = \frac{n_i}{n},\ i = 0, 1, 2, ..., s-1 \tag{3.4}$$

where $s$ is the total number of symbols in the image, $n_i$ is the number of times that the $i^{th}$ symbol appears in the image, and $n$ is the number of total windows in the image.

In chapter 2, it was discussed that in local image processing algorithms, the response of the mask, $r_{ij}$, solely depends on the pixels in the local window, $w_{ij}$, covered by the mask (equation 2.11). All the windows in the image that contain similar pixels are identified by one symbol, $s_i$. Thus, for a given algorithm, a symbol $s_i$ will produce the same response for $n_i$ times. This means that much of the mask operations sets that are applied to symbol $s_i$ are unnecessary or redundant.

Ideally, for each symbol in an image, the mask operations must be performed only once. Therefore, for a given algorithm, the number of the mask operations sets that are absolutely necessary to apply to an image, in order to complete processing the image is equal to the number of symbols present in the image ($s$).

We define the *computational redundancy* ($Comp_r$) of an image as the percentage of the mask operations sets that are not necessary to perform, in order to complete processing the image. In other words, the computational redundancy of an image indicates what percentage of mask operations sets are redundant. For each symbol $s_i$, only one set of mask operations is absolutely necessary. Thus, the rest of the mask operations sets, which are equal to $n_i - 1$ are redundant. The number of the redundant mask operations sets for all $s$ symbols in the image will be $n - s$. Thus, the percentage of the redundant mask operations sets in an image is:

$$\begin{aligned} Comp_r &= \frac{n-s}{n} \\ &= 1 - \frac{s}{n} \end{aligned} \tag{3.5}$$

where $s$ and $n$ are the total number of symbols and the total number of windows in the image, respectively. It is seen that the computational redundancy depends on the total number of windows and the number of symbols present in the image. An image with small number of symbols will have a higher computational redundancy. In contrast, the

computational redundancy will be 0 if the number of symbols is equal to the total number of windows in the image ($s = n$). This, of course, is very rare for typical images where the number of symbols present in the image is usually less than the number of windows in the image. Therefore, the number of mask operations sets that an algorithm applies to an image is usually higher than that absolutely necessary.

### 3.1.2 Perfect Window Memoization

We define an abstract processing model, *perfect window memoization*, which minimizes the number of redundant mask operations sets performed on an image. Perfect window memoization is a high level model that gives an estimation (upper-bound) of performance gain in software and hardware, obtained by eliminating the redundant mask operations sets.

Perfect window memoization uses a *reuse table* to store symbols, $s_i$, and the corresponding responses, $r_i$. The reuse table in perfect window memoization is in fact a *perfect cache*. The perfect cache never inserts the same data again. In other words, each symbol is inserted only once and when inserted, it is never evicted. It should be mentioned that perfect cache is used for theoretical study of window memoization and it cannot be used in actual implementations of window memoization. Instead, as it will be discussed in chapters 4 and 5, a practical memory architecture will be used as the reuse table in software and hardware implementations of window memoization. Nevertheless, with the assumption of even distributions of symbols across the image, which will be discussed in chapter 4, section 4.6.1, perfect cache can be used as a high level model for the actual implementations of the reuse table.

When perfect window memoization receives a window for the first time, there is no matching symbol in the reuse table and thus, a *miss* occurs. Therefore, it applies the mask operations set to the window and inserts its matching symbol and its response into the reuse table. Upon encountering the same window again in the future, this time there is a matching symbol in the reuse table and thus, a *hit* occurs. As a result, perfect window memoization does not perform the mask operations set on the recent window. Instead, it looks up or *reuses* the corresponding response from the reuse table. In other words, each symbol is inserted into the reuse table only once and the mask operations set is applied

to its corresponding windows only once regardless of the probability of occurrence of the symbol in the image. This means that when a symbol is inserted into the reuse table, it is never replaced with another symbol. Eventually, all symbols of an image along with their responses will be stored in the reuse table.

For a symbol whose windows appear in the image $n_i$ times, $n_i - 1$ mask operations sets can be skipped and their corresponding responses can be reused. In other words, the number of hits for the windows of a symbol with the population of $n_i$ will be $n_i - 1$. The number of hits for all windows in the image will be $n - s$ where $s$ is the number of symbols in the image. The *hit rate* ($HR_{pc}$) is defined as the percentage of the times that the incoming windows find a matching symbol in the reuse table and therefore, reuse their previously calculated responses. The subscript *pc* (perfect cache) indicates that the reuse table used by perfect window memoization is a perfect cache.

In order to insert all symbols of an image into the reuse table, the reuse table size must be equal to the number of symbols in the image. In practice, however, the required size may be too large to afford. Therefore, perfect window memoization explores inserting a variable portion of symbols in the image into the reuse table. The symbols with higher probabilities of occurrence will contribute more to performance gain if inserted into the reuse table. The reason is that for such symbols, higher number of mask operations sets can be skipped.

For a reuse table of $k$ entry size, the number of hits will be the sum of populations of the $k$ most frequent symbols (the total number of windows that belong to $k$ inserted symbols) minus the number of inserted symbols (or the number of misses), which is $k$.

$$HR_{pc}(k) = \frac{n_0 + n_1 + ... + n_{k-1}}{n} - \frac{k}{n} \tag{3.6}$$

where $n_0, n_1, n_2, ..., n_{k-1}$ are the populations of $k$ most frequent symbols in the image, and $n$ and $k$ are the total number of windows in the image and the number of inserted symbols (or the reuse table size), respectively. Although hit rate is a discrete function of $k$ (the reuse table size), to simplify the analysis, we model its behaviour with a continuous function. Assume that $P$ is a continuous function that models the probability density function ($PDF$) of the image symbols, which have been sorted in descending order. We can write:

$$HR_{pc}(k) = \int_0^{k-1} P(x)dx - \frac{k}{n} \tag{3.7}$$

where $\int_0^{k-1} P(x)dx$ is the probability of $k$ most frequent symbols (or inserted symbols).

In contrast to normal caches in processor architecture where the cache size is always smaller than the number of unique entries, in window memoization, when input images are simple (*i.e.* small number of symbols), it is possible that the reuse table size is larger than the total number of symbols in the image (*i.e.* $k > s$). In this case, the reuse table will not be filled with the symbols, which means that the total number of misses will be equal to the total number of symbols ($s$), rather than the reuse table size ($k$). Thus, a more accurate equation for hit rate is:

$$HR_{pc}(k) = \int_0^{k-1} P(x)dx - \frac{min(k,s)}{n} \tag{3.8}$$

In other words, if $k \leq s$ then $k$ indicates the total number of misses. Otherwise, the total number of misses will be $s$.

Ignoring some details of implementations of perfect window memoization in software and hardware, $HR_{pc}(k)$ indicates what percentage of mask operations sets can be skipped and their corresponding responses can be reused, given that $k$ most frequent symbols have been inserted into the reuse table.

#### Computational Redundancy versus Hit Rate

In this chapter, the relationship of the coding and interpixel redundancy of an image with the computational redundancy of the image is investigated. As an intermediate step, we need to know the relationship between the hit rate of perfect window memoization for an image ($HR_{pc}$) and the computational redundancy of the image ($Comp_r$). This will help to tie the coding and interpixel redundancy of an image with the performance gain achieved by perfect window memoization.

As the first step in proving the chain of relations (relation 3.1), we want to show that for any size of the reuse table, the hit rate of perfect window memoization for an image has a positive relationship with the computational redundancy of the image:

$$HR_{pc} \propto^{+} Comp_r \tag{3.9}$$

Relation 3.9 can be rewritten as:

$$(\forall k,\ HR_{pc1}(k) \leq HR_{pc2}(k)) \implies (Comp_{r1} \leq Comp_{r2}) \tag{3.10}$$

Substituting $HR_{pc}(k)$ and $Comp_r$ into the relation above from equations 3.8 and 3.5, respectively, gives:

$$\left(\forall k,\ \int_0^{k-1} P_1(x)dx - \frac{min(k,s_1)}{n} \leq \int_0^{k-1} P_2(x)dx - \frac{min(k,s_2)}{n}\right) \implies \left(1 - \frac{s_1}{n} \leq 1 - \frac{s_2}{n}\right) \tag{3.11}$$

Because the left hand side of the relation above is correct for any size of $k$, we can assume that $k$ is a very large number such that it is larger than both $s_1$ and $s_2$ meaning that $min(k, s_1) = s_1$ and $min(k, s_2) = s_2$. By having a very large $k$, the two integrals will become 1 and we will have [1]:

$$\left(1 - \frac{s_1}{n} \leq 1 - \frac{s_2}{n}\right) \implies \left(1 - \frac{s_1}{n} \leq 1 - \frac{s_2}{n}\right) \tag{3.12}$$

The relation above proves that the hit rate of perfect window memoization for an image has a positive relationship with the computational redundancy of the image.

$$HR_{pc}(k) \propto^{+} Compr_r \tag{3.13}$$

It is observed that the computational redundancy of an image gives the maximum hit rate that the perfect window memoization can possibly achieve for the image.

### 3.1.3 Tolerant Memoization

The human vision system cannot distinguish small amounts of error in an image, with respect to a reference image. Therefore, small errors can usually be tolerated in an image

[1] It should be noted that in reality, in order for the integrals to become 1, $k$ only needs to surpass $s$.

processing system. Perfect window memoization benefits from such tolerance to increase the performance gain. In section 3.1.1, we defined the condition, in which case a window belongs to a symbol:

$$\forall pix \in win,\ \forall pix' \in sym,\ pix = pix' \implies win \in sym \tag{3.14}$$

In the above definition, a window belongs to a symbol if all the pixels in the window are identical to the corresponding pixels in the symbol. We relax the equality requirement such that similar but not necessarily identical windows may belong to one symbol:

$$\forall pix \in win,\ \forall pix' \in sym,\ MSB(d, pix) = pix' \implies win \in sym \tag{3.15}$$

where $MSB(d, pix)$ represents $d$ most significant bits of pixel $pix$ in window $win$ and $pix'$ has $d$ bits. In the ideal case (*i.e.* definition 3.14), $d = 8$. By reducing $d$, windows that are similar but not identical are assigned to one symbol. From the perfect window memoization perspective, this means that the response of one window may be assigned to a similar but not necessarily identical window. As $d$ decreases, more windows with minor differences are assumed equal and thus, the hit rate of perfect window memoization increases drastically. Assigning the response of a window to a similar but not necessarily identical window introduces inaccuracy in the result of the algorithm to which perfect window memoization is applied. However, in practice, the accuracy loss in responses is usually negligible.

The error in an image ($Img$) with respect to a reference image ($R_{Img}$) is usually measured by *signal-to-noise ratio* or SNR as follows [23] :

$$SNR = 20log_{10}(\frac{A_{signal}}{A_{noise}}) \tag{3.16}$$

where $A$ is the RMS (root mean squared) amplitude. $A^2_{noise}$ is defined as:

$$A^2_{noise} = \frac{1}{rc}\sum_{i=0}^{r-1}\sum_{j=0}^{c-1}(Img(i,j) - R_{Img}(i,j))^2 \tag{3.17}$$

where $r \times c$ is the size of $Img$ and $R_{Img}$. If SNR of an image is higher than $30dB$, the error in the image is nearly indistinguishable by the observer [1]. As it will be discussed

in chapter 4, we have found that for most algorithms, $d$ can be decreased to as low as 4 while the average SNR of result images are maintained slightly below $30db$ (*i.e.* $29.52db$). Therefore, in our experiments in the remaining of this chapter, we choose $d$ to be 4. Nevertheless, the entire argument is valid and can be verified by empirical data for any reasonable value of $d$ (*i.e.* $d \in \{2, 3, 4, .., 8\}$).

### 3.1.4 Reusability of an Image

In order to investigate the relationship between the coding and interpixel redundancy of an image and the hit rate of perfect window memoization for the image (relation 3.1), we introduce an intermediate step, which we call *reusability of image*. On one hand, the reusability of image should be tied to the coding and interpixel redundancy and on the other hand, it must be connected to the hit rate of perfect window memoization. The relationships of the reusability of image with coding/interpixel redundancy and the hit rate of perfect window memoization will be discussed in section 3.3.

In section 2.1.3, we discussed that the interpixel redundancy of an image can be measured using three different methods: mapping transforms, Markov model, and auto-correlation. We reasoned that for our purpose, mapping transform is the most suitable method to measure interpixel redundancy because it is both easy to compute and directly linked to coding redundancy. To define the reusability of an image, first, we define a mapping transform, which converts the input image into an image whose interpixel redundancy has been extracted. Transform $icp$ (information at central pixel) takes a local window as input and outputs the new information, $\delta$, carried by the pixel at the center of the window. The new information carried by the central pixel is defined with respect to the pixels in the local window.

$$\delta = icp(w_{ij}) \tag{3.18}$$

The transformed image ($Img^{icp}$) will be produced when the transform $icp$ is applied to all windows in the image.

$$Img^{icp} = icp(Img) \tag{3.19}$$

It is possible that *icp* maps windows that do not belong to one symbol to a single value because two windows can be different while their central pixels carry exactly the same new information. This means that more than one symbol may be mapped to a given $\delta$ in the transformed image $Img^{icp}$. Thus, as discussed in section 2.1.3, the number of bins in the histogram of the transformed image is considerably less than the number of bins in the histogram of symbols of the original image, which makes it computationally inexpensive to calculate the interpixel redundancy of an image.

We define the *reusability of image (R)* based on the *icp* transform. $R(\delta)$ is the average number of the redundant mask operations sets per symbol for symbols in the image whose central pixels carry new information in the range $[-\delta, +\delta]$. To calculate the number of redundant mask operations sets for a given $\delta$, we need to calculate the number of symbols in the image whose central pixel carry the new information $\delta$ (*i.e.* $n_{sym}(\delta)$) and the number of windows that belong to such symbols (*i.e.* $n_{win}(\delta)$). In other words, for a given $\delta$, $n_{sym}(\delta)$ indicates the number of necessary mask operations sets and $n_{win}(\delta)$ is the total number of mask operations sets. Thus, for a given $\delta$, the number of redundant mask operations sets, $mask_{red}(\delta)$, is calculated by:

$$mask_{red}(\delta) = n_{win}(\delta) - n_{sym}(\delta) \tag{3.20}$$

The average number of the redundant mask operations sets per symbol is:

$$\begin{aligned} \overline{mask_{red}}(\delta) &= \frac{mask_{red}(\delta)}{n_{sym}(\delta)} \\ &= \frac{n_{win}(\delta) - n_{sym}(\delta)}{n_{sym}(\delta)} \\ &= \frac{n_{win}(\delta)}{n_{sym}(\delta)} - 1 \end{aligned} \tag{3.21}$$

The equation above can be normalized with respect to the total number of windows in the image, $n$:

$$\overline{mask_{red}}(\delta) = \frac{\frac{n_{win}(\delta)}{n}}{\frac{n_{sym}(\delta)}{n}} - 1 \tag{3.22}$$

Let $P^{icp}$ be the probability density function of $Img^{icp}$. $P^{icp}(\delta)$ is also the probability of occurrences of symbols in the original image $Img$ whose central pixels carry the same new information, $\delta$ [2]. For short, we call $P^{icp}$ the probability density function of symbols in the original image based on $\delta$. Therefore, $\frac{n_{win}(\delta)}{n}$ in equation 3.22 can be replaced by $P^{icp}(\delta)$:

$$\overline{mask_{red}}(\delta) = \frac{P^{icp}(\delta)}{\frac{n_{sym}(\delta)}{n}} - 1 \tag{3.23}$$

$\frac{n_{sym}(\delta)}{n}$ is the ratio of the number of symbols in the original image $Img$ for a given $\delta$, and the total number of windows in the image. This ratio is not a probability distribution because the total number of symbols present in image (*i.e.* $\int_{+\infty}^{+\infty} n_{sym}(x)dx$) is usually less than the total number of windows in image, $n$. In addition, the total number of symbols present in an image may be different for different images. We define $P_u^{icp}$ as a pseudo probability distribution, which for a given $\delta$ indicates the ratio of the population of symbols and the total number of windows in the original image:

$$P_u^{icp}(\delta) = \frac{n_{sym}(\delta)}{n} \tag{3.24}$$

We call $P_u^{icp}$ the *uniqueness probability density function of symbols* in the original image based on $\delta$ since it indicates the number of unique symbols present in the image whose central pixels carry new information $\delta$, normalized by the total number of windows in the image. Substituting $P_u^{icp}(\delta)$ into equation 3.23 gives:

$$\overline{mask_{red}}(\delta) = \frac{P^{icp}(\delta)}{P_u^{icp}(\delta)} - 1 \tag{3.25}$$

As defined earlier, the reusability of image $R(\delta)$ is the average number of the redundant mask operations sets per symbol for symbols in the image whose central pixels carry new

[2] As it will be discussed in section 3.1.6, we model $P^{icp}$ with a continuous function. The probability of a continuous function should be defined over an interval (*i.e.* $P^{icp}(\delta \pm \epsilon)$) where $\epsilon = \frac{1}{2\times(2^{d+1}-1)}$ and $(2^{d+1} - 1)$ is the number of possible values for $\delta$ (see equation 3.36). ($d$ is the number of the most significant bits used for assigning windows to symbols). If $d = 4$, $\delta$ will be in the range $[-15, 15]$, which gives 31 bins and thus, $\epsilon = \frac{1}{2\times 31}$. However, to keep the writing of the following discussions clear and easy to follow, we sacrifice the mathematical rigor by ignoring $\pm\epsilon$. Therefore, what is meant by $P^{icp}(\delta)$ is in fact $P^{icp}(\delta \pm \epsilon)$.

information in the range $[-\delta, +\delta]$. Thus, we can write:

$$\begin{aligned} R(\delta) &= \int_{-\delta}^{+\delta} \overline{mask_{red}}(x)dx \\ &= \frac{\int_{-\delta}^{+\delta} P^{icp}(x)dx}{\int_{-\delta}^{+\delta} P_u^{icp}(x)dx} - 1 \end{aligned} \tag{3.26}$$

The equation above indicates that for a given range, $[-\delta, +\delta]$, how many mask operation responses can be reused per one symbol.

In sections 3.1.6 and 3.1.7, we will show that both the probability and uniqueness probability of symbols of an image based on $\delta$ (*i.e.* $P^{icp}$ and $P_u^{icp}$) can be modeled using a Laplace distribution.

### 3.1.5 Ideal Reusability of an Image

From equation 3.26, it is seen that the reusability of image depends on both the probability of occurrences of symbols ($\int_{-\delta}^{+\delta} P^{icp}(x)dx$) and the uniqueness probability of symbols ($\int_{-\delta}^{+\delta} P_u^{icp}(x)dx$) whose central pixels carry new information in the range $[-\delta, +\delta]$. As discussed before, the reusability of image must be connected with the coding and interpixel redundancy of the image. In section 3.2, we will show that there is a mathematical relationship between the coding and interpixel redundancy of an image and the probability of occurrences of symbols in the image based on $\delta$ ($\int_{-\delta}^{+\delta} P^{icp}(x)dx$). Nevertheless, there is no apparent relationship between the coding and interpixel redundancy of an image and the uniqueness probability of symbols of the image based on $\delta$ ($\int_{-\delta}^{+\delta} P_u^{icp}(x)dx$). Therefore, first, we define an idealized version of the image reusability, which ignores the effect of the uniqueness probability of symbols on reusability and investigate the relationship between the coding/interpixel redundancy and ideal reusability of the image. Afterward, we study the the relationship between the ideal reusability and reusability of the image.

We define the ideal reusability of an image as:

$$R_{ideal}(\delta) = \int_{-\delta}^{+\delta} P^{icp}(x)dx \tag{3.27}$$

With this simplification, the probability density function of $Img^{icp}$ is the only information that we need in order to obtain knowledge about the the ideal reusability of the original image. In section 3.2, we will show that there is a positive relationship between the sum of coding ($C_r$) and interpixel redundancy ($IP_r$) of an image and the ideal reusability of the image:

$$C_r + IP_r \propto^{+} R_{ideal} \tag{3.28}$$

In section 3.3, we will show that there is a positive relationship between the ideal reusability and the reusability of an image:

$$R_{ideal}(\delta) \propto^{+} R(\delta) \tag{3.29}$$

In section 3.3, we will also show that there is a positive relationship between the reusability of an image and the hit rate of perfect window memoization for the image:

$$R(\delta) \propto^{+} HR_{pc}(k) \tag{3.30}$$

Given that in section 3.1.2, we showed that the hit rate of perfect window memoization for an image has a positive relationship with the computational redundancy of the image ($HR_{pc} \propto^{+} Comp_r$), ultimately we will show that:

$$(C_r + IP_r) \ \propto^{+} \ R_{ideal}(\delta) \ \propto^{+} \ R(\delta) \ \propto^{+} \ HR_{pc}(k) \ \propto^{+} \ Comp_r \tag{3.31}$$

For our experiments in the following sections of this chapter, we use local windows of $3 \times 3$ pixels. The experiments are repeatable for larger sizes of local windows (*i.e.* $5 \times 5$ and $7 \times 7$).

### 3.1.6 Laplace Model for the Probability of Symbols

The *icp* transform extracts the new information carried solely by one pixel at the center of a local window. Such a transform must be designed with respect to the local window's

size. For a window of 2 adjacent pixels, the new information carried by one pixel can be defined as the difference between the gray levels of the two pixels.

$$Img^{icp}(i,j) = Img(i,j) - Img(i,j+1) \tag{3.32}$$

Because adjacent pixels in image are usually similar, the probability density function of $Img^{icp}$ has a peak at zero and it drops rapidly as the differences between adjacent pixels become larger. The probability density function of symbols of $Img^{icp}$ (or the probability density function of symbols of the original images based on $\delta$) can be modeled by a zero mean Laplace distribution [40] [45] as shown in the equation below [3].

$$P^{icp}(x) = \frac{1}{\sqrt{2}\lambda^{icp}} e^{\frac{-\sqrt{2}|x|}{\lambda^{icp}}} \tag{3.33}$$

where $\lambda^{icp}$ is the scale factor of $Img^{icp}$, which is equal to the standard deviation of $Img^{icp}$ multiplied by a constant value:

$$\lambda^{icp} = \kappa \times \sigma^{icp} \tag{3.34}$$

The *icp* transform can be defined over a window of pixels. For $3 \times 3$ windows, we define the new information carried by the pixel at the center of the window, $w_{ij}$, which is located at the location $(i,j)$ in image, as follows:

$$\begin{aligned} icp(w_{ij}) &= \frac{1}{8} \sum_{m=-1}^{1} \sum_{n=-1}^{1} (w_{ij}(m,n) - w_{ij}(0,0)) \\ &= \delta_{ij} \end{aligned} \tag{3.35}$$

By applying the transform to the entire image, we will have:

$$Img^{icp}(i,j) = \frac{1}{8} \sum_{m=-1}^{1} \sum_{n=-1}^{1} (Img(i+m, j+n) - Img(i,j)) \tag{3.36}$$

The equation above calculates the average difference between the pixel at the center and the eight pixels surrounding the central pixel. It is not surprising to see that the

[3] This is somewhat similar to the energy distribution of natural images in the frequency domain where the lowest frequency has the largest amplitude and increasing the frequency causes the amplitude to decrease by a factor of $\frac{1}{f}$ (*i.e.* $P(f) = \frac{1}{f}$) [19].

probability density function of symbols of such a transformed image is also a zero mean Laplace distribution. In natural images, symbols with small $\delta$ (*i.e.* little differences in their pixel gray levels) usually belong to one object, texture or background. In contrast, symbols with large $\delta$ (*i.e.* large differences in their pixel gray levels) usually belong to boundaries of the objects in the image. The boundaries constitute a small fraction of the whole image and most of image is usually background or objects with consistent surfaces. Therefore, the probability density function of $Img^{icp}$ has a peak at $\delta = 0$ and as $\delta$ increases/decreases, it drops drastically. Moreover, on average, the probability of symbols whose central pixels carry new information $\delta$ is almost the same as that of symbols whose central pixels carry new information $-\delta$. Thus, the probability density function of $Img^{icp}$ is symmetric around 0. As we mentioned in section 3.1.3, in our experiments, we use 4 most significant bits of each pixel (*i.e.* $d = 4$). Thus, the range for $\delta$ will be $[-15, 15]$.

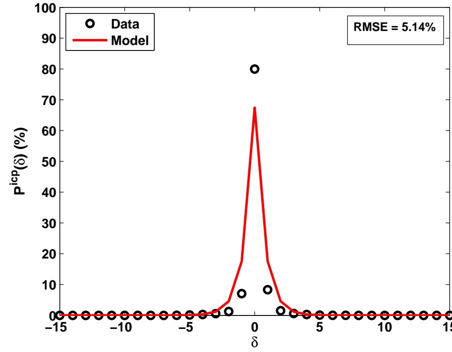
Figure 3.1 shows three samples from natural images along with the probability density functions of the transformed images and their corresponding Laplace curves. The *root mean squared error (RMSE)* of the curve fits are 5.28%, 3.37% and 3.55%, respectively. We have done the curve fit for all 40 natural images and the average RMSE for 40 images is 3.60%.

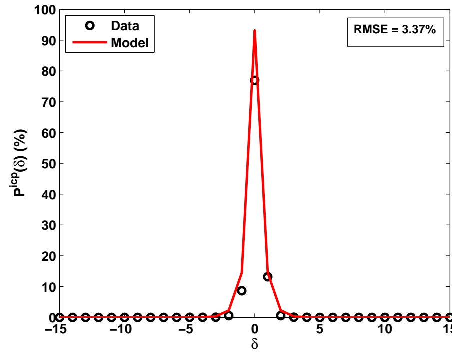
### 3.1.7 Laplace Model for Uniqueness Probability of Symbols

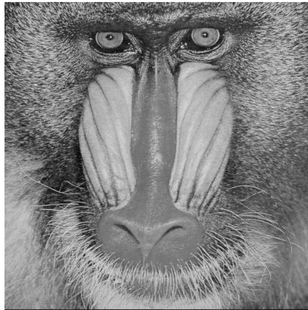
Similar to the probability density function of symbols in image based on $\delta$ ($P^{icp}$), the uniqueness probability of symbols ($P_u^{icp}(\delta)$) can be modeled by a Laplace distribution. However, as we discussed in the previous section, $P_u^{icp}$ is not an actual probability density function and therefore, the area under $P_u^{icp}$ curve is not 1. As a result, we model $P_u^{icp}$ by a general form of the Laplace distribution:

$$Pu^{icp}(x) = ae^{-b|x|} \tag{3.37}$$

We have performed the curve fits for the uniqueness probabilities of all 40 natural images and it has been observed that the model matches the experimental data with a high accuracy. The average RMSE for 40 natural images is 5.06%. Figure 3.2 shows three sample natural images along with their uniqueness probability of symbols.





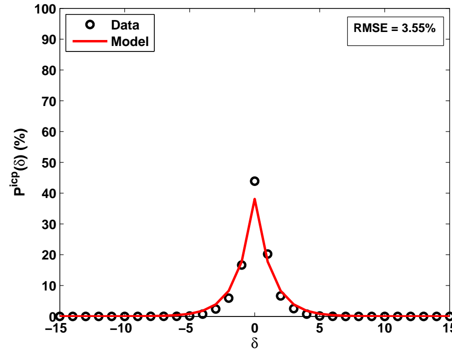
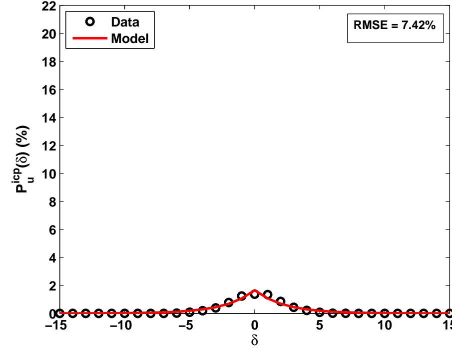


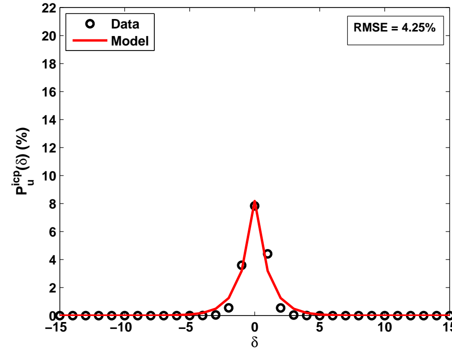

Figure 3.1: Left: original images. Right: the probability density function of symbols in the original image based on $\delta$.

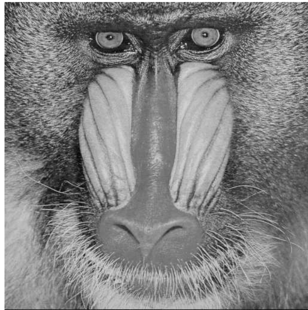
## 3.2 Ideal Reusability of Image Data versus Coding and Interpixel Redundancy of the Image

In this section, we prove that the sum of coding and interpixel redundancy of an image has a positive relationship with the ideal reusability of an image.

$$C_r + IP_r \propto^{+} R_{ideal} \tag{3.38}$$





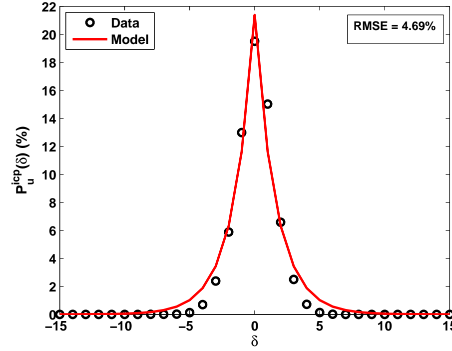



Figure 3.2: Left: original images. Right: the uniqueness probability density function of symbols in the original image based on $\delta$.

In order to prove relation 3.38, we break down the problem into smaller pieces and each one is proved in the following sections:

1. Section 3.2.1: The sum of coding and interpixel redundancies of the original image $(C_r + IP_r)$ has a negative relationship with the entropy of the transformed image $(H^{icp})$:

$$(C_r + IP_r) \propto^{-} H^{icp}$$

2. Section 3.2.2: The entropy of the transformed image ($H^{icp}$) has a positive relationship with the standard deviation of the transformed image ($\sigma^{icp}$):

$$H^{icp} \propto^{+} \sigma^{icp}$$

3. Section 3.2.3: The standard deviation of the transformed image ($\sigma^{icp}$) has a negative relationship with the ideal reusability of the original image ($R_{ideal}$):

$$\sigma^{icp} \propto^{-} R$$

### 3.2.1 Entropy of the Transformed Image versus Coding and Interpixel Redundancy of the Original Image

In this section, we show that the sum of coding and interpixel redundancy of the original image has a negative relationship with the entropy of the transformed image:

$$(C_r + IP_r) \propto^{-} H^{icp}$$

The relation above indicates that as the coding and interpixel redundancy of an image increases, the entropy of the transformed image decreases. As we discussed in section 2.1.3, by applying a mapping transform (*icp*) to an image, which extracts the interpixel redundancy of the image, a transformed image is created whose interpixel redundancy has been removed. Therefore, the entropy of the transformed image is much less than that of the original image. The decrease in entropy from the original image ($H^{orig}$) to the transformed image ($H^{icp}$) indicates the amount of interpixel redundancy in the original image:

$$IP_r = H^{orig} - H^{icp} \tag{3.39}$$

From equation 2.3 in section 2.1.2, we have $H^{orig} = log_2(GL) - C_r$. Substituting $H^{orig}$ into equation 3.39 gives:

$$IP_r = log_2(GL) - C_r - H^{icp} \tag{3.40}$$

which gives:

$$C_r + IP_r = log_2(GL) - H^{icp} \tag{3.41}$$

This means that the sum of coding and interpixel redundancy of the original image has a negative relationship with the entropy of the transformed image.

$$(C_r + IP_r) \propto^{-} H^{icp} \tag{3.42}$$

### 3.2.2 Standard Deviation of the Transformed Image versus Entropy of the Transformed Image

In this section, we show that the entropy of the transformed image has a positive relationship with the standard deviation of the transformed image:

$$H^{icp} \propto^{+} \sigma^{icp}$$

In section 3.1.6, we showed that the probability density function of $Img^{icp}$ can be modeled by a zero mean Laplace distribution. Because we have a continuous model for the probability density function of $Img^{icp}$, we can calculate the continuous entropy of $Img^{icp}$ ($H_c^{icp}$) based on its probability density function .

$$H_c^{icp} = -\int_{-\infty}^{+\infty} P^{icp}(x) \times log_2(P^{icp}(x))dx \tag{3.43}$$

where $P^{icp}(x) = \frac{1}{\sqrt{2}\lambda^{icp}} e^{\frac{-\sqrt{2}|x|}{\lambda^{icp}}}$. In equation 3.43, we can transform $log_2$ into the natural logarithm using:

$$\begin{aligned} log_2 P^{icp}(x) &= log_2 e \times ln P^{icp}(x) \\ &= log_2 e \times ln[\frac{1}{\sqrt{2}\lambda^{icp}} e^{\frac{-\sqrt{2}|x|}{\lambda^{icp}}}] \\ &= log_2 e \times [ln \frac{1}{\sqrt{2}\lambda^{icp}} - \frac{\sqrt{2}|x|}{\lambda^{icp}}] \end{aligned} \tag{3.44}$$

Substituting equation 3.44 into equation 3.43, we will have:

$$\begin{aligned} H_c^{icp} &= -\int_0^{+\infty} \frac{2}{\sqrt{2}\lambda^{icp}} e^{\frac{-\sqrt{2}x}{\lambda^{icp}}} \times log_2 e \times [ln \frac{1}{\sqrt{2}\lambda^{icp}} - \frac{\sqrt{2}x}{\lambda^{icp}}] dx \\ &= \frac{2log_2 e}{\sqrt{2}\lambda^{icp}} \times [ln(\sqrt{2}\lambda^{icp}) \int_0^{+\infty} e^{\frac{-\sqrt{2}x}{\lambda^{icp}}} dx + \frac{\sqrt{2}}{\lambda^{icp}} \int_0^{+\infty} x e^{\frac{-\sqrt{2}x}{\lambda^{icp}}} dx] \\ &= \frac{2log_2 e}{\sqrt{2}\lambda^{icp}} \times [ln(\sqrt{2}\lambda^{icp}) \times \frac{\lambda^{icp}}{\sqrt{2}} + \frac{\sqrt{2}\lambda^{icp}}{2}] \\ &= log_2 e \times (ln[\sqrt{2}\lambda^{icp}] + 1) \end{aligned} \tag{3.45}$$

Substituting $\lambda^{icp}$ from equation 3.34 into equation 3.45 yields:

$$H_c^{icp} = log_2 e \times (ln[\sqrt{2}\kappa\sigma^{icp}] + 1) \tag{3.46}$$

The equation above indicates that the continuous entropy of the transformed image ($H_c^{icp}$) has a positive relationship with the standard deviation of the transformed image ($\sigma^{icp}$).

$$H_c^{icp} \propto^+ \sigma^{icp} \tag{3.47}$$

It is known that the continuous entropy of a distribution ($H_c^{icp}$) is not exactly the same as its discrete entropy ($H^{icp}$) [12].

$$H_c^{icp} = Lim_{\Delta \to 0}(H^{icp} + Log_2 \Delta) \tag{3.48}$$

where $H_c^{icp}$ and $H^{icp}$ are the continuous entropy and discrete entropy of the distribution, respectively, and $\Delta$ represents the widths of bins to which the range of discrete entropy has been divided. Equation 3.48 can be estimated by [12]:

$$H_c^{icp} \approx H^{icp} + Log_2 \Delta \tag{3.49}$$

which gives:

$$H^{icp} \propto^{+} H_c^{icp} \tag{3.50}$$

From relations 3.47 and 3.50, it is concluded that:

$$(H^{icp} \propto^{+} H_c^{icp}) \wedge (H_c^{icp} \propto^{+} \sigma^{icp}) \implies H^{icp} \propto^{+} \sigma^{icp} \tag{3.51}$$

The relation above is what we wanted to prove for the entropy and the standard deviation of $Img^{icp}$.

We have run the experiments for the set of 40 natural images and the empirical result matches equation 3.45. Figure 3.3 shows the entropy of the transformed images versus their standard deviations. The circles represent the empirical data and the solid curve plots equation 3.45. The RMSE for the curve fit is 4.85%. From figure 3.3, it is seen that there are a few outlier images. The reason that these images do not exactly follow equation 3.45 is that their probability density function of symbols based on $\delta$ do not exactly match the Laplace distribution. This is the case for images that have isolated regions with inconsistent gray levels compared to neighboring pixels .

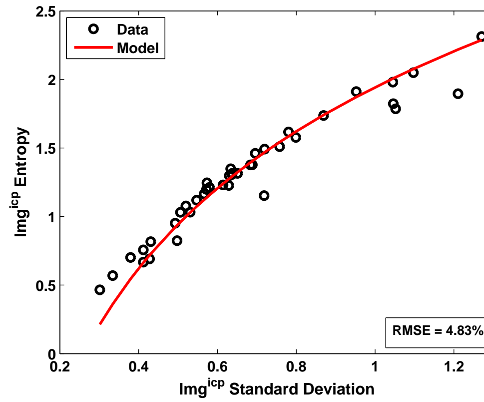



Figure 3.3: Entropy versus standard deviation for the transformed images

### 3.2.3 Ideal Reusability of an Image versus Standard Deviation of the Transformed Image

In this section, we show that the standard deviation of the transformed image (*i.e.* $\sigma^{icp}$) has a negative relationship with the ideal reusability of the original image (*i.e.* $R_{ideal}$):

$$\sigma^{icp} \propto^{-} R_{ideal}$$

The relation above means that as the standard deviation of the transformed image decreases, the number of windows in the original image whose central pixels carry new information in the interval $[-\delta, +\delta]$ increases. As mentioned before, the probability density function of $Img^{icp}$ can be modeled by a zero mean Laplace distribution as shown in the equation below.

$$P^{icp}(x) = \frac{1}{\sqrt{2}\kappa\sigma^{icp}} e^{\frac{-\sqrt{2}|x|}{\kappa\sigma^{icp}}}$$

where $\sigma^{icp}$ is the standard deviation of $Img^{icp}$.

Based on the definition of the ideal reusability of image in section 3.1.5, equation 3.27, we can write:

$$\begin{aligned} R_{ideal}(\delta) &= \int_{-\delta}^{+\delta} P^{icp}(x)dx \\ &= \frac{1}{\sqrt{2}\kappa\sigma^{icp}} \int_{-\delta}^{+\delta} e^{\frac{-\sqrt{2}|x|}{\kappa\sigma^{icp}}} dx \\ &= \frac{2}{\sqrt{2}\kappa\sigma^{icp}} \int_{0}^{+\delta} e^{\frac{-\sqrt{2}x}{\kappa\sigma^{icp}}} dx \\ &= 1 - e^{\frac{-\sqrt{2}\delta}{\kappa\sigma^{icp}}} \end{aligned} \quad (3.52)$$

We use equation 3.52 to show that increasing $\sigma^{icp}$ will cause $R_{ideal}(\delta)$ to decrease.

$$\begin{aligned} \sigma_1^{icp} \leq \sigma_2^{icp} \Longrightarrow e^{\frac{-\sqrt{2}\delta}{\kappa\sigma_1^{icp}}} &\leq e^{\frac{-\sqrt{2}\delta}{\kappa\sigma_2^{icp}}} \\ \Longrightarrow 1 - e^{\frac{-\sqrt{2}\delta}{\kappa\sigma_1^{icp}}} &\geq 1 - e^{\frac{-\sqrt{2}\delta}{\kappa\sigma_2^{icp}}} \\ \Longrightarrow R_{ideal1}(\delta) &\geq R_{ideal2}(\delta) \end{aligned} \quad (3.53)$$

The relation above shows that the standard deviation of the transformed image has a negative relationship with the ideal reusability of the original image:

$$\sigma^{icp} \propto^{-} R_{ideal} \quad (3.54)$$

More precisely:

$$\sigma_1^{icp} \leq \sigma_2^{icp} \implies \forall\delta,\ R_{ideal1}(\delta) \geq R_{ideal2}(\delta) \tag{3.55}$$

Figure 3.4 shows the ideal reusability of 40 natural images versus the standard deviations of the transformed images for different $\delta$. The circles represent the empirical data and the solid curves plot equation 3.52. Table 3.1 shows the RMSEs of the curve fits for the ideal reusability plots for $\delta$ being equal to 0, 1, 2 and 3 [4].

Table 3.1: RMSE for ideal reusability plots

| $\delta$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| RMSE (%) | 4.30 | 0.78 | 0.31 | 0.25 |

It is seen that as $\delta$ increases the ideal reusability of all images become closer to 1 (or 100%). This is due to the fact that the ideal reusability is the cumulative density function (CDF) of the transformed image; $\int_{-\delta}^{+\delta} P^{icp}(x)dx$.

### 3.2.4 Summary of Ideal Reusability versus Coding and Interpixel Redundancy

Through sections 3.2.1 to 3.2.3, we proved that the sum of coding and interpixel redundancy of an image has a positive relationship with the ideal reusability of the image ($(C_r + IP_r) \propto^+ R_{ideal}$). In order to prove this, we proved the followings:

1. Section 3.2.1: The sum of coding and interpixel redundancy of the original image has a negative relationship with the entropy of the transformed image:

$$(C_r + IP_r) = log_2(GL) - H^{icp} \implies (C_r + IP_r) \propto^- H^{icp} \tag{3.56}$$

2. Section 3.2.2: The entropy of the transformed image has a positive relationship with the standard deviation of the transformed image:

[4] Note that $R^d_{ideal}(\delta) = R^c_{idea}(\delta + 1)$ where $d$ and $c$ represent the discrete and continues functions, respectively.

