1-1-2010

Window Memoization In Software As Applied To Image Processing Algorithms

Tahir Jaffer
Ryerson University

Follow this and additional works at: <http://digitalcommons.ryerson.ca/dissertations>

Part of the [Electrical and Electronics Commons](#)

Recommended Citation

Jaffer, Tahir, "Window Memoization In Software As Applied To Image Processing Algorithms" (2010). *Theses and dissertations*. Paper 1450.

This Thesis is brought to you for free and open access by Digital Commons @ Ryerson. It has been accepted for inclusion in Theses and dissertations by an authorized administrator of Digital Commons @ Ryerson. For more information, please contact bcameron@ryerson.ca.

WINDOW MEMOIZATION IN SOFTWARE AS APPLIED TO IMAGE PROCESSING ALGORITHMS

by

Tahir Jaffer

Bachelor of Engineering in Electrical Engineering, Ryerson University 2007

A thesis

presented to Ryerson University

in partial fulfillment of the
requirements for the degree of

Master of Applied Science

in the program of
Electrical Engineering

Toronto, Ontario, Canada, 2010

©Tahir Jaffer 2010

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis or dissertation.

I authorize Ryerson University to lend this thesis or dissertation to other institutions or individuals for the purpose of scholarly research.

Tahir Jaffer

I further authorize Ryerson University to reproduce this thesis or dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Tahir Jaffer

Abstract

Window Memoization In Software As Applied To Image Processing Algorithms

Master of Applied Science, 2010

Tahir Jaffer

Electrical and Computer Engineering, Ryerson University

A new local image processing algorithm, the Tahir algorithm, is an adaptation to the standard low-pass filter. Its design is for images that have the spectrum of pixel intensity concentrated at the lower end of the intensity spectrum.

Window memoization is a specialization of memoization. Memoization is a technique to reduce computational redundancy by skipping redundant calculations and storing results in memory.

An adaptation for window memoization is developed based on improved symbol generation and a new eviction policy. On implementation, the mean lower-bound speed-up achieved was between 0.32 (slowdown of approximately 3) and 3.70 with a peak of 4.86. Lower-bound speed-up is established by accounting for the time to create and delete the cache. Window memoization was applied to: the convolution technique, Trajkovic corner detection algorithm and the Tahir algorithm.

Window memoization can be evaluated by calculating both the speed-up achieved and the error introduced to the output image.

Acknowledgments

First and foremost I would like to thank God for giving me the opportunity to pursue my education and providing me with all I need to succeed in both my academic endeavors and worldly pursuits.

I wish to express a great deal of gratitude to my supervisor, Dr. Kaamran Raahemifar, for allowing me to work with him and under his supervision. I am truly lucky to have received his guidance and to have learnt from him and his vast experiences. His encouragement of me from my time as an undergraduate student has been sincere and unwavering. He was not only willing to take me on as a student when it seemed all was lost, but championed my cause throughout my graduate career. Without his support I could not have completed this work today. For all his help and insight, I am truly grateful; thank you Professor Raahemifar.

I am deeply thankful to my family for all the help that they have given me and the opportunities that they have provided me with. I am grateful to my parents for their wholehearted support and constant encouragement of me throughout my whole life. Without them to lean on, this thesis would not have been possible.

Lastly, I would like to thank my wife Fatemah, who has provided me much support and continuous encouragement while I have been working on my thesis.

Dedication

To my late grandfather Mohamed Hussein Kermali Alibhai and my grandmother Nargis Kermali Alibhai.

Without your love and support this would not have been possible.

Table of Contents

AUTHOR'S DECLARATION.....	ii
Abstract.....	iii
Acknowledgments.....	iv
Dedication.....	v
Table of Contents.....	vi
List of Tables.....	ix
List of Figures.....	xi
Chapter 1 Introduction.....	1
1.1 Motivation.....	1
1.2 Objective.....	2
1.3 Methodology.....	2
1.4 Contributions.....	3
1.5 Organization.....	4
Chapter 2 Literature Survey.....	5
2.1 Image Data Redundancy.....	5
2.1.1 Coding Redundancy.....	5
2.1.2 Inter-pixel Redundancy.....	7
2.1.3 Psycho-visual Redundancy.....	11
2.2 Previous Work in Memoization and Computational Redundancy.....	11
2.2.1 Data Locality.....	12
2.2.2 Memoization in Software.....	14
2.3 Local Image Processing Algorithms.....	17
2.3.1 Convolution Algorithm.....	17
2.3.2 Trajkovic Corner Detection Algorithm	20

2.3.3 Proposed Image Processing Algorithm (Tahir algorithm).....	23
2.4 Set of Test Images.....	25
2.5 Summary.....	26
Chapter 3 Memoization.....	28
3.1 Theory of Memoization.....	28
3.1.1 Memoization Process.....	29
3.1.2 Ideal Memoization.....	31
3.1.3 Tolerant Memoization.....	32
3.1.4 Computational Redundancy.....	33
3.1.5 Window Memoization.....	34
3.1.6 Summary.....	35
3.2 Software Implementation of Memoization.....	36
3.2.1 Software Memoization Steps.....	36
3.2.2 Check Cache.....	39
3.2.3 Eviction Policy.....	44
3.2.4 Tolerant Memoization in Software.....	46
3.2.5 Summary.....	47
3.3 Summary.....	48
Chapter 4 Proposed Implementation of Memoization and Challenges Encountered.....	50
4.1 Memoization in Software:.....	50
4.2 Specific Innovations in the Proposed Implementation.....	53
4.2.1 Generating Symbols.....	53
4.2.2 Hash Table.....	54
4.2.3 Eviction Policy.....	55
4.2.4 Hash Functions.....	56

4.2.5 Tolerant Memoization.....	58
4.3 Methodology.....	58
4.3.1 Platform.....	59
4.3.2 Implementation Method.....	59
4.3.3 Testing and Evaluation Technique.....	60
4.4 Time Requirements and Speed-up.....	63
4.5 Error.....	68
4.5.1 Mean Pixel Error.....	68
4.5.2 SNR.....	69
4.6 Summary.....	71
Chapter 5 Results and Observations.....	73
5.1 Resulting Speed-up.....	73
5.2 Resulting Error.....	76
5.3 Comparison and Analysis of Speed-Up vs. Error.....	78
5.4 Results Comparison with Results from Khalvati [13] Technique.....	89
5.5 General Observations and Summary of Results.....	90
Chapter 6 Conclusion.....	92
Appendix A	
More Results.....	96
Bibliography.....	100

List of Tables

Table 2.1: Horizontal Mask Operator.....	18
Table 2.2: Vertical Mask Operator.....	18
Table 2.3: Combined Mask Operator.....	18
Table 2.4: Sample input image (5x5).....	19
Table 2.5: Output from convolution algorithm for sample input image from Table 2.4.....	19
Table 2.6: Trajkovic corner detection algorithm 8-neighbour definition.....	21
Table 2.7: Pseudo-code summary of the Trajkovic corner detection algorithm [19], [25].....	23
Table 3.1: Sub-steps required to check reuse table for previous results matching current input.....	39
Table 3.2: Generating a symbol.....	40
Table 4.1: Generating symbol using the overlapping windows.....	54
Table 5.1: Comparison of memoized algorithm speed-up for each algorithm at input