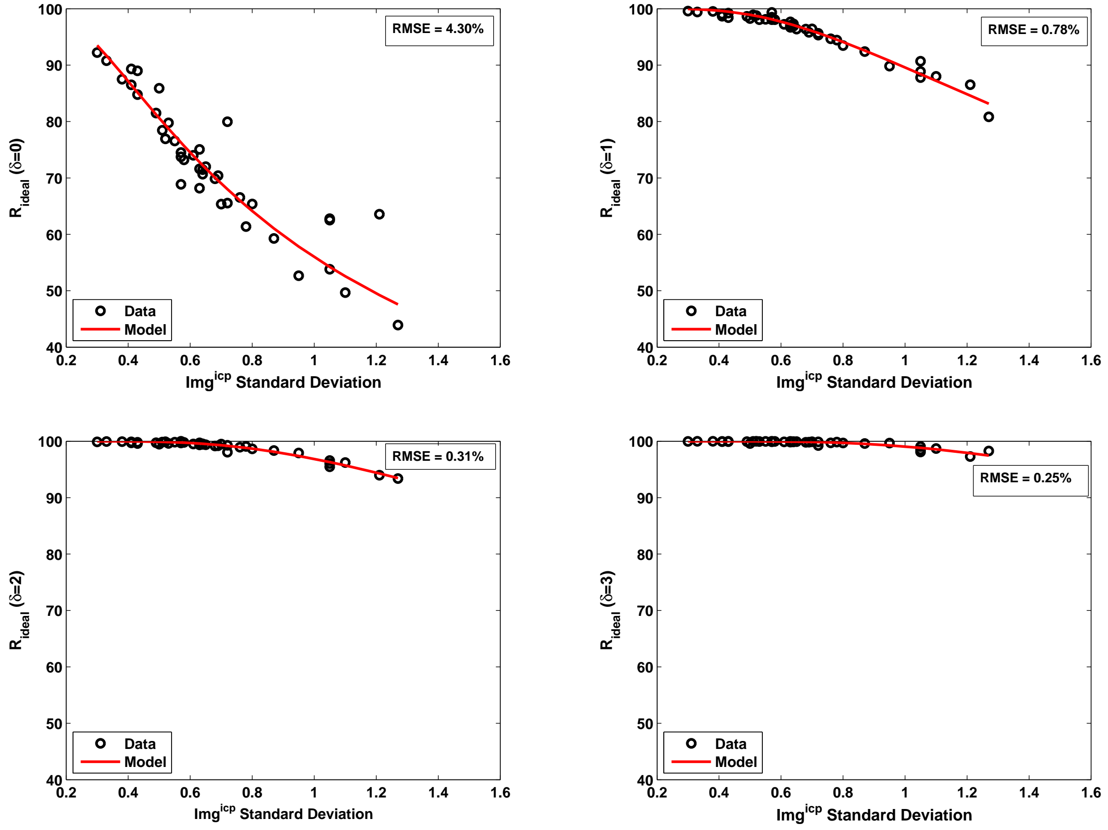



Figure 3.4: Ideal reusability of images versus the standard deviation of transformed images for different $\delta$ ($\delta = 0, 1, 2, 3$)

$$\begin{aligned}(H_c^{icp} &= log_2 e \times (ln[\sqrt{2}\kappa\sigma^{icp}] + 1) \wedge (H_c^{icp} \approx H^{icp} + Log_2\Delta) \\ &\implies H^{icp} \propto^{+} \sigma^{icp}\end{aligned} \tag{3.57}$$

3. Section 3.2.3: The standard deviation of the transformed image has a negative relationship with the ideal reusability of the image:

$$R_{ideal}(\delta) \;=\; 1 - e^{\frac{-\sqrt{2}\delta}{\kappa\sigma^{icp}}} \implies \sigma^{icp} \propto^{-} R_{ideal} \tag{3.58}$$

4. From (1) and (2), it is concluded that

$$((C_r + IP_r) \propto^{-} H^{icp}) \wedge (H^{icp} \propto^{+} \sigma^{icp}) \implies (C_r + IP_r) \propto^{-} \sigma^{icp} \tag{3.59}$$

5. From (3) and (4), it is concluded that

$$((C_r + IP_r) \propto^{-} \sigma^{icp}) \wedge (\sigma^{icp} \propto^{-} R_{ideal}) \implies (C_r + IP_r) \propto^{+} R_{ideal} \tag{3.60}$$

Relation 3.60 is the one that we wanted to prove in this section. We proved that there is a positive relationship between the coding and interpixel redundancy of an image and the ideal reusability of the image. This means that as the coding and interpixel redundancy of an image increase, the number of windows in the image whose central pixels carry new information in the interval $[-\delta, +\delta]$ will increase.

## 3.3 Hit Rate versus Coding and Interpixel Redundancy

In section 3.2, it was shown that the ideal reusability of an image depends positively on the coding and interpixel redundancy of the image.

$$C_r + IP_r \propto^+ R_{ideal} \tag{3.61}$$

In this section, we want to show that the ideal reusability of an image has a positive relationship with the hit rate of perfect window memoization for the image.

$$R_{ideal}(\delta) \propto^+ HR_{pc}(k) \tag{3.62}$$

In order to show the relation above, first we show that the ideal reusability of an image has a positive relationship with the reusability of the image:

$$R_{ideal}(\delta) \propto^+ R(\delta) \tag{3.63}$$

Then, we will show that the reusability of an image has a positive relationship with the hit rate of perfect window memoization for the image:

$$R(\delta) \propto^+ HR_{pc}(k) \tag{3.64}$$

By proving relations 3.63 and 3.64, relation 3.62 will be proved. Given that we have already proved relation 3.61 in the previous section, proving relation 3.62 will give:

$$C_r + IP_r \propto^+ HR_{pc}(k) \tag{3.65}$$

The relation above combined with relation 3.13 ($HR_{pc}(k) \propto^{+} Comp_r$) proved in section 3.1.2 will complete our ultimate goal, which is to prove:

$$C_r + IP_r \propto^{+} Compr_r \tag{3.66}$$

### 3.3.1 Reusability versus Ideal Reusability of an Image

In this section, our goal is to show that the ideal reusability of an image has a positive relationship with the reusability of the image

$$\forall\delta,\ R_{ideal1}(\delta) \leq R_{ideal2}(\delta) \implies \forall\delta,\ R_1(\delta) \leq R_2(\delta) \tag{3.67}$$

In section 3.1.4, we defined a measure for the reusability of image using the probability of symbols and the uniqueness probability of symbols in the image based on the new information carried by the central pixels of symbols ($\delta$).

$$R(\delta) = \frac{\int_{-\delta}^{+\delta} P^{icp}(x)dx}{\int_{-\delta}^{+\delta} P_u^{icp}(x)dx} - 1 \tag{3.68}$$

Also, in section 3.1.5, we defined the ideal reusability as:

$$R_{ideal}(\delta) = \int_{-\delta}^{+\delta} P^{icp}(x)dx \tag{3.69}$$

The reusability of image (equation 3.68), can be written based on the ideal reusability (equation 3.69) and the uniqueness probability of symbols:

$$R(\delta) = \frac{R_{ideal}(\delta)}{\int_{-\delta}^{+\delta} P_u^{icp}(x)dx} - 1 \tag{3.70}$$

By substituting the definition of $R(\delta)$ from equation 3.70 into equation 3.67, we can write:

$$(\forall\delta,\ R_{ideal1}(\delta) \leq R_{ideal2}(\delta)) \implies \left(\forall\delta,\ \frac{R_{ideal1}(\delta)}{\int_{-\delta}^{+\delta} P_{u1}^{icp}(x)dx} \leq \frac{R_{ideal2}(\delta)}{\int_{-\delta}^{+\delta} P_{u2}^{icp}(x)dx}\right) \tag{3.71}$$

In order to prove relation 3.71, we only need to show:

$$(\forall\delta,\ R_{ideal1}(\delta) \leq R_{ideal2}(\delta)) \implies \left(\forall\delta,\ \int_{-\delta}^{+\delta} P_{u1}^{icp}(x)dx \geq \int_{-\delta}^{+\delta} P_{u2}^{icp}(x)dx\right) \quad (3.72)$$

As we proved in section 3.2.3, the left hand side relation in 3.72 has been derived from:

$$\sigma_1^{icp} \geq \sigma_2^{icp} \quad (3.73)$$

where $\sigma^{icp}$ is the standard deviation of the transformed image, $Img^{icp}$. Thus, the relation that we want to prove (relation 3.72) can be written as:

$$\left(\sigma_1^{icp} \geq \sigma_2^{icp}\right) \implies \left(\forall\delta,\ \int_{-\delta}^{+\delta} P_{u1}^{icp}(x)dx \geq \int_{-\delta}^{+\delta} P_{u2}^{icp}(x)dx\right) \quad (3.74)$$

In relation 3.74, the left hand side and right hand side relations are related to two different distributions: the probability of symbols based on $\delta$ ($i.e.$ $P^{icp}$) and the uniqueness probability of symbols based on $\delta$ ($i.e.$ $P_u^{icp}$), respectively. Therefore, it is necessary to discover a relationship between the two distributions. In doing so, we investigate the boundaries of the two distributions where $P^{icp}$ and $P_u^{icp}$ are maximum and minimum. In our arguments in the remaining of this section, we only consider the case where $\delta \geq 0$. All arguments are also valid for $\delta \leq 0$ because the distributions of $P^{icp}$ and $P_u^{icp}$ are symmetric around 0.

We start with the left hand side of relation 3.74 to investigate that for $P^{icp}$, what can be concluded if $\sigma_1^{icp} \geq \sigma_2^{icp}$. As defined in section 3.1.6, $P^{icp}$ is modeled as:

$$P^{icp}(\delta) = \frac{1}{\sqrt{2}\kappa\sigma^{icp}} e^{\frac{-\sqrt{2}|\delta|}{\kappa\sigma^{icp}}} \quad (3.75)$$

where $\kappa$ is a constant number and $\sigma^{icp}$ is the standard deviation of the transformed image $Img^{icp}$. For $P^{icp}$, the boundary points are $P^{icp}(\delta = 0)$ and $P^{icp}(\delta = e)$. $e$, which we call *endpoint* is the point where $P^{icp}$ becomes very small [5] or $P^{icp}(e) = \epsilon > 0$. For $\delta = 0$, we will have:

[5] For images of $n$ windows, $\epsilon < \frac{1}{n}$.

$$P^{icp}(0) = \frac{1}{\sqrt{2}\kappa\sigma^{icp}} \tag{3.76}$$

From equation 3.76, we can conclude:

$$\sigma_1^{icp} \geq \sigma_2^{icp} \implies P_1^{icp}(0) \leq P_2^{icp}(0) \tag{3.77}$$

Figure 3.5 shows two hypothetical distributions of $P_1^{icp}$ and $P_2^{icp}$ with standard deviations of $\sigma_1^{icp}$ and $\sigma_2^{icp}$, respectively where $\sigma_1^{icp} \geq \sigma_2^{icp}$. We showed that $P_1^{icp}(0) \leq P_2^{icp}(0)$. We want to show that the endpoint of $P_1^{icp}$ is farther than that of $P_2^{icp}$ or $e_1 \geq e_2$. In doing so, first, we must show that $P_1^{icp}$ and $P_2^{icp}$ can have at most one intersection. This is verified by solving the equation $P_1^{icp}(x) = P_2^{icp}(x)$ where for generality, we consider the general form of a Laplace distribution $P^{icp}(\delta \geq 0) = ae^{-b\delta}$:

$$\begin{aligned} P_1^{icp}(\delta) &= P_2^{icp}(\delta) \\ a_1 e^{-b_1\delta} &= a_2 e^{-b_2\delta} \\ \delta &= \frac{Ln(\frac{a_1}{a_2})}{b_1 - b_2} \end{aligned} \tag{3.78}$$

The equation above indicates that two Laplace distributions can intersect each other at most once.

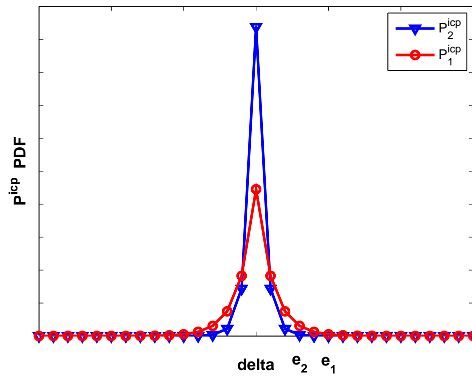



Figure 3.5: Probability density function of symbols based on $\delta$

Let's assume that $e_1 \geq e_2$ does not hold. If $e_1 < e_2$, because $P_1^{icp}(0) \leq P_2^{icp}(0)$ and the fact that $P_1^{icp}$ and $P_2^{icp}$ can have at most one intersection, then the area under curve

for $P_1^{icp}$ will be smaller than that of $P_2^{icp}$. However, this can not be the case because the the area under curve for both $P_1^{icp}$ and $P_2^{icp}$ are equal to 1. Therefore, $e_1 \geq e_2$ must hold. To summarize, in figure 3.5, we have:

$$\sigma_1^{icp} \geq \sigma_2^{icp} \implies (e_1 \geq e_2) \wedge (P_1^{icp}(0) \leq P_2^{icp}(0)) \tag{3.79}$$

By knowing the relationship between the endpoints of $P_1^{icp}$ and $P_2^{icp}$, we can derive a conclusion for endpoints of $P_{u1}^{icp}$ and $P_{u2}^{icp}$ because $P_u^{icp}$ stretches as far as $P^{icp}$ does. In other words, if after a certain $\delta$, $P^{icp}$ becomes very small (*i.e.* $P^{icp} = \epsilon$), $P_u^{icp}$ will also become very small (*i.e.* $P_u^{icp} = \epsilon$). Figure 3.6 shows two distributions of $P_{u1}^{icp}$ and $P_{u2}^{icp}$. Thus we can write:

$$e_1 \geq e_2 \implies e_1' \geq e_2' \tag{3.80}$$

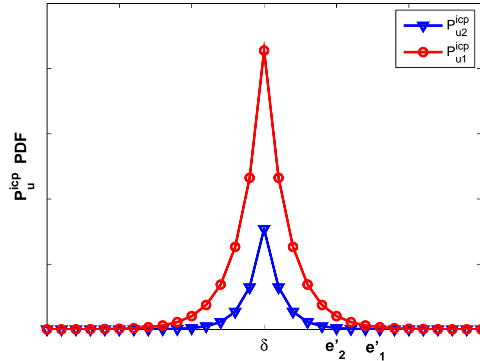



Figure 3.6: Uniqueness probability density function of symbols based on $\delta$

To further investigate the properties of $P_u^{icp}$, we examine the empirical data. It is observed that:

$$\sigma_1^{icp} \geq \sigma_2^{icp} \implies P_{u1}^{icp}(0) \geq P_{u2}^{icp}(0) \tag{3.81}$$

The relation above is true for our set of natural images with 95% accuracy and 8% error margin [6]. Relation 3.81 indicates that if the probability of symbols for $\delta = 0$ is

[6]The definition of accuracy and error margin is given in appendix A.

higher, the uniqueness probability of symbols for $\delta = 0$ will be lower. In other words, for $\delta = 0$, if $image_1$ has higher population of windows than $image_2$ then $image_1$ also has less number of unique symbols at $\delta = 0$ than $image_2$. This is an interesting phenomena. The higher coding and interpixel redundancy not only means that at $\delta = 0$ there will be high number of windows, it also means that the windows with $\delta = 0$ will belong to less number of symbols.

$P_{u1}^{icp}$ and $P_{u2}^{icp}$ are both Laplace functions and hence, as discussed before, they can intersect each other at most once. Therefore, because $P_{u1}^{icp}(0) \geq P_{u2}^{icp}(0)$ (relation 3.81) and $e_1' \geq e_2'$ (relation 3.80), it is concluded that the two curves ($P_{u1}^{icp}$ and $P_{u2}^{icp}$) do not intersect each other. As a result, we will have:

$$\int_{-\delta}^{+\delta} P_{u1}^{icp}(x)dx \geq \int_{-\delta}^{+\delta} P_{u2}^{icp}(x)dx \tag{3.82}$$

which means that relation 3.74 and consequently relation 3.67 holds.

By showing that the ideal reusability of an image has a positive relationship with the reusability of the image (relation 3.63: $R_{ideal} \propto^{+} R$), there is one more step left to complete our entire proof: the reusability of an image has a positive relationship with the hit rate of perfect window memoization for the image (relation 3.64).

### 3.3.2 Hit Rate of Perfect Window Memoization versus Reusability of an Image

In this section, we show the last step of our proof, which is the relationship between the reusability of an image and the hit rate of the perfect window memoization for the image. More precisely, we want to show:

$$(\forall \delta,\ R_1(\delta) \leq R_2(\delta)) \implies (\forall k,\ HR_1(k) \leq HR_2(k)) \tag{3.83}$$

where $k$ is the reuse table size and $\delta$ is the new information carried by the central pixels of the windows. We begin with the left hand side in the relation above. Substituting the reusability equation from 3.68 gives:

$$
\begin{aligned}
(\forall \delta,\ R_1(\delta) \leq R_2(\delta)) \quad &\Longrightarrow \quad \left( \frac{\int_{-\delta}^{+\delta} P_1^{icp}(x)dx}{\int_{-\delta}^{+\delta} P_{u1}^{icp}(x)dx} - 1 \leq \frac{\int_{-\delta}^{+\delta} P_2^{icp}(x)dx}{\int_{-\delta}^{+\delta} P_{u2}^{icp}(x)dx} - 1 \right) \\
&\Longrightarrow \quad \left( \frac{\int_{-\delta}^{+\delta} P_1^{icp}(x)dx}{\int_{-\delta}^{+\delta} P_{u1}^{icp}(x)dx} \leq \frac{\int_{-\delta}^{+\delta} P_2^{icp}(x)dx}{\int_{-\delta}^{+\delta} P_{u2}^{icp}(x)dx} \right)
\end{aligned} \tag{3.84}
$$

If $\delta$ goes to infinity, we will have:

$$
\begin{aligned}
\left( Lim_{\delta \to \infty}^{R_1(\delta)} \leq Lim_{\delta \to \infty}^{R_2(\delta)} \right) \quad &\Longrightarrow \quad \left( \frac{\int_{-\infty}^{+\infty} P_1^{icp}(x)dx}{\int_{-\infty}^{+\infty} P_{u1}^{icp}(x)dx} \leq \frac{\int_{-\infty}^{+\infty} P_2^{icp}(x)dx}{\int_{-\infty}^{+\infty} P_{u2}^{icp}(x)dx} \right) \\
&\Longrightarrow \quad \left( \frac{1}{\int_{-\infty}^{+\infty} P_{u1}^{icp}(x)dx} \leq \frac{1}{\int_{-\infty}^{+\infty} P_{u2}^{icp}(x)dx} \right) \\
&\Longrightarrow \quad \left( \int_{-\infty}^{+\infty} P_{u1}^{icp}(x)dx \geq \int_{-\infty}^{+\infty} P_{u2}^{icp}(x)dx \right) \\
&\Longrightarrow \quad \left( \frac{s_1}{n} \geq \frac{s_2}{n} \right) \\
&\Longrightarrow \quad (s_1 \geq s_2)
\end{aligned} \tag{3.85}
$$

where $s_1$ and $s_2$ are the total number of unique symbols in image1 and image2, respectively, and $n$ is the total number of windows in both images. The relation above indicates that an image with higher reusability when all windows are considered ($i.e.\ \delta \to \infty$) will have a lower number of unique symbols, which is what we expect. Lower number of unique symbols means that there is higher average number of windows per unique symbol.

Using relation 3.85, we simplify relation 3.83 to:

$$
(s_1 \geq s_2) \implies (\forall k,\ HR_1(k) \leq HR_2(k)) \tag{3.86}
$$

On the right hand side of relation 3.86, we have hit rate. Hit rate was defined in section 3.1.2 as (equation 3.8):

$$
HR(k) = \int_0^{k-1} P(x)dx - \frac{min(k,s)}{n} \tag{3.87}
$$

where, $k$ is the reuse table size, $s$ is the total number of unique symbols in the image, and $n$ is the total number of windows in the image. It is important to note that $P$

is different than $P^{icp}$. The former is the probability density function of symbols in the original image ($Img$), which are sorted in descending order and the latter is the probability density function of symbols in the transformed image $Img^{icp}$. $P^{icp}$ is also the probability density function of symbols in the original image based on the new information carried by the central pixels of the symbols $\delta$.

Substituting the hit rate equation into the right hand side of relation 3.86 and assuming that the total number of symbols in image is larger than the reuse table size ($i.e.$ $s > k$), which is a more probable case, we will have [7]:

$$(\forall k,\ HR_1(k) \leq HR_2(k)) \equiv \left(\int_0^{k-1} P_1(x)dx - \frac{k}{n} \leq \int_0^{k-1} P_2(x)dx - \frac{k}{n}\right) \tag{3.88}$$

Thus, the relation that we want to prove (relation 3.86) can be rewritten as:

$$(s_1 > s_2) \implies \left(\int_0^{k-1} P_1(x)dx - \frac{k}{n} \leq \int_0^{k-1} P_2(x)dx - \frac{k}{n}\right) \tag{3.89}$$

In order to prove relation 3.89, it is sufficient to prove:

$$(s_1 > s_2) \implies \left(\int_0^{k-1} P_1(x)dx \leq \int_0^{k-1} P_2(x)dx\right) \tag{3.90}$$

In order to prove relation 3.90, we need to have some information about the distribution of $P$. As mentioned earlier, $P$ is the probability density function (or histogram) of symbols in the original image ($Img$), which are sorted in descending order. $P$ has a peak at 0 and moving along x axis, it usually drops rapidly. $s$ is the total number of unique symbols in the image. Therefore, $s$ is the endpoint of $P$ curve where $P$ becomes very small ($i.e.$ $P(x = s) = \epsilon$) [8] where the $s - 1^{th}$ symbol is the least frequent symbol in the image. This means that the area under curve $P$ for the interval $[0, s-1]$ is approximately 1.

$$\int_0^{s-1} P(x)dx \approx 1 \tag{3.91}$$

[7] It can be shown that the discussion is also valid for less probable cases where $s \leq k$.

[8] For images of $n$ windows, $\epsilon < \frac{1}{n}$.

Figure 3.7 shows the probability density functions of symbols of two hypothetical images where $s_1$ and $s_2$ are the endpoints of $P$ curves of $Img_1$ and $Img_2$, respectively and $s_1 > s_2$. In order to prove relation 3.90, let's assume that the two distributions (*i.e.* $P_1$ and $P_2$) are exponential. We will verify this assumption later in this section. Because both distributions are exponential, with the same argument that we made for the Laplace distributions earlier, they intersect each other at most once at $i$. Furthermore, because the area under curves of both $P_1$ and $P_2$ are unity, from $s_1 > s_2$ it is concluded that $P_1(0) < P_2(0)$.

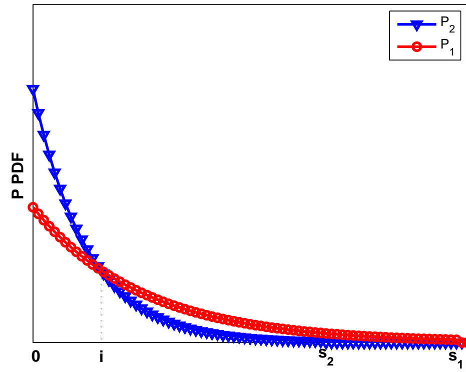



Figure 3.7: Probability density function of symbols

To show that relation 3.90 holds, we consider all possbile locations of $k-1$ where it can be in either of the three intervals:

- $k-1 \leq i$: In this case, as it is seen from figure 3.7, the area under curve $P_1$ is smaller than that of $P_2$ or $\int_0^{k-1} P_1(x)dx \leq \int_0^{k-1} P_2(x)dx$
- $i < k-1 \leq s_2$: The areas under the two curves in the interval $[0, s-1]$ are almost equal. So we can write:

$$\begin{aligned} \int_0^{s_1} P_1(x)dx &\approx \int_0^{s_2} P_2(x)dx \\ \int_0^{k-1} P_1(x)dx + \int_{k-1}^{s_1} P_1(x)dx &\approx \int_0^{k-1} P_2(x)dx + \int_{k-1}^{s_2} P_2(x)dx \end{aligned} \tag{3.92}$$

Because $k-1 > i$, in the interval $[k-1, +\infty)$ the $P_1$ curve overshadows the $P_2$

curve, which means:

$$\int_{k-1}^{s_1} P_1(x)dx \geq \int_{k-1}^{s_1} P_2(x)dx \tag{3.93}$$

From 3.92 and 3.93, we can conclude: $\int_0^{k-1} P_1(x)dx \leq \int_0^{k-1} P_2(x)dx$

- $k-1 > s_2$: In this case, the whole area under curve $P_2$ is covered, which is almost equal to 1. However, the the area under curve $P_1$ is partially covered, which means it is less than 1. Thus: $\int_0^{k-1} P_1(x)dx \leq \int_0^{k-1} P_2(x)dx$

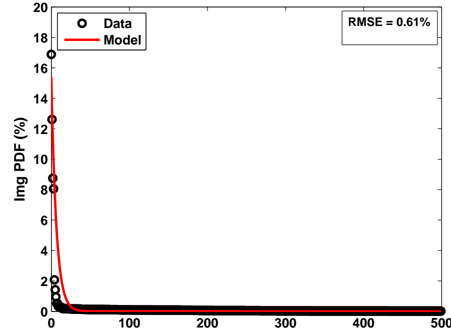
In order to verify that the probability density functions of image symbols are exponential with the area under curves of unity (*i.e.* $P(x) = ae^{-ax}$), we have performed the curve fitting for all 40 natural images. The experimental data shows that the exponential curve models the probability density functions of image symbols very accurately. The average RMSE for 40 images is 0.54%. Figure 3.8 depicts three sample images along with their probability density functions of symbols and the fitted curves. The RMSE for the three images are 0.61%, 0.64%, and 0.34%, respectively. It must be mentioned that the reason for very low RMSEs for the probability density functions of image symbols is the large number of data points with small populations. Although in figure 3.8 the probability density functions are shown up to $x = 500$, in reality the the probability density functions of symbols stretch well over 500. For example, the sample images shown in figure 3.8 have about 18,000, 44,000, and 173,000 unique symbols, respectively.

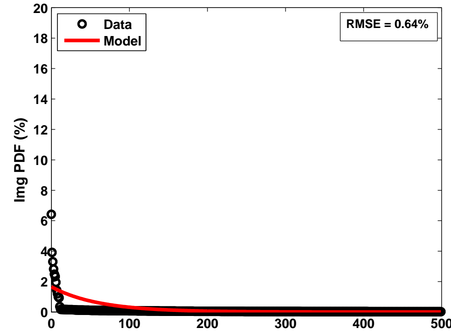
By showing that relation 3.90 holds, we have shown that the coding and interpixel redundancy of an image has a positive relationship with the hit rate of perfect window memoization for the image:

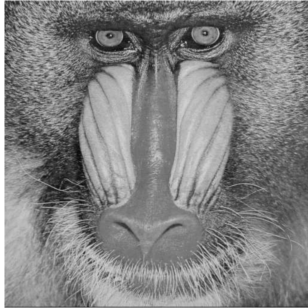
$$C_r + IP_r \propto^{+} HR(k) \tag{3.94}$$

## 3.4 Summary

The main goal of this chapter was to prove that the sum of the coding and interpixel redundancy of an image has a positive relationship with the computational redundancy of the image:





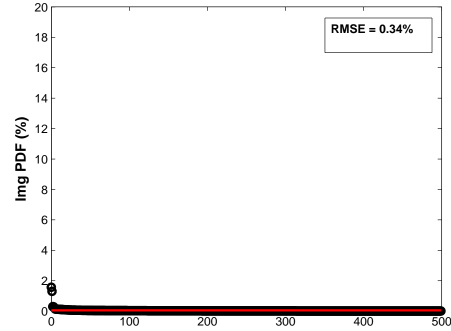



Figure 3.8: Left: original images. Right: the probability density functions of image symbols.

$$(C_r + IP_r) \propto^{+} Comp_r \tag{3.95}$$

In order to prove relation 3.95, several intermediate steps had to be proven. The followings list the intermediate steps that were taken in order to prove relation 3.95.

1. Section 3.2.1: The sum of coding and interpixel redundancy of the original image

has a negative relationship with the entropy of the transformed image:

$$(C_r + IP_r) \propto^- H^{icp} \tag{3.96}$$

2. Section 3.2.2: The entropy of the transformed image has a positive relationship with the standard deviation of the transformed image:

$$H^{icp} \propto^+ \sigma^{icp} \tag{3.97}$$

3. Section 3.2.3: The standard deviation of the transformed image has a negative relationship with the ideal reusability of the image:

$$\sigma^{icp} \propto^- R_{ideal}(\delta) \tag{3.98}$$

4. Section 3.3.1: The ideal reusability of an image has a positive relationship with the reusability of the image:

$$R_{ideal}(\delta) \propto^+ R(\delta) \tag{3.99}$$

5. Section 3.3.2: The reusability of an image has a positive relationship with the hit rate of perfect window memoization for the image:

$$R(\delta) \propto^+ HR(K) \tag{3.100}$$

6. Section 3.1.2: The hit rate of perfect window memoization for an image has a positive relationship with the computational redundancy of the image:

$$HR(K) \propto^+ Comp_r \tag{3.101}$$

In short, the whole chain that we proved in this chapter is:

$$(C_r + IP_r) \ \propto^- \ H^{icp} \ \propto^+ \ \sigma^{icp} \ \propto^- \ R_{ideal}(\delta) \ \propto^+ \ R(\delta) \ \propto^+ \ HR(k) \ \propto^+ \ Comp_r \tag{3.102}$$

which yields:

$$(C_r + IP_r) \propto^{+} Comp_r \tag{3.103}$$

This is a simple yet revealing relationship, which gives useful information on the potential performance gain obtained by window memoization for an image, only based on the coding and interpixel redundancy of the image.

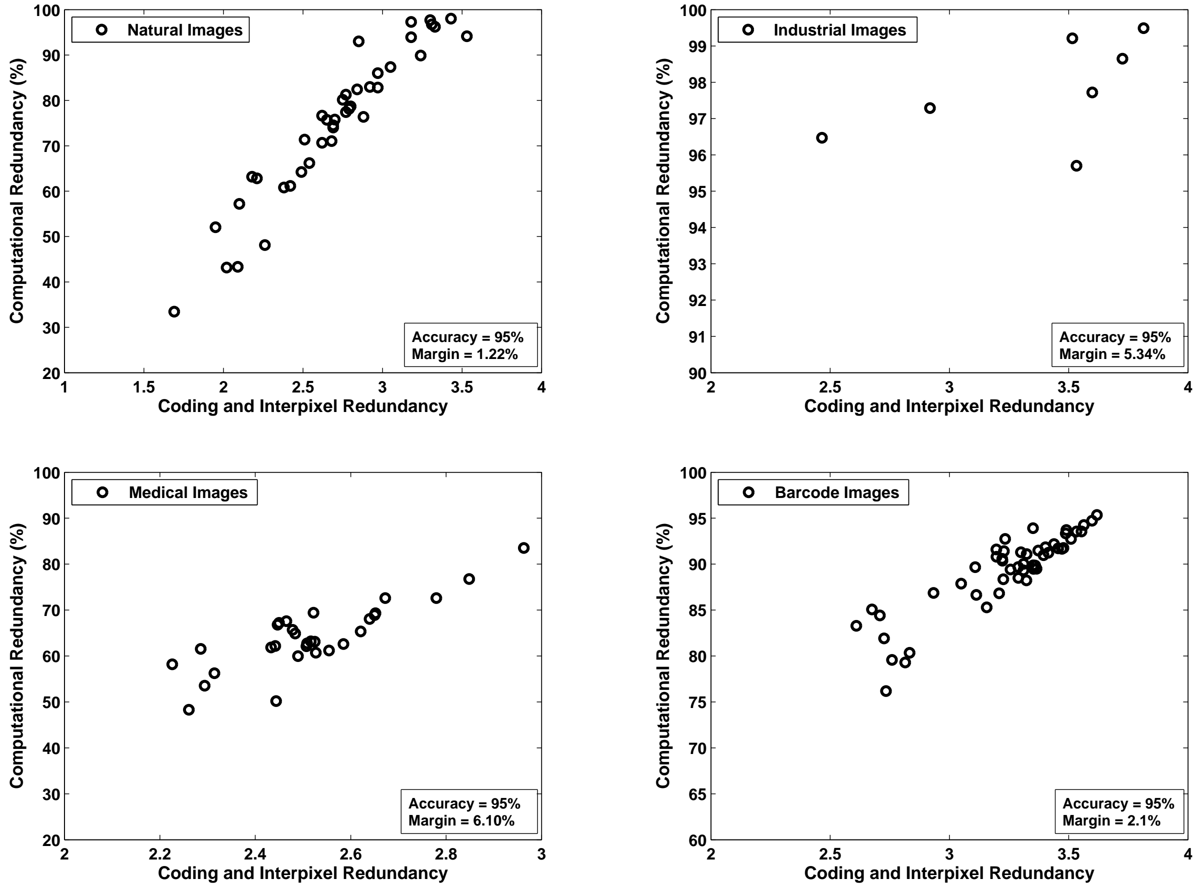



Figure 3.9: Computational redundancy versus coding and interpixel redundancy

In order to verify relation 3.103, we have run an experiment on four sets of images (natural, industrial, medical, and barcode). The results are shown in figure 3.9. The experimental data matches relation 3.103 with 95% of accuracy and error margins [9] in the range 1.22% to 6.10%. This, in fact, completes the goal of this chapter, which was to show that the sum of the coding and interpixel redundancy of an image has a positive relationship with the computational redundancy of the image (relation 3.103).

[9] The definition of accuracy and error margin is given in appendix A.

# Chapter 4

# Window Memoization in Software

In chapter 3, perfect window memoization was introduced as a high level model, which takes advantage of coding and interpixel redundancy in image data to improve the performance of local image processing algorithms. We also showed that the coding and interpixel redundancy of an image have a positive relationship with the computational redundancy of the image. In this chapter, we present the implementation of window memoization in software. We have applied the technique to several case studies implemented in *C* and run the experiments on three different processors. The typical speedups range from 1.2 to 7.9 with a maximum factor of 40. We also present a model that for a given processor and algorithm, predicts the speedup of all images in a data set with minimum required information. Finally, we show that the computational redundancy of an image has a positive relationship with the speedup obtained for the image by window memoization. This leads to the fact that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in software. This enables us to compare images based on the performance gain obtained by window memoization in software based on only the coding and interpixel redundancy of the images, without actually implementing window memoization.

For our experiments throughout this chapter, we have used the four sets of images given in section 2.6. As we will discuss later in this chapter, it is expected that for a given algorithm and image, different speedups will be achieved by window memoization on different processors. Therefore, in all our experiments, we use three different processors, which are listed in table 4.1.

Table 4.1: Processors used in experiments

| |
|---|
| 1. Processor 1: Intel(R) Xeon(TM), CPU: 3.20GHz, cache size: 1024 KB. |
| 2. Processor 2: Intel(R) XEON(TM), CPU: 1.80GHz, cache size: 512 KB. |
| 3. Processor 3: Embedded PowerPC 405 processor on Xilinx ML403 board, CPU: 300MHz, cache size: 16 KB. |

The selected set of processors covers a wide range of performance and cache size. It includes a low-end embedded processor (processor 3), a mid-range server (processor 2), and a high-end server (processor 1). Evaluating window memoization on different processors and demonstrating that it yields performance improvements on all processors validates the practicality and portability of the technique across different platforms.

The outline of this chapter is as follows. In section 4.1, we present the implementation of window memoization in software. In Section 4.2, window memoization in software is discussed in more detail. In section 4.3, we present a model for the overhead time incurred by the memoization mechanism. In section 4.4, we present a model for speedup and validate the model using empirical data. Section 4.5 presents empirical speedup results for all case study algorithms run on three processors using four sets of images. In section 4.6, we show mathematically and empirically that the coding and interpixel redundancy of an image have a positive relationship with the speedup of the image obtained by window memoization in software. Finally, section 4.7 gives the summary of the chapter.

## 4.1 Window Memoization Technique in Software

In this section, we present the architecture of window memoization in software. We also propose a model for actual speedups obtained by window memoization in software and discuss the factors that affect the speedup.

Figure 4.1 shows the flowchart of window memoization in software. The heart of window memoization is the memoization mechanism and a reuse table. Upon arrival of a window, if the memoization mechanism is able to find the matching symbol in the reuse table, a hit occurs. In this case, the response of the window is looked up from the reuse table and the actual computations (*i.e.* mask operations set) are skipped. Otherwise

(miss), the mask operations set is applied to the window to produce the response and the reuse table is updated with the symbol to which the window belongs and its response.

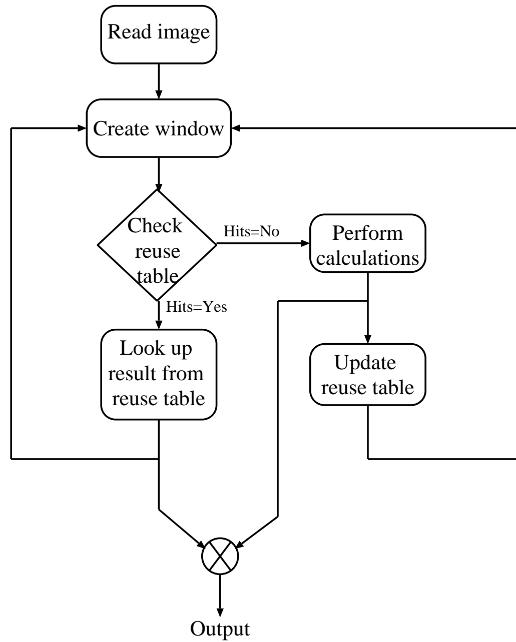



Figure 4.1: Flowchart of window memoization

Perfect window memoization uses a perfect cache as reuse table. As discussed in section 3.1.2, once an entry is inserted into a perfect cache, it is never evicted in the future. Therefore, if the size of the perfect cache is smaller than the total number of symbols in an image, those symbols are inserted into the perfect cache (or reuse table), which have higher probabilities of occurrence than others. This, of course, requires a priori knowledge about the probability density function of the symbols in the image, which is not available in real-world implementations. In order to implement window memoization in software, a different mapping scheme is required, which does not need the information about the probability density function of the symbols in the image. The mapping scheme will decide how to map the incoming windows into the reuse table and in case of a miss, which reuse table entry to evict.

Window memoization in software combines the concepts of cache and hash table. Similar to cache, window memoization uses a mapping scheme and an eviction policy to map incoming windows into the reuse table and, in case of a miss, to replace a symbol in the reuse table with the symbol of current window. Instead of using the symbol of

the incoming window directly to generate the reuse table address, similar to hash table, window memoization uses a hash function to convert the symbol of the incoming window into an address which is within the range of the reuse table size. In general, the symbol is a large number (*e.g.* 36-bit) and it is impossible to have such a large reuse table that allocates one unique location for each value of symbols. Thus, multiple symbols may be mapped into one location in the reuse table. As a result, the symbol itself must be stored in the reuse table for the purpose of future comparisons. This leads to a reuse table whose elements contain three fields: valid bit, symbol, and the result (or response). The valid bit indicates whether the data stored in this address is valid. The symbol field represents the stored symbol. Finally, the outcome of applying the mask operations set to the window is stored in the result field.

In the software implementation of window memoization, first, the incoming window is transformed into its corresponding symbol. The symbol then is mapped into an address in the reuse table by building a *key*. Afterward, the content of the reuse table at the location where the key points to is read and compared against symbol of the incoming window. If symbol of the incoming window matches the content of the reuse table at that particular location, a hit occurs; the response is read from the reuse table and the mask operations set for the incoming window is skipped. If an incoming window is mapped into a location in the reuse table, which is either empty or already occupied by a symbol to which the incoming window does not belong, a miss will occur. In this case, the eviction policy used by window memoization selects a symbol in the reuse table to be replaced by the symbol corresponding to the incoming window. Moreover, the mask operations set of the algorithm is performed on the incoming window and the reuse table is updated by its response.

In both cases of hit and miss, extra operations are introduced by window memoization. In case of a hit, the extra operations are: generating symbol, generating key, reading from reuse table and comparing the symbol of the incoming window to the content of the reuse table. In case of a miss, in addition to the extra operations similar to hit, there is another operation: writing the symbol of the incoming window and its response to the reuse table. As a result, the software implementation of perfect window memoization introduces an overhead in processing time, which we call *memoization overheard time*. The memoization

overhead time will affect the actual speedup obtained by window memoization in software. Thus, the overhead time should be taken into account when modeling the performance improvement of window memoization in software.

The following shows the flow of window memoization in software and the corresponding time for each step, based on whether a hit or a miss occurs:

- If a hit occurs:
  - build symbol and key based on the incoming window ($t_{key}$).
  - read the symbol from the reuse table, stored in the address that the key points to ($t_{read}$).
  - compare the symbol that just read from the reuse table with the symbol to which the incoming window belongs ($t_{compare}$).
  - the total time consumed by a window for which, a hit occurs, $T_{w_hit}$, will be:

$$T_{w_hit} = t_{key} + t_{read} + t_{compare} \tag{4.1}$$

- If a miss occurs:
  - build symbol and key based on the incoming window ($t_{key}$).
  - read the symbol from the reuse table, stored in the address that the key points to ($t_{read}$).
  - compare the symbol that just read from the reuse table with the symbol to which the incoming window belongs ($t_{compare}$).
  - perform the mask operations set on the incoming window ($t_{mask}$).
  - update the reuse table by writing the symbol of the incoming window and its response to the reuse table ($t_{write}$).
  - the total time consumed by a window for which, a miss occurs, $T_{w_miss}$, will be:

$$T_{w_miss} = t_{key} + t_{read} + t_{compare} + t_{mask} + t_{write} \tag{4.2}$$

Assume that $HR_{sw}$ is hit rate; the number of windows in an image that find a matching symbol in the reuse table divided by the total number of windows in the image. For those

windows that find a matching symbol in the reuse table, the required processing time is $HR_{sw} \times T_{w_hit}$ and similarly, for the remaining windows, the required processing time is $(1 - HR_{sw}) \times T_{w_miss}$. Note that both times ($T_{w_hit}$ and $T_{w_miss}$) are normalized with respect to the total number of windows in image. We can combine $T_{w_hit}$ and $T_{w_miss}$, using hit rate ($HR_{sw}$) as follows:

$$
\begin{aligned}
T_w &= HR_{sw} \times T_{w_hit} + (1 - HR_{sw}) \times T_{w_miss} \\
&= HR_{sw} \times (t_{key} + t_{read} + t_{compare}) + \\
&\quad (1 - HR_{sw}) \times (t_{key} + t_{read} + t_{compare} + t_{mask} + t_{write}) \\
&= (t_{key} + t_{read} + t_{compare}) + (1 - HR_{sw}) \times t_{write} + (1 - HR_{sw}) \times t_{mask}
\end{aligned}
\tag{4.3}
$$

It is seen from the equation above that except for the portion with $t_{mask}$, the rest are the overhead time required by the implementation of window memoization in software, or memoization overhead time (*i.e.* $t_{memo}$):

$$
t_{memo} = (t_{key} + t_{read} + t_{compare}) + (1 - HR_{sw}) \times t_{write} \tag{4.4}
$$

In writing the equation for $t_{memo}$ and consequently for speedup, for simplicity, first we assume that all memoization overhead time components (*i.e.* $t_{key}$, $t_{read}$, $t_{compare}$, and $t_{write}$) have a linear effect on $t_{memo}$ and thus on speedup. In section 4.3, we will revisit this assumption and show that in fact one of the memoization overhead time components (*i.e.* $t_{write}$) has a nonlinear (quadratic) effect on speedup. The speedup equation will also be modified accordingly. However, for our current purpose, which is to identify the parameters that influence the speedup, a linear relationship suffices. By substituting $t_{memo}$ from equation 4.4 in equation 4.3, we can write:

$$
T_w = t_{memo} + (1 - HR_{sw}) \times t_{mask} \tag{4.5}
$$

Similar to any performance improving technique, the speedup of window memoization applied to an algorithm is defined as the ratio of the time required to process the conventional algorithm ($T_c$) and the algorithm with window memoization, ($T_w$).

$$speedup = \frac{T_c}{T_w} \tag{4.6}$$

In equation 4.6, similar to $T_w$, $T_c$ is also normalized with respect to the total number of windows in image. Therefore, $T_c$ is in fact the time required by the mask operations set performed on one window or $t_{mask}$. Replacing $T_w$ and $T_c$ in equation 4.6 with equation 4.5 and $t_{mask}$, respectively, gives:

$$speedup = \frac{t_{mask}}{t_{memo} + (1 - HR_{sw}) \times t_{mask}} \tag{4.7}$$

where $HR_{sw}$ is hit rate, $t_{mask}$ is the time required to perform mask operations set, and $t_{memo}$ is the memoization overhead time.

Equation 4.7 indicates that the performance improvement in software obtained by window memoization (*i.e.* speedup) depends on three different parameters: $HR_{sw}$, $t_{mask}$, and $t_{memo}$. Hit rate ($HR_{sw}$) which itself depends on the memoization mechanism and the input image, has two effects on speedup. Increasing hit rate directly increases speedup and reduces $t_{memo}$, which also leads to a higher speedup. Therefore, hit rate is an important parameter in the performance improvement obtained by window memoization in software.

The time required for mask operations set ($t_{mask}$), which depends on the complexity of the algorithm under study, also has a positive relationship with speedup. The more complex the algorithm under study, the higher the achieved speedup will be. The memoization overhead time ($t_{memo}$) has a negative relationship with speedup. A higher $t_{memo}$ will cause speedup to decrease. The memoization overhead time results from multiple components: memory operations ($t_{read}$ and $t_{write}$), hit rate ($HR_{sw}$), comparison ($t_{compare}$), and key generation ($t_{key}$), which all depend only on the memoization mechanism, except $HR_{sw}$ which also depends on the input image.

Because $t_{memo} \neq 0$, equation 4.7 can be rewritten as:

$$speedup = \frac{\frac{t_{mask}}{t_{memo}}}{1 + (1 - HR_{sw}) \times \frac{t_{mask}}{t_{memo}}} \tag{4.8}$$

Equation 4.8 indicates that for a fixed hit rate, speedup depends on the ratio $\frac{t_{mask}}{t_{memo}}$. In other words, the more complex the algorithm under study (*i.e.* larger $t_{mask}$) and smaller memoization overhead time (*i.e.* smaller $t_{memo}$), a higher speedup will be achieved. As a result, first, for a fixed hit rate, depending on the complexity of the algorithm under study, different algorithms will yield different speedups. Second, in order to obtain a higher speedup, it is desired that $t_{memo}$ is minimized. Another important observation from this equation is that the ratio $\frac{t_{mask}}{t_{memo}}$ may be different for different processors. This is mainly due to the fact that for different processors, the ratio of arithmetic operations to memory operations (*i.e.* processor-memory performance gap) is different. The result is that for a particular algorithm and image, different speedups may be achieved on different processors.

## 4.2 Memoization Mechanism for Window Memoization in Software

In this section, we discuss the components of the memoization mechanism that affect speedup and present an optimized memoization mechanism, which takes these parameters into account to yield high speedups.

The memoization mechanism of window memoization in software consists of three steps: symbol generation, mapping scheme, and hash function. Each of these steps affect speedup by affecting the memoization overhead time ($t_{memo}$) and hit rate ($HR_{sw}$) in the equation of speedup (equation 4.7). $t_{key}$ in $t_{memo}$ is affected by both the symbol generation and hash function. Symbol generation converts each incoming window into a single number (symbol) and the hash function converts the symbol into an address. The memory operations ($t_{read}$, $t_{write}$) and $t_{compare}$ in $t_{memo}$ are all affected by the mapping scheme, which decides which symbol to evict when a miss occurs. $HR_{sw}$ is affected by both the hash function and mapping scheme.

When a window is received by window memoization, the first step is to produce the

corresponding symbol of the window (symbol generation reflected in $t_{key}$), by choosing the $d$ most significant bits from each pixel in the window (*i.e.* tolerant memoization). To achieve high speedups (low $t_{key}$), generating the symbol from the incoming window must be fast. Once the symbol is generated, a *mapping* scheme must map the symbol into the reuse table. The mapping scheme will have multiple effects on speedup through $t_{memo}$. First, the way that symbols are mapped to the reuse table will determine the overhead time caused by memory operations, which are represented by $t_{read}$ and $t_{write}$ in $t_{memo}$. Second, the number of comparisons that the mapping scheme requires (*i.e.* $t_{compare}$) will affect $t_{memo}$. Third, depending on the mapping scheme, a hash function might be needed for generating an address (or key) from a symbol, which will also affect $t_{memo}$ by affecting $t_{key}$. In addition, the mapping scheme and hash function will also have a direct effect on hit rate, which affects speedup. Thus, the choice of a method which maps symbols to the reuse table quickly (*i.e.* low $t_{key}$, $t_{read}$ and $t_{write}$ and thus low $t_{memo}$) and at the same time yields high hit rates is crucial. In table 4.2, the list of parameters that affect speedup along with their causing roots is shown.

Table 4.2: Design decisions for memoization mechanism in software

| Parameter/Affected by | Symbol Generation | Mapping Scheme | Hash Function |
|---|---|---|---|
| $HR_{sw}$ | | √ | √ |
| $t_{key}$ | √ | | √ |
| $t_{read}$ | | √ | |
| $t_{write}$ | | √ | |
| $t_{compare}$ | | √ | |

In the following sections, we will discuss the design decisions that must be made based on table 4.2, in order to develop an optimized memoization mechanism for window memoization in software.

### 4.2.1 Generating Symbols

In section 3.1.3, we defined a condition, based on which a window *win* belongs to a symbol *sym*:

$$\forall pix \in win, \forall pix' \in sym,\ MSB(d, pix) = pix' \implies win \in sym \tag{4.9}$$

where $MSB(d, pix)$ represents the $d$ most significant bits of the pixel $pix$ in the window $win$. A symbol represents the $d$ most significant bits of each pixel in the window. As discussed in section 3.1.3, decreasing $d$ causes the response of a window to be assigned to windows that are similar but not necessarily identical to the window. This leads to higher hit rates and thus higher speedups, with a small cost in the accuracy of results.

Window memoization in software extracts the $d$ most significant bits of each pixel of a window to generate the corresponding symbol. The symbol is a number, which consists of chunks of the $d$ most significant bits from pixels of the window. In general, it does not matter in which order the the $d$ most significant bits of each pixel appear in the symbol. However, to reduce the time required to generate symbols, a certain order is required, which will be explained shortly. To generate a symbol for each window of $m \times m$ pixels in the image, the $d$ most significant bits of each pixel in the window are shifted and $OR$ed with each other such that they build a $d \times m^2$ bit number ($d$ bits per pixel), as listed in table 4.3.

Table 4.3: Generating symbol

1. Input image $Img$
2. For each window $win$ of $m \times m$ pixels in the image, perform:
    a. Initialize symbol: sym=0.
    b. For each pixel $pix$ in the window $win$, starting from the leftmost column, perform:
        $sym = (sym \ll d\ )\ OR\ (pix \gg 8 - d)$

In the algorithm in table 4.3, '$\ll$' represents a left shift operator and '$OR$' is a logical *or* operator. The right hand side of $OR$ ($i.e.\ pix \gg 8-d$) shows how the $d$ most significant bits of each pixel is selected.

As mentioned before, to gain better performance for window memoization in software, the memoization overhead time ($t_{memo}$ and thus $t_{key}$) must be minimized, which means that the symbol of each window must be generated quickly. In doing so, we benefit from the overlap between the neighboring windows to build the symbol incrementally as the mask moves across the image. The pixels of each window are used to generate its symbol

in a left-to-right order. For the first windows in each row, the algorithm is the same as the one in table 4.3. For the rest of windows, the pixels that belong to the leftmost column in the window are removed from the symbol and instead, the pixels that belong to the new rightmost column are added to the symbol. The algorithm given in table 4.3 requires $m^2$ shift and $OR$ operations in order to build symbols for each window. By taking advantage of overlapping windows, for each window, only $m$ shift and $OR$ operations are needed in order to build symbol (table 4.4) [1].

Table 4.4: Generating symbol using the overlapping windows

---

1. input image $Img$
2. For each window $win$ of $m \times m$ pixels in the image perform:
  a. If $win$ is the first window in the row, then:
    - Initialize symbol: sym=0.
    - For each pixel $pix$ in $win$, starting from the leftmost column,
      perform: $sym = (sym \ll d\ )\ OR\ (pix \gg 8 - d)$.
  b. Else:
    - For each pixel $pix$ in the rightmost column of $win$, perform (insert the rightmost column and shift out the leftmost column):
      $sym = (sym \ll d\ )\ OR\ (pix \gg 8 - d)$.

---

### 4.2.2 Mapping Scheme for Memoization Mechanism

Once the symbol of an incoming window is generated, a mapping scheme is required to map the symbol into the reuse table. Once again, a mapping scheme must be chosen such that the overhead time (*i.e.* $t_{read}$, $t_{write}$, and $t_{compare}$) is small. Different mapping algorithms have been developed and used in processors cache hierarchy design including *direct-mapped*, *fully associative*, and *set-associative* [49]. In the following, each of the mapping schemes is reviewed. We also discuss that direct-mapped mapping scheme is suitable for window memoization in software.

Direct-mapped method which performs a many-to-one mapping, always maps a particular symbol to a single location. *Fully associative* method performs an any-to-any

[1] This method of fast symbol generation, which benefits from overlapping windows in the image, is similar to Huang's method for fast median filter [27], as discussed in chapter 2, section 2.5.5.

mapping between the incoming windows and the addresses of the reuse table. In this method, a symbol can reside anywhere in the reuse table and thus, in order to find a match between an incoming window and the stored symbols, the whole reuse table must be searched. *Set associative* scheme is a combination of direct-mapped and fully associative methods. In this method, first, a symbol is mapped to a range of addresses in the reuse table, similar to direct-mapped method. Afterward, the symbol can be stored in any address, which is in that range, similar to fully associative method.

In set associative and fully associative methods, a heuristic must be used in order to decide where to store the symbol of the incoming window and which address to evict. Three commonly used methods in processor cache hierarchy design are: *first in first out* (FIFO), *least recently used* (LRU), and *random*. FIFO evicts the oldest entry in the reuse table. LRU evicts the entry that has been referenced the least. Finally, random eviction method chooses randomly which entry to evict.

Fully associative and set associative methods require many memory and comparison operations (*i.e.* high $t_{read}$, $t_{write}$, and $t_{compare}$). In hardware, because these operations can be performed in parallel, the two methods are used extensively. In contrast, the serial nature of software leads to a very poor performance for window memoization that uses fully associative or set associative methods as the mapping scheme. In other words, these two schemes yield very high $t_{memo}$ such that in equation 4.7, for any normal range of values (*i.e.* $t_{mask}$ and $HR_{sw}$), it almost becomes infeasible to achieve speedups higher than 1. As a result, the software implementation of window memoization uses a direct-mapped mapping scheme for the memoization mechanism.

### 4.2.3 Hash Function for Direct-Mapped Mapping Scheme

The memoization mechanism of window memoization in software requires a hash function to convert each symbol into a smaller number *hash_key*, which is in the range of the reuse table size. This will cause more than one symbol to be mapped to each location in the reuse table leading to *collisions*. In conventional hashing schemes in software, one generally solves the collisions by two major methods: *open addressing* and *chaining*. Both methods, however, impose an extra overhead time to the memoization mechanism (higher $t_{read}$, $t_{write}$, and $t_{compare}$). Moreover, in window memoization, when a collision occurs,

the conventional mask operations set can be performed on the window to obtain the response. This is in contrast to conventional hashing methods where the goal is to find a target in the hash table and there is no alternate in case of a collision. As a result, it is more beneficiary for window memoization to consider collision as a miss and hence, to perform the mask operations set if a collision occurs, rather than employing conventional collision resolution methods.

There are two commonly used hash functions in the literature: division method and multiplication method [10]. The division method maps a symbol $sym$ into one of the reuse table slots by taking the remainder of the symbol by the reuse table size ($RT_{size}$):

$$\begin{aligned} hash_key &= h(sym) \\ &= mod(symbol, RT_{size}) \end{aligned} \tag{4.10}$$

In order to obtain a better result in terms of having hash-keys evenly distributed over the hash table, $RT_{size}$ must not be a power of 2. A good choice for $RT_{size}$ is a prime number, which is not too close to a power of 2.

An alternative scheme to the division method is the multiplication method. In this method, first, the symbol is multiplied by a number, $A$, in the range $(0, 1)$ and the fractional part is extracted. Afterward, the result is multiplied by $RT_{size}$ and the floor of the final result is the outcome of the hash function, which is hash-key:

$$hash_key = floor[RT_{size} \times mod((sym \times A), 1)] \tag{4.11}$$

In this method, there is no constraint on the value of $RT_{size}$. However, it is usually chosen to be a power of 2 so that the multiplication becomes a shift operation. In order to use integer operations instead of floating point, $A$ is defined as $A = \frac{s}{2^b}$ where $s$ is an integer in the range $(0, 2^b)$ and $b$ is the number of bits of symbol $sym$. First, $sym$ is multiplied by $s = A \times 2^b$. This gives a 2b-bit number: $2^b.r_1 + r_0$. The most significant portion of the number, $r_1$, is the result of multiplying $A$ by $2^b$ and therefore, it is ignored. $r_0$ is the fractional part of $sym \times A$, which has been shifted to the left by $b$ bits. In order

to find the result of $r_0 \times RT_{size}$, where $RT_{size} = 2^z$, we need to pick $z$ most significant bits of $r_0$.

The only parameter that must be chosen for this method is the constant $A$. Depending on the characteristics of data being hashed, an optimal value can be chosen for $A$. Knuth [10] has reported that for many applications, the golden ratio $A = \frac{\sqrt{5}-1}{2} \approx 0.6118$ is an optimal choice, which we have used for our experiments.

As shown in table 4.2, in order to compare the two hash functions (*i.e.* division and multiplication), both $HR_{sw}$ and $t_{key}$ must be compared. $HR_{sw}$ and $t_{key}$ have a positive and negative relationships with speedup, respectively (equation 4.12). Therefore, a hash function with higher $HR_{sw}$ and lower $t_{key}$ will yield higher speedups.

$$speedup = \frac{t_{mask}}{t_{memo} + (1 - HR_{sw}) \times t_{mask}} \tag{4.12}$$

where

$$t_{memo} = t_{key} + t_{read} + t_{compare} + (1 - HR_{sw}) \times t_{write} \tag{4.13}$$

To evaluate the efficiency of multiplication and division methods for hash function, we have run an experiment on the three different processors using the set of natural images. For each reuse table size in the range $1K$ to $256K$ (*i.e.* $1K$, $2K$, $4K$, $8K$, $16K$, $32K$, $64K$, $128K$, $256K$), we measure $t_{key}$ and average $HR_{sw}$ over the set of images. To measure $t_{key}$, we use the C *clock()* function. For the given input images (*e.g.* natural images), $HR_{sw}$ and $t_{key}$ depend only on the reuse table size and the processor on which the experiments are run, respectively. For each hash function, figure 4.2 left shows $t_{key}$ versus different processors and figure 4.2 right shows $HR_{sw}$ versus the reuse table size.

As figure 4.2 left shows, in all cases, the multiplication method gives a smaller $t_{key}$ than the division method. This is mainly due to the fact that the multiplication method uses inexpensive operations (*i.e.* shift) while the division method uses an expensive operation (*i.e.* mod function). It is seen from figure 4.2 right that in the majority of cases, the multiplication method also gives higher hit rates than the division method. For a given processor and the reuse table size, if the multiplication method gives both higher hit

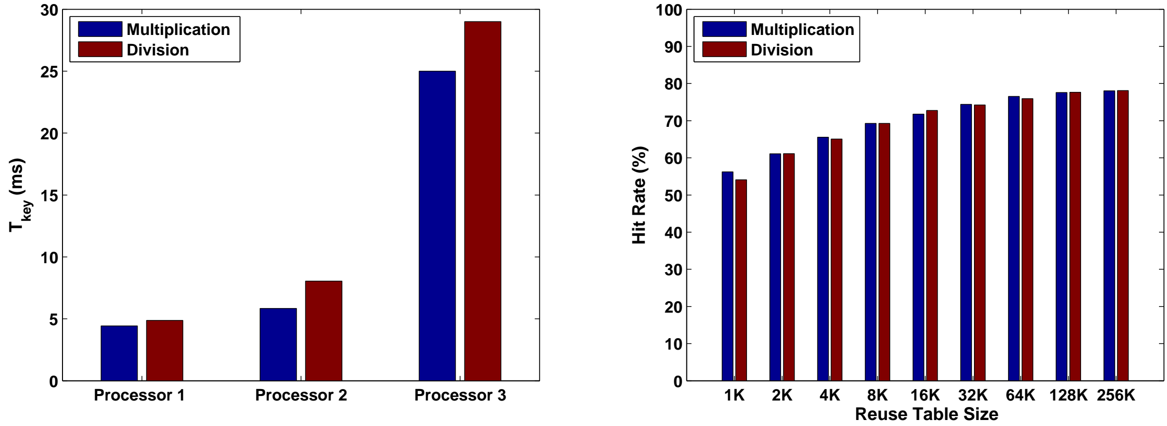



Figure 4.2: Left: $t_{key}$ for all three processors. Right: hit rate versus the reuse table size.

rate and lower $t_{key}$, which is true in most cases, then the speedup for the multiplication method will be higher than that of the division method. The only case (the worst case) that speedup for the multiplication method might be lower than that of the division method is for $16K$ entries reuse table where the hit rate of the multiplication method is 0.98% lower than that of the division method.

To decide between the multiplication and division method, we perform the following experiment. We calculate speedup using $t_{key}$ and $HR_{sw}$ of each hash function. We use typical values for the remaining parameters in the equation of speedup (equation 4.12). The difference in speedups of the multiplication and division methods (*i.e.* $speedup_{mul} - speedup_{div}$) is shown in figure 4.3. As it is seen, there is only one noticeable scenario (*i.e.* $16K$ entries reuse table) where the multiplication method gives a lower speedup than the division method (*i.e.* 6%, 5%, and 12% lower). In all other cases, either the speedup of the multiplication method is higher than that of the division method or the difference is negligible (*i.e.* less than 1%). Based on this observation and the fact that in most cases, a $16K$ entries is not the reuse table size that gives the maximum speedups (sections 4.4.1 and 4.4.2), we have chosen the multiplication method as the hash function for the software implementation of window memoization.

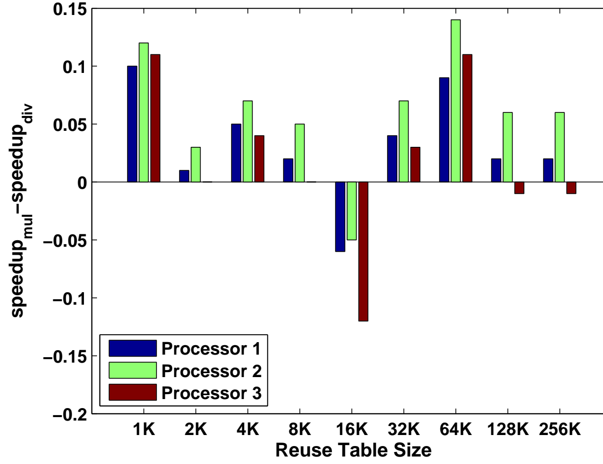



Figure 4.3: The difference in speedups of the multiplication and division methods

### 4.2.4 Tolerant Memoization in Software

In this section, we explore the possibility of assigning similar but not necessarily identical windows to one symbol, in order to achieve high hit rates and hence, high speedups. We introduced the concept of tolerant memoization in chapter 3, section 3.1.3 where in assigning windows to symbols, the $d$ most significant bits of each pixel are used.

We perform an experiment, using our memoization mechanism, which was presented in the previous sections, to pick an optimal $d$ that gives high hit rates with small inaccuracy in results. For our experiments, we use an ideal algorithm which outputs the central pixel of the input $3 \times 3$ window as its response. We use different values for $d$ from 1 to 8. As the input images, we use the set of natural images. The accuracy of the results is calculated as SNR, given in equation 3.16, presented in section 3.1.3. In calculating SNR, we replace an infinite SNR with 100, in order to calculate the average of SNRs of all images.

Figure 4.4 shows the average hit rate and SNR of the result for each value of $d$. It is seen that as $d$ decreases, hit rate increases and at the same time, SNR decreases. The error in an image with SNR of $30dB$ is nearly indistinguishable by the observer [1]. Therefore, we pick $d$ to be 4 because it gives an average SNR of $29.68dB$, which is slightly below the value $30dB$. Reducing $d$ from 8 to 4 increases the average hit rate from 10% to 66%. For our experiments in the remaining of this chapter, we will choose $d$ to be 4.

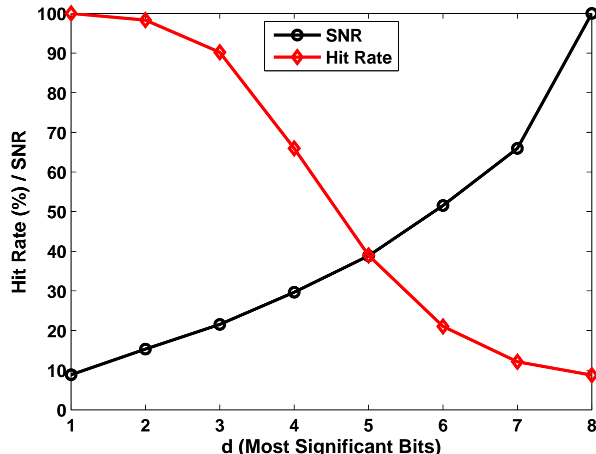



Figure 4.4: Average hit rate and SNR versus the number of the most significant bits used for assigning windows to symbols. Infinite SNRs have been replaced by SNR of 100.

### 4.2.5 Summary of Design Decisions for Memoization Mechanism

In this section, we discussed several parameters that affect the speedup achieved by window memoization in software. It was shown that each parameter depends on a design decision of developing the memoization mechanism. A fast symbol generation mechanism was presented which yields a small $t_{key}$. As the mapping scheme, the direct-mapped method was chosen because it produces small overhead time for memory operations and comparisons (*i.e.* $t_{read}$, $t_{write}$, and $t_{compare}$). The multiplication method was selected as the hash function for direct-mapped mapping, which yields high hit rate and low $t_{key}$. Finally, to achieve high hit rates, 4 bits of each pixel in a window are considered to determine whether the window belongs to a symbol.

## 4.3 Revisiting the Memoization Overhead Time Equation

In section 4.1, a theoretical model for calculating speedup obtained by window memoization in software was presented (equation 4.14).

$$speedup = \frac{t_{mask}}{t_{memo} + (1 - HR_{sw}) \times t_{mask}} \tag{4.14}$$

Equation 4.14 indicates that for a given algorithm and image, in order to calculate speedup, the memoization overhead time ($t_{memo}$) must be known. In the previous sections, we discussed that $t_{memo}$ depends on different parameters: $t_{key}$, $t_{read}$, $t_{write}$, $t_{compare}$, and $HR_{sw}$. Based on these parameters, we designed an optimized memoization mechanism that uses the multiplication method for hash function in a direct-mapped style for mapping the symbols of incoming windows into the reuse table. We also proposed a model for the memoization overhead time ($t_{memo}$) based on the assumption that hit rate ($HR_{sw}$) has a linear relationship with $t_{memo}$.

In this section, using empirical data, first we evaluate the accuracy of the linear model for the memoization overhead time proposed in section 4.1. In order to achieve higher accuracy of curve fits of the model and empirical data, we modify $t_{memo}$ to a quadratic equation . Afterward, we simplify the quadratic model such that by using the empirical data for only two extreme cases of images in a data set, $t_{memo}$ can be predicted for all images in the data set. Figure 4.8 summarizes the comparison of the three models (*i.e.* linear, quadratic, and simplified quadratic). In section 4.4, we use the simplified quadratic model of $t_{memo}$ to modify the speedup equation (*i.e.* equation 4.14) accordingly. We then use the modified speedup equation to pick the optimal reuse table sizes to achieve maximum speedup for each case study algorithm.

### 4.3.1 Memoization Overhead Time: A Linear Model

In the previous section, it was assumed that there is a linear relationship between $t_{memo}$ and its constituting elements:

$$t_{memo} = t_{key} + t_{read} + t_{compare} + (1 - HR_{sw}) \times t_{write} \tag{4.15}$$

We study $t_{memo}$ for a given processor and the reuse table size. By using a particular processor, $t_{key}$ and $t_{compare}$ become constant values. Fixing the the reuse table size makes the memory operations (*i.e.* $t_{read}$ and $t_{write}$) constant values as well. As a result, $t_{memo}$ becomes dependent only on hit rate ($HR_{sw}$).

To verify the linear equation for $t_{memo}$, we use empirical data. In order to measure $t_{memo}$, we run window memoization in software without applying any mask operations set. In other words, the code only includes the window memoization mechanism where the windows' size is $3 \times 3$ pixels. We run the experiments on three different processors given in table 4.1 to calculate $t_{memo}$ for each image in our set of 40 natural images. In addition, we use different reuse table sizes, ranging from $1K$ entries up to $256K$ entries (*i.e.* $1K$, $2K$, $4K$, $8K$, $16K$, $32K$, $64K$, $128K$, $256K$). For each processor, this yields 9 sets of data where each set consists of $t_{memo}$ data for 40 images. Table 4.5 shows the RMSEs of the measured $t_{memo}$ against the linear model for each processor and each reuse table size.

Table 4.5: RMSEs (%) for linear curve fit for $t_{memo}$ for different $RT$ sizes on three processors

| Processor/$RT_{size}$ | $1K$ | $2K$ | $4K$ | $8K$ | $16K$ | $32K$ | $64K$ | $128K$ | $256K$ | Average |
|---|---|---|---|---|---|---|---|---|---|---|
| Processor 1 | 5.02 | 6.10 | 4.70 | 4.45 | 4.11 | 4.24 | 4.10 | 3.98 | 4.70 | 4.60 |
| Processor 2 | 3.86 | 4.21 | 4.34 | 4.43 | 3.82 | 4.61 | 4.85 | 5.01 | 5.13 | 4.47 |
| Processor 3 | 0.00 | 1.28 | 2.01 | 2.70 | 2.98 | 3.39 | 3.65 | 3.41 | 3.27 | 2.52 |

Figure 4.5 shows the experimental data and linear model for $t_{memo}$ versus $HR_{sw}$ for a particular reuse table size ($16K$ entries) for all three processors. Although the RMSEs given in table 4.5 seem reasonable, observing the $t_{memo}$ versus $HR_{sw}$ curves (figure 4.5) reveals a non-linear effect in the $t_{memo}$ and $HR_{sw}$ relationship. This is more visible especially in the data sets of processor 1 and 2.

In order to investigate the nonlinear behavior of the empirical data for $t_{memo}$ versus $HR_{sw}$, we must further study the constituting elements of $t_{memo}$. As it is shown in equation 4.15, $t_{memo}$ has the following components: $t_{key}$, $t_{read}$, $t_{compare}$, and $t_{write}$. It is seen that $t_{key}$, $t_{read}$ and $t_{compare}$ are all independent of hit rate $HR_{sw}$. However, $t_{write}$ is the only parameter that can be affected by hit rate. We hypothesize that the nonlinear behavior observed in the empirical data of $t_{memo}$ versus hit rate is due to the write operation, which is reflected as $t_{write}$ in $t_{memo}$. When a miss occurs in window memoization, the mask operations set is applied to the current window and the symbol of the window along with its response are written to the reuse table. This causes the

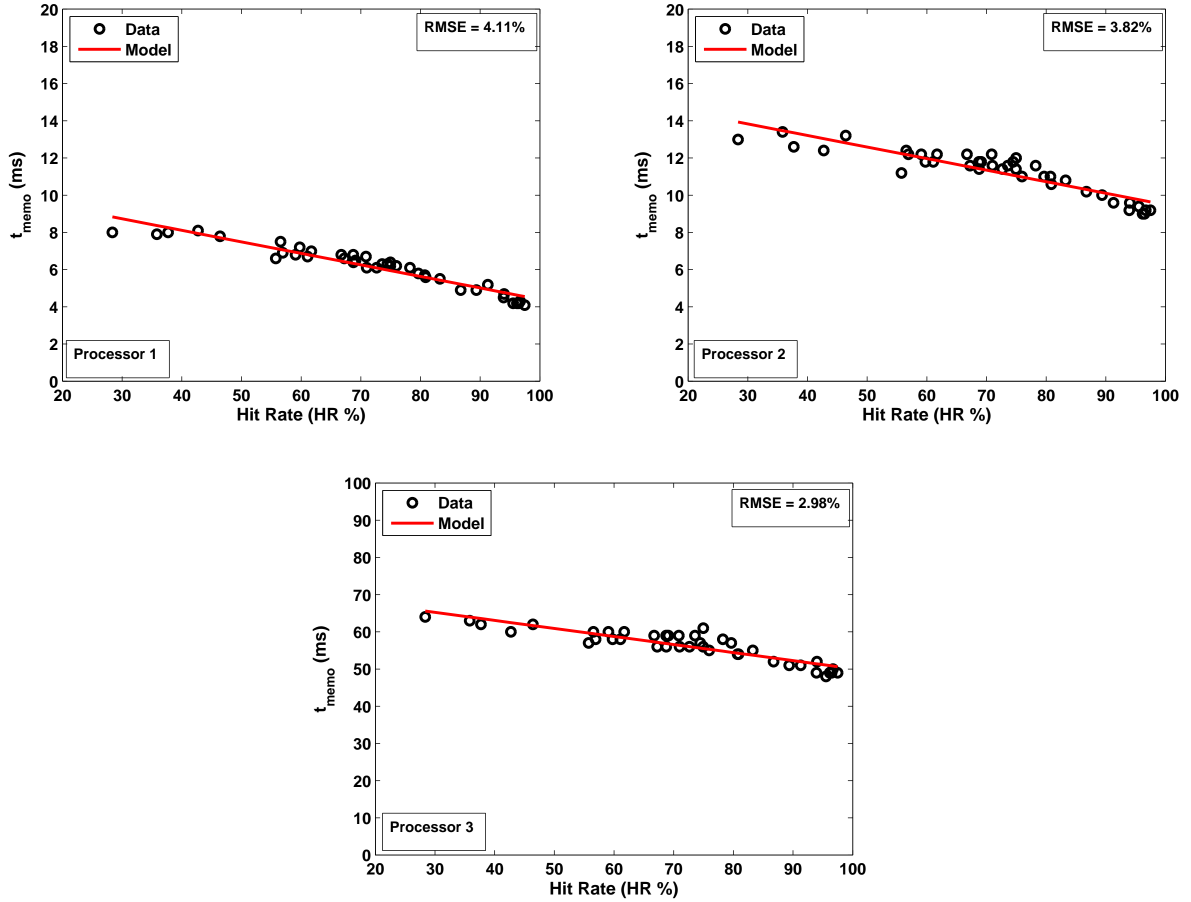



Figure 4.5: Linear model for $t_{memo}$ versus $HR_{sw}$ for a $16K$ entries reuse table

caching mechanism of the processor to update all copies of the reuse table that reside in different cache levels. Updating a data block in a cache hierarchy is done by either *write-through* or *write-back* method [49]. In a write-through cache, a write to a cache is propagated through all levels of the cache once a block of data in the highest level of cache is updated. In a write-back cache, updating the lower levels of cache is delayed until it is necessary. In either case, updating the cache may cause the pipeline to stall due to data hazards leading to the nonlinear effect of $t_{write}$ on $t_{memo}$. On the other hand, increasing hit rate decreases the number of cache updates that the pipeline must perform. Thus, $t_{memo}$ has a nonlinear relationship with $HR_{sw}$.

### 4.3.2 Memoization Overhead Time: A Nonlinear Model

To improve the accuracy of curve fits of the model and empirical data for the memoization overhead time ($t_{memo}$), we model $t_{memo}$ with a nonlinear equation:

$$t_{memo}(HR_{sw}) = -a \times HR_{sw}^2 - b \times HR_{sw} - c \tag{4.16}$$

The RMSEs are listed in table 4.6. As it is seen, the average RMSEs of curve fits for nonlinear model have decreased by 1.36%, 0.99%, and 0.35% in comparison to the linear model, for processor 1, processor 2, and processor 3, respectively.

Table 4.6: RMSEs (%) for quadratic curve fit for $t_{memo}$ for different $RT$ sizes on three processors

| Processor/$RT_{size}$ | $1K$ | $2K$ | $4K$ | $8K$ | $16K$ | $32K$ | $64K$ | $128K$ | $256K$ | Average |
|---|---|---|---|---|---|---|---|---|---|---|
| Processor 1 | 3.13 | 3.87 | 2.69 | 2.84 | 2.73 | 3.17 | 3.34 | 3.20 | 4.21 | 3.24 |
| Processor 2 | 3.15 | 3.28 | 3.16 | 3.07 | 2.70 | 3.77 | 3.84 | 4.15 | 4.21 | 3.48 |
| Processor 3 | 0.00 | 1.16 | 1.71 | 2.27 | 2.51 | 2.88 | 3.16 | 2.99 | 2.82 | 2.17 |

Figure 4.6 shows the experimental data and nonlinear model for $t_{memo}$ versus $HR_{sw}$ for a particular reuse table size ($16K$ entries) for all three processors.

### 4.3.3 Memoization Overhead Time: A Simplified Nonlinear Model

Our main goal of modeling the memoization overhead time ($t_{memo}$) is to predict speedup for images in a data set with the minimum required information. This enables us to quickly pick the optimal reuse table sizes that yield maximum speedups. The equation of speedup (equation 4.14) indicates that for a given algorithm, in order to predict the speedups for images of a data set, three parameters are required for each image: $t_{mask}$, $t_{memo}$, and $HR_{sw}$. $t_{mask}$ is usually similar for different images in a data set since it depends on the complexity of the algorithm. Thus, as we will show in the next section, to calculate speedup, measuring $t_{mask}$ for only two images is usually sufficient. The current nonlinear model of $t_{memo}$ requires that memoization overhead time be measured for at least three images in the data set because there are three paraments ($a$, $b$, and $c$) in

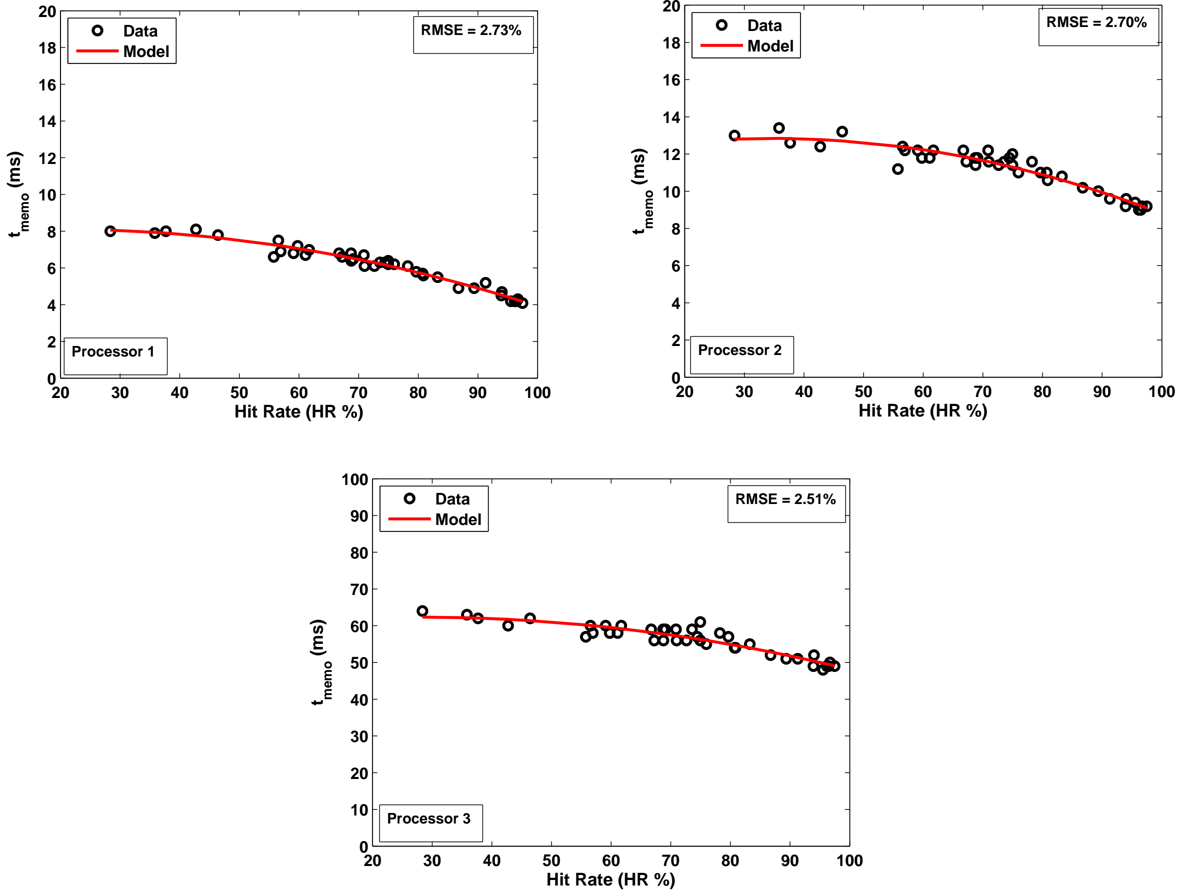



Figure 4.6: Nonlinear model for $t_{memo}$ versus $HR_{sw}$ for a 16$K$ entries reuse table

equation 4.16. We will simplify the model such that it requires measuring $t_{memo}$ for only two images of a data set, in order to predict $t_{memo}$ for all images in the data set. With this simplification, to predict $t_{memo}$ for all images in a data set, the only parameter that is required to be calculated for all images is hit rate $HR_{sw}$. Hit rate is a characteristic of image and is independent of the processor on which window memoization is run. Thus, a high level model can be used to calculate the hit rates of all images easily.

The current equation for memoization overhead time is:

$$t_{memo}(HR_{sw}) = -a \times HR_{sw}^2 - b \times HR_{sw} - c \tag{4.17}$$

Assuming that we have the hit rate of all images in the data set, it is seen that to calculate the coefficients of the quadratic equation in 4.17 (*i.e.* $a$, $b$, and $c$), at least three images

are required to measure $t_{memo}$. To reduce the minimum number of images required, we simplify equation 4.17 to:

$$t_{memo}(HR_{sw}) = -a \times HR_{sw}^2 - b \tag{4.18}$$

In order to determine the coefficients of the equation above (*i.e.* $a$ and $b$ ), it is only required to measure $t_{memo}$ for two images. To obtain a better accuracy of model, we pick two images with extreme hit rates: images with minimum and maximum hit rate. To predict $t_{memo}$ for all images in a data set, first, we measure $t_{memo}$ for two images with extreme hit rates in the data set. By having $t_{memo}$ for two images, we are able to calculate the coefficients of equation 4.18 ($a$ and $b$). By having $a$ and $b$ determined, for any image in the data set (*i.e.* any $HR_{sw}$), $t_{memo}$ can be calculated. The equations below show how $a$ and $b$ are calculated based on the memoization overhead times of images with maximum and minimum hit rates in a set of images.

$$a = \frac{t_{memo1} - t_{memo2}}{HR_{sw2}^2 - HR_{sw1}^2} \tag{4.19}$$

$$\begin{aligned} b &= -t_{memo1} - a \times HR_{sw1}^2 \\ &= -t_{memo1} - \left( \frac{t_{memo1} - t_{memo2}}{HR_{sw2}^2 - HR_{sw1}^2} \right) \times HR_{sw1}^2 \end{aligned} \tag{4.20}$$

where the subscripts 1 and 2 indicate $t_{memo}$ and $HR_{sw}$ of images with minimum and maximum hit rates, respectively.

To validate our simplified model for $t_{memo}$, we use equation 4.18 to perform curve fits for the empirical data of $t_{memo}$ for the same set of natural images used in the previous sections. We validate the simplified model for $t_{memo}$ by comparing the model with the empirical data using the RMSEs of the curve fits (table 4.7).

As it can be seen, although the accuracy has slightly dropped in comparison to the model using equation 4.17 (nonlinear model using $t_{memo}$ for three images), the model still matches the empirical data with reasonably low error (*i.e.* average RMSEs less than 4%)
.

Table 4.7: RMSEs (%) for simplified quadratic curve fit for $t_{memo}$ for different $RT$ sizes on three processors, using only two extreme images

| Processor/$RT_{size}$ | $1K$ | $2K$ | $4K$ | $8K$ | $16K$ | $32K$ | $64K$ | $128K$ | $256K$ | Average |
|---|---|---|---|---|---|---|---|---|---|---|
| Processor 1 | 4.45 | 6.20 | 3.19 | 3.39 | 4.05 | 3.67 | 3.46 | 3.19 | 4.07 | 3.96 |
| Processor 2 | 3.15 | 3.72 | 4.06 | 4.42 | 3.66 | 4.33 | 3.92 | 4.18 | 4.45 | 3.99 |
| Processor 3 | 0.00 | 1.23 | 1.79 | 2.30 | 2.56 | 3.22 | 3.78 | 3.57 | 3.72 | 2.48 |

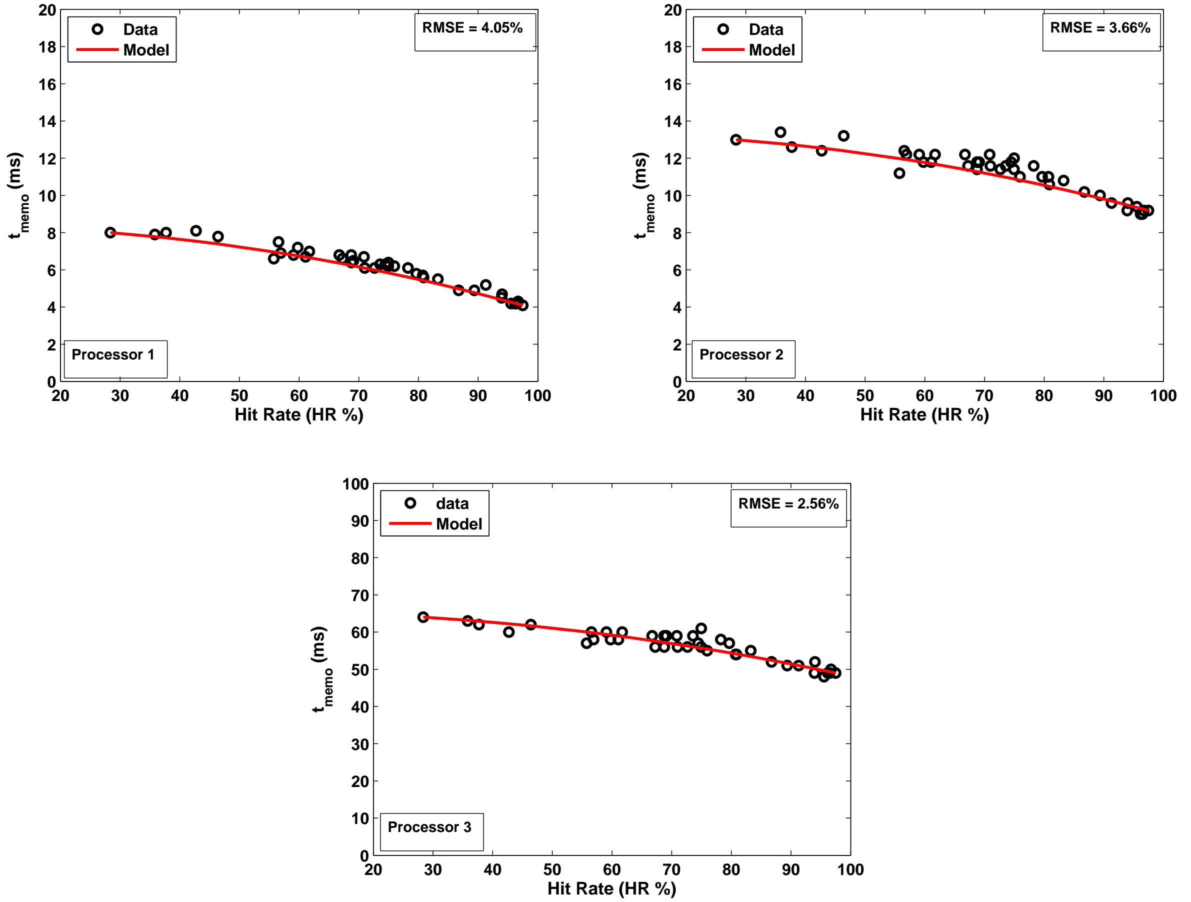



Figure 4.7: Simplified nonlinear model for $t_{memo}$ versus $HR_{sw}$ for a $16K$ entries reuse table using only two extreme images

Figure 4.7 shows the experimental data and the simplified quadratic equation (4.17) used to model the data for all three processors using a $16K$ reuse table. This model only uses two extreme case images for curve fit.

Figure 4.8 shows the RMSEs of all three models (linear, quadratic, and simplified

quadratic) for different reuse table sizes and three processors. It is seen that the quadratic model gives the best accuracy and in most cases, the simplified quadratic model gives more accurate result than the linear model.

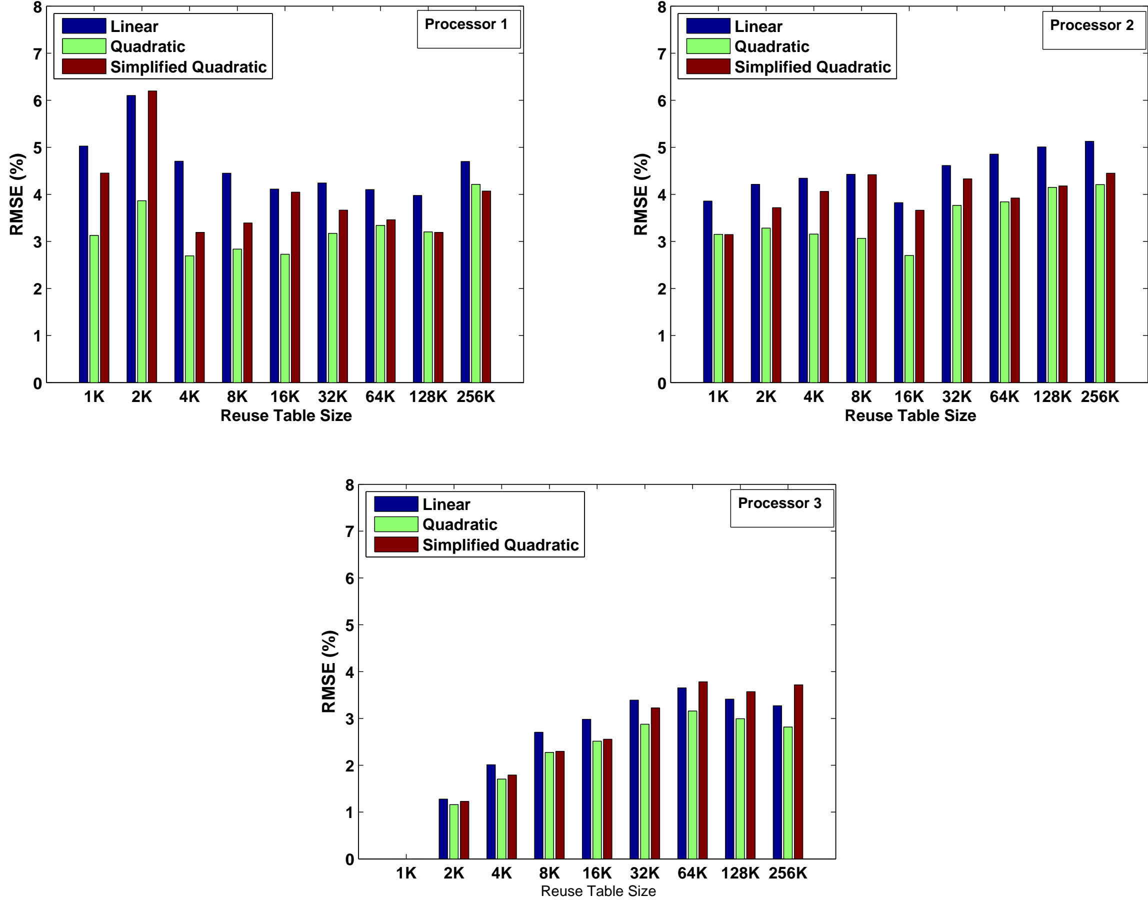



Figure 4.8: RMSEs for different reuse table sizes for linear, quadratic and simplified quadratic model for $t_{memo}$ versus $HR$

## 4.4 Speedup Model Validation

In section 4.3, we presented a model to predict the memoization overhead time for all images in a data set. The only information that the model needs is to measure the memoization overhead time ($t_{memo}$) for two images, which have minimum and maximum hit rates in a data set. With this information, for any image (*i.e.* for any hit rate), the model predicts the memoization overhead time. In this section, we modify the speedup

equation (equation 4.14) by replacing $t_{memo}$ with the equation presented in section 4.3. Afterward, we validate our model for speedup with empirical data for six case study algorithms presented in section 2.5.

In section 4.4.1, we present the empirical data and the results of our model for speedup for all six case studies run on processor 1 for different reuse table sizes. This gives two curves of speedup versus reuse table size. For each case study algorithm, we calculate the RMSE of our model result with respect to the empirical data to validate our model for speedup. In section 4.4.2, with the confidence achieved from high accuracy of the speedup model for processor 1, we validate the model for processors 2 and 3 for only two case study algorithms. We use the model to predict speedups for the rest of case study algorithms run on processors 2 and 3. It will be shown that for a given algorithm run on a given processor, there is an optimal reuse table size, which gives the maximum speedup. We use our model to predict the optimal reuse table size for a given algorithm and processor. In section 4.5, we will use the optimal reuse sizes obtained in this section to run experiments and measure empirical speedup results for all six case study algorithms run on the three processors using four different sets of images.

### 4.4.1 Model Validation for Processor 1

In this section, we present the empirical data and the results of our model for speedup for all six case study algorithms run on processor 1. In section 4.1, the the speedup equation was given as:

$$speedup = \frac{t_{mask}}{t_{memo} + (1 - HR_{sw}) \times t_{mask}} \tag{4.21}$$

where $t_{mask}$, $HR_{sw}$, and $t_{memo}$ are:

- $t_{mask}$: the time required to process the image by the conventional algorithm
- $HR_{sw}$: the hit rate of the image
- $t_{memo}$: the memoization overhead time

In section 4.3, we presented a model for $t_{memo}$ as:

$$t_{memo} = -a \times HR_{sw}^2 - b \tag{4.22}$$

Substituting equation 4.22 in 4.21 gives:

$$speedup = \frac{t_{mask}}{(-a \times HR_{sw}^2 - b) + (1 - HR_{sw}) \times t_{mask}} \tag{4.23}$$

In order to use equation 4.23 to calculate the speedup, first, the hit rates of all images are calculated. Hit rate is a characteristic of the input image and it is independent of the processor on which the program is run. Thus, it can be calculated using any high level tool (*e.g.* Matlab). Afterward, for the given processor and algorithm, we measure $t_{memo}$ and $t_{mask}$ for only two extreme images: images with minimum and maximum hit rates. Using $t_{memo}$ for two extreme case images, we can calculate $a$ and $b$ in the speedup equation (equation 4.23). For $t_{mask}$, we use the average of $t_{mask}$ for the two extreme case images. We calculate the speedup for each image in the set of natural images (see section 2.6) for each case study algorithm run on processor 1. In addition, we run the experiments for a range of reuse table sizes from $1K$ entries to $256K$ entries.

Figure 4.9 shows the speedup curves for both empirical data and the model for all six case study algorithms run on processor 1. The plots show the speedups for different reuse table sizes, ranging from $1K$ entries to $256K$ entries. For each reuse table size, the speedup curve shows the average speedup of all 40 images in the set of natural images. The RMSEs for each case study algorithm (our model versus empirical data) are shown in Table 4.8. It is seen that the model matches the experimental data with a reasonably high accuracy with an average RMSE of 3.28%.

Table 4.8: RMSEs (%) for speedups on processor 1

| RMSE/Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| RMSE (%) | 2.56 | 1.77 | 7.95 | 2.05 | 3.59 | 1.79 |

The speedup curves shown in Figure 4.9 demonstrate an interesting feature; almost in all cases, increasing the reuse table size beyond some point (*e.g.* $16K$) degrades the speedup. In other words, further than a particular reuse table size, although hit rate still

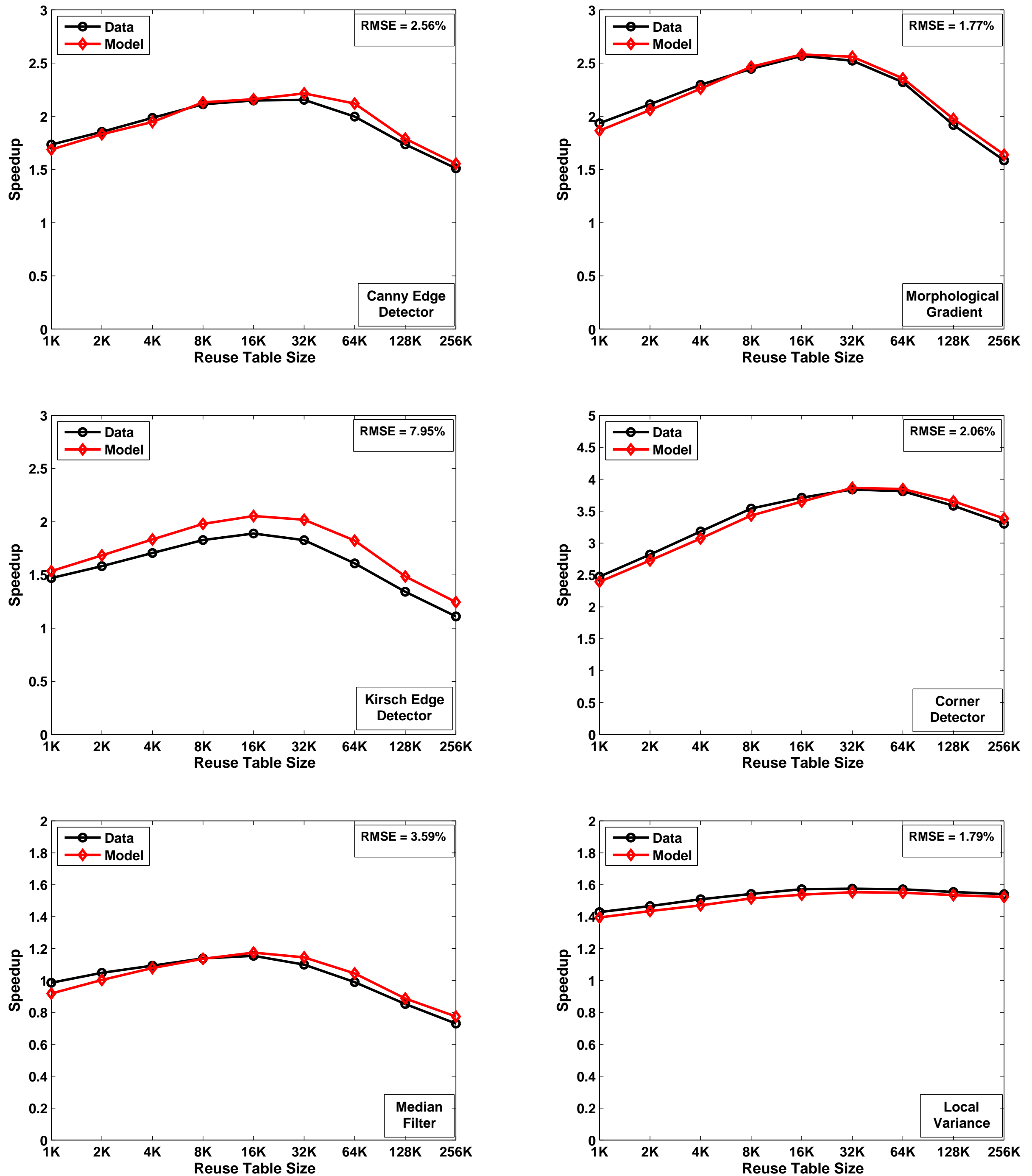



Figure 4.9: Speedup curves for the empirical data and the model for processor 1

increases, which on surface should cause speedup to increase (according to equation 4.23), it is seen that speedup decreases. The reason is that as the size of the reuse table increases, it is more likely that the reuse table is located in larger cache memories, which belong to

the lower levels of the cache hierarchy in the processor. This increases the time required to perform memory operations (*i.e.* $t_{read}$ and $t_{write}$) due to high cache miss cost. As a result, for larger reuse sizes, $t_{memo}$ increases faster than hit rate. Therefore, it is not always the case that a larger reuse table will give a better speedup for window memoization.

For a given algorithm and processor, there is an optimal reuse table size, which gives the maximum speedup. For processor 1, the optimal reuse table sizes for case study algorithms are listed in table 4.9.

Table 4.9: Optimal $RT_{size}$ for for processor 1

| $RT_{size}$/Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| $RT_{size}$ | $32K$ | $16K$ | $16K$ | $32K$ | $16K$ | $32K$ |

The advantage of our speedup model (equation 4.23) is that by having the memoization overhead time for only two images, we can calculate speedups of all images for the full range of the reuse table sizes. This will yield the optimal reuse table size for the given algorithm and processor, to be used by window memoization for actual experiments.

### 4.4.2 Model Validation for Processors 2 and 3

In the previous section, we showed that for all six case study algorithms, our model for speedup matches the empirical data with a high accuracy. In this section, in order to further validate our speedup model, we present the empirical data and our model result for speedup for two algorithms run on processors 2 and 3.

To generate empirical data for speedup on processors 2 and 3, we have chosen the morphological gradient and Kirsch edge detector because $t_{mask}$ for these two algorithms are the smallest in comparison to the other case study algorithms. If $t_{mask}$ is too large then the effect of $t_{memo}$ will almost be negligible and hence, regardless of accuracy of our model for $t_{memo}$ in equation 4.18, the error in speedup model with respect to empirical data will be negligible. In addition, as shown for processor 1 in the previous section, the worst case accuracy of our speedup model is for the Kirsch edge detector. Therefore, it is necessary to calculate the accuracy of the model for the worst case for processors 2 and 3. The RMSEs for the morphological gradient and Kirsch edge detector for processors 2

and 3 are shown in table 4.10. The speedup plots for all six algorithms on processors 2

Table 4.10: RMSEs (%) for speedups on processors 2 and 3

| Processor/Algorithm | Morphological | Kirsch |
|---|---|---|
| Processor 2 | 4.22 | 13.73 |
| Processor 3 | 2.70 | 11.73 |

and 3 are shown in figures 4.10 and 4.11, respectively.

As it can be seen from the speedup curves for processors 2 and 3, each algorithm gives an optimal reuse table size to achieve the maximum speedup. Tables 4.11 and 4.12 list the optimal reuse table sizes for processors 2 and 3, respectively.

Table 4.11: Optimal $RT_{size}$ for for processor 2

| $RT_{size}$/Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| $RT_{size}$ | $16K$ | $16K$ | $16K$ | $32K$ | $8K$ | $64K$ |

Table 4.12: Optimal $RT_{size}$ for for processor 3

| $RT_{size}$/Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| $RT_{size}$ | $256K$ | $64K$ | $64K$ | $128K$ | $256K$ | $256K$ |

## 4.5 Empirical Speedup Results

In the previous section, we determined the optimal reuse table sizes for each case study algorithm run on each of the three processors. In this section, we use these optimal reuse table sizes to run window memoization for all case study algorithms on different image sets (*i.e.* natural, industrial, medical, and barcode images presented in section 2.6) to measure the actual speedups. We run the experiments on all three processors with maximized compiler optimization settings.

Figure 4.12 shows the speedups for all six case study algorithms using optimal reuse table sizes for all three processors. For each processor, it is seen that medical images

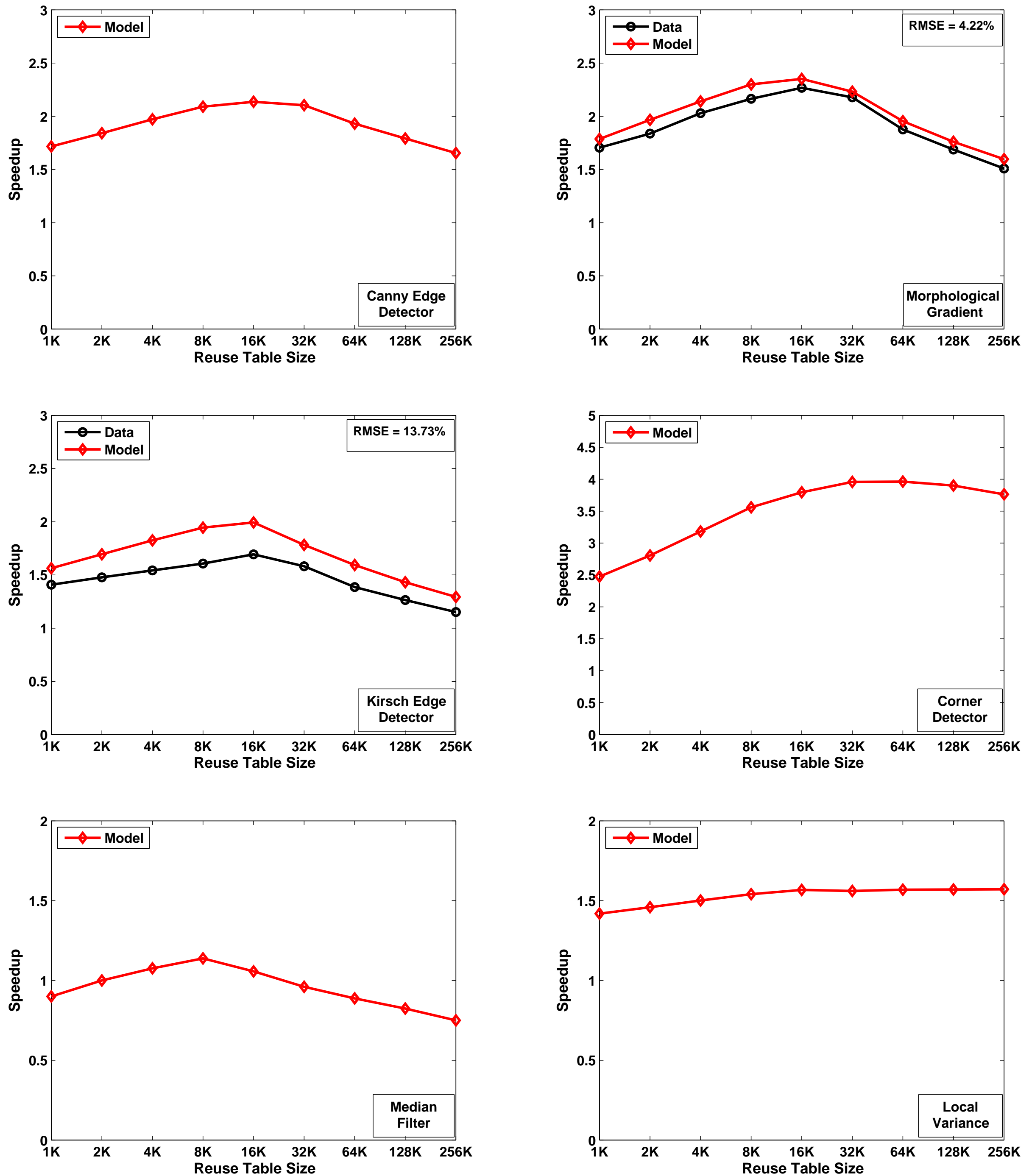



Figure 4.10: Speedup curves generated by the model for processor 2

(*i.e.* ultrasound images) give the lowest speedups. The reason is that medical images are usually very noisy. A large amount of noise decreases the coding and interpixel redundancy and causes similar windows to become dissimilar leading to low hit rates

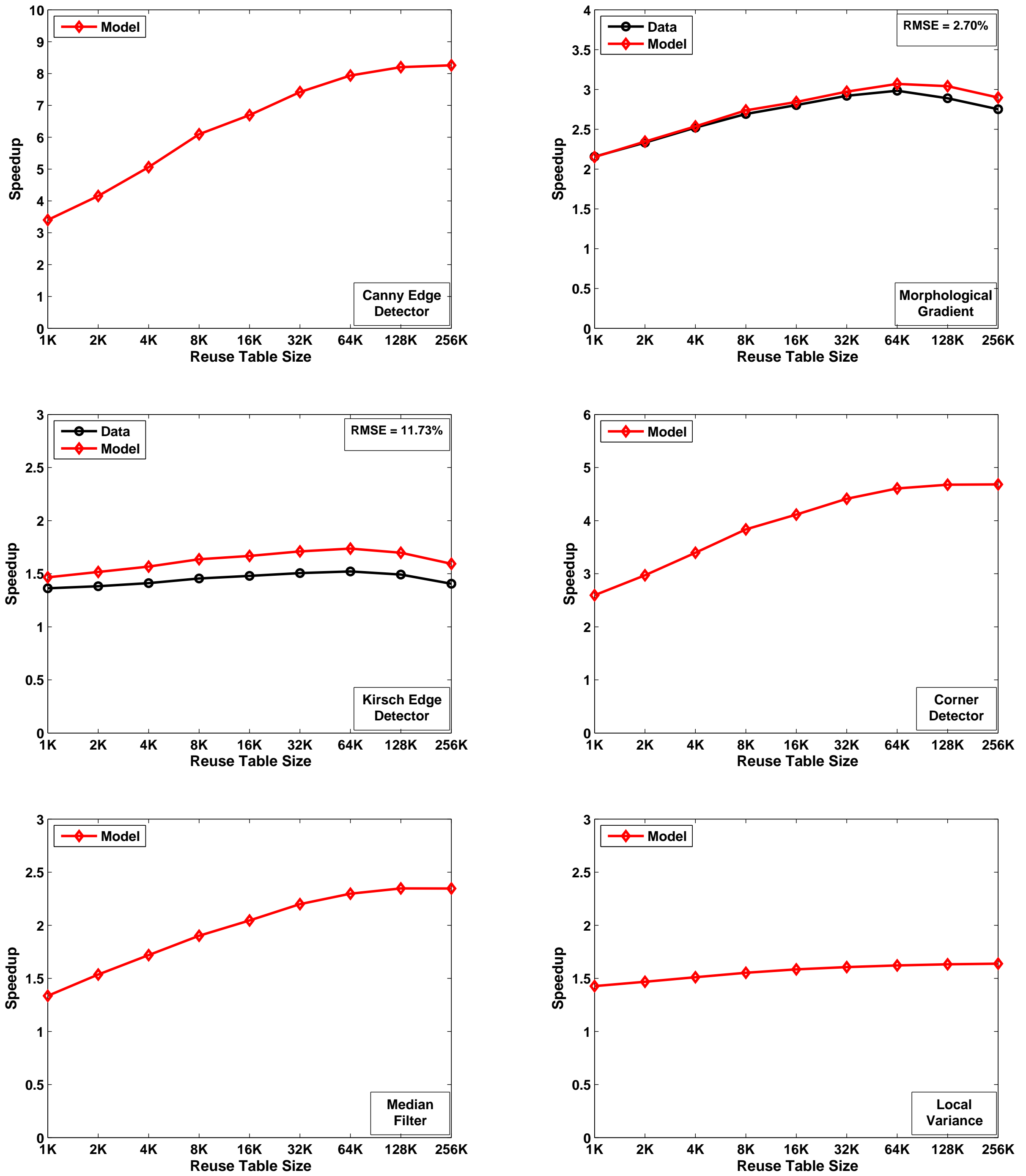



Figure 4.11: Speedup curves generated by the model for processor 3

and hence low speedups (equation 4.23). Nevertheless, if the medical images are filtered first to reduce the noise, which is usually the case in the medical applications, window memoization can yield higher speedups. The highest speedups are achieved by industrial

images (*i.e.* text classification, quality control, and cell imaging). These images do not contain complex patterns and hence many windows are similar (*i.e.* high coding and interpixel redundancy and high hit rate). Among processors, processor 3 which is an embedded processor with a low-end CPU ($300MHz$) gives the highest speedups. For low-end processors, the relative speed of processor in comparison to memory is slower than that for mid-range and high-end processors (*i.e.* higher $\frac{t_{mask}}{t_{memo}}$ in equation 4.8 for processor 3). This leads to higher speedups in low-end processors. The numerical values of speedups are presented in appendix C, tables C.1, C.2, and C.3.

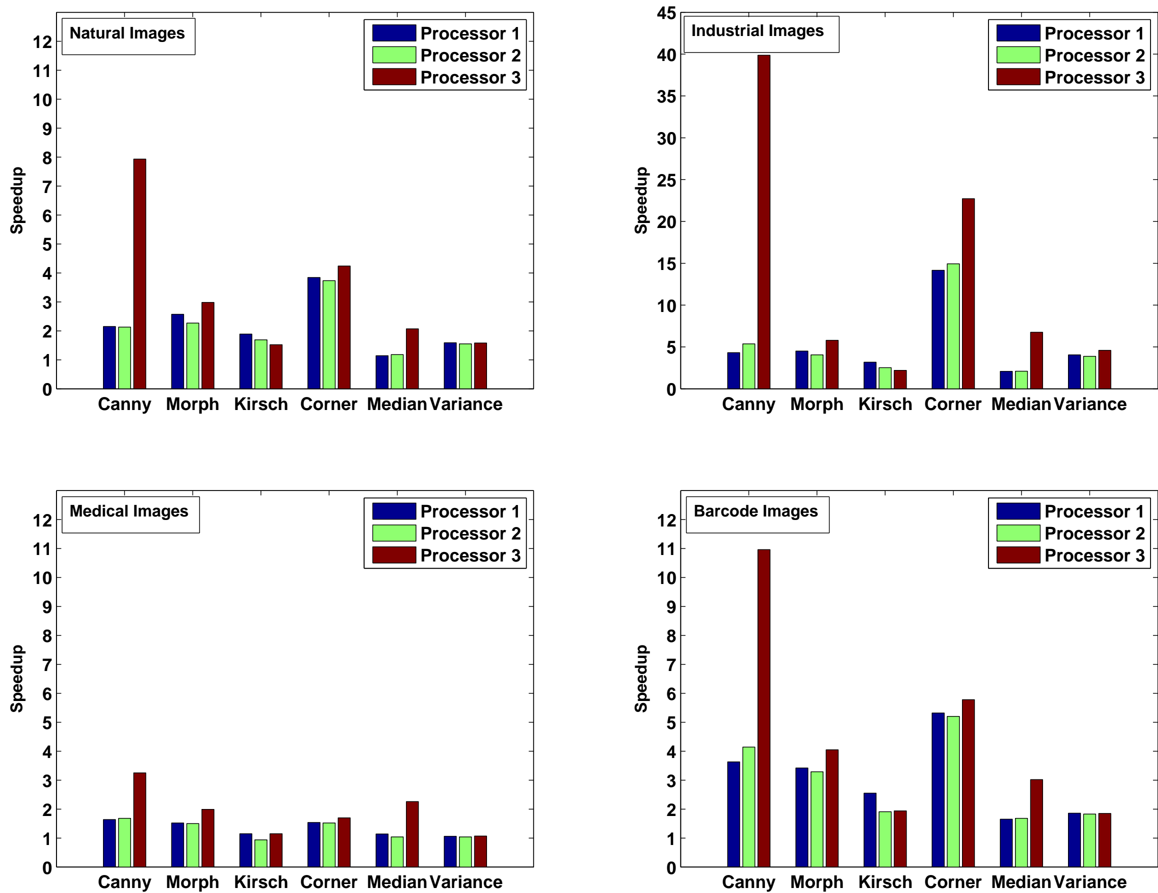



Figure 4.12: Average empirical speedup results

We measure the accuracy of window memoization for an algorithm by comparing the output of window memoization against the reference image calculated by a conventional implementation of the algorithm (*i.e.* without memoization). For binary output images (*i.e.* Canny edge detection, Morphological gradient, Kirsch edge detection, and

Corner detection) we use misclassification error (equation 4.24), which calculates the percentage of the background pixels that have been assigned to foreground incorrectly and vice versa [48]. In Equation 4.24, $B_{Ref}$ and $F_{Ref}$ are the reference edge/corner map background and foreground, respectively; and $B_{Test}$ and $F_{Test}$ are the background and foreground of the window memoization result, respectively.

$$ME = 1 - \frac{|B_{Ref} \cap B_{Test}| + |F_{Ref} \cap F_{Test}|}{|B_{Ref}| + |F_{Ref}|} \tag{4.24}$$

For algorithms with gray-level outputs (*i.e.* median filter and local variance), we use signal-to-noise ratio, given in section 3.1.3.

Figure 4.13 shows the accuracy of the results for all six case study algorithms. For binary results, the accuracy of algorithms is in the range 96.01% to 99.93%. For gray-level results, the SNR ranges from 29.73 to 47.74. The numerical values of results accuracy are presented in appendix C, table C.4.

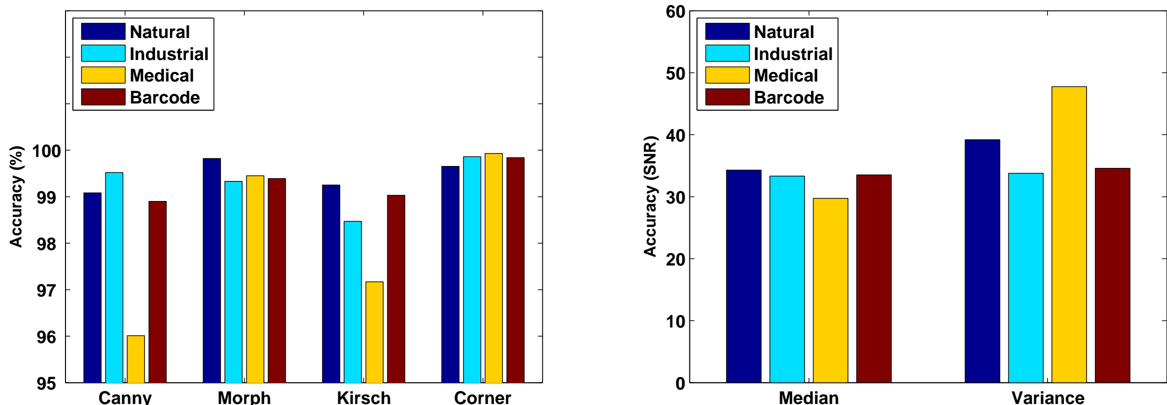



Figure 4.13: Accuracy of results

For natural images, the original results for a sample image along with the results for window memoization for all six algorithms are shown in figures 4.14 and 4.15. Almost in all cases, the accuracy is so high that the difference between the original result and window memoization result is hardly distinguishable. The results for industrial, medical, and barcode images are presented in appendix C, figures C.1 to C.6.

The images in the bottom of figures 4.14 and 4.15 show the *difference images* of the results of the algorithms with and without window memoization. For binary output

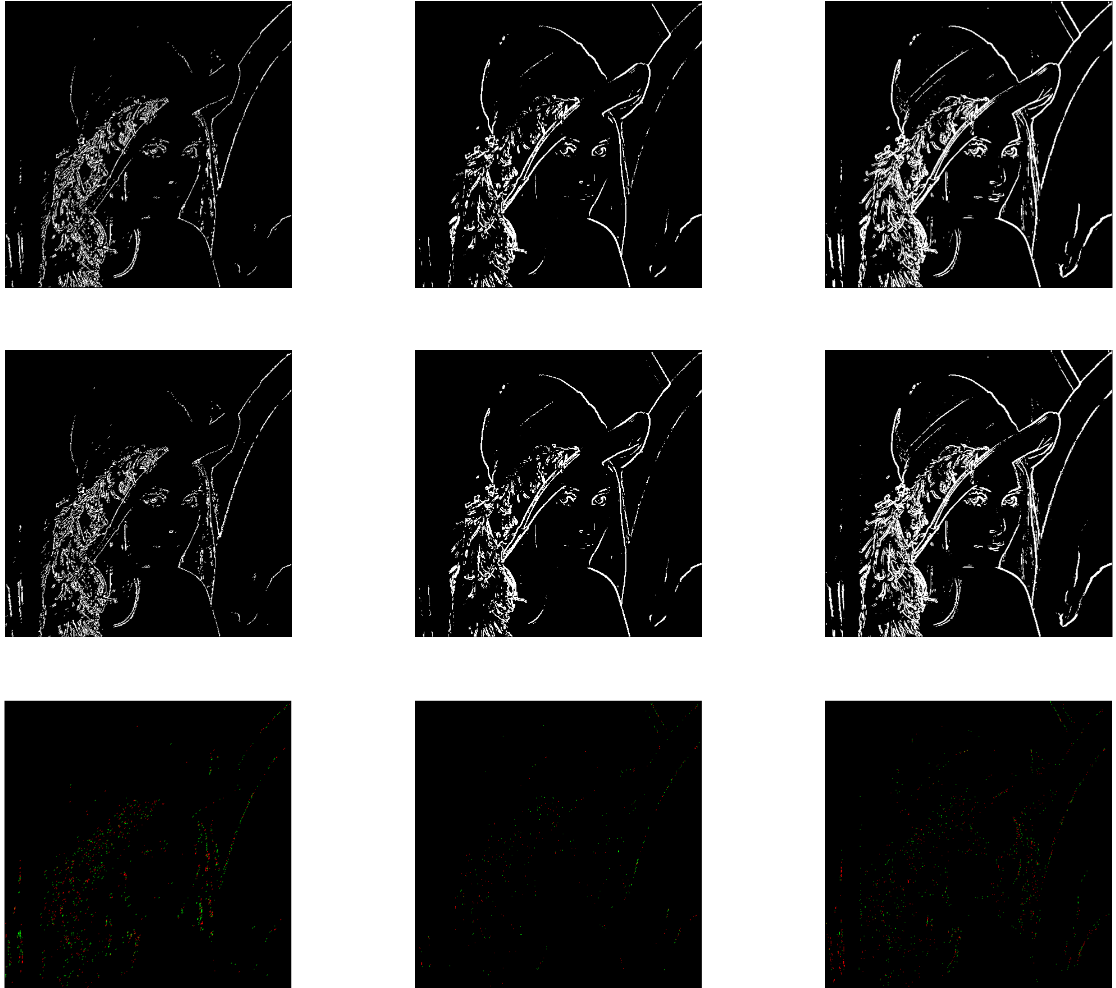


Figure 4.14: Results for natural images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, middle: window memoization results, bottom: difference images.

algorithms (*i.e.* Canny, Kirsch, and morphological edge detectors and corner detector), the difference images show two sets of marks: the locations that contain edges in the original results and non-edges in the window memoization results (marked with red edges) and the locations that contain non-edges in the original results and edges in the window memoization results (marked with green edges). It is seen that the error in the window memoization results is mostly in the area that the original results contain edges. This is expected because those windows whose responses are edges contain more information compared to the windows that belong to the background. For windows with higher

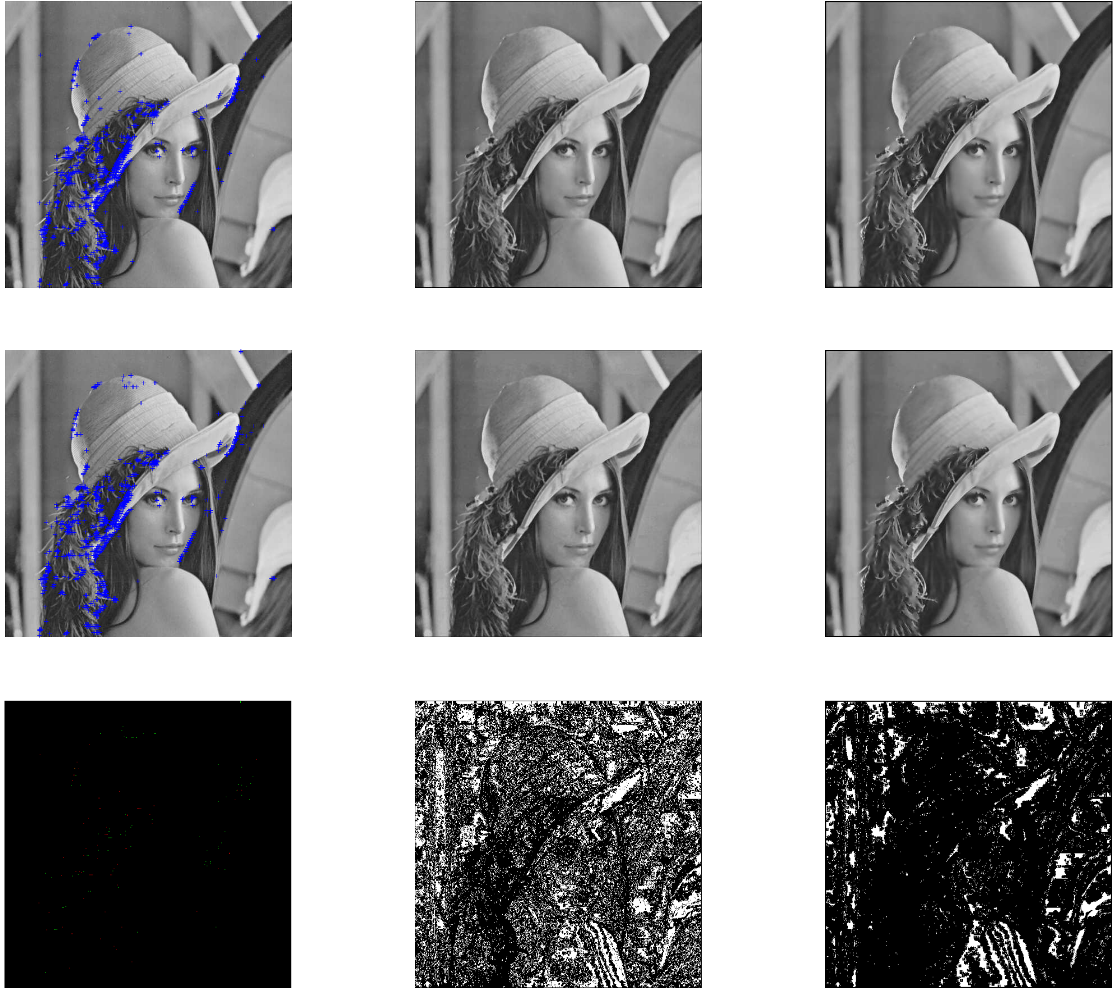


Figure 4.15: Results for natural images. Left to right: Corner detection, median filter, and local variance. Top: original results, middle: window memoization results, bottom: difference images.

information, it is more likely that the 4 least significant bits of each pixel play a role in determining the response of the window. As a result, by ignoring the 4 least significant bits of each pixel in window memoization, it is more likely that the inaccuracy in the result is introduced in the locations that edges exist.

For gray-level output algorithms (*i.e.* median filter and local variance), the difference image is the gray-level difference of the result of the original algorithm and the window memoization result, which has been normalized with respect to gray level 255. It is seen that the inaccuracy introduced by window memoization is almost spread all over the

image (figure 4.15, bottom center and bottom right).

## 4.6 Speedup versus Coding/Interpixel Redundancy

In this section, we show that the coding and interpixel redundancy of an image have a positive relationship with the actual speedup obtained by window memoization in software for the image (relation 4.28). In chapter 3, we showed that the coding and interpixel redundancy of an image have a positive relationship with the computational redundancy of the image.

$$C_r + IP_r \propto^{+} Comp_r \tag{4.25}$$

In this section, first, we show that the computational redundancy of an image has a positive relationship with the hit rate obtained by window memoization in software for the image ($HR_{sw}$).

$$Comp_r \propto^{+} HR_{sw} \tag{4.26}$$

Afterward, we show that the hit rate of an image has a positive relationship with the speedup obtained by window memoization in software for the image.

$$HR_{sw} \propto^{+} speedup \tag{4.27}$$

From relations 4.25, 4.26, and 4.27, it will be concluded that:

$$C_r + IP_r \propto^{+} speedup \tag{4.28}$$

### 4.6.1 Hit Rate of Window Memoization in Software versus Computational Redundancy

In this section, we show that the computational redundancy of an image has a positive relationship with the hit rate obtained by window memoization in software for the image.

$$Comp_r \propto^+ HR_{sw} \tag{4.29}$$

In chapter 3, computational redundancy was defined as:

$$Comp_r = 1 - \frac{s}{n} \tag{4.30}$$

In equation 4.30, $s$ and $n$ are the total number of symbols and windows in image, respectively. For reasonably complex images, it is likely that in many cases the symbols are almost evenly distributed across the image. In other words, for any subimage in the image of size $n'$ windows, the probability of occurrence of each symbol $s_i$ in the subimage ($P^{sub}(s_i)$) is almost equal to the probability of occurrence of the symbol in the image ($P(s_i)$):

$$P^{sub}(s_i) \approx P(s_i) \tag{4.31}$$

For very small subimages, however, the windows usually belong to one object or background, which means that the windows are similar and thus, they do not represent many different symbols. As a result, depending on the structure and geometry of images, for each image, there is a minimum subimage size, above which the distribution of its symbols becomes even. For an image with a uniform distribution of symbols, equation 4.31 will hold for any subimage size. On the contrary, for an image with an absolutely non-even distribution of symbols, equation 4.31 will only hold if the subimage is as large as the image itself.

For real-world images, the distribution of symbols usually becomes almost even when the subimage size reaches a certain point (e.g $4K$ windows; see figure 4.16). In the context of window memoization, the subimage size, based on which we investigate the evenness of distributions of symbols across the image is in fact the reuse table size. Assume that we have two images $Img_1$ and $Img_2$ with the same size ($n$ windows) and $s_1$ and $s_2$ symbols, respectively, where $s_1 > s_2$. Based on equation 4.30, $s_1 > s_2$ implies that $Comp_{r1} < Comp_{r2}$. Also, assume that both images have an even distribution of symbols across the image for subimages larger than $n'$ windows. From $s_1 > s_2$, we can conclude that, on average, the probability of occurrence of each symbol in $Img_2$ (*i.e.* $P_2(s_j)$) is

higher than that in $Img_1$ (*i.e.* $P_1(s_i)$) or $P_1(s_i) < P_2(s_j)$ where $i \in [0, s_1 - 1]$ and $j \in [0, s_2 - 1]$.

Due to the even distribution of symbols in both images, $P_1(s_i) < P_2(s_j)$ will cause the probability of occurrence of each symbol across any subimage in $Img_2$ to be higher than that in $Img_1$ or $P_1^{sub}(s_i) < P_2^{sub}(s_j)$. This means that, with the direct-mapped mapping scheme used by window memoization in software using a reuse table larger than $n'$ entries, the frequency of reusing the result of each symbol for $Img_2$ (or hit rate) will be higher than that of $Img_1$ or $HR_{sw1} < HR_{sw2}$. Given that we already showed $Comp_{r1} < Comp_{r2}$, by showing that $HR_{sw1} < HR_{sw2}$ it is shown that relation 4.29 holds for most typical cases.

To summarize, if images have even distributions of symbols for subimages larger than $n'$ windows, the computation redundancy of an image has a positive relationship with the hit rate of the image obtained by window memoization, provided that the reuse table size is larger than $n'$ entries.

$$Comp_r \propto^+ HR_{sw} \tag{4.32}$$

Figure 4.16 shows the error margins for relation 4.32 with an accuracy equal to or above 95% for different reuse table sizes and image sets. It is seen that as the reuse table size grows, the error margin decreases such that after a certain reuse table size, it becomes zero or negligible because the distributions of symbols in most images become even for large subimages (*e.g.* reuse tables larger than 4K entries). A larger reuse table will lead to less evictions and hence hit rate will approach to that of a perfect cache, which as shown in chapter 3, has a positive relationship with the computational redundancy of the image (section 3.1.2).

Figure 4.16 indicates that the reuse table size beyond which the error margin is negligible is different for each set of images. For example, medical images require a reuse table of $8K$ entries to achieve negligible error margin while for natural images, a $4K$ entries reuse table eliminates the error margin. The reason is that the symbols in different sets of images are distributed with a different degree of evenness. For images with more even

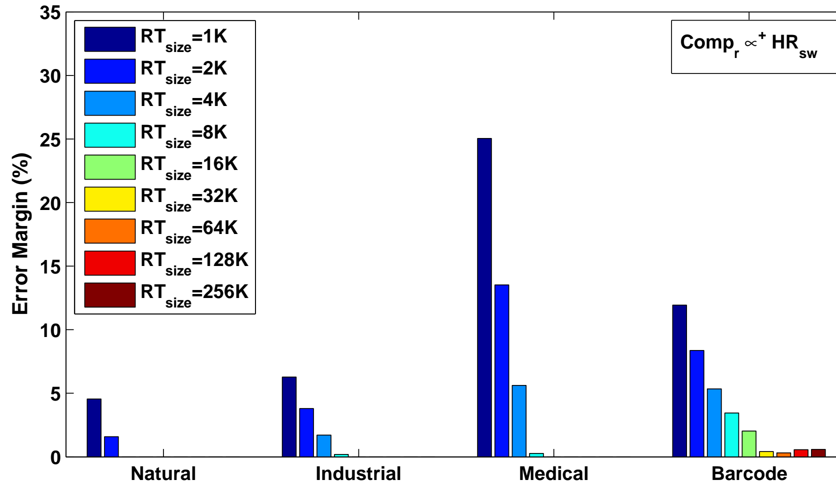



Figure 4.16: Error margins for relation $Comp_r \propto^+ HR_{sw}$ with an accuracy equal to or above 95%

distribution of symbols, relation 4.32 will hold with a high accuracy with smaller reuse table sizes, in comparison to images with with less even distribution of symbols.

In section 4.4, we discussed that for each algorithm and processor, there is an optimal reuse table size, which yields the maximum speedup. As it was shown in tables 4.9, 4.11, and 4.12, for all our case study algorithms, the minimum reuse table size that leads to maximum speedups is $8K$ entries. Figure 4.16 shows that for almost all sets of images, relation 4.32 holds with very high accuracy for reuse table sizes equal to or greater than $8K$ entries.

As an example, figure 4.17 shows relation 4.29 for natural images for reuse tables of $1K$, $2K$, and $16K$ entries sizes. Table C.5 to C.8 in appendix C show the numerical values of accuracy and error margin for relation 4.32 for different sets of images.

### 4.6.2 Speedup versus Hit Rate of Window Memoization In Software

In the previous section, we showed that the computational redundancy of an image has a positive relationship with the hit rate obtained by window memoization in software for the image.

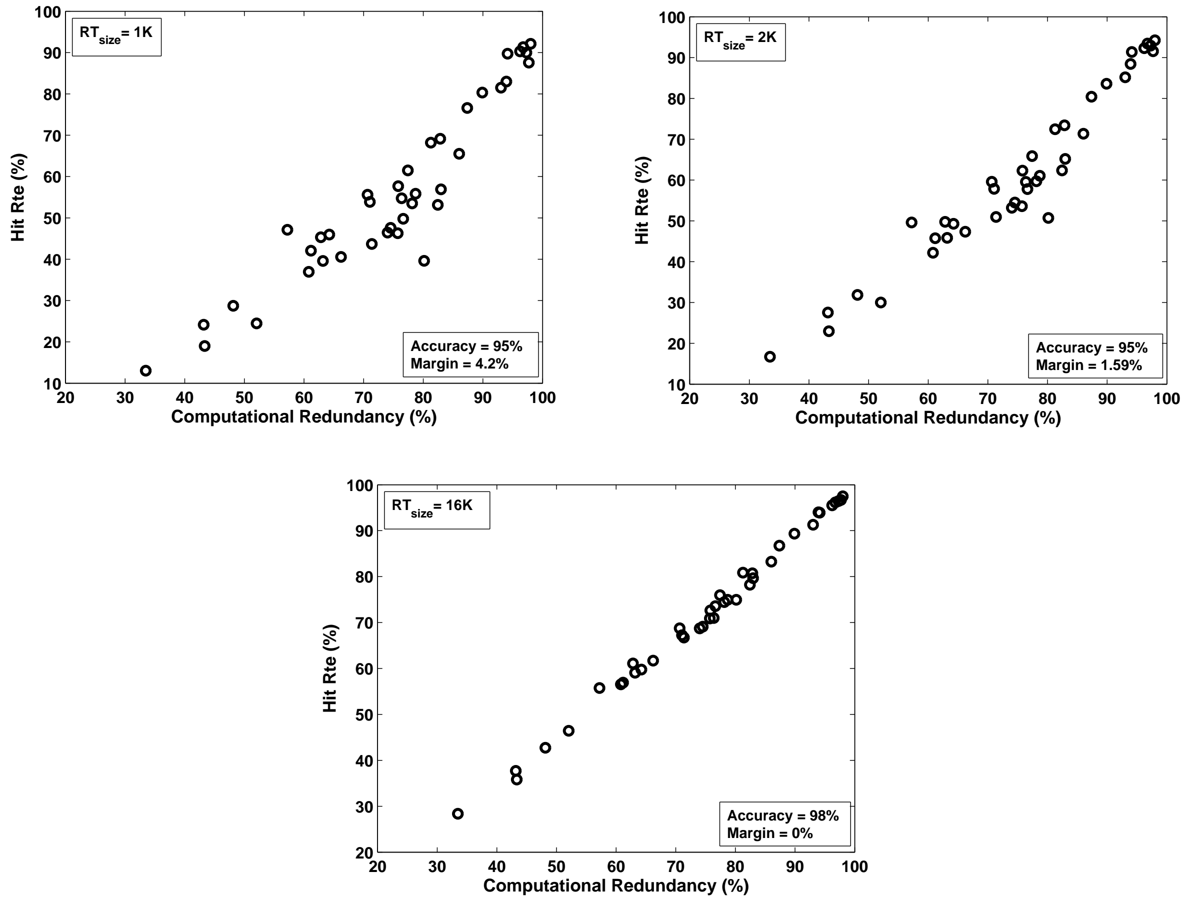



Figure 4.17: Hit rate versus computational redundancy for natural images

$$Comp_r \propto^+ HR_{sw} \tag{4.33}$$

In this section, we show that the hit rate of an image obtained by window memoization in software has a positive relationship with the speedup of the image obtained by window memoization in software

$$HR_{sw} \propto^+ speedup \tag{4.34}$$

In section 4.4.1, we showed that the speedup for an image obtained by window memoization in software has the following relationship with hit rate:

$$speedup = \frac{t_{mask}}{(-a \times HR_{sw}^2 - b) + (1 - HR_{sw}) \times t_{mask}} \tag{4.35}$$

where as discussed in section 4.3.3, $a$ and $b$ are calculated as:

$$a = \frac{t_{memo1} - t_{memo2}}{HR_{sw2}^2 - HR_{sw1}^2} \tag{4.36}$$

$$b = -t_{memo1} - a \times HR_{sw1}^2 \tag{4.37}$$

In the equations above, the subscripts 1 and 2 indicate $t_{memo}$ and $HR_{sw}$ of images with minimum and maximum hit rates, respectively. Experimental data shows that in equation 4.36, the numerator is always a positive number. This is expected because higher hit rates, which is the case for the image with the maximum hit rate, reduce the number of writes to the reuse table and therefore, the memoization overhead time decreases (*i.e.* $t_{memo1} > t_{memo2}$). Because the denominator of equation 4.36 is also always positive, it is concluded that $a$ is always a positive number. This means that $b$ (*i.e.* equation 4.37) is a negative number. Because $a$ is always positive and $b$ is always negative, equation 4.35 indicates that for a given algorithm (*i.e.* fixed $t_{mask}$) run on a given processor (*i.e.* fixed $a$ and $b$), the hit rate of an image has a positive relationship with the speedup of the image, which means relation 4.34 holds.

Our experiments show that for very simple images (*e.g.* industrial images), for a given algorithm and processor, $t_{mask}$ is not exactly the same for all images in the data set. Very simple images tend to have smaller $t_{mask}$ than other images. The root cause of this seems to be the optimizations performed at the microarchitectural level of the processor. Having different $t_{mask}$ for different images introduces an error in relation 4.34. As Figure 4.18 shows, among the four sets of images, the error margin of relation 4.34 for industrial images is the highest since the set of industrial images contains the simplest images. The second simplest images are barcode images, for which relation 4.34 gives the second poorest accuracy among the sets of images (figure 4.18).

Table C.9 to C.12 in appendix C show the numerical values of accuracy and error margin for relation 4.34 for different sets of images.

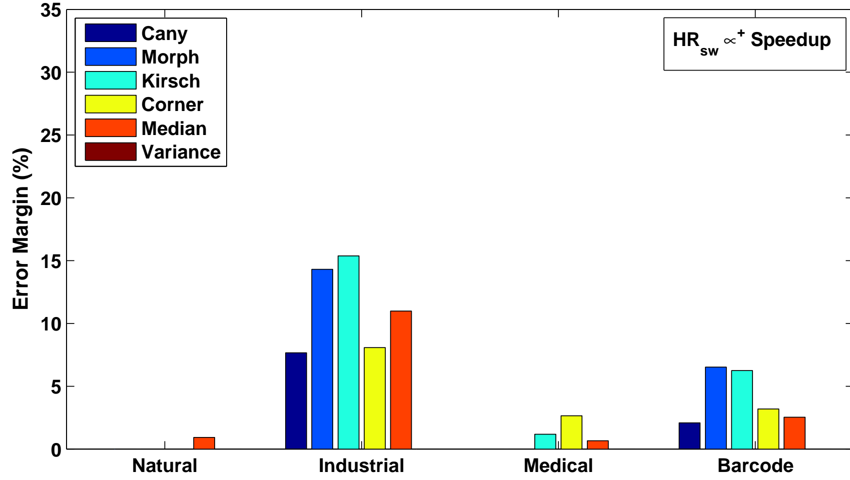



Figure 4.18: Error margins for relation $HR_{sw} \propto^{+} speedup$ with an accuracy equal to or above 95%

### 4.6.3 Speedup of Window Memoization in Software versus Coding/Interpixel Redundancy

In sections 4.6.2 and 4.6.3, we showed that:

$$Comp_r \propto^{+} HR_{sw} \tag{4.38}$$

and

$$HR_{sw} \propto^{+} speedup \tag{4.39}$$

From the above two relations, it is concluded that the computational redundancy of an image has a positive relationship with the speedup of the image

$$Comp_r \propto^{+} speedup \tag{4.40}$$

Given that, in chapter 3, we have shown that the coding and interpixel redundancy of an image have a positive relationship with the computational redundancy of the image (relation 4.25), it is concluded that the coding and interpixel redundancy of an image has a positive relationship with the speedup of the image.

$$C_r + IP_r \propto^+ speedup \tag{4.41}$$

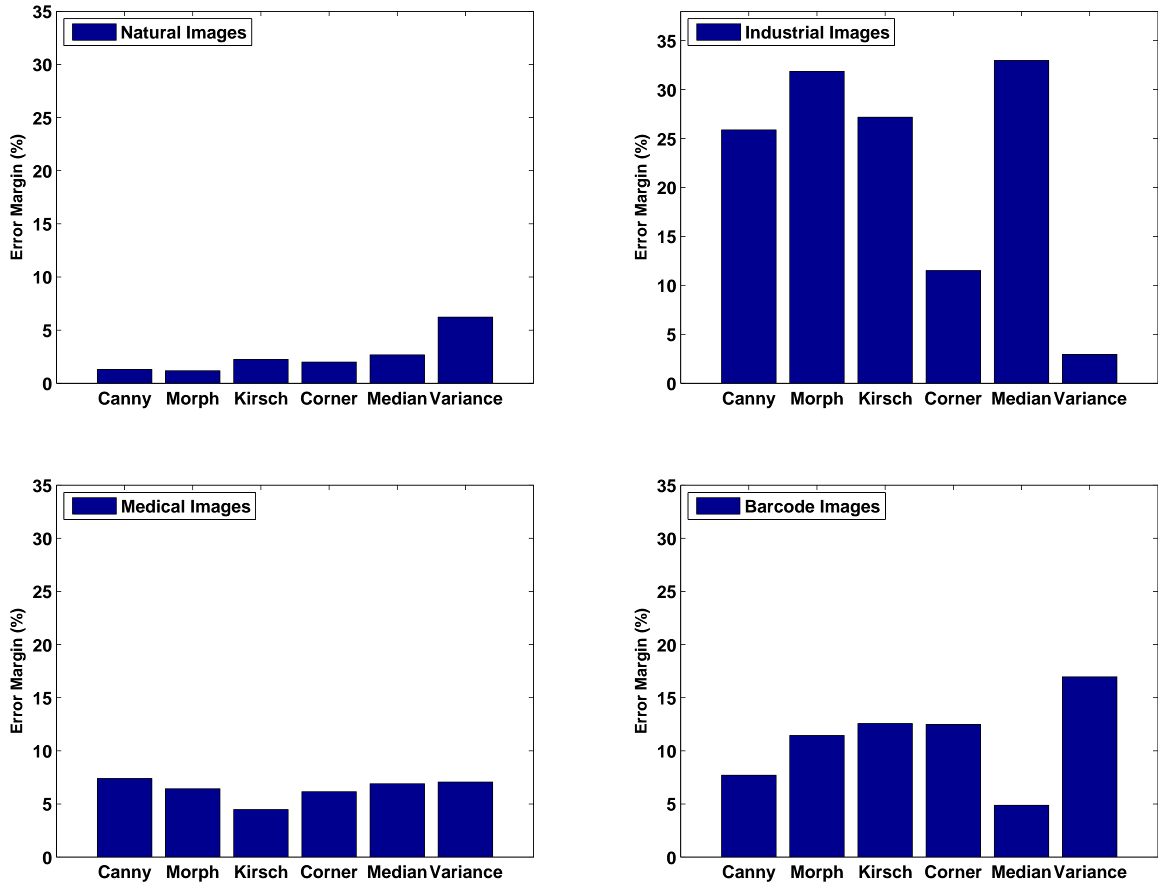



Figure 4.19: Error margin for relation $C_r + IP_r \propto^+ speedup$ with 95% accuracy

Figure 4.19 shows the error margins of relation 4.41 with 95% accuracy for all sets of images and case study algorithms run on processor 1. It is seen that the natural and industrial images give the best and worst accuracy on relation 4.41, respectively. To be able to understand the underlying roots of errors, we have drawn a plot (figure 4.20) that shows the average error margins at each intermediate step of relation 4.41. The relations whose average error margins are shown in figure 4.20 are:

$$C_r + IP_r \propto^+ Comp_r \tag{4.42}$$

$$Comp_r \propto^+ HR_{sw} \tag{4.43}$$

$$HR_{sw} \propto^+ speedup \tag{4.44}$$

$$C_r + IP_r \propto^+ speedup \tag{4.45}$$

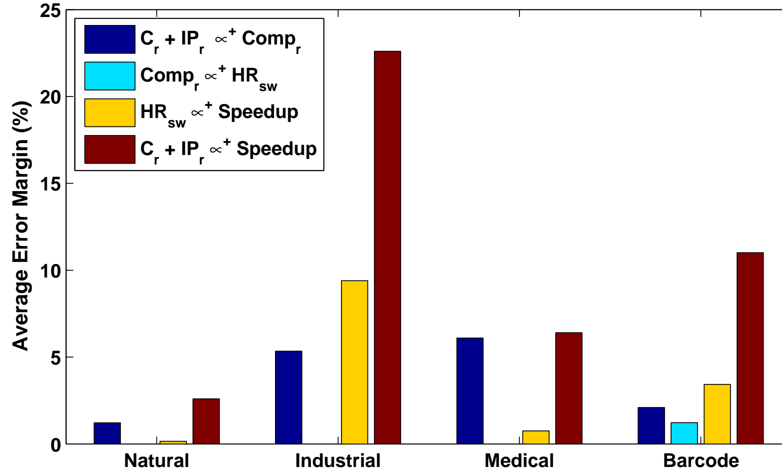



Figure 4.20: Average error margins for relation $C_r + IP_r \propto^+ speedup$ and its intermediate steps with an accuracy equal to or above 95%

It is seen that the first source of error in final relation (*i.e.* relation 4.45) is relation 4.42 ($C_r + IP_r \propto^+ Comp_r$). As discussed in chapter 3, the more the probability density function of the transformed image (*i.e.* $Img^{icp}$) follows a Laplace distribution, the higher the accuracy of relation 4.42 will be. The probability density functions of transformed images for very simple images (*e.g.* industrial images) and very noisy images (*e.g.* medical images) tend to match the Laplace distribution less in comparison to natural images. As seen from figure 4.20, for industrial and medical images, the error for relation 4.42 is 5.34% and 6.10%, respectively where for natural and barcode images, the error is 1.22% and 2.10%, respectively.

The other important source of error in final relation (*i.e.* relation 4.45) is relation 4.44 ($HR_{sw} \propto^+ speedup$). As discussed in the previous section, this error arises because of the fact that for very simple images, $t_{mask}$ slightly differs for different images. This causes

that in the speedup equation (equation 4.35), in addition to hit rate, $t_{mask}$ to affect speedup, too. Due to the fact that industrial and barcode images contain simple images, the error in relation 4.44 for these two sets of images is higher than that for natural and medical images. As it is seen from figure 4.20, for industrial and barcode images, the error for relation 4.44 is 9.40% and 4.43%, respectively where for natural and medical images, the error is 0.15% and 0.75%, respectively.

For the final relation ($C_r + IP_r \propto^+ speedup$), figure 4.20 shows that the set of natural images, which contain 40 images of $512 \times 512$ pixels, yields the best total average accuracy (error margin of 2.60%). For this set of images, relation 4.42 ($C_r + IP_r \propto^+ Comp_r$) is the major source of total error. Medical images, which contain 30 images of $280 \times 400$ pixels yield the second best total average accuracy (error margin of 6.41%) where major source of error is also relation 4.42 ($C_r + IP_r \propto^+ Comp_r$). The third best total average accuracy (error margin of 11.01%) is achieved by barcode images, which contain 50 images of $480 \times 640$ pixels where all three relations 4.43, 4.43, and 4.44 contribute to the total error. Finally, industrial images, which contain 8 images of $512 \times 512$ pixels yield the worst total average accuracy (error margin of 22.60%). For this set of images, both relation 4.42 ($C_r + IP_r \propto^+ Comp_r$) and 4.44 ($HR_{sw} \propto^+ Speedup$) are the major sources of total error.

Figure 4.21 shows the speedup versus coding/interpixel redundancies for natural images for all six case study algorithms run on processor 1. The plots for speedup versus coding/interpixel redundancies for industrial, medical, and barcode images for all six case study algorithms run on processor 1 are shown in appendix C, figures C.7, C.8, and C.9, respectively.

The error margins for relation 4.45 ($C_r + IP_r \propto^+ speedup$) for processor 2 is almost the same as that for processor 1. Processor 3, however, yields better accuracy for relation 4.45 because for processor 3, relation 4.44 ($HR_{sw} \propto^+ speedup$) introduces almost negligible error. The reason is that processor 3 is a low-end embedded processor with possibly less optimization at microarchitectural level. Hence, for a given algorithm, it yields similar $t_{mask}$ for all images.

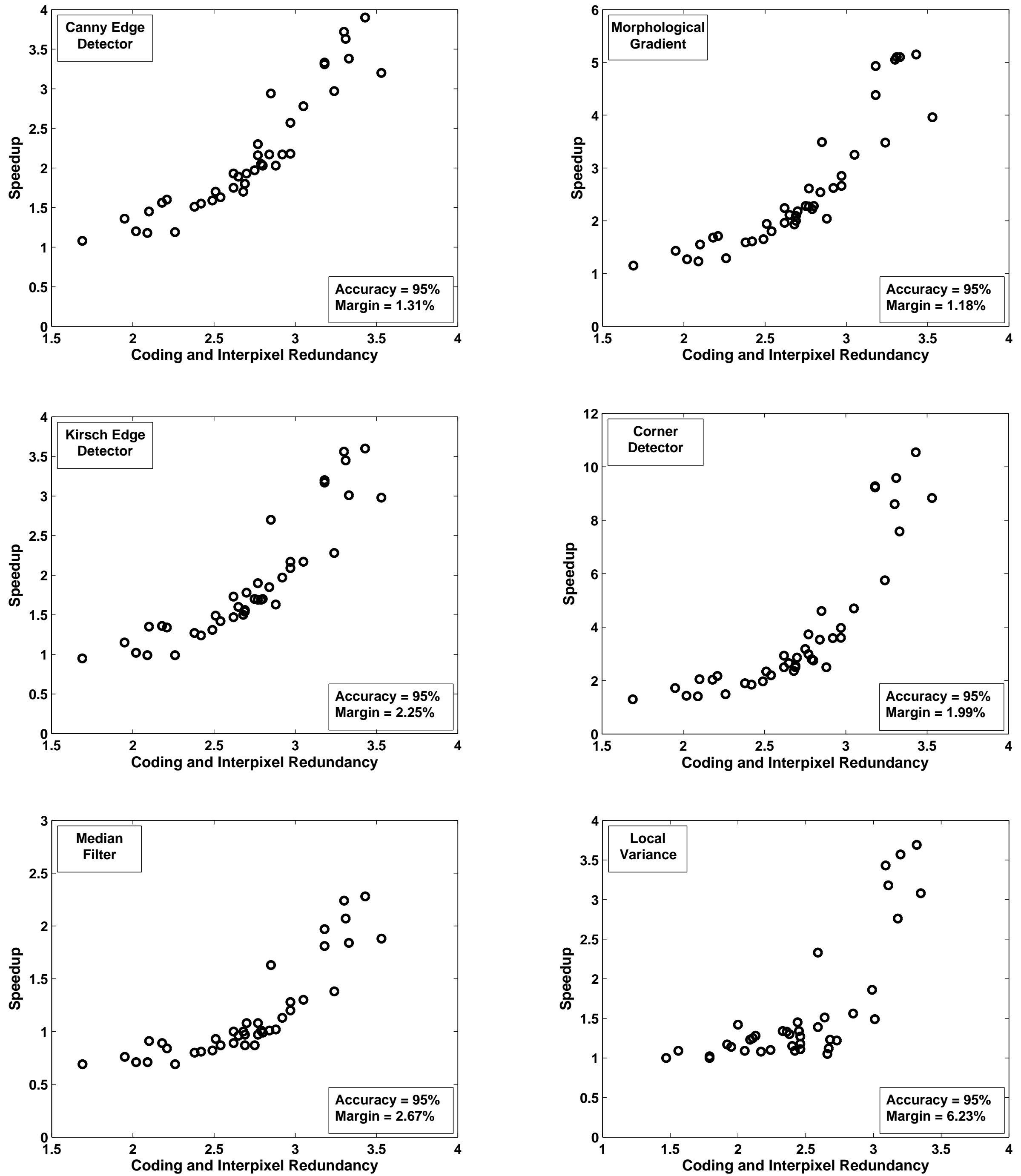



Figure 4.21: Speedup versus coding/interpixel redundancy for natural images run on processor 1

## 4.7 Summary

In this chapter, we presented an optimized architecture for the implementation of window memoization in software. We also presented a model for the speedup obtained by window

memoization in software, which requires the minimum amount of information to predict the speedup of all images in a data set. In order to achieve high speedups, we used tolerant memoization, presented a fast symbol generation mechanism, and used the multiplication method as the hash function for the direct-mapped mapping scheme of the memoization mechanism. We applied the window memoization technique in software to six case study algorithms and demonstrated that our technique can improve the performance of different case study algorithms and input images across different processors significantly (typical speedups: 1.2 to 7.9), while maintaining the accuracy of the results reasonably high (*i.e.* above 96% and $29dB$). Finally, we showed mathematically and empirically that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in software.

# Chapter 5

# Window Memoization in Hardware

In this chapter, we present an architecture for the hardware implementation of the window memoization technique. In hardware, window memoization improves the performance of local image processing algorithms with lower cost (*i.e.* hardware area) in comparison to conventional performance improvement techniques. We implement window memoization in hardware as a 2-wide superscalar pipeline. Conventional 2-wide superscalar pipelines require twice the hardware of a scalar pipeline. In contrast, our superscalar pipeline with window memoization only needs extra hardware to implement the reuse mechanism, rather than duplicating the original hardware. We have applied the window memoization technique to two case study algorithms and implemented them at the register-transfer-level using VHDL. We have used an Altera FPGA as the target device for our designs. For typical images, our technique improves the performance by 58% with 40% less hardware area in comparison to conventional 2-wide superscalar pipelines.

The outline of this chapter is as follows. In section 5.1, we introduce the measure of relative efficiency which is used to compare the performance/area tradeoffs of two designs. In section 5.2, we present the architecture of window memoization in hardware. In section 5.3, the design decisions for window memoization in hardware are presented and discussed. In section 5.4, we present an architecture for parallel reuse tables based on Bloom filters. In section 5.5, we demonstrate tolerant memoization in hardware. In

section 5.6, an optimized architecture for window memoization in hardware is presented. In section 5.7, the results for speedup, sprawl, accuracy, and the relative efficiency for two case study algorithms are given. In section 5.8, we show mathematically that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in hardware. This shows that the coding and interpixel redundancy of images can be used as a measure for classifying images based on their potential performance gain obtained by the window memoization technique in hardware, without any actual implementation of the technique. Finally, section 5.9 gives the summary of the chapter. ‘

## 5.1 Efficiency of a Design

In this section, we introduce a measure, *relative efficiency*, which evaluates the effectiveness of an optimized hardware design with respect to a base design considering two parameters: performance improvement and hardware area cost. The measure enables us to compare different designs not only based on their performance improvement, also based on the hardware area cost that they require to achieve the performance improvement. As it will be discussed in the following sections, in hardware we implement window memoization as a superscalar pipeline. We will use relative efficiency to compare the effectiveness of window memoization applied to image processing algorithms in hardware (superscalar pipeline) against the conventional implementations of the algorithms in hardware (scalar pipeline).

### 5.1.1 Relative Efficiency of an Optimized Design versus a Base Design

We define the efficiency of a hardware design as the ratio of the performance of the design (*perf*) and the hardware area cost of the design (*area*).

$$efficiency = \frac{perf}{area} \tag{5.1}$$

The performance of a hardware design is characterized by *throughput*, which is measured as the number of parcels of data that is produced by the design per unit time [49].

The cost of a hardware design is measured based on the amount of hardware area that the design consumes. In ASIC design, the consumed hardware area is measured based on the amount of the silicon area that the design uses. In FPGA design, the consumed hardware area is measured based on the number of FPGA cells (or logic elements) and memory bits used by the design. In order to unify the amount of hardware area consumed by logic elements and memory bits, the area consumed by memory bits is measured in terms of the hardware area consumed by logic elements. It is estimated that the hardware area that each memory bit in an FPGA consumes is equal to 6% of the area consumed by one logic element in the FPGA [14].

Equation 5.1 represents the effectiveness of one design independent of any other design. In many cases, it is required to measure the relative efficiency of an optimized design with respect to the original design (or base design). We define relative efficiency ($eff_{Rel}$) as the ratio of the efficiencies of the optimized design ($eff_{Opt}$) and the base design ($eff_{Base}$).

$$eff_{Rel} = \frac{eff_{Opt}}{eff_{Base}} \tag{5.2}$$

By substituting the efficiency of the optimized and base design from equation 5.1 into equation 5.2 and considering that the performance of a design is characterized by throughput, we obtain:

$$eff_{Rel} = \frac{\frac{thru_{Opt}}{thru_{Base}}}{\frac{area_{Opt}}{area_{Base}}} \tag{5.3}$$

The ratio in the numerator of equation 5.3 (*i.e.* $\frac{thru_{Opt}}{thru_{Base}}$) is in fact the speedup of the optimized design with respect to the base design.

$$speedup = \frac{thru_{Opt}}{thru_{Base}} \tag{5.4}$$

The ratio in the denominator of equation 5.3 (*i.e.* $\frac{area_{Opt}}{area_{Base}}$) indicates the amount of increase in hardware area cost in the optimized design with respect to the base design. We call the increase in the hardware area of the optimized design with respect to the base design *sprawl*, which is calculated as:

$$sprawl = \frac{area_{Opt}}{area_{Base}} \quad (5.5)$$

The equation for relative efficiency (equations 5.3) can be rewritten in terms of speedup and sprawl (equations 5.4 and 5.5).

$$eff_{Rel} = \frac{speedup}{sprawl} \quad (5.6)$$

Equation 5.6 evaluates the relative efficiency of an optimized design with respect to a base design considering both speedup and sprawl. In many real-world applications, the defining factor for the hardware design is both performance requirements and hardware area limitations. Therefore, when comparing two designs, it is important to consider the performance improvements achieved by the designs (*i.e.* speedup) and the cost that each design had to pay in order to achieve the performance improvements (*i.e.* sprawl).

### 5.1.2 Relative Efficiency of Superscalar Pipeline versus Scalar Pipeline

In digital hardware design, *pipelining* is a fundamental optimization technique, which increases the throughput of a design by overlapping the execution of operations. A pipeline usually contains two parts: *datapath* and *control circuitry*. The datapath is responsible for performing some operations on each parcel of data that enters the pipeline to produce an output parcel. The control circuitry is in charge of handling the internal states of the pipeline based on inputs.

In general, pipelines are categorized into two classes: scalar and superscalar. A pipeline is called *scalar* if its maximum throughput is 1. In other words, a scalar pipeline is able to fetch (and thus to output) at most one parcel of data per clock cycle. A *superscalar* pipeline, on the other hand, can fetch and output at most $n$ parcels of data per clock cycle, which yields a maximum throughput of $n$. A superscalar pipeline that is able to fetch at most $n$ parcels of data per clock cycle is called *$n$-wide*.

The reason that the throughput of a superscalar (scalar) pipeline is not always $n$ (1) is that due to different hazards in a pipeline (*i.e.* structural, data, and control hazard),

the pipeline is not able to fetch new parcels of data at each clock cycle. In order to resolve the hazards, the pipeline must stall and stop fetching new parcels of data for some clock cycles.

An n-wide superscalar pipeline is created by replicating the hardware of a scalar pipeline for $n$ times. Figure 5.1 shows a scalar pipeline and a 2-wide superscalar pipeline. The 2-wide superscalar pipeline (figure 5.1 right) is created by duplicating the hardware of the scalar pipeline. As it can be seen from the figure, the 2-wide superscalar pipeline can fetch and output at most 2 parcels of data per clock cycle. In figure 5.1, *core* represents the hardware of the base design or the scalar pipeline.

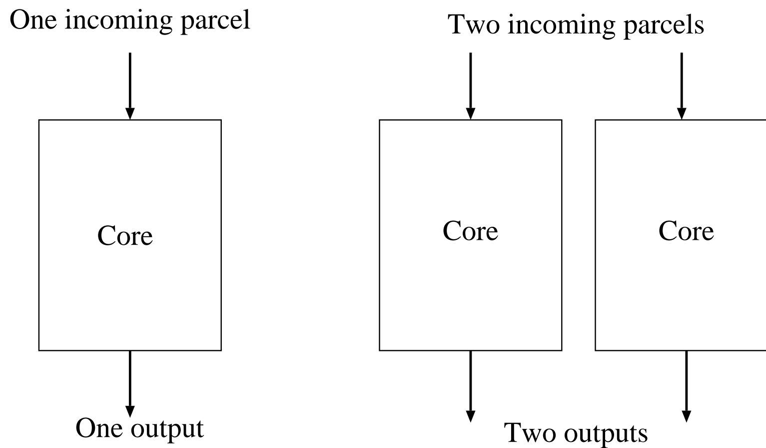



Figure 5.1: Right: scalar pipeline. Left: 2-wide superscalar pipeline.

We can use equation 5.6 to calculate the relative efficiency of an n-wide superscalar pipeline with respect to the scalar pipeline on which the superscalar pipeline has been based. In doing so, the speedup and sprawl of the n-wide superscalar pipeline must be calculated with respect to the scalar pipeline. As equation 5.4 indicates, speedup is calculated as the ratio of the throughput of the superscalar pipeline (optimized design) to the scalar pipeline (base design).

$$speedup = \frac{thru_{superscalar}}{thru_{scalar}} \tag{5.7}$$

In calculating throughput, if it is assumed that the clock speed of the two designs under study (*i.e.* scalar and superscalar pipelines) are the same, instead of seconds, the

clock cycle of the pipeline can be used as unit time. Thus, the throughput of a pipeline can be written as:

$$thru = \frac{pcl}{cyc} \tag{5.8}$$

In the equation above, *pcl* represents the total number of parcels of data that enter the pipeline and *cyc* is the total number of clock cycles elapsed to fetch all *pcl* parcels of data. When comparing a superscalar pipeline to a scalar pipeline, we assume that the total number of parcels of data that enter both pipelines are equal. Therefore, the speedup equation (equation 5.7) can be rewritten as:

$$\begin{aligned} speedup &= \frac{\frac{pcl}{cyc_{superscalar}}}{\frac{pcl}{cyc_{scalar}}} \\ &= \frac{cyc_{scalar}}{cyc_{superscalar}} \end{aligned} \tag{5.9}$$

As the equation above indicates, the speedup of an n-wide superscalar pipeline with respect to a scalar pipeline, assuming that both pipelines process the same amount of data with equal clock speeds, is equal to the ratio of the total number of clock cycles that the scalar pipeline and the superscalar pipeline require to fetch all parcels of data.

As equation 5.5 indicates, sprawl of a n-wide superscalar pipeline with respect to a scalar pipeline is calculated as the ratio of the hardware area consumed by the superscalar pipeline ($area_{superscalar}$) to scalar pipeline ($area_{scalar}$).

$$sprawl = \frac{area_{superscalar}}{area_{scalar}} \tag{5.10}$$

A conventional n-wide superscalar pipeline usually yields a maximum relative efficiency of 1 because it speeds up the computations at most by a factor of $n$ with respect to a scalar pipeline and at the same time, it consumes almost $n$ times the hardware area that a scalar pipeline does. In contrast, as it will be discussed in the upcoming sections, an n-wide superscalar pipeline with window memoization gives a relative efficiency larger

than 1. This means that window memoization is able to improve performance with a cost (hardware area) which is less than that of conventional techniques.

## 5.2 Window Memoization Technique in Hardware

In this section, first, we present the architecture of window memoization in hardware and discuss how speedup and sprawl is calculated for window memoization in hardware. Afterward, we discuss the factors that affect speedup and sprawl and hence the relative efficiency of window memoization in hardware.

### 5.2.1 Architecture of Window Memoization in Hardware

We implement window memoization in hardware as a 2-wide superscalar pipeline. In contrast to conventional 2-wide superscalar pipelines, which are created by duplicating the hardware of a scalar pipeline (*i.e.* two cores), a 2-wide superscalar pipeline with window memoization only adds a memoization mechanism to the scalar pipeline (*i.e.* one core). As a result, instead if having two cores, a 2-wide superscalar pipeline with window memoization has only one core and a memoization mechanism. The memoization mechanism consumes a small amount of hardware area in comparison to the core and increases the throughput of the pipeline

The memoization mechanism includes a reuse table and control circuitry. The reuse table is a dual port memory array (one read and one write port), which is used to store the symbols of incoming windows and the corresponding responses. The control circuitry is responsible for mapping the symbols of the incoming windows into the reuse table and determining whether there is a matching symbol in the reuse table for each incoming window. Depending on whether a hit or miss occurs, the control circuitry is also responsible for stalling the pipeline and deciding which window can enter the core and updating the reuse table with the response of windows generated by the core.

Figure 5.2 shows a 2-wide superscalar pipeline with window memoization. The 2-wide superscalar pipeline with window memoization accepts two windows of pixels as inputs at each clock cycle. One window, $win_A$, tries to find a matching symbol in the reuse table while the other window, $win_B$, is sent through a fifo ($Fifo_1$) to keep both windows in

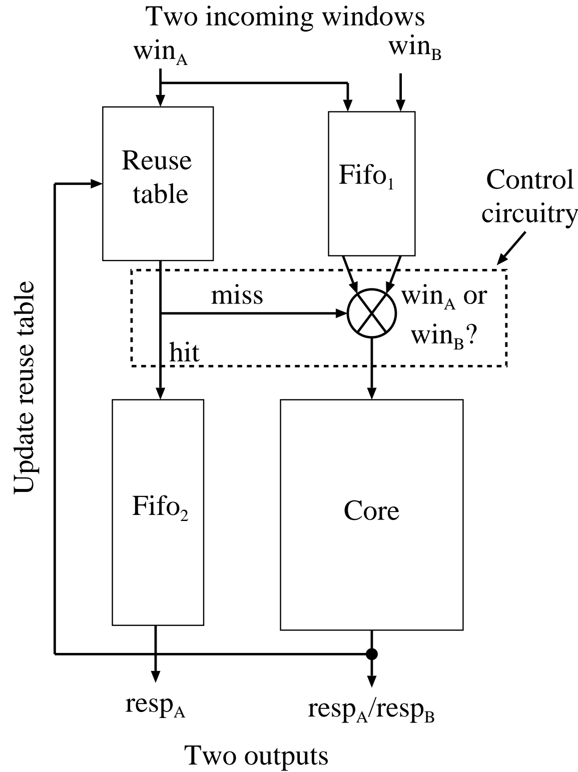



Figure 5.2: 2-wide superscalar pipeline with window memoization

sync. By the time $win_B$ exits $Fifo_1$, the control circuitry has determined whether a hit or miss has occurred for $win_A$. In case of a hit, the control circuitry sends the response of $win_A$ ($resp_A$), which has already been looked up from the reuse table, through $Fifo_2$ and at the same time, $win_B$ is sent to the core. $Fifo_2$ ensures that the responses of both input windows ($resp_A$ and $resp_B$) exit the pipeline at the same clock cycle. In other words, in case of a hit, the pipeline is able to fetch and output two windows per clock cycle.

If the control circuitry determines that $win_A$ was not able to locate a matching symbol in the reuse table (miss), then both windows require using the same resource (core) at the same time, causing a structural hazard. To solve the structural hazard, the control circuitry stalls the pipeline; $win_A$ is sent to the core, followed by $win_B$. As a result, the two responses ($resp_A$ and $resp_B$) exit the pipeline in two consecutive clock cycles. In other words, in case of a miss, the pipeline is able to fetch and output only one window per clock cycle. Finally, the reuse table is updated with the symbol and response of either $win_A$ or $win_B$, depending on the design.

In this architecture, only one input window (*i.e.* $win_A$) is checked against the reuse table. It may seem that checking both input windows against the reuse table can increase hit rate. However, for each pair of input windows, the increase in throughput is the same if either one window hits the reuse table or both windows hit. The reason is that the pipeline can accept at most two input windows per cycle, regardless of the number of hits for a pair of windows. On the other hand, adjacent windows are usually similar in an image. Therefore, if one window hits (misses) the reuse table, it is very likely that its adjacent window will also hit (miss) the reuse table. Thus, in order to make control circuitry simple, it only checks one input window against the reuse table.

### 5.2.2 Speedup of Window Memoization in Hardware

In section 5.1.2, we showed that the speedup of a 2-wide superscalar pipeline with respect to a scalar pipeline is calculated as:

$$speedup = \frac{cyc_{scalar}}{cyc_{superscalar}} \tag{5.11}$$

where $cyc_{scalar}$ and $cyc_{superscalar}$ are the number of clock cycles that the scalar pipeline and the superscalar pipeline require to fetch all parcels of data, respectively. In deriving equation 5.11, we assumed that both pipelines process the same number of parcels of data and both have the same clock speeds.

For a scalar pipeline, assuming that there is no data dependency between the operations, the total number of clock cycles required to fetch all parcels of data, $pcl$, is equal to the total number of the parcels.

$$cyc_{scalar} = pcl \tag{5.12}$$

For a 2-wide superscalar pipeline with window memoization, depending on the mapping scheme used, the pipeline has to stall for some cycles in order to access (read from and/or write to) the reuse table. Assume that on average, in case of a hit or miss, the

number of cycles that the pipeline has to stall due to memory access is $mem_{stall}$ (memory stall). Also, assume that $HR_{hw}^{pair}$ is the number of input pairs of windows that find a matching symbol in the reuse table divided by the total number of pairs of windows that enter the pipeline. If a hit occurs, the pipeline requires one clock cycle to fetch one pair of windows and $mem_{stall}$ clock cycles to access the reuse table. Therefore, the total number of clock cycles required to fetch the pairs of windows, for which a hit will occur is:

$$cyc_{hit} = HR_{hw}^{pair} \times \frac{pcl}{2} \times (1 + mem_{stall}) \tag{5.13}$$

where $pcl$ is the total number of windows and thus, $\frac{pcl}{2}$ is the total number of pairs of windows that enter the pipeline. In case of a miss, the pipeline requires two clock cycles to fetch one pair of windows (because pipeline must stall and both windows must go through the core) and $mem_{stall}$ clock cycles to access the reuse table. Therefore, the total number of clock cycles required to fetch the pairs of windows, for which a miss will occur is:

$$cyc_{miss} = (1 - HR_{hw}^{pair}) \times \frac{pcl}{2} \times (2 + mem_{stall}) \tag{5.14}$$

The total number of clock cycles required to fetch all pairs of windows will be:

$$\begin{aligned} cyc_{superscalar} &= cyc_{hit} + cyc_{miss} \\ &= HR_{hw}^{pair} \times \frac{pcl}{2} \times (1 + mem_{stall}) + (1 - HR_{hw}^{pair}) \times \frac{pcl}{2} \times (2 + mem_{stall}) \\ &= \frac{pcl}{2} \times (2 + mem_{stall} - HR_{hw}^{pair}) \end{aligned} \tag{5.15}$$

We can rewrite the equation above in terms of $HR_{hw}$; the number of input windows that find a matching symbol in the reuse table divided by the total number of windows that enter the pipeline. In the 2-wide superscalar pipeline with window memoization, for each pair of input windows at most only one hit can occur. In other words, out of two incoming windows, only one is checked against the reuse table. The other window is always sent to the core. Therefore, the number of hits for both single windows and

pairs of windows is the same. However, hit rate for pairs of windows (*i.e.* $HR_{hw}^{pair}$) is calculated with respect to the total number of pairs of input windows while hit rate for single windows (*i.e.* $HR_{hw}$) is calculated with respect to the total number of single input windows, which means: $HR_{hw}^{pair} = 2 \times HR_{hw}$. In other words, for a 2-wide superscalar pipeline with window memoization, the maximum hit rate is 50% (*i.e.* $HR_{hw} = 50\%$). Therefore, equation 5.15 can be rewritten as:

$$cyc_{superscalar} = pcl \times (1 + \frac{mem_{stall}}{2} - HR_{hw}) \tag{5.16}$$

Substituting equations 5.12 and 5.16 into the speedup equation (equation 5.11) gives:

$$\begin{aligned} speedup &= \frac{pcl}{pcl \times (1 + \frac{mem_{stall}}{2} - HR_{hw})} \\ &= \frac{1}{1 + \frac{mem_{stall}}{2} - HR_{hw}} \end{aligned} \tag{5.17}$$

Equation 5.17 is the speedup equation for a 2-wide superscalar pipeline with window memoization with respect to a scalar pipeline.

### 5.2.3 Sprawl of Window Memoization in Hardware

In section 5.1.2, we showed that the sprawl of a 2-wide superscalar pipeline with respect to a scalar pipeline is calculated as:

$$sprawl = \frac{area_{superscalar}}{area_{scalar}} \tag{5.18}$$

where $area_{superscalar}$ and $area_{scalar}$ are the hardware area consumed by the superscalar pipeline and scalar pipeline, respectively. For a scalar pipeline, $area_{scalar}$ is the total number of logic elements consumed by the pipeline, which we call $LE_{core}$.

$$area_{scalar} = LE_{core} \tag{5.19}$$

For a 2-wide superscalar pipeline with window memoization, in addition to the core, the pipeline also includes the memoization mechanism, which consists of the control

circuitry, the reuse table and two fifos. Therefore, the total hardware area consumed by a 2-wide superscalar pipeline with window memoization will be:

$$area_{superscalar} = LE_{core} + LE_{ctl} + 0.06 \times mem_{bits} \tag{5.20}$$

where $LE_{core}$ and $LE_{ctl}$ are the number of logic elements consumed by the core and control circuitry, respectively. $mem_{bits}$ is the total number of memory bits used by the reuse table and the two fifos. In equation 5.20, the coefficient 0.06 reflects the fact that the hardware area that each memory bit in an FPGA consumes is equal to 6% of the area consumed by one logic element in the FPGA [14].

Substituting equations 5.19 and 5.20 into the sprawl equation (equation 5.18) gives:

$$sprawl = \frac{LE_{core} + LE_{ctl} + 0.06 \times mem_{bits}}{LE_{core}} \tag{5.21}$$

Equation 5.21 is the sprawl equation for a 2-wide superscalar pipeline with window memoization with respect to a scalar pipeline.

## 5.3 Design Decisions for Window Memoization in Hardware

In this section, we discuss the factors that affect the speedup and sprawl of window memoization in hardware. For each factor, we present an optimized design, which achieves high speedup and low sprawl.

The factors that affect speedup are the mapping scheme used by the control circuitry, the core latency, and the reuse table size. Sprawl is affected by the sizes of the reuse table and fifos, the control circuitry, and the core. The control circuitry is responsible for mapping the symbols of incoming windows into the reuse table using a mapping scheme, determining whether a hit or miss occurs, stalling the pipeline (in case of a miss), and updating the reuse table. The reuse table stores the symbols and responses of windows and the two fifos keep the windows in order throughout the pipeline. The core size is the

hardware area consumed by the original scalar pipeline, and finally, the core latency is the number of clock cycles from the input of the core to its output.

In table 5.1, the list of parameters that affect speedup and sprawl is shown. Fifos used in window memoization are usually small and hence, at this stage, their effect on sprawl can be ignored. Therefore, they have not been listed in the table. In the following sections, we will explain how each component/factor affects speedup or sprawl. For each component/factor, we will present a solution to achieve high speedup and low sprawl.

Table 5.1: Design decisions for memoization mechanism in hardware

| | Mapping Scheme | Core Latency | RT Size | Control Circuitry | Core Size |
|---|---|---|---|---|---|
| Speedup | √ | √ | √ | | |
| Sprawl | | | √ | √ | √ |

### 5.3.1 Speedup

Speedup is affected by the mapping scheme, core latency, and the reuse table size. The effect of the reuse table size on hit rate and hence on speedup is obvious. A larger reuse table will produce a higher hit rate. In the following sections, we study the effect of the mapping scheme and core latency on speedup in more details. For each case, we present a solution, which yields high speedups.

#### Mapping Scheme

The equation for speedup (equation 5.17) indicates that in addition to hit rate ($HR_{hw}$), the number of pipeline stalls caused by the reuse table access ($mem_{stall}$) also affects speedup. For a 2-wide superscalar pipeline with window memoization, with the maximum hit rate (*i.e.* $HR_{hw} = 50\%$), in order to achieve a speedup of at least 1, $mem_{stall}$ must be equal to or less than 1. In reality, hit rate is usually below 50% (*e.g.* 35%), which means that $mem_{stall}$ larger than 0 may lead to speedups below 1. Therefore, the mapping scheme for a 2-wide superscalar pipeline with window memoization should not impose any extra pipeline stalls. Similar to window memoization in software (section 4.2.2), among the

three mapping schemes in processor cache hierarchy design (*i.e.* direct-mapped, fully associative, and set-associative), only direct-mapped mapping scheme is able to give $mem_{stall}$ of 0 for window memoization in hardware. Fully associative and set-associative mapping schemes require multiple clock cycles in order to map/read an entry into/from memory. As a result, the hardware implementation of window memoization uses a direct-mapped mapping scheme for the memoization mechanism. With a direct-mapped mapping scheme (*i.e.* $mem_{stall} = 0$), the speedup of a 2-wide superscalar pipeline with window memoization is calculated as:

$$speedup = \frac{1}{1 - HR_{hw}} \tag{5.22}$$

**Core Latency**

The latency of the scalar pipeline (*core*) to which the window memoization technique is applied can affect hit rate and hence speedup. The core latency causes the response of a window to be generated and the reuse table to be updated a few cycles after the window enters the pipeline. On the other hand, many similar windows in an image are usually located very close to each other. This is due to the fact that neighboring windows usually belong to one object or background in the image and hence, they contain similar pixels. When the immediate following windows, which are most probably similar to the current window, enter the pipeline, the reuse table has not been updated yet with the response of the current window. As a result, many potential hits are turned into misses.

In order to eliminate the effect of the core latency on hit rate, we use two levels of reuse tables instead of one. The level 1 reuse table, which is located at the top of the pipeline stores the symbols of the incoming windows and the level 2 reuse table, which is located at the bottom of the pipeline stores the responses of the incoming windows. When a window enters the pipeline for the first time, its symbol is inserted into the level 1 reuse table immediately while it takes a few cycles, due to the core latency, to generate the response of the window and update the level 2 reuse table. As a result, as soon as another window enters the pipeline, it is determined whether a hit or miss occurs. Nevertheless, the response is actually reused (looked up) a few cycles later. This enables a current window to find a matching symbol that belongs to windows that are spatially

close to the current window, eliminating the effect of the latency of the core on hit rate and leading to high speedups.

It must be mentioned that the total number of memory bits used by a 2-level reuse table is the same as that consumed by a single reuse table. Therefore, using a 2-level reuse table instead of a single reuse table does not affect sprawl.

### 5.3.2 Sprawl

Sprawl is affected by the size of the control circuitry, core, and the reuse table. The equation of sprawl (equation 5.21) can be rewritten as:

$$sprawl = 1 + \frac{LE_{ctl} + 0.06 \times RT_{bits}}{LE_{core}} \tag{5.23}$$

In the equation above, $mem_{bits}$ have been replaced by $RT_{bits}$ because the number of memory bits consumed by the two fifos are usually negligible.

As equation 5.23 indicates, in order to achieve low sprawl, the relative size of the control circuitry and memory bits used by window memoization should be small with respect to the original scalar pipeline (*i.e.* small $\frac{LE_{ctl}}{LE_{core}}$ and $\frac{RT_{bits}}{LE_{core}}$). A typical core consumes about 1000 logic elements (*i.e.* $LE_{core} = 1000$). Using conventional digital design optimization techniques, the control circuitry can be optimized such that the hardware area that it consumes is no more than 15% of a typical core (*i.e.* $\frac{LE_{ctl}}{LE_{core}} = 0.15$). Nevertheless, for small cores (*e.g.* $LE_{core} < 250$), obtaining low sprawls might require further reducing the amount of hardware area consumed by the control circuitry.

To obtain small $\frac{RT_{bits}}{LE_{core}}$, the reuse tables must be space-efficient. Window memoization in hardware uses the direct-mapped mapping scheme. In direct-mapped mapping method, a portion of each incoming window's symbol is used as an address to the reuse table while the remaining bits of symbol along with the response of the window are stored in the reuse table. The conventional implementations of the reuse table usually consume a large amount of hardware area leading to high sprawls.

For a typical image, to achieve speedup of 1.50, the reuse table size must be above 512 entries. Assuming that for windows of $3 \times 3$ only 2 most significant bits are used to

determine whether a window belongs to a symbol, each symbol will have $3 \times 3 \times 2 = 18$ bits. For a reuse table of 512 entries, 9 bits of symbols will be used as address to the reuse table. The remaining 9 bits will be stored in the reuse table to check whether there is a hit. Assuming that the responses of the windows are binary, the reuse table will have $(9 + 1) \times 512 = 5120$ bits, which is equal to the hardware area consumed by 307 logic elements. This amount of hardware area adds to the sprawl and thus, degrades the relative efficiency of the design. For a typical core ($LE_{core} = 1000$) and assuming that the control circuitry consumes 150 logic elements, sprawl will be $1 + \frac{1000+150+307}{1000} = 1.46$. For typical speedups of 1.50, the relative efficiency will be $\frac{1.50}{1.46} = 1.03$. Our goal is to achieve higher relative efficiencies by reducing sprawl.

To achieve low sprawls, we present a space-efficient architecture for the reuse table, which is based on *parallel Bloom filters*. We will discuss our architecture for space-efficient reuse table in section 5.4. In section 5.5, we will present tolerant memoization in hardware where, similar to window memoization in software, we will determine the optimal number of most significant bits of pixels used in assigning windows to symbols, which gives high hit rates and high accuracy of results. We will also show how the tolerant memoization is implemented with parallel reuse tables.

In section 5.3.1, we discussed that in order to achieve high speedups, the direct-mapped mapping scheme and a 2-level reuse table must be used. In section 5.6, we will present an optimized architecture for window memoization in hardware, which takes into account all the design decisions discussed in section 5.3.1 (*i.e.* direct-mapped mapping scheme and a 2-level reuse table), in this section (*i.e.* parallel reuse tables), and in section 5.6 (*i.e.* tolerant memoization). The optimized architecture of window memoization in hardware uses a direct-mapped mapping scheme, a 2-level parallel reuse tables based on Bloom filters, and tolerant memoization to achieve high speedups and low sprawls. Finally, in section 5.7, we will present the results on speedup, accuracy, sprawl, and relative efficiency of designs.

## 5.4 Parallel Reuse Tables Based on Bloom Filters

In this section, we present a space-efficient architecture for the reuse table used by the hardware implementation of window memoization. Our architecture, which is based on *parallel Bloom filters* consumes a considerably smaller number of bits compared to conventional architectures. It yields high hit rates with a small number of false positive hits. In the following sections, first, a brief background on Bloom filters is given. Afterward, we present the architecture of parallel reuse tables based on Bloom filters.

### 5.4.1 Parallel Bloom Filters

A Bloom filter is a space-efficient data structure, which is used to determine whether an element is a member of a set [3]. For a set $S$ of $n$ elements, the Bloom filter is an array of $z$ bits, which are all initially set to 0. In order to insert each element $x$ in $S$ into the Bloom filter, $k$ independent hash functions are used to generate $k$ hash_keys in the range $\{1, .., z\}$ and all $k$ bits in the Bloom filter in the locations where the $k$ hash_keys point to are set to 1. To look up an element from the Bloom filter, first, $k$ hash_keys are generated. Afterward, all the bits in the Bloom filter in the location where the $k$ hash_keys point to are read. If all the bit are set to 1 then it is assumed that the element is in the set $S$. Otherwise, the element is not in the set $S$ [4].

The structure of the Bloom filter is such that it may lead to a *false positive*, meaning that the Bloom filter may suggest that an element is in the set $S$ while it is not. This is due to the fact that the hash_keys of one element may find all the corresponding bits in the Bloom filter set to 1 while those bit have been set to 1 by multiple inserted elements rather than by just one element. Assuming that the hash functions map each element in the set $S$ to a random number in the range $\{1, .., z\}$ uniformly, the probability of a false positive ($FP$) is calculated as [4]:

$$FP = (1 - (1 - \frac{1}{z})^{k \times n})^k \tag{5.24}$$

where $z$ is the size of the Bloom filter, $n$ is the number of inserted elements, and $k$ is the number of hash_keys generated for each element.

For parallel Bloom filters, the standard Bloom filter is divided by $k$ equal filters where each filter has $\frac{z}{k}$ bits. Each filter can be accessed in parallel with the other filters and hence, the access time is much less than that of the standard Bloom filter. With the same assumption that the hash functions map each element in the set $S$ to a random number in the range $\{1, .., \frac{z}{k}\}$ uniformly, the probability of a false positive ($FP$) for parallel Bloom filter is calculated as [4]:

$$FP = (1 - (1 - \frac{k}{z})^n)^k \tag{5.25}$$

Because $(1 - \frac{k}{z})^n \leq (1 - \frac{1}{z})^{k \times n}$, the probability of false positives of parallel Bloom filters is slightly higher than that of a standard Bloom Filter [6] [4].

Parallel Bloom filters require much less space than normal reuse table. To achieve maximum hit rates with a regular reuse table, elements of $b$ bit wide require $2^b$ bit memory. In contrast, parallel Bloom filters can use, for example, 2 reuse tables in parallel, each of $2^{\frac{b}{2}}$ bits, which means that the total number of the consumed bits will be $2 \times 2^{\frac{b}{2}} = 2^{1+\frac{b}{2}}$. Thus, parallel Bloom filters will require $\frac{2^{1+\frac{b}{2}}}{2^b} = 2^{1-\frac{b}{2}}$ the space (*i.e.* the number of bits) that a regular reuse table does. For example, if the input elements are 10 bit wide (*i.e.* $b = 10$), parallel Bloom filters will require $2^{1-\frac{10}{2}} = \frac{1}{16}$ the space that a regular reuse table consumes.

### 5.4.2 Parallel Reuse Tables as Parallel Bloom Filters

In this section, we present a space-efficient architecture for the reuse table based on parallel Bloom filters. We also present an analytical approach to estimate the probability of false positive hits. Due to the nature of image processing data (*i.e.* correlation between pixels in a window), the analytical probability of false positives is much less than that of empirical data. Nevertheless, the analytical method can be used to explore the effect of different parameters on the probability of false positives.

The major difference between our architecture and the regular parallel Bloom filters is that each entry in the regular parallel Bloom filters is a single bit. In contrast, in our architecture, each entry can be multiple bits. As we will discuss shortly, having wider entries will reduce probability of false positives.

In section 5.3.1, we presented a 2-level architecture for the reuse table, which eliminates the effect of the core latency on speedup. The level 1 reuse table stores the symbols of the windows to determine hit/miss and the level 2 reuse table keeps the responses of the windows. To reduce the amount of hardware area that the reuse table consumes, for the level 1 reuse table, we use multiple memory arrays in parallel, rather than a single memory array. The level 2 reuse table uses a single memory array as before. For windows of $m \times m$ pixels, $m^2 - 1$ reuse tables ($RT_1$, $RT_2$,...,$RT_{m^2-1}$) are used to store the symbols of the windows (*i.e.* the level 1 reuse tables). For each pixel in the window ($pix_i$), except the central pixel, there is a corresponding reuse table ($RT_i$). From each pixel $pix_i$, a hash_key is generated as an address to each corresponding reuse table $RT_i$. A portion of the (or the whole) central pixel in the window is stored in all reuse tables, in locations where the corresponding *hash_keys* point to (Figure 5.3).

In order to check whether a current window matches a symbol already stored in the level 1 reuse tables, first, the contents of all level 1 reuse tables at locations where the corresponding hash_keys of the current window point to (*i.e.* $RT_1(key_1)$, $RT_2(key_2)$, ..., $RT_8(key_8)$) are read. If all the values read from the level 1 reuse tables are equal to the hash_key of the central pixel of the current window ($key_0$), then a hit occurs. Otherwise, a miss will occur. In case of a hit, the response is read from the level 2 reuse table, using $key_0$ as the address.

The response of the window is stored in a separate reuse table (level 2 reuse table) whose address is a hash_key generated from a certain pixel in the window (*e.g.* central pixel). In fact, the address to the level 2 reuse table that holds responses could be extracted from any pixel in the window so along as the same pixel is used all the time.

To insert the symbol of a window into the reuse tables, the hash_key generated from the central pixel $key_0$ is inserted to the level 1 reuse tables at locations where the corresponding hash_keys of the current window point to (*i.e.* $RT_1(key_1)$, $RT_2(key_2)$, ..., $RT_8(key_8)$). To insert the response of a window into the level 2 reuse table, $key_0$ is used as the address to the level 2 reuse table.

Figure 5.3 shows the parallel reuse tables for windows of $3 \times 3$ pixels.

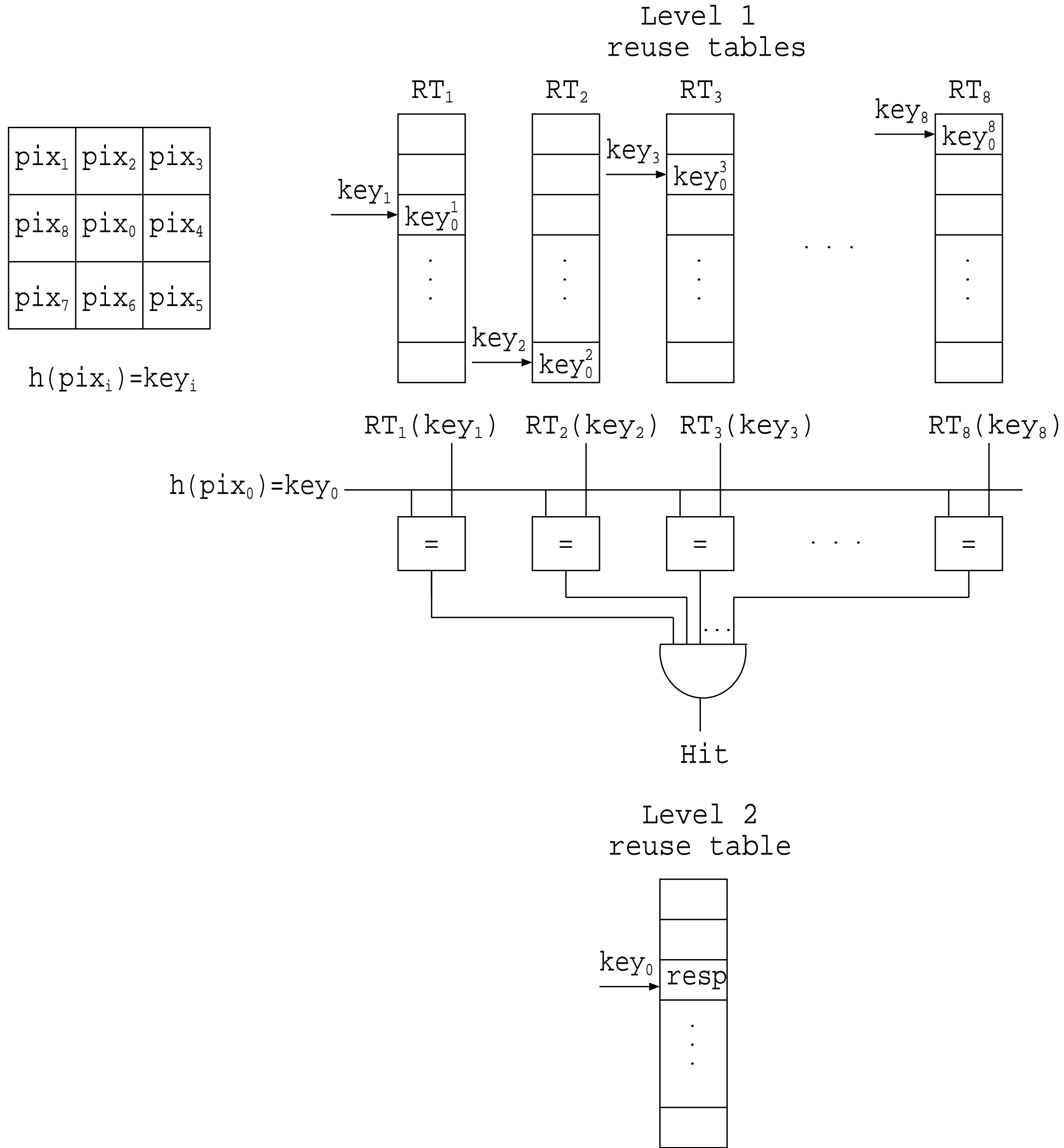



Figure 5.3: Parallel reuse tables for window memoization in hardware

### 5.4.3 The Probability of False Positives for Parallel Reuse Tables

Similar to regular parallel Bloom filters, the parallel reuse tables may cause false positives because certain locations in the level 1 reuse tables, which belong to an incoming window may have been filled by different windows. Assume that the length and width of each reuse table are $RT_{length}$ and $RT_{width}$, respectively. For windows of $m \times m$ pixels, there will be $m^2 - 1$ reuse tables. Assuming that the hash_keys are random numbers uniformly distributed over the range $\{1, 2, ..., RT_{length}\}$, the probability that a certain location in

reuse table $RT_i$ has not been inserted with any value is $1-\frac{1}{RT_{length}}$. Each reuse table entry is $RT_{width}$ bits wide. Thus, each entry can have $2^{RT_{width}}$ possible values. The probability that a certain location in reuse table $RT_i$ has not been inserted with a certain value is $1 - \frac{1}{RT_{length} \times 2^{RT_{width}}}$. After inserting $n$ symbols, the probability that a certain location in reuse table $RT_i$ has not been inserted with a certain value is $(1 - \frac{1}{RT_{length} \times 2^{RT_{width}}})^n$. Thus, the probability that a certain location in reuse table $RT_i$ has been inserted with a certain value is $1 - (1 - \frac{1}{RT_{length} \times 2^{RT_{width}}})^n$. The probability that certain locations in all $m^2 - 1$ reuse tables have been inserted with a certain value will be $(1 - (1 - \frac{1}{RT_{length} \times 2^{RT_{width}}})^n)^{m^2-1}$. Given that all hash_keys are assumed to be random numbers, the probability that certain locations in all $m^2 - 1$ reuse tables have been inserted with a certain value indicates the probability of a false positive because random numbers cannot all point to locations that represent a certain value. Thus, the probability of a false positive $FP$ will be:

$$FP = (1 - (1 - \frac{1}{RT_{length} \times 2^{RT_{width}}})^n)^{m^2-1} \tag{5.26}$$

From the equation above, it is seen that the probability of false positives in parallel reuse tables depends on the reuse table size ($RT_{length}$ and $RT_{width}$), the number of inserted symbols ($n$) and the size of the windows (or the number of hash functions $m^2 - 1$). A larger reuse table and/or window will decrease the probability of false positives. As the number of inserted symbols increases, the probability of false positives increases as well.

In contrast to regular parallel Bloom filters whose entries are 1 bit wide, the parallel reuse tables ($RT_{width}$) can have multi-bit wide entries. From equation 5.26, it is seen that increasing $RT_{width}$ is more effective in reducing false positives than increasing $RT_{length}$ while both $RT_{width}$ and $RT_{length}$ affect the total number of memory bits equally. In other words, having multi-bit entry reuse tables decreases the probability of false positives with no extra cost in memory size in comparison to single-bit entry reuse tables (*i.e.* regular parallel Bloom filters).

Figure 5.4 shows the probability of false positives versus the total number of memory bits used for two approaches: multi-bit entry reuse tables and single-bit entry reuse tables. In figure 5.4, it has been assumed that the number of inserted elements is 1000 ($n = 1000$)

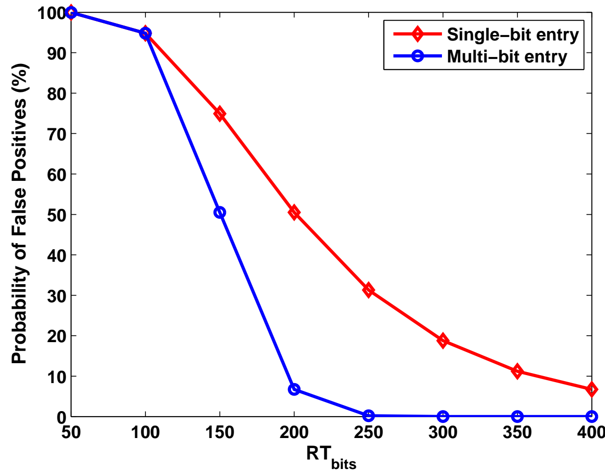



Figure 5.4: The probability of false positives versus $RT_{width}$

and the input windows are $3 \times 3$ pixels. It is seen that for the same number of memory bits, the multi-bit entry reuse tables give lower probability of false positives compared to the single-bit entry reuse tables.

## 5.5 Tolerant Memoization in Hardware

Similar to software, tolerant memoization in hardware allows us to use the $d$ most significant bits of each pixel in a window to determine whether the window belongs to a symbol. Combining the tolerant memoization and parallel reuse tables, we use the $d$ most significant bits of each pixel in a window as hash_keys for the level 1 reuse tables (*i.e.* $key_1$, $key_2$, ...,$key_8$) and the level 2 reuse table (*i.e.* $key_0$) where $key_i = MSB(d, pix_i)$. This means that the length and width of reuse table will be equal to $2^d$ and $d$, respectively ($RT_{length} = 2^d$, $RT_{width} = d$). On the other hand, in a 2-wide superscalar with window memoization, one window of each pair of input windows always update the reuse tables. Therefore, the reuse tables almost always hold only the symbols and responses of the most recent windows. For reuse tables of $2^d$ long, at each snapshot, the symbols of last $2^d$ windows are held. As a result, at each time that a symbol is checked against the level 1 reuse tables, the number of inserted symbols to the reuse tables is at most $2^d$ ($n = 2^d$). The equation of probability of false positives (equation 5.26) can be rewritten as:

$$
\begin{aligned}
FP &= (1 - (1 - \frac{1}{2^d \times 2^d})^{2^d})^{m^2-1} \\
&= (1 - (1 - 2^{-2\times d})^{2^d})^{m^2-1}
\end{aligned}
\tag{5.27}
$$

The total number of memory bits used by the level 1 reuse tables will be $(m^2 - 1) \times (2^d \times d)$. The level 2 reuse table, which is used to store the responses of windows, will require up to $(2^d \times 8)$ bits (8 bits required to store a gray level response). Thus, the total number of bits consumed by all reuse tables will be:

$$
\begin{aligned}
RT_{bits} &= (m^2 - 1) \times (2^d \times d) + 2^d \times 8 \\
&= ((m^2 - 1) \times d + 8) \times 2^d
\end{aligned}
\tag{5.28}
$$

In order to pick an optimal number of most significant bits used for assigning windows to symbols ($d$), we run a simulation to calculate both speedup and results accuracy for different $d$ (1 to 8) on the set of natural images. We use the ideal algorithm to generate the response of windows. As defined in section 4.2.4, the ideal algorithm outputs the central pixel of each window as the response of the window. In calculating SNR, we have replaced an infinite SNR with 100, in order to calculate the average of SNRs of all images. Figure 5.5 shows the average SNR and speedups of natural images versus the number of bits used for comparison for the ideal algorithm. It can be seen that using a $d$ of 2 yields an average SNR over $30dB$ (*i.e.* $32.19dB$) with an average speedup of 1.58.

In chapter 4, it was shown that in most cases, using 4 most significant bits of pixels for comparison (*i.e.* $d = 4$) yields reasonably high accuracy of results, which is higher than that required for hardware (*i.e.* $d = 2$). There are two reasons for this difference. First, in hardware, at most 50% of window are reused and hence, the mask operations set are applied to at least half of windows. Second, in hardware, the reuse table is updated by every other window in the image. Thus, whenever a window is checked against the reuse table, it is very likely that the reuse table holds its nearby neighbors, which are similar to the window. Therefore, window memoization in hardware can achieve the same accuracy

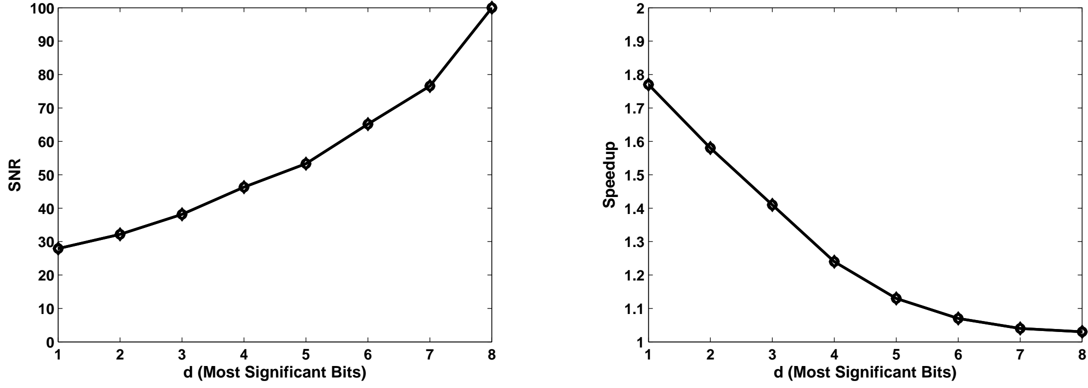



Figure 5.5: Left: average SNR versus the number of the most significant bits used for assigning windows to symbols. Infinite SNRs have been replaced by SNR of 100. Right: speedup versus the number of the most significant bits used for assigning windows to symbols.

of results as in software with fewer bits used to determine whether a window belongs to a symbol.

For windows of $3 \times 3$ pixels ($m = 3$), if 2 most significant bits are used for comparison ($d = 2$), the probability of false positives will be (equation 5.27):

$$\begin{aligned} FP &= (1 - (1 - 2^{-2\times d})^{2^d})^{m^2-1} \\ &= (1 - (1 - 2^{-2\times 2})^{2^2})^{3^2-1} \\ &= 7.18 \times 10^{-4}\% \end{aligned} \tag{5.29}$$

However, as it will be shown in section 5.7, the probability of false positives for empirical data is much higher than the analytical result (*i.e.* average $FP$ of 4.28%). The reason is that the hash_keys in window memoization (*i.e.* the pixels of windows) are not random variables.

The total number of bits of the reuse tables for window memoization in hardware where $d = 2$ and $m = 3$ will be (equation 5.28):

$$\begin{aligned} RT_{bits} &= ((m^2 - 1) \times d + 8) \times 2^d \\ &= ((3^2 - 1) \times 2 + 8) \times 2^2 = 96 \end{aligned} \tag{5.30}$$

The equivalent number of logic elements for 96 memory bits is almost 6, which is negligible in comparison to a typical core with 1000 logic elements.

## 5.6 An Optimized Architecture for Window Memoization in Hardware

In the previous sections (*i.e.* sections 5.3.1, 5.3.2, 5.4, and 5.5), we discussed the factors that affect speedup and sprawl of a 2-wide superscalar pipeline with window memoization. To achieve high speedups, we chose the direct-mapped method as the mapping scheme (section 5.3.1). To eliminate the effect of the core latency on speedup, we proposed a 2-level reuse table where symbols and responses of windows are stored in different levels of reuse tables (section 5.3.1). To achieve low sprawl, we proposed parallel reuse tables which are based on parallel Bloom filters (sections 5.3.2 and 5.4). Finally, we explored different $d$ (number of the most significant bits of pixels) to be used for comparison and found out that by using only 2 most significant bits of pixels, speedup can be increased further while maintaining high accuracy of results (section 5.5). We also showed that how the tolerant memoization is implemented using the parallel reuse tables (section 5.5). In this section, we present an optimized architecture for window memoization in hardware, which considers all the design decisions discussed in the previous sections to yield high speedups and low sprawl.

Figure 5.6 shows the optimized architecture of a 2-wide superscalar pipeline with window memoization in hardware. The difference between figure 5.6 and figure 5.2 in section 5.2.1 is that figure 5.6 uses a 2-level reuse table rather than a 1-level. As it can be seen from figure 5.6, the architecture includes one core to implement the image processing algorithm, control circuitry, two fifos ($Fifo_1$ and $Fifo_2$), and two levels of the reuse tables. At each cycle, two windows enter the pipeline: $win_A$ and $win_B$. Out of each pair of input windows ($win_A$ and $win_B$), one window (*i.e.* $win_A$) is always checked against the level 1 reuse tables to determine whether there is a matching symbol in the reuse table. The other window (*i.e.* $win_B$) is always sent to the core and never checked against the reuse table. As a result, the level 1 reuse tables are always read by $win_A$ and in case of a hit, the level 2 reuse table is also read by $win_A$. On the other hand, the level

1 and the level 2 reuse tables are always updated with the symbol and response of $win_B$, respectively.

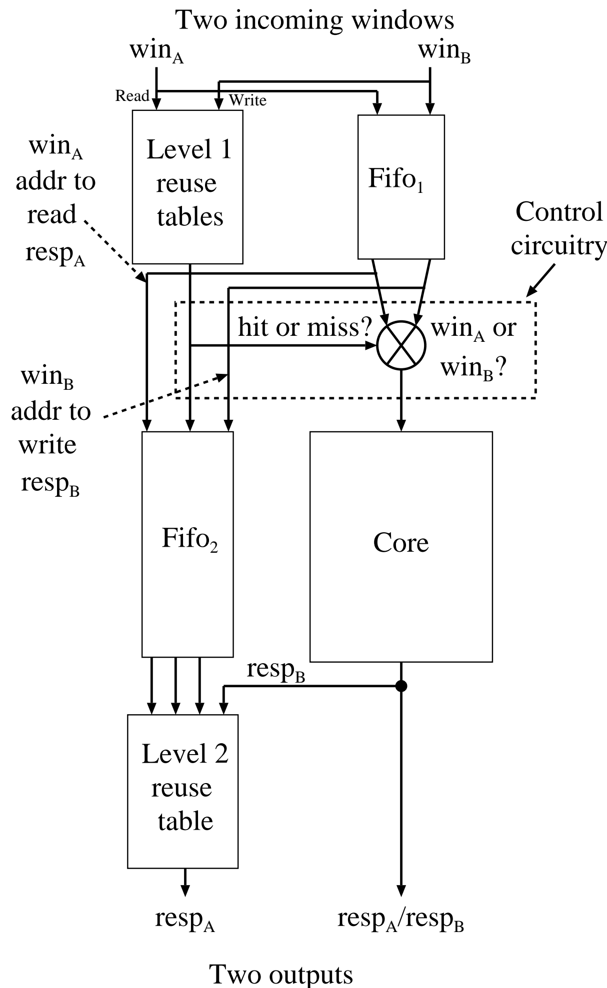



Figure 5.6: An optimized architecture for 2-wide superscalar pipeline with window memoization

At the same cycle that $win_A$ is checked against the level 1 reuse tables, the symbol of $win_B$ is written into the level 1 reuse tables. At this stage, the response of $win_B$ has not been generated yet. Both windows are also sent through $Fifo_1$ in order to keep them in sync with the flow of the pipeline. By the time that the control circuitry determines whether $win_A$ has found a matching symbol in the level 1 reuse tables, both windows have exited $Fifo_1$. In case of a hit, $win_B$ is sent to the core and the central pixel of $win_A$ is sent through $Fifo_2$ to be used as the address to the level 2 reuse table to look up the response of $win_A$ ($resp_A$). When the core generates the response of $win_B$ ($resp_B$),

$resp_A$ has also been read from the level 2 reuse table. Therefore, both responses exit the pipeline at the same cycle. Finally, the level 2 reuse table is updated by $resp_B$, at the location where the central pixel of $win_B$ points to.

If $win_A$ is not able to find a matching symbol in the level 1 reuse tables (miss) the pipeline stalls to prevent new windows from entering the pipeline. In the stalled cycle, $win_A$ is sent to the core first to generate its response ($resp_A$). Following $win_A$, $win_B$ is also sent to the core to generate $resp_B$. The responses of both window will exit the core consecutively in two cycles. The level 2 reuse table is updated by $resp_B$.

It should be noted that in our optimized architecture for window memoization, regardless of whether a miss or hit occurs, the reuse tables are updated by only $win_B$ and its response $resp_B$. In case of hit, $win_A$ is already in the reuse tables and thus, it is beneficial to update the reuse tables with a new information, $win_B$. In case of miss, if the reuse tables are updated by $win_A$ instead of $win_B$, the control circuitry will be slightly less complicated compared to the scenario where the reuse tables are updated only by $win_B$. However, our simulations show that updating the reuse tables by $win_A$ (in case of a miss) decreases hit rate (and hence speedup) slightly. Therefore, the overall efficiency of the design remains essentially unchanged regardless of the design decision.

## 5.7 Results

In this section, we present the speedup, sprawl and accuracy results for the optimized architecture of window memoization in hardware. Both speedup and sprawl are calculated for a 2-wide superscalar pipeline with window memoization for a case study algorithm and compared to the scalar pipeline of the case study algorithm. Speedup is independent of the case study algorithm and it only depends on the input images. Sprawl depends on the hardware area consumed by the scalar pipeline of the case study algorithm (core) and the memoization mechanism. Finally, accuracy of results depends on both the case study algorithm and the input images.

We have implemented all designs (*i.e.* scalar pipeline and 2-wide superscalar pipeline with window memoization for each case study algorithm) at the register-transfer-level (RTL) using VHDL on an Altera FPGA. We have chosen the Kirsch edge detector and

median filter as our case study algorithms. These two algorithms consume small hardware area compared to other algorithms used as case studies in software. Therefore, they give the upper-bound for sprawl (small $LE_{core}$ in equation 5.23) and thus, the lower-bound for the efficiency. Moreover, these algorithms produce both binary and gray-level outputs, which allows us to evaluate the accuracy for both classes of results. We have run simulations to calculate the speedups obtained for all four sets of images (natural, industrial, medical, and barcode images). In addition, the sprawl of each design has been calculated using the number of logic elements and memory bits reported by the synthesis tool. The clock speeds for all designs, including the two cores (*i.e.* scalar pipelines for Kirsch and median) and the two 2-wide superscalar pipelines with window memoization for Kirsch and median are the same at $235MHz$.

### 5.7.1 Speedup and Accuracy Results

We have run the hardware simulation for all four sets of images and measured the speedups and the accuracy of results. Figure 5.7 shows the average hit rates and false positives for each set of images. It is seen that as the complexity of images grow, the probability of false positives increases (least complex images, industrial, $FP = 1.75\%$, most complex images, medical, $FP = 7.89\%$).

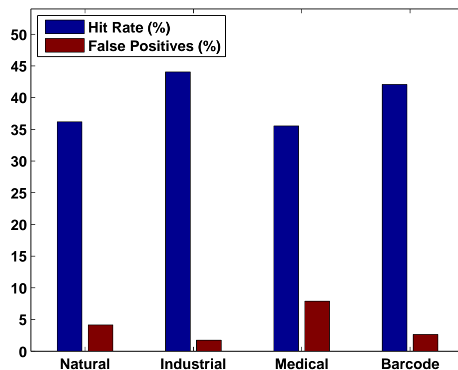



Figure 5.7: Average hit rates and false positives

Speedup is calculated based on hit rate using the equation below:

$$speedup = \frac{1}{1 - HR_{hw}} \tag{5.31}$$

Table 5.2 shows the average speedups and accuracy of the results for each of the sets of images for the Kirsch edge detector and median filter. To calculate the accuracy of the results, for the Kirsch edge detector, we use misclassification error (given in section 4.5, equation 4.24) and for median filter, we use signal-to-noise ratio (given in section 3.1.3, equation 3.16).

Table 5.2: Average speedups and results accuracy for in hardware

| Images | Natural (40) $512 \times 512$ | Industrial (8) $512 \times 512$ | Medical (30) $400 \times 280$ | Barcode (50) $640 \times 480$ |
|---|---|---|---|---|
| Speedup | 1.58 | 1.80 | 1.55 | 1.73 |
| Kirsch: Accuracy | 98.70% | 99.34% | 97.21% | 98.63% |
| Median: SNR | 33.85 | 37.08 | 29.40 | 37.35 |

From table 5.2, it is seen that even the most complex images (*i.e.* medical images) yield reasonably high average speedups (*i.e.* 1.55). It is also seen that the result accuracy of both algorithms is high for all four sets of images (Kirsch: above 97%, median: above $29dB$).

Figures 5.8 and 5.9 show the results of conventional algorithms and window memoization for samples of natural image for the Kirsch edge detector and median filter, respectively. As it can be seen, due to high accuracy of results, the difference between the results of conventional algorithms and algorithms with window memoization are either very small or indistinguishable. The results for industrial, medical, and barcode images are presented in appendix D, figures D.1 to D.6.

The images in the bottom of figures 5.8 and 5.9 show the difference images of the results of the algorithms with and without window memoization. For the Kirsch edge detector algorithm, the difference images show two sets of marks: the locations that contain edges in the original results and non-edges in the window memoization results (marked with red edges) and the locations that contain non-edges in the original results and edges in the window memoization results (marked with green edges). As discussed

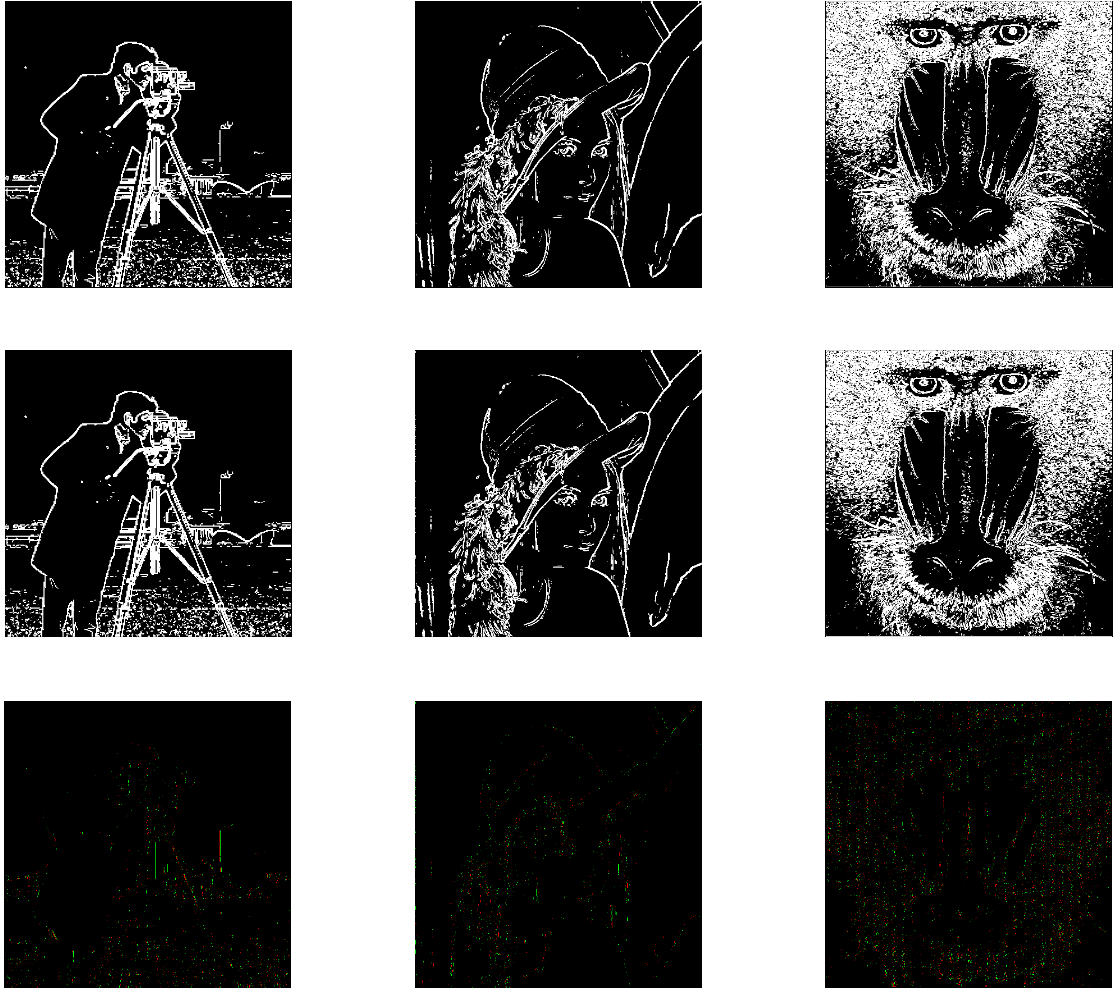


Figure 5.8: Results for natural images for Kirsch edge detector. Top: original results, middle: window memoization results, bottom: difference images.

in chapter 4, section 4.5, the error in the window memoization results for the Kirsch edge detector is mostly in the area that the original results contain edges. For the median filter, it is seen that the inaccuracy introduced by window memoization is almost spread all over the image.

### 5.7.2 Sprawl Results

To calculate sprawl for each case study algorithm, first, we implement each case study algorithm as a conventional scalar pipeline (core) at the register-transfer-level and calculate the hardware area consumed by this conventional implementation. Afterward,

Figure 5.9: Results for natural images for median filter. Top: original results, middle: window memoization results, bottom: difference images.

we implement the case study algorithm using 2-wide superscalar pipelines with window memoization and calculate the consumed hardware area. In each case, we have used Altera Cyclone II FPGA [20] as the target device.

Table 5.3 lists the hardware area consumed by different components of each design. As discussed in section 5.5, we use 2 most significant bits of each pixel for determining whether windows belong to a symbol (*i.e.* $d = 2$). Therefore, each level 1 reuse table is $2^d \times d = 8$ bits. Given that the case study algorithms use windows of $3 \times 3$ pixels, there are 8 level 1 reuse tables, consuming $8 \times 16 = 64$ bits of memory. The level 2 reuse table is $4 \times x$ where $x$ is the number of bits per each output pixel. For the Kirsch edge detector,

$x = 1$ since the result is a binary image. For median filter, which outputs a gray-level image, $x = 8$.

Other memory arrays used by window memoization are the two fifos. In our design, $Fifo_1$ has been implemented using registers because it is no more than 3 entries deep. Therefore, the consumed area by $Fifo_1$ is considered as a part of the control circuitry. $Fifo_2$ has different sizes for each of the case studies. For the Kirsch edge detector, $Fifo_2$ is 8 entries deep, consuming $8 \times 4 = 32$ bits (2 bits for $key_0$ of each window $win_A$ and $win_B$). Median filter requires a fifo, which is 16 entries deep and consumes $16 \times 4 = 64$ bits.

Table 5.3: Hardware area consumed by different components of the designs

| Algorithm | core | control circuitry | level 1 reuse tables | level 2 reuse table | $Fifo_2$ | memory as LEs | sprawl |
|---|---|---|---|---|---|---|---|
| Kirsch | 1028 LEs | 121 LEs | 64 bits | 4 bits | 32 bits | 6 LEs | 1.12 |
| Median | 1002 LEs | 160 LEs | 64 bits | 32 bits | 64 bits | 9.6 LEs | 1.17 |

Sprawl is calculated using the equation below:

$$sprawl = \frac{LE_{core} + LE_{ctl} + 0.06 \times mem_{bits}}{LE_{core}} \tag{5.32}$$

As it can be seen from table 5.3, sprawls for the Kirsch edge detector and median filter are 1.12 and 1.17, respectively. In other words, the hardware area cost incurred by the 2-wide superscalar pipeline with window memoization is 12% and 17% of the original scalar pipelines.

### 5.7.3 Efficiency Results

As we discussed in section 5.1, the relative efficiency of a 2-wide superscalar pipeline with window memoization with respect to the original scalar pipeline is calculated as:

$$eff_{Rel} = \frac{speedup}{sprawl} \tag{5.33}$$

Tables 5.4 and 5.5 show the relative efficiency for each sets of images for the Kirsch edge detector and median filter, respectively. It is seen that Kirsch edge detector gives higher relative efficiencies because its sprawl is lower than that of median filter. The relative efficiency for both algorithms is between 1.33 and 1.60.

Table 5.4: Relative efficiency for Kirsch edge detector

| Images | Natural (40) $512 \times 512$ | Industrial (8) $512 \times 512$ | Medical (30) $400 \times 280$ | Barcode (50) $640 \times 480$ |
|---|---|---|---|---|
| Speedup | 1.58 | 1.80 | 1.55 | 1.73 |
| Sprawl | 1.12 | 1.12 | 1.12 | 1.12 |
| Efficiency | 1.41 | 1.60 | 1.38 | 1.54 |

Table 5.5: Relative efficiency for median filter

| Images | Natural (40) $512 \times 512$ | Industrial (8) $512 \times 512$ | Medical (30) $400 \times 280$ | Barcode (50) $640 \times 480$ |
|---|---|---|---|---|
| Speedup | 1.58 | 1.80 | 1.55 | 1.73 |
| Sprawl | 1.17 | 1.17 | 1.17 | 1.17 |
| Efficiency | 1.35 | 1.54 | 1.33 | 1.48 |

## 5.8 Speedup versus Coding/Interpixel Redundancy

In chapter 3, we showed mathematically that the coding and interpixel redundancy of an image have a positive relationship with both the reusability and computational redundancy of the image. In chapter 4, we showed that the computational redundancy of an image has a positive relationship with the speedup obtained for the image by window memoization in software. Thus, it was concluded that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in software. In this section, we show that the reusability of an image has a positive relationship with the speedup obtained for the image by window memoization in hardware. This will lead to the fact that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image

by window memoization in hardware (equation 5.34).

$$(C_r + IP_r) \propto^{+} speedup \tag{5.34}$$

In section 5.3.1, we showed that the hit rate of an image in hardware has a positive relationship with the speedup of the image in hardware:

$$speedup = \frac{1}{1 - HR_{hw}} \implies HR_{hw} \propto^{+} speedup \tag{5.35}$$

Thus, to show that relation 5.34 holds, it is sufficient to show that the relation below holds.

$$(C_r + IP_r) \propto^{+} HR_{hw} \tag{5.36}$$

For window memoization in software, in order to show that $(C_r + IP_r) \propto^{+} HR_{sw}$ holds, we had to take the following intermediate steps (chapter 3 and chapter 4, section 4.6):

$$(C_r + IP_r) \ \propto^{+} \ R_{ideal}(\delta) \ \propto^{+} \ R(\delta) \ \propto^{+} \ HR_{pc} \ \propto^{+} \ Comp_r \propto^{+} HR_{sw} \tag{5.37}$$

As we will discuss shortly, for window memoization in hardware, in order to show that $(C_r + IP_r) \propto^{+} HR_{hw}$ holds, the following intermediate steps are sufficient:

$$(C_r + IP_r) \ \propto^{+} \ R_{ideal}(\delta) \ \propto^{+} \ R(\delta) \ \propto^{+} \ HR_{hw} \tag{5.38}$$

In other words, in contrast to window memoization in software, in order to show that relation 5.36 holds, the information about the hit rate of perfect cache and computational redundancy is not necessary. The reason is that in window memoization in software, the reuse is done globally while in window memoization in hardware, the reuse is done locally. In window memoization in software, the response of one window in the image can be reused for a similar window located in any spatial distance from the first window because the reuse table is not always updated. It is updated only in case of a miss. Therefore, it is possible that a symbol from the beginning of an image is inserted into the reuse table

and reused for a long time without being evicted. Also, window memoization in software usually uses large reuse tables (*e.g.* 32K entries), which leads to non-local reuse.

In contrast, in the hardware implementation of window memoization, the reuse table is always updated by one window of each pair of input windows. Thus, the reuse table usually holds only the symbols and responses of the most recent windows. Given that the reuse table size for window memoization in hardware is very small (*i.e.* 4 entries) , only those similar windows reuse each other's responses that are located in very close distances from one another.

In chapter 3, section 3.3, we showed that for two images $Img_1$ and $Img_2$, relation 5.39 holds.

$$(C_{r1} + IP_{r1}) \leq (C_{r2} + IP_{r2}) \implies P_{u1}^{icp}(\delta) > P_{u2}^{icp}(\delta) \tag{5.39}$$

where $P_u^{icp}$ is the uniqueness probability density function of an image. Figure 5.10 illustrates relation $P_{u1}^{icp}(\delta) > P_{u2}^{icp}(\delta)$.

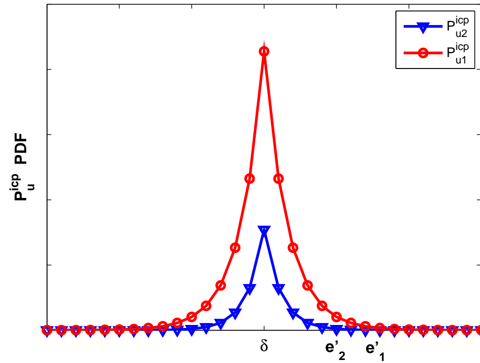



Figure 5.10: Uniqueness probability density functions

Relation $P_{u1}^{icp}(\delta) > P_{u2}^{icp}(\delta)$ means that for any $\delta$, the number of unique symbols for $Img_1$ is higher than that for $Img_2$. In hardware, window memoization compares only those windows that are located very close to each other in the image. Close windows in image are usually similar and thus, they usually have similar $\delta$. Thus, in hardware, window memoization only compares windows with similar $\delta$. Therefore, having fewer unique symbols for a given $\delta$ means, on average, fewer misses and hence more hits.

$$P_{u1}^{icp}(\delta) > P_{u2}^{icp}(\delta) \implies HR_{hw1} < HR_{hw2} \tag{5.40}$$

In contrast, in software, having fewer unique symbols for a given $\delta$ does not necessarily mean fewer misses because a miss can also occur due to comparing non_matching windows, which have different $\delta$. In other words, in software, there are two sources for misses: comparing non_matching windows with similar $\delta$ and comparing non_matching windows with different $\delta$. Having fewer unique symbols for a given $\delta$ only affects the misses caused by comparing non_matching windows with similar $\delta$, which is the case in hardware.

From relations 5.39 and 5.40, it is concluded that:

$$(C_{r1} + IP_{r1}) \leq (C_{r2} + IP_{r2}) \implies HR_{hw1} < HR_{hw2} \tag{5.41}$$

The relation above can be rewritten as:

$$(C_r + IP_r) \propto^{+} HR_{hw} \tag{5.42}$$

Figure 5.11 shows the speedup versus the coding/interpixel redundancies for all four different sets of images. It is seen that in all cases, relation 5.42 holds with very high accuracy.

## 5.9 Summary

In this chapter, we presented an optimized architecture for window memoization in hardware (*i.e.* 2-wide superscalar pipeline with window memoization) that gives a high relative efficiency with respect to the design (scalar pipeline) on which window memoization is based. We implemented window memoization at the register-transfer-level using VHDL and applied the technique to two case study algorithms. To achieve high speedups, we used tolerant memoization, a 2-level reuse table, and the direct-mapped mapping scheme for the memoization mechanism. To achieve low sprawls, we presented parallel reuse

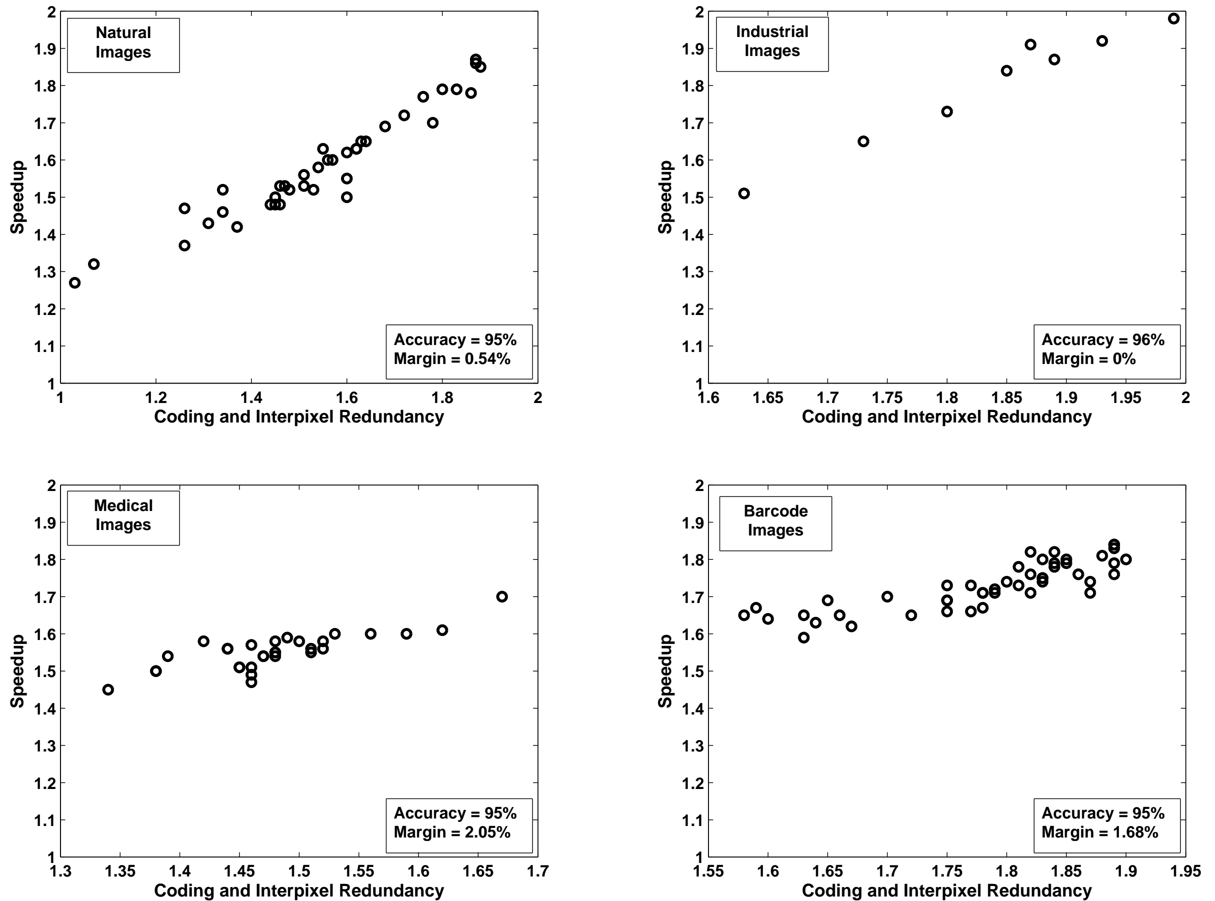



Figure 5.11: Speedup versus coding/interpixel redundancy

tables based on Bloom filters. We demonstrated that the optimized architecture for window memoization in hardware can improve performance by up to a factor of 1.8 (typical speedup: 1.6) with up to 60% (on average: 40%) less hardware area than the conventional 2-wide superscalar pipelines while maintaining reasonably high accuracy of results (*i.e.* above 97% and $29dB$). Finally, we showed mathematically and empirically that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained for the image by window memoization in hardware.

# Chapter 6

# Conclusion

In this work, we introduced an innovative performance improvement technique, window memoization, that benefits from data redundancy in images to improve the performance and efficiency of local image processing algorithms. Window memoization combines the memoization techniques proposed in software and hardware with the repetitive nature of image data to speed up the computations in both software and hardware. We also showed mathematically that the speedup obtained by window memoization in both software and hardware is directly inherited from fundamental characteristics of image data: coding redundancy and interpixel redundancy.

Although different memoization techniques have previously been proposed in software and hardware, their practicality has usually been limited. In software, the proposed techniques usually require detailed profiling information about the runtime behaviour of the program to benefit from the value locality of data [15] [55]. In other words, the techniques are usually generic methods, which do not concentrate on any particular class of input data or algorithms. In contrast, window memoization is a specific technique for local image processing algorithms where the speedup is directly affected by the characteristics of input image data.

In hardware (*i.e.* processor design), no memoization technique has been implemented in a real design yet. Handling the dependencies between instructions in a program and maintaining the coherency of the memoization table would require large content-addressable memory arrays with multiple write ports. The complexity of these memory

arrays and the difficulty of pipelining them overshadow the theoretical gains in performance [49]. Image processing algorithms usually do not have data dependencies among the operations. This makes it feasible to implement the memoization mechanism using simple dual-port memories. Moreover, the memoization techniques proposed in hardware are mostly for processor designs. In contrast, our memoization technique in hardware targets the specialized hardware implementations of local image processing algorithms.

In software, we discussed several parameters that affect the speedup achieved by window memoization by affecting hit rate and the memoization overhead time. It was shown that each parameter depends on a design decision from developing the memoization mechanism. We presented a fast symbol generation mechanism whose contribution to the memoization overhead time is small. As the mapping scheme, the direct-mapped method was chosen because it produces small overhead times for memory operations and comparisons. The multiplication method was selected as the hash function for the direct-mapped mapping since it yields high hit rates and small contributions to the memoization overhead time. Finally, to achieve high hit rates with reasonably high accuracy of results, 4 bits of each pixel in a window are considered to determine whether the window belongs to a symbol.

We also presented a model to predict the speedup obtained by window memoization in software for all images in a data set. The only information that the model needs is to measure the memoization overhead time for the two images that have the minimum and maximum hit rates in a data set. With this information, for any image, the model predicts the speedup. The reuse table size is a key parameter in obtaining high speedups. Although larger reuse tables usually lead to higher hit rates, we showed that for a given algorithm and a processor, after certain size of reuse table, speedup degrades. Therefore, for a given algorithm and processor, there is an optimal reuse table size, which yields maximum speedups. Our speedup model enables the developer to pick the optimal reuse table size for a set of images by using only two images and before integrating the memoization mechanism into the algorithm at hand.

In software, we implemented window memoization in $C$, where six local image processing algorithms were used as case studies. We used the optimal reuse table sizes obtained by the speedup model. Our experiments showed that window memoization yields

significant speedups for different algorithms with various input images run on different processors. The typical speedups range from 1.2 to 7.9 with a maximum factor of 40.

In hardware, we presented an optimized architecture for window memoization, which yields high speedups with a small overhead in hardware area (*i.e.* sprawl). We implemented window memoization in hardware as a 2-wide superscalar pipeline. In contrast to conventional 2-wide superscalar pipelines that consume twice the hardware area of a scalar pipeline, the only extra hardware that a 2-wide superscalar pipeline with window memoization consumes with respect to a scalar pipeline is the memoization mechanism, which is typically 20% of the scalar pipeline. We also defined a measure, relative efficiency, which evaluates the speedup/sprawl tradeoffs of a 2-wide superscalar pipeline with window memoization with respect to a scalar pipeline.

We discussed the factors that affect speedup in hardware, including the mapping scheme used by the control circuitry, the core latency, and the reuse table size. We showed that sprawl is affected by the hardware area consumed by the reuse table, the control circuitry, and the core. We discussed that in order to achieve high speedups, the direct-mapped mapping scheme and a 2-level reuse table must be used. To achieve low sprawls, we presented a space-efficient architecture for the reuse table, which is based on parallel Bloom filters. Finally, we presented an optimized architecture for window memoization in hardware, which uses a direct-mapped mapping scheme and a 2-level parallel reuse table based on Bloom filters to achieve high speedups and low sprawls.

In contrast to software, where using 4 most significant bits of pixels for comparison yields reasonably high accuracy of results, in hardware only 2 most significant bits are sufficient. The reason is that in hardware, the mask operations set are applied to at least half the windows in the image. Moreover, in hardware, window responses are usually reused by nearby windows, because the reuse table is always updated by half the windows. This also allows the reuse table required for window memoization in hardware to be much smaller than that in software (e.g. 4 entries versus $32K$ entries).

In hardware, we implemented window memoization at the register-transfer-level using VHDL targeting an Altera FPGA where two local image processing algorithms were used as case studies. For typical images, window memoization yields a relative efficiency in the range 1.4 to 1.6, while the relative efficiency of conventional superscalar pipelines is 1.

We introduced the reusability of an image as a measure that gives the average number of redundant sets of computations per each reuse table cell. We also presented computational redundancy as a quantitative measure that indicates the percentage of unnecessary mask operations sets applied to an image. We showed that both the reusability and the computational redundancy of an image are inherited from two principal redundancies in image data: coding redundancy and interpixel redundancy. We proved that the coding and interpixel redundancy of an image have a positive relationship with both the reusability and computational redundancy of the image.

Computational redundancy considers all of the windows across an image whose responses need not be calculated. In other words, in calculating computational redundancy, is is assumed that the response of a window can be reused for similar windows, regardless of the spatial distance in the image between the two windows, which is the case in window memoization in software. In other words, in window memoization in software, the reuse is done globally. We showed mathematically that the computational redundancy of an image has a positive relationship with the speedup obtained by window memoization in software. Therefore, it has been concluded that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained by window memoization in software.

Reusability is based on the new information carried by the central pixels of windows, or $\delta$. We showed that for any $\delta$, the reusability of an image has a negative relationship with the number of unique symbols of the image. In hardware, window memoization compares only those windows that are located very close to each other in the image. In other words, in window memoization in hardware, the reuse is done locally. Nearby windows in an image are usually similar and thus, they usually have similar values of $\delta$. Thus, in hardware, window memoization only compares windows with similar values of $\delta$. Therefore, having fewer unique symbols for a given $\delta$ means on average, fewer misses and hence more hits. This means that the reusability of an image has a positive relationship with the speedup obtained by window memoization in hardware. Therefore, it has been concluded that the coding and interpixel redundancy of an image have a positive relationship with the speedup obtained by window memoization in hardware.

In software, the mask operations sets are applied only to those windows that cannot

find matching symbols in the reuse table. In contrast, in hardware, the mask operations sets must be applied to at least half the windows (*i.e.* every other window in the image). This limits the maximum speedup in hardware to a factor, which is equal to the pipeline width (*e.g.* 2), while in software, the speedup can be much higher (*e.g.* 40). As a result, window memoization in software performs the reuse globally, while in hardware, window memoization performs the reuse locally. This is the main difference between the software and hardware implementations of window memoization. Nevertheless, as discussed earlier, in both cases, the root cause of having similar windows (either globally or locally) is coding and interpixel redundancy of the image. This is a simple concept, which connects basic characteristics of image data to performance improvement obtained by window memoization in both software and hardware.

Future work for this research will proceed in three directions: theory, software, and hardware. In theory, we will investigate the mathematical relationship between the two different measures of interpixel redundancy: mapping transform and Markov model. We expect that the two measures have a positive relationship with each other, since both represent the same characteristic of image data. We will also apply window memoization to color images. For algorithms that use the three basic elements of color (*i.e.* red, blue, and green) separately, we anticipate similar speedups to gray-level images to be obtained. For algorithms that use a combination of colors, although the reuse mechanism will be more complicated and hit rates may not be as high as for gray-level images, we believe that there is a potential to obtain reasonable speedups.

In software, we will use processor simulation tools (*e.g.* SimpleScalar [35]) to investigate the exact effects of the processor and cache hierarchy architecture (*e.g.* different branch and caching schemes) on speedup. This will enable us to customize a configurable embedded processor for window memoization.

In hardware, we will use wider superscalar pipelines (*e.g.* 4-wide and 8-wide) to implement window memoization. In addition, we will apply window memoization to recursive image processing algorithms where there are data dependencies between the operations. For recursive algorithms, their recursive nature requires feedback paths that often make it difficult to increase the clock speed by conventional pipeline optimization techniques. To overcome this performance limitation, we will extend window memoization in such a way

that it is applicable to recursive image processing algorithms. We expect that window memoization will improve the throughput of the hardware implementations of recursive algorithms.

Finally, we will develop an automated version of window memoization for software and hardware. Automated window memoization will be a tool that accepts an existing algorithm (*i.e.* core) that the user wants to optimize. In both hardware and software, the core, written in VHDL or C, will be given to automated window memoization along with a set of input images and the final results of the core. Automated window memoization will then explore different design options to maximize the performance gain obtained by window memoization for the existing algorithm. After setting the parameters, a new VHDL/C code will be generated, which embodies the window memoization technique applied to the existing algorithm in hardware or software. Automated window memoization will use learning algorithms to find the optimal point for the window memoization parameters quickly and robustly. By automating window memoization and customizing it for a given algorithm, many existing image processing algorithms and new algorithms, in software and hardware, will be optimized with minimal cost and human intervention. This will enable the image processing software and embedded system designers to optimize their existing systems with the push of a button and design new high-performance systems more efficiently.

# Appendix A

# Notations and Terminology

- $\propto^+$ (Positive relationship): two variables are considered to have positive relationship if one variable increases/decreases then the other variable also increases/decreases.
- $\propto^-$ (Negative relationship): two variables are considered to have negative relationship if one variable increases/decreases then the other variables decreases/increases.
- $Comp_r$ (Computational redundancy of an image): the percentage of the mask operations sets that are not necessary to perform, in order to complete processing the image.
- Transform *icp*: A transform (information at the central pixel) that takes a local window as input and outputs the new information carried by the central pixel.
- $Img^{icp}$: A transformed format of original image, based on the information carried by the pixels at the centers of symbols.
- $P$: The probability of occurrence of symbols in the original image, which has been sorted in descending order.
- $P^{icp}$: The probability of occurrence of symbols in the transformed image, which is equal to the probability of occurrence of symbols in the original image based on the new information carried by the central pixels of symbols $\delta$.
- $P_u^{icp}$: The uniqueness probability of symbols in the original image based on the new information carried by the central pixels of symbols $\delta$.

- $HR$: The percentage of the times that the incoming windows find a matching symbol in the reuse table.
- $C_r$: Average coding redundancy (bits per pixel).
- $IP_r$: Average interpixel redundancy (bits per pixel).
- $H$: Average content of information (bits per pixel).
- $GL$: Number of gray levels of an image.
- $R(\delta)$ (Reusability of an image): the average number of the redundant mask operations sets per symbol for symbols in the image whose central pixels carry new information in the range $[-\delta, +\delta]$.
- $R_{ideal}(\delta)$ (Ideal Reusability of an image): an idealized version of the image reusability, which ignores the effect of the uniqueness probability of symbols on reusability.
- $\mu$: Mean value of the original image.
- $\sigma$: Standard deviation of the original image.
- $\sigma^{icp}$: Standard deviation of $Img^{icp}$.
- Accuracy and error margin: Assume we have a function $f : S- > S'$ where $S$ and $S'$ are the finite subsets of real numbers. The accuracy and error margin for the relation $\forall x, y \in S,\ x \leq y \implies f(x) \leq f(y)$ is defined as follows:
  - For a given $\epsilon$: Accuracy $=100 \times \frac{k}{n}$ where $k$ is the total number of cases that the relation $\forall x, y \in S,\ x \leq y \implies f(x) \leq f(y) + \epsilon$ holds, and $n$ is the total number of possible pairs of elements from set $S$, which is equal to $\frac{|S| \times |S| - 1}{2}$.
  - Error Margin$=100 \times \frac{\epsilon}{max(S')}$ where the relation $\forall x, y \in S,\ x \leq y \implies f(x) \leq f(y) + \epsilon$ holds.

# Appendix B

# Autocorrelation as a Measure for Interpixel Redundancy

In this appendix we use autocorrelation (equation 2.10) to measure the amount of interpixel redundancy of the pairs of adjacent pixels in image. :

$$
\begin{aligned}
IP_r &= AC(1) \\
&= \frac{E[(Img_i - \mu)(Img_{i+1} - \mu)]}{E[(Img - \mu)^2]} \\
&= \frac{E[(Img_i - \mu)(Img_{i+1} - \mu)]}{\sigma^2} \\
&= \frac{1}{\sigma^2}(E[Img_i Img_{i+1}] - \mu(E[Img_{i+1}] + E[Img_i]) + \mu^2)
\end{aligned} \tag{B.1}
$$

where $\sigma^2$ and $\mu$ are the variance and mean value of the original image, respectively. $Img_i$ and $Img_{i+1}$ are defined based on the original image, $Img$:

$$Img = x_0 x_1 x_2 ... x_{n-1} \tag{B.2}$$

$$Img_i = x_0 x_1 x_2 ... x_{n-2} \tag{B.3}$$

$$Img_{i+1} = x_1 x_2 ... x_{n-1} \tag{B.4}$$

Using the equations above, $Img_{diff}$ is defined as:

$$Img_{diff} = Img_i - Img_{i+1} \tag{B.5}$$

$Img_{diff}$ is in fact $Img^{icp}$ where the transform $icp$ takes pairs of adjacent pixels as input and outputs the information carried by one of the pixels.

In equation B.1, $E[Img_i]$ and $E[Img_{i+1}]$ can be written in terms of $E[Img]$ or $\mu$:

$$\begin{aligned} E[Img_i] &= \frac{1}{n-1}\sum_{0}^{n-2} x_i \\ &= \frac{1}{n-1}(\sum_{0}^{n-1} x_i - x_{n-1}) \\ &= \frac{1}{n-1}(nE[Img] - x_{n-1}) \\ &= \frac{n\mu}{n-1} \\ &\approx \mu \end{aligned} \tag{B.6}$$

and similarly:

$$E[Img_{i+1}] \approx \mu \tag{B.7}$$

Replacing $E[X_i]$ and $E[X_{i+1}]$ in equation B.1 with equations B.6 and B.7 yields:

$$IP_r = AC(1) = \frac{1}{\sigma^2}(E[Img_i Img_{i+1}] - \mu^2) \tag{B.8}$$

We can calculate the variance of $Img_{diff}$:

$$\sigma^2_{diff} = E[(Img_{diff} - \mu_{diff})^2] \tag{B.9}$$

As mentioned in chapter 3, section 3.1.6, the probability density function of $Img_{diff}$ is modeled by a zero mean Laplace distribution, which means $\mu_{diff} = 0$, or:

$$\sigma^2_{diff} = E[Img^2_{diff}] \tag{B.10}$$

Substituting $Img_{diff} = Img_i - Img_{i+1}$ from equation B.5 gives:

$$\begin{aligned}\sigma^2_{diff} &= E[(Img_i - Img_{i+1})^2] \\ &= E[Img^2_i] + E[Img^2_{i+1}] - 2E[Img_i Img_{i+1}]\end{aligned} \tag{B.11}$$

$E[Img^2_i]$ and $E[Img^2_{i+1}]$ can be written in terms of $E[Img^2]$:

$$\begin{aligned}E[Img^2_i] &= \frac{1}{n-1}\sum_0^{n-2} x_i^2 \\ &= \frac{1}{n-1}(\sum_0^{n-1} x_i^2 - x^2_{n-1}) \\ &= \frac{1}{n-1}(nE[Img^2] - x^2_{n-1}) \\ &= \frac{nE[Img^2]}{n-1} \\ &\approx E[Img^2]\end{aligned} \tag{B.12}$$

and similarly:

$$E[Img^2_{i+1}] \approx E[Img^2] \tag{B.13}$$

Replacing $E[Img^2_i]$ and $E[Img^2_{i+1}]$ in equation B.11 with equations B.12 and B.13 yields:

$$\sigma^2_{diff} \approx 2(E[Img^2] - E[Img_i Img_{i+1}]) \tag{B.14}$$

We also have:

$$E[Img^2] = \sigma^2 + \mu^2 \tag{B.15}$$

Substituting the equation above into equation B.14 yields:

$$\sigma^2_{diff} \approx 2(\sigma^2 + \mu^2 - E[Img_i Img_{i+1}]) \tag{B.16}$$

which gives:

$$E[Img_i Img_{i+1}] \approx (\sigma^2 + \mu^2) - \frac{\sigma^2_{diff}}{2} \tag{B.17}$$

Substituting $E[Img_i Img_{i+1}]$ into equation B.8 yields:

$$\begin{aligned} IP_r &= AC(1) \\ &\approx 1 - \frac{\sigma^2_{diff}}{2\sigma^2} \end{aligned} \tag{B.18}$$

which gives:

$$\sigma^2_{diff} \approx 2\sigma^2(1 - IP_r) \tag{B.19}$$

In the chain of relations that we proved in chapter 3 (relation 3.102), we needed to prove:

$$(C_r + IP_r) \propto^- \sigma_{diff}$$

Let's assume that the coding redundancy is constant for all images under study. Thus, the histograms of images under study will be the same since the equation for coding redundancy depends only on the histogram (chapter 2, equations 2.1 and 2.3). Images with the same histograms will have identical variances ($\sigma^2$). Therefore, in equation B.19, $\sigma^2_{diff}$ will only depend on $IP_r$.

$$\sigma^2_{diff} \approx 2\sigma^2(1 - IP_r) \implies IP_r \propto^- \sigma_{diff} \tag{B.20}$$

The relation above is a part of the chain of relations proved in chapter 3, relation 3.102. However, eliminating the assumption that all the images under study have the same histogram or coding redundancy will cause that in equation B.19, $\sigma^2_{diff}$ depend on both

$IP_r$ and $\sigma^2$. Although images with the same coding redundancies have the same variances, there is no mathematical relationship between the image variance and its coding redundancy. In other words, $\sigma_{diff1} < \sigma_{diff2}$ does not imply $C_{r1} > C_{r2}$.

As a result, when working with images with different histograms (*i.e.* different variances), having information about the variance does not give any information about the coding redundancy of the image. Therefore, it is not possible to draw any conclusion about the relationship between $\sigma_{diff}$ and coding redundancy using equation B.18. Thus, for our purpose, which is to prove the relationship between $\sigma_{diff}$ and both coding and interpixel redundancies, autocorrelation is not a good measure for interpixel redundancy.

# Appendix C

# Results for Window Memoization in Software

Table C.1: Speedups for processor 1

| Images<br>Algorithms | Natural (40)<br>512 × 512 | Industrial (8)<br>512 × 512 | Medical (30)<br>400 × 280 | Barcode (50)<br>640 × 480 |
|---|---|---|---|---|
| Canny Edge Detection | 2.15 | 4.31 | 1.64 | 3.63 |
| Morphological Gradient | 2.57 | 4.51 | 1.52 | 3.42 |
| Kirsch Edge Detection | 1.89 | 3.17 | 1.15 | 2.55 |
| Corner Detection | 3.84 | 14.16 | 1.54 | 5.32 |
| Median Filter | 1.14 | 2.08 | 1.14 | 1.65 |
| Local Variance | 1.59 | 4.06 | 1.06 | 1.86 |

Table C.2: Speedups for processor 2

| Images<br>Algorithms | Natural (40)<br>512 × 512 | Industrial (8)<br>512 × 512 | Medical (30)<br>400 × 280 | Barcode (50)<br>640 × 480 |
|---|---|---|---|---|
| Canny Edge Detection | 2.13 | 5.37 | 1.68 | 4.14 |
| Morphological Gradient | 2.27 | 4.05 | 1.50 | 3.29 |
| Kirsch Edge Detection | 1.69 | 2.52 | 0.94 | 1.91 |
| Corner Detection | 3.73 | 14.94 | 1.52 | 5.20 |
| Median Filter | 1.18 | 2.10 | 1.04 | 1.68 |
| Local Variance | 1.55 | 3.88 | 1.04 | 1.83 |

Table C.3: Speedups for processor 3

| Images<br>Algorithm | Natural (40)<br>512 × 512 | Industrial (8)<br>512 × 512 | Medical (30)<br>400 × 280 | Barcode (50)<br>640 × 480 |
|---|---|---|---|---|
| Canny Edge Detection | 7.93 | 39.87 | 3.25 | 10.96 |
| Morphological Gradient | 2.98 | 5.79 | 1.99 | 4.05 |
| Kirsch Edge Detection | 1.52 | 2.20 | 1.15 | 1.94 |
| Corner Detection | 4.24 | 22.72 | 1.70 | 5.78 |
| Median Filter | 2.07 | 6.75 | 2.26 | 3.02 |
| Local Variance | 1.58 | 4.60 | 1.07 | 1.85 |

Table C.4: Accuracy of the results

| Images<br>Algorithms | Natural (40)<br>512 × 512 | Industrial (8)<br>512 × 512 | Medical (30)<br>400 × 280 | Barcode (50)<br>640 × 480 |
|---|---|---|---|---|
| Canny Edge Detection | 99.08% | 99.52% | 96.01% | 98.90% |
| Morphological Gradient | 99.82% | 99.33% | 99.45% | 99.39% |
| Kirsch Edge Detection | 99.25% | 98.47% | 97.17% | 99.03% |
| Corner Detection | 99.65% | 99.86% | 99.93% | 99.84% |
| Median Filter (SNR) | 34.28 | 33.33 | 29.73 | 33.52 |
| Local Variance (SNR) | 39.19 | 33.78 | 47.74 | 34.58 |

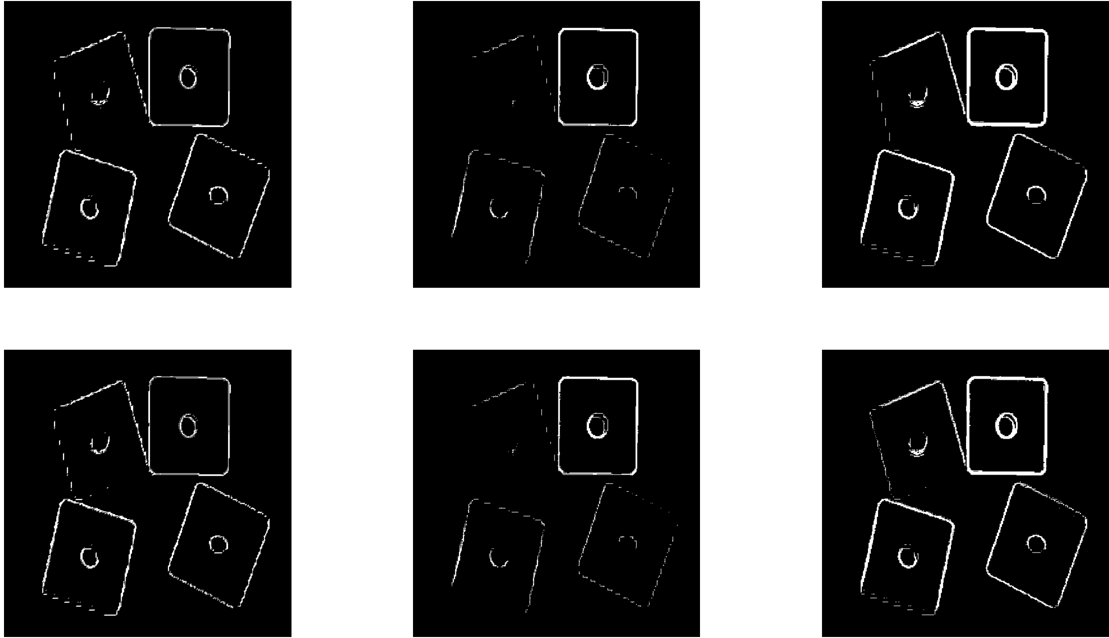


Figure C.1: Results for industrial images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, bottom: window memoization results.

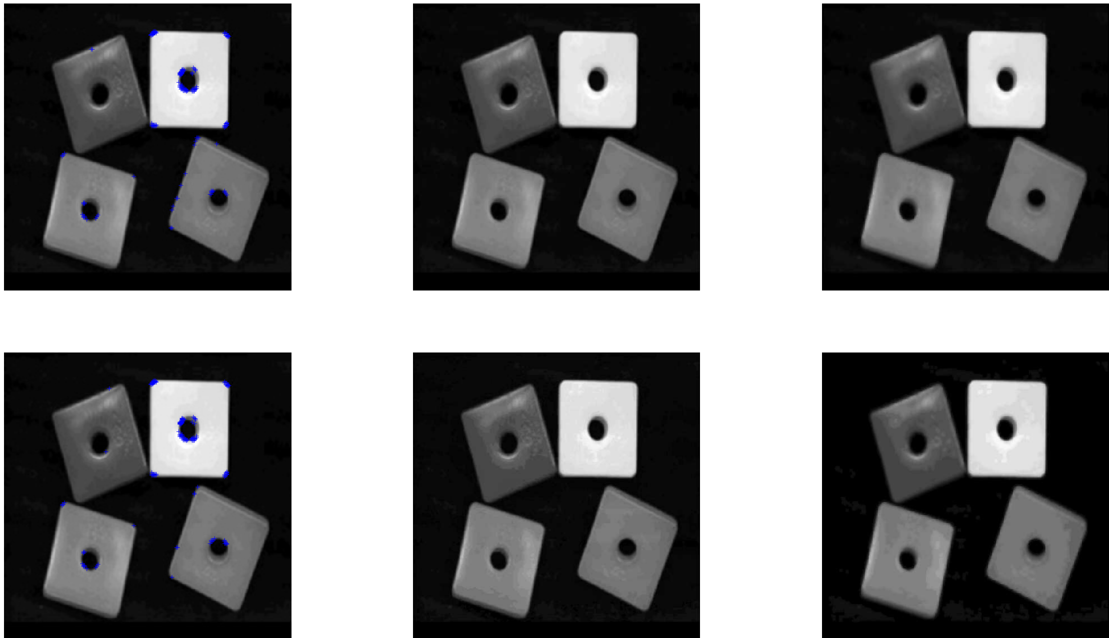


Figure C.2: Results for industrial images. Left to right: corner detection, median filter, and local variance. Top: original results, bottom: window memoization results.

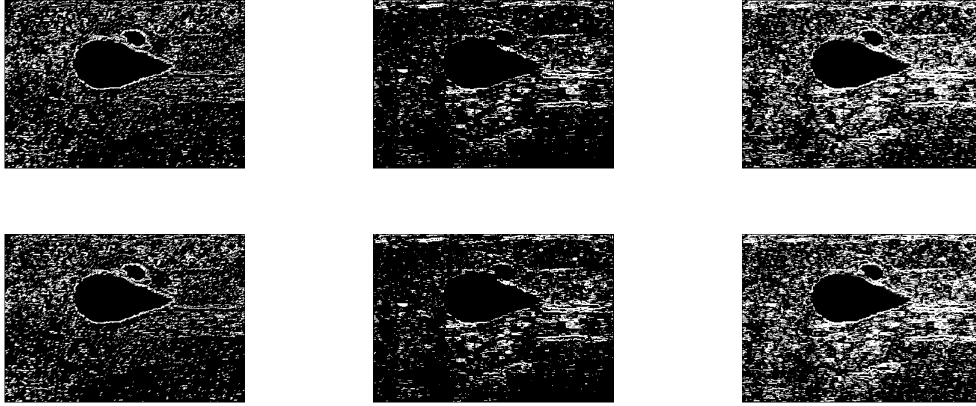


Figure C.3: Results for medical images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, bottom: window memoization results.

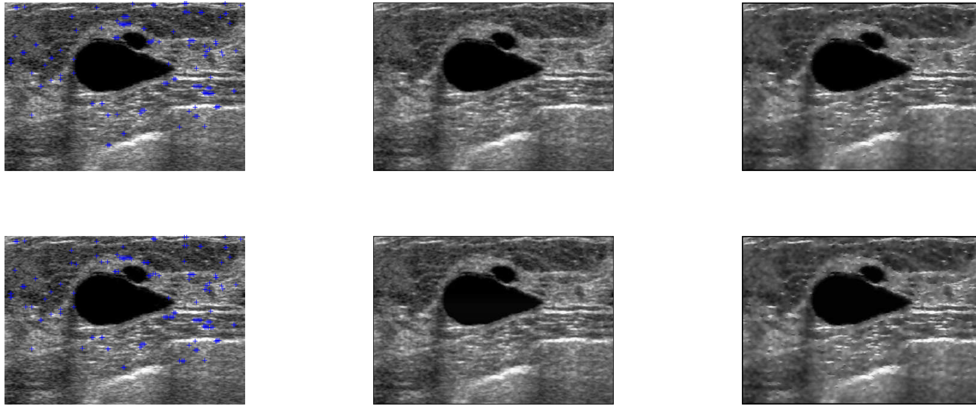


Figure C.4: Results for medical images. Left to right: corner detection, median filter, and local variance. Top: original results, bottom: window memoization results.

Table C.5: Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for natural images

| $RT_{size}$ | $1K$ | $2K$ | $4K$ | $8K$ | 16 | $32K$ | $64K$ | $128K$ | $256K$ |
|---|---|---|---|---|---|---|---|---|---|
| Accuracy (%) | 95 | 95 | 96 | 97 | 98 | 98 | 99 | 99 | 99 |
| Error Margin (%) | 4.55 | 1.59 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

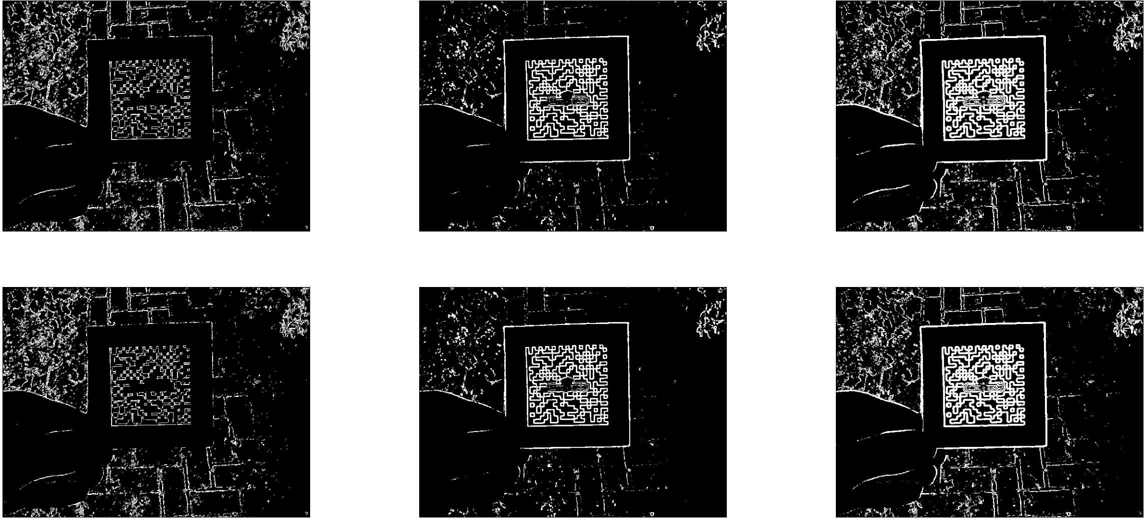


Figure C.5: Results for barcode images. Left to right: Canny, morphological, and Kirsch edge detectors. Top: original results, bottom: window memoization results.

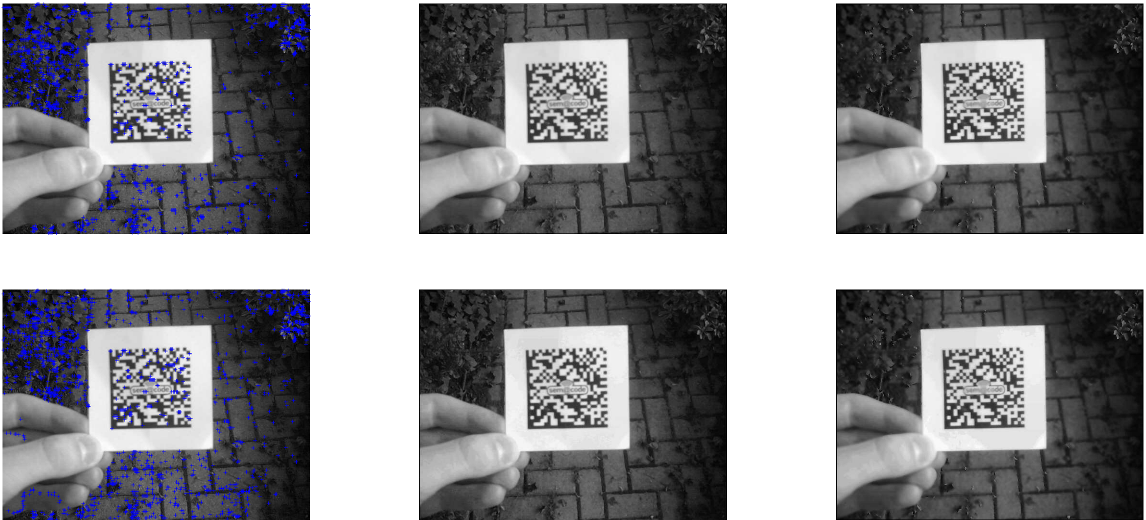


Figure C.6: Results for barcode images. Left to right: corner detection, median filter, and local variance. Top: original results, bottom: window memoization results.

Table C.6: Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for industrial images

| $RT_{size}$ | $1K$ | $2K$ | $4K$ | $8K$ | 16 | $32K$ | $64K$ | $128K$ | $256K$ |
|---|---|---|---|---|---|---|---|---|---|
| Accuracy (%) | 95 | 95 | 95 | 95 | 96 | 96 | 100 | 100 | 100 |
| Error Margin (%) | 6.28 | 3.80 | 1.71 | 0.19 | 0 | 0 | 0 | 0 | 0 |

Table C.7: Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for medical images

| $RT_{size}$ | $1K$ | $2K$ | $4K$ | $8K$ | 16 | $32K$ | $64K$ | $128K$ | $256K$ |
|---|---|---|---|---|---|---|---|---|---|
| Accuracy (%) | 95 | 95 | 95 | 95 | 96 | 97 | 97 | 97 | 96 |
| Error Margin (%) | 25.04 | 13.52 | 5.62 | 0.27 | 0 | 0 | 0 | 0 | 0 |

Table C.8: Accuracy and error margin for relation $comp \propto^{+} HR_{sw}$ for barcode images

| $RT_{size}$ | $1K$ | $2K$ | $4K$ | $8K$ | 16 | $32K$ | $64K$ | $128K$ | $256K$ |
|---|---|---|---|---|---|---|---|---|---|
| Accuracy (%) | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 |
| Error Margin (%) | 11.93 | 8.37 | 5.35 | 3.45 | 2.03 | 0.42 | 0.31 | 0.56 | 0.58 |

Table C.9: Accuracy and error margin for relation $HR_{sw} \propto^{+}$ *speedup* for natural images

| Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| Accuracy (%) | 98 | 98 | 96 | 97 | 95 | 99 |
| Error Margin (%) | 0 | 0 | 0 | 0 | 0.92 | 0 |

Table C.10: Accuracy and error margin for relation $HR_{sw} \propto^{+}$ *speedup* for industrial images

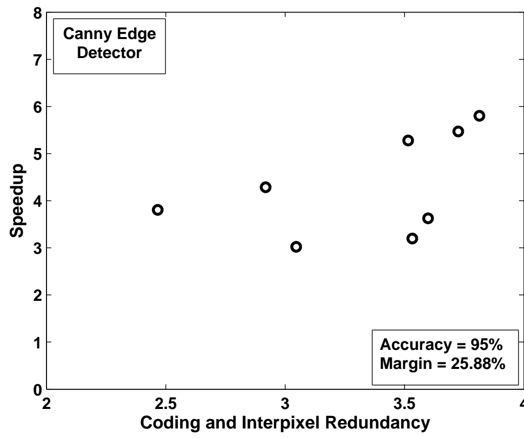
| Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| Accuracy (%) | 95 | 95 | 95 | 95 | 95 | 100 |
| Error Margin (%) | 7.66 | 14.31 | 15.38 | 8.08 | 10.99 | 0 |

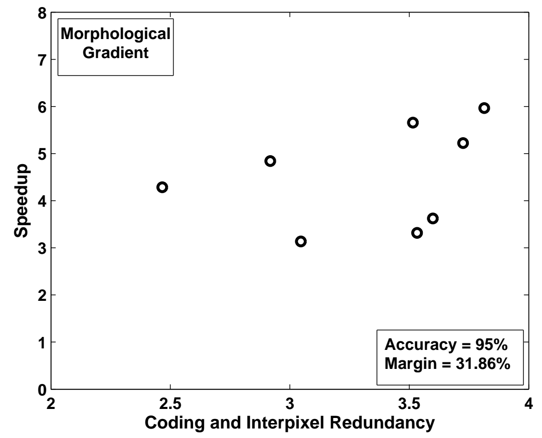
Table C.11: Accuracy and error margin for relation $HR_{sw} \propto^{+}$ *speedup* for medical images

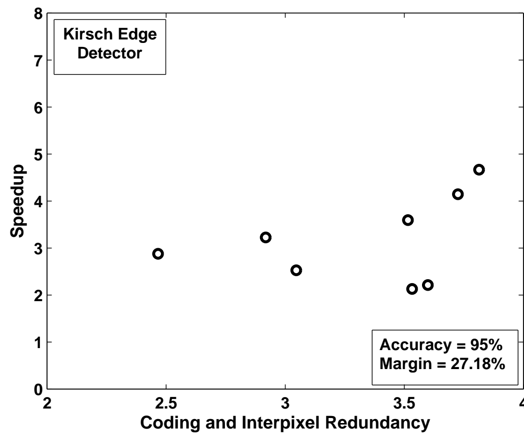
| Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| Accuracy (%) | 97 | 95 | 95 | 95 | 95 | 97 |
| Error Margin (%) | 0 | 0 | 1.18 | 2.65 | 0.66 | 0 |

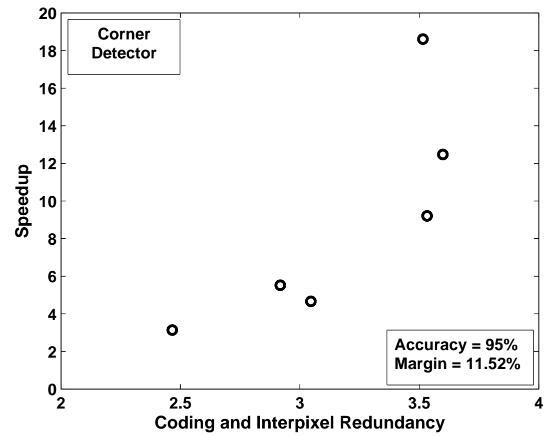
Table C.12: Accuracy and error margin for relation $HR_{sw} \propto^{+}$ *speedup* for barcode images

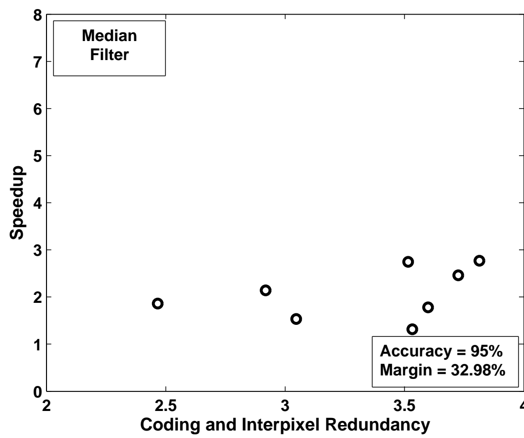
| Algorithm | Canny | Morphological | Kirsch | Corner | Median | Variance |
|---|---|---|---|---|---|---|
| Accuracy (%) | 95 | 95 | 95 | 95 | 95 | 99 |
| Error Margin (%) | 2.09 | 6.52 | 6.25 | 3.19 | 2.53 | 0 |



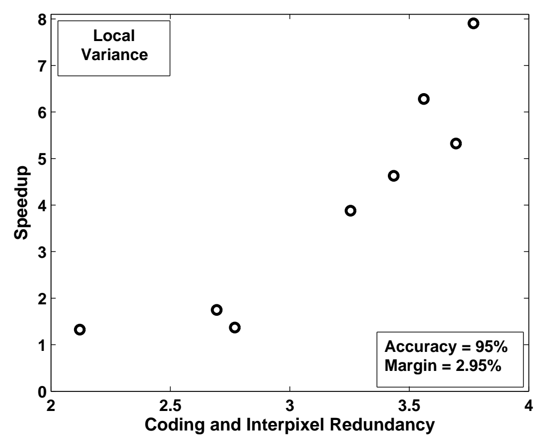
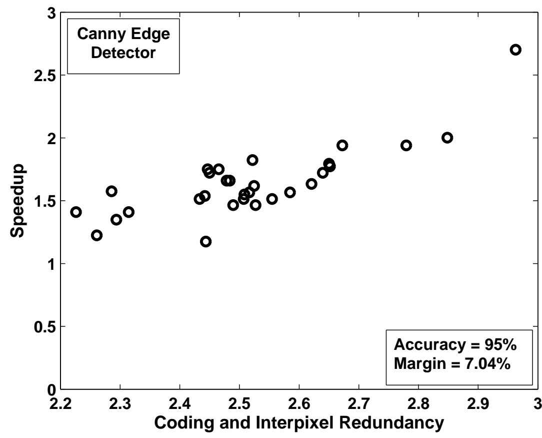
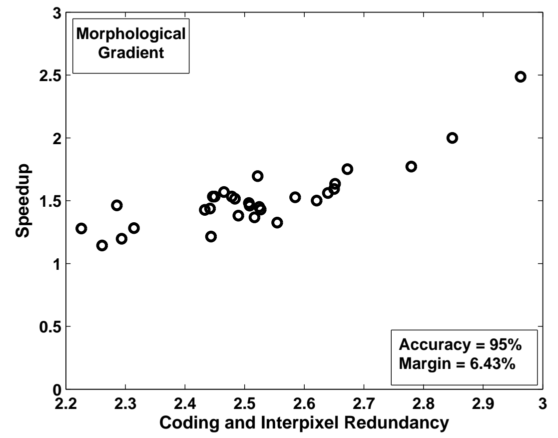
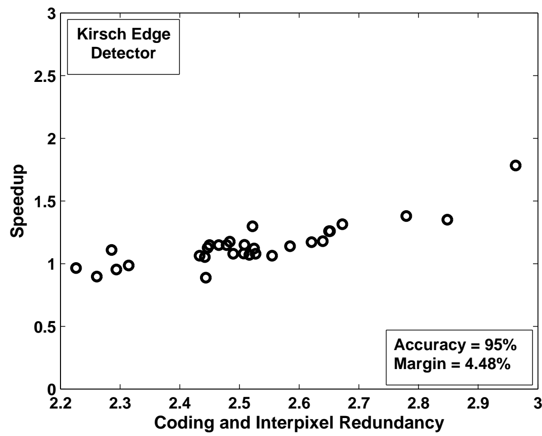
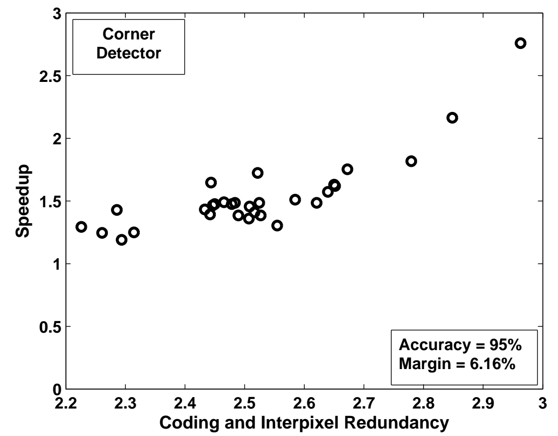



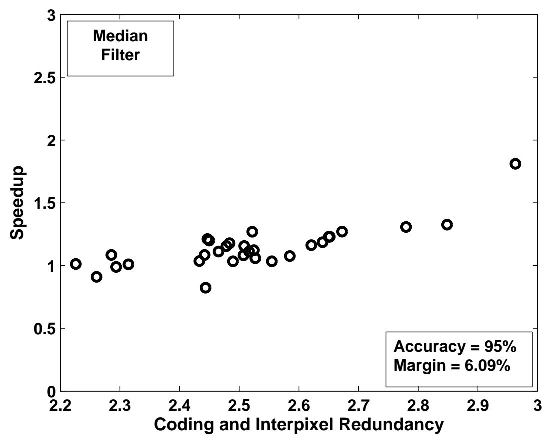

Figure C.7: Speedup versus coding/interpixel redundancy for industrial images run on processor 1.



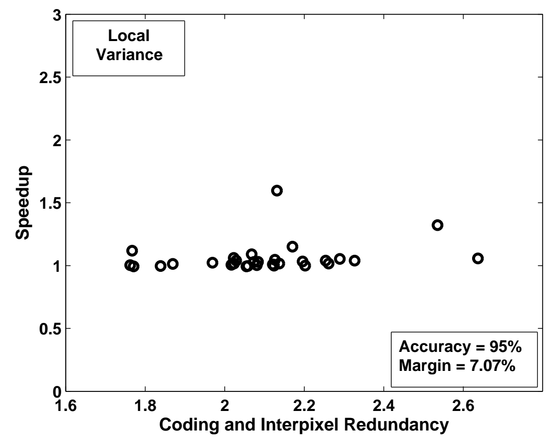




Figure C.8: Speedup versus coding/interpixel redundancy for medical images run on processor 1.

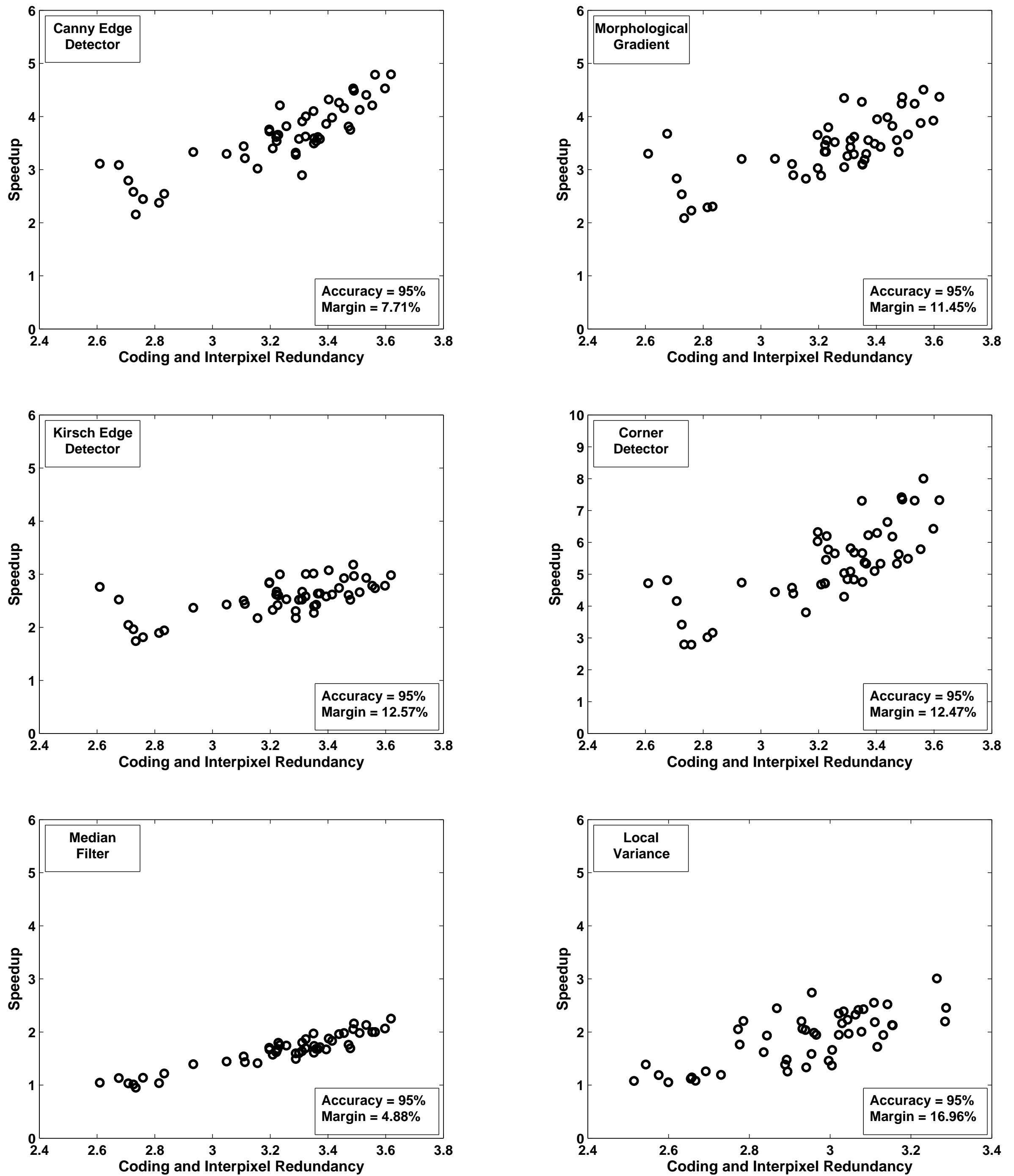



Figure C.9: Speedup versus coding/interpixel redundancy for barcode images run on processor 1.

# Appendix D

# Results for Window Memoization in Hardware

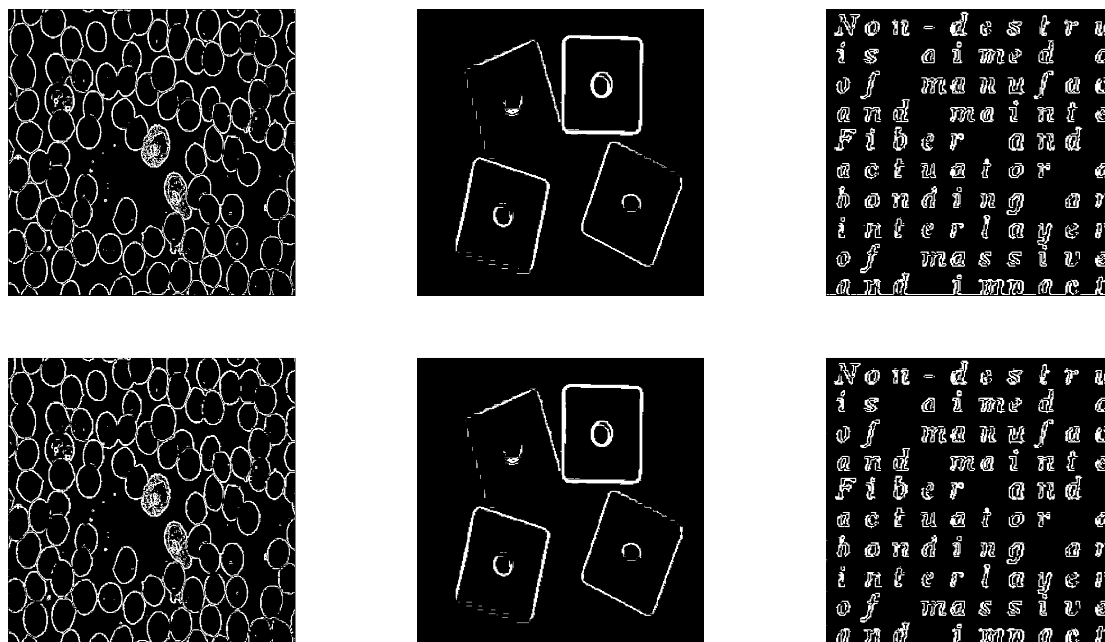



Figure D.1: Results for industrial images for Kirsch edge detector. Top: original results, bottom: window memoization results.

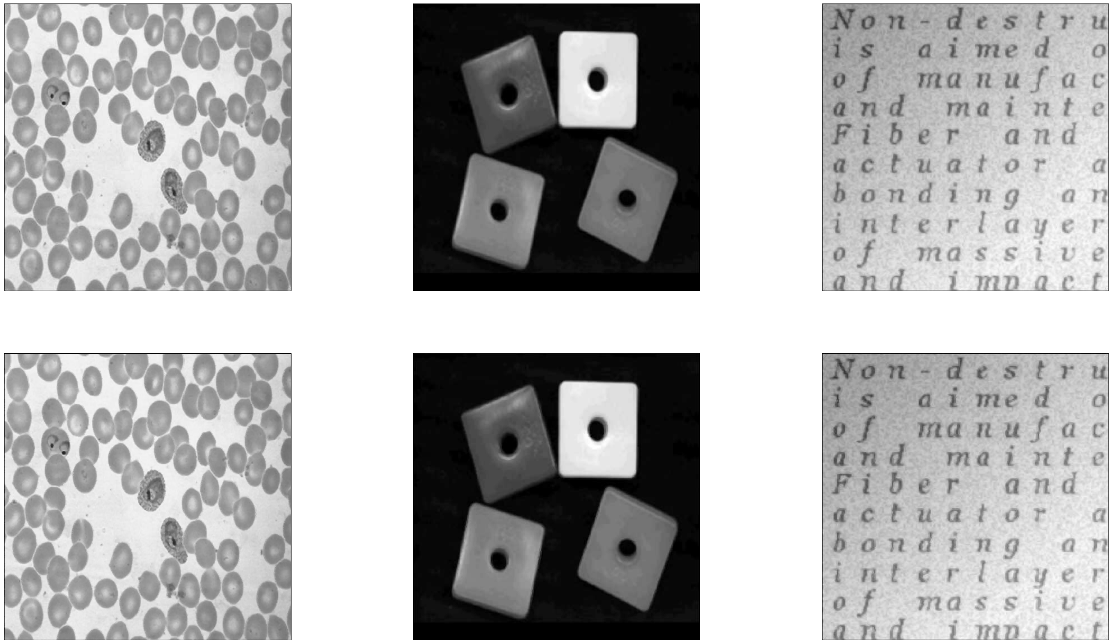



Figure D.2: Results for industrial images for median filter. Top: original results, bottom: window memoization results.

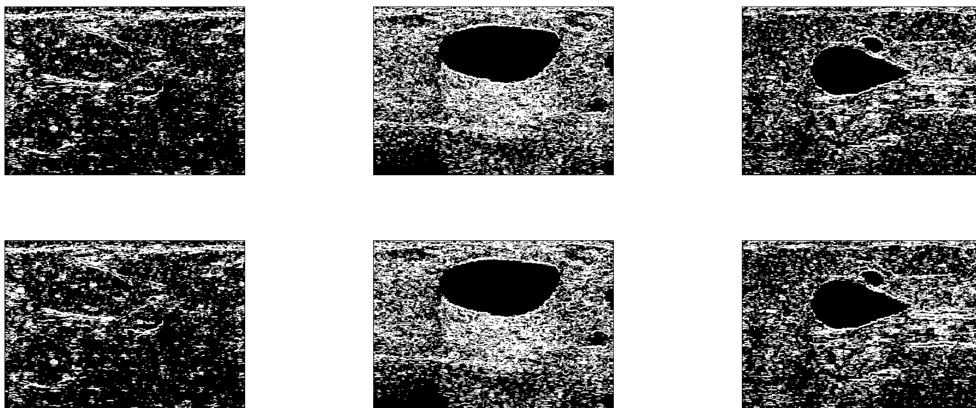


Figure D.3: Results for medical images for Kirsch edge detector. Top: original results, bottom: window memoization results.

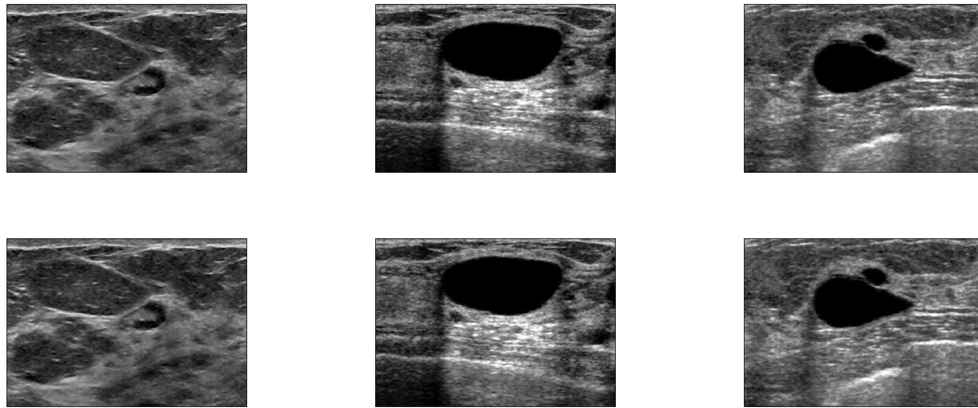


Figure D.4: Results for medical images for median filter. Top: original results, bottom: window memoization results.

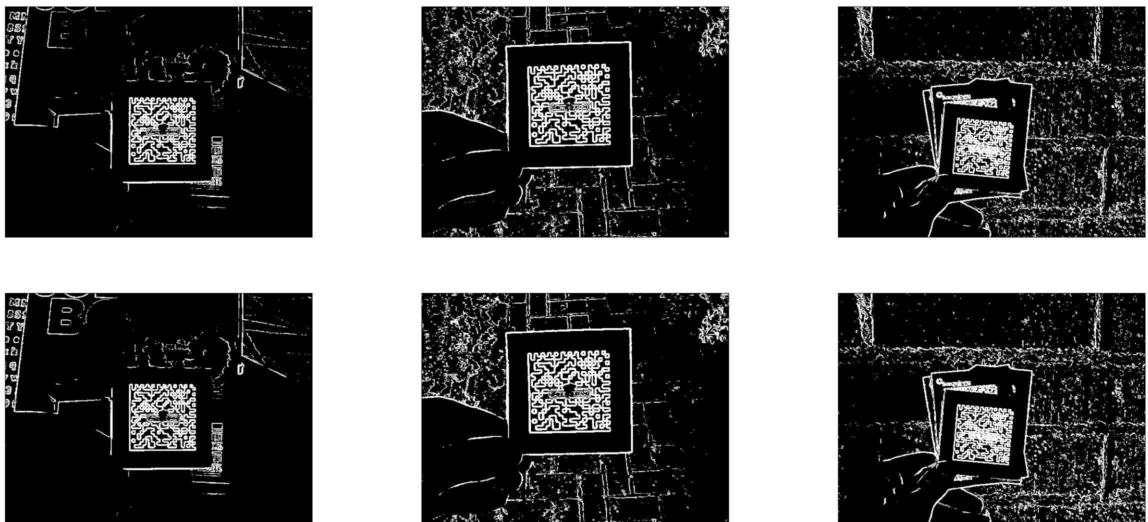


Figure D.5: Results for barcode images for Kirsch edge detector. Top: original results, bottom: window memoization results.

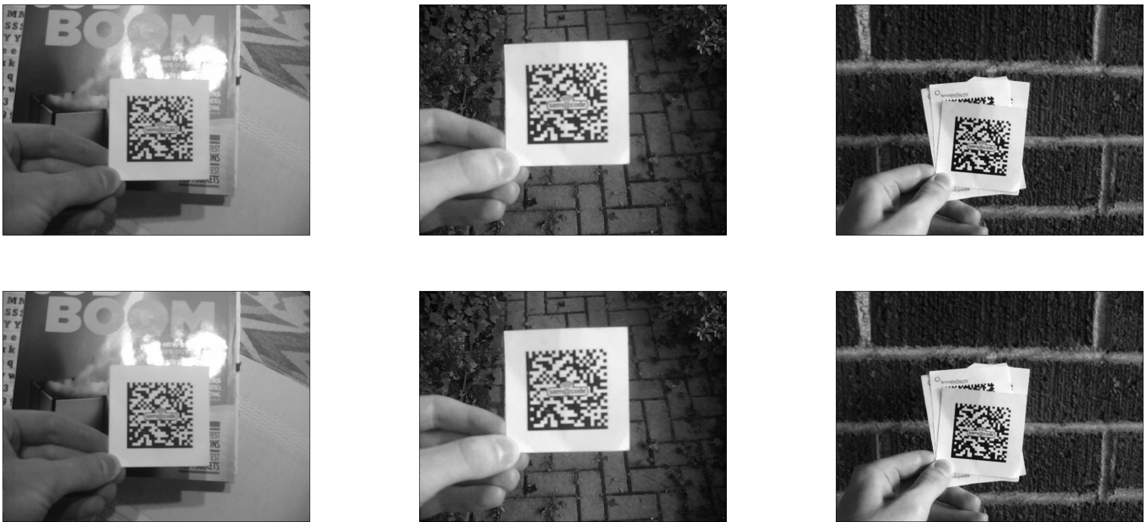



Figure D.6: Results for barcode images for median filter. Top: original results, bottom: window memoization results.

# Bibliography

[1] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, July 2005.

[2] Richard S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, 13(7):422–426, 1970.

[4] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[5] Octavia I. Camps and Tapas Kanungo. Gray-scale structuring element decomposition. *IEEE Transactions on Image Processing*, 5-1:111–120, 1996.

[6] Yang Chen, Abhishek Kumar, and Jun (Jim) Xu. New design of bloom filter for packet inspection speedup. In *IEEE Global Telecommunications Conference*, pages 1–5, 2007.

[7] Daniel Citron, Dror Feitelson, and Larry Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 252–261, 1998.

[8] Daniel Citron and Dror G. Feitelson. Hardware memoization of mathematical and trigonometric functions. Technical report, Hebrew University of Jerusalem, 2000.

[9] Daniel Citron and Dror G. Feitelson. "Look It Up" or "Do the Math": An energy, area, and timing analysis of instruction reuse and memoization. In *Power-Aware Computer Systems: Third International Workshop*, volume LNCS 3164, pages 101–116. Springer-Verlag, 2004.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[11] Semacode Corporation. URL: http://www.semacode.com. Last checked: 2008/12/9.

[12] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley and Sons Inc., 1991.

[13] Richard N. Czerwinski, Douglas L. Jones, and William D. OBrien. Line and boundary detection in speckle images. *IEEE Transactions on Image Processing*, 7(12):1700–1714, 1998.

[14] Andre DeHon. Reconfigurable architectures for general-purpose computing. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1996.

[15] Yonghua Ding and Zhiyuan Li. Operation reuse on handheld devices. In *Languages and Compilers for Parallel Computing (LCPC-03)*, volume 2958 / 2004, pages 273–287. Springer Berlin/Heidelberg, 2003.

[16] Roger Easton. *Basic Principles of Imaging Science*. Chester F. Carlson Center for Imaging Science, Rochester Institute of Technology, 2005.

[17] Michael Egmont-Petersen, Dick de Ridder, and Heinz Handels. Image processing with neural networks, a review. *Pattern Recognition*, 35:2279–2301, 2002.

[18] Ming Fang and Gerd Hausler. Modified rapid transform. *Journal of Applied Optics*, 28(6):1257–1262, 1989.

[19] David J. Field. Relations between the statistics of natural images and the response properties of cortical cells. *Journal of the Optical Society of America A: Optics, Image Science, and Vision*, 4(12):2379–2394, 1987.

[20] Altera Cyclone II FPGAs. URL: http://www.altera.com/products/devices/cyclone2/cy2-index.jsp. Last checked: 2008/12/9.

[21] William T. Freeman and Edward H. Adelson. The design and use of steerable filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(9):891–906, 1991.

[22] V.S Frost, J.A. Stiles, A. Josephine, K.S Shanmugan, and J.C Holtzman. A model for radar images and its application to adaptive digital filtering of multiplicative noise. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(2):157–166, 1982.

[23] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2008.

[24] Chris Harris and Mike Stephens. A combined corner and edge detection. In *The Fourth Alvey Vision Conference*, pages 147–151, 1988.

[25] John Haslett. Maximum likelihood discriminant analysis on the plane using a markovian model of spatial context. *Pattern Recognition*, 18(3-4):287–296, 1985.

[26] Jian Huang and David J. Lilja. Extending value reuse to basic blocks with compiler support. *IEEE Transactions on Computers*, 49:331–347, 2000.

[27] T. S. Huang, G. J. Yang, and G. Y. Tang. A fast two-dimensional median filtering algorithm. *IEEE Trans. on Acoustics, Speech and Signal Processing*, ASSP-27(1):13–18, 1979.

[28] John Hughes. Lazy memo-functions. In *A Conference on Functional Programming Languages and Computer Architecture*, pages 129–146. Springer-Verlag New York, Inc., 1985.

[29] Philips Breast Images. URL: http://www.medical.philips.com/main/products/ultrasound. Last checked: 2008/12/9.

[30] Byeungwoo Jeon and David A. Landgrebe. Classification with spatio-temporal interpixel class dependency contexts. *IEEE Transactions on Geoscience and Remote Sensing*, 30(4):663–671, 1992.

[31] Krishna Kavi and Peng Chen. Dynamic function result reuse. In *International Conference on Advanced Computing and Communication (ADCOM-03)*, 2003.

[32] Russell A. Kirsch. Computer determination of the constituent structure of biological images. *Computers and Biomedical Research*, 4:315–328, 1971.

[33] Robarts Imaging Research Laboratories. URL: http://www.imaging.robarts.ca. Last checked: 2008/12/9.

[34] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value locality and load value prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 138–147, 1996.

[35] SimpleScalar LLC. URL: http://www.simplescalar.com. Last checked: 2008/12/9.

[36] D. G. Lowe. Object recognition from local scale-invariant features. In *The International Conference on Computer Vision ICCV-99, Corfu*, pages 1150–1157, 1999.

[37] J. Mayfield, T. Finin, and M. Hall. Using automatic memoization as a software engineering tool in real-world AI systems. In *The 11th Conference on Artificial Intelligence for Applications (CAIA-95)*, pages 87–93, 1995.

[38] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[39] Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical report, 1980.

[40] Arun N. Netravali and Barry G. Haskell. *Digital Pictures: Representation, Compression and Standards*. Plenum Press, 1995.

[41] Hochong Park and R. T. Chin. Decomposition of arbitrary shaped morphological structuring elements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17-1:1606–1617, 1995.

[42] Donovan Parks and Jean-Philippe Gravel. URL:http://www.cim.mcgill.ca/ dparks/ CornerDetector/trajkovic8.htm. Last checked: 2008/12/9.

[43] William Pugh. An improved replacement strategy for function caching. In *The 1988 ACM conference on LISP and functional programming (LFP-88)*, pages 269–276. ACM, 1988.

[44] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *The 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.

[45] Majid Rabbani and Paul W. Jones. *Digital Image Compression Techniques*. Spie Optical Engineering Press, 1991.

[46] Stephen E. Richardson. Exploiting trivial and redundant computation. In *IEEE Symposium on Computer Arithmetics*, pages 220–227, 1993.

[47] Torsten Seemann. *Digital Image Processing using Local Segmentation*. PhD thesis, School of Computer Science and Software Engineering, Faculty of Information Technology, Monash University, Australia, 2002.

[48] Mehmet Sezgin and Blent Sankur. Survey over image thresholding techniques and quantitiative performance evaluation. *Journal of Electronic Imaging*, 13(1):146, 2004.

[49] John P. Shen and Mikko H. Lipasti. *Modern Processor Design*. McGraw-Hill, 2004.

[50] Stephen M. Smith and J. Michael Brady. Susan - a new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78, 1997.

[51] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *International Symposium on Computer Architecture (ISCA-97)*, pages 194–205, 1997.

[52] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision*. PWS, 1999.

[53] Miroslav Trajkovi and Mark Hedley. Fast corner detection. *Elsevier Science: Image and Vision Computing*, 16:75–87, 1998.

[54] Tinne Tuytelaars and Krystian Mikolajczyk. Survey on local invariant features. *FnT Computer Graphics and Vision*, 1(1):1–94, 2008.

[55] Weidong Wang, Anand Raghunathan, and Niraj K. Jha. Profiling driven computation reuse: An embedded software synthesis technique for energy and performance optimization. In *IEEE International Conference on VLSI Design (VLSID-04 Design)*, page 267, 2004.

[56] Tai-Sen Yu and King-Sun Fu. Recursive contextual classification using a spatial stochastic model. *Pattern Recognition*, 16:89–108, 1983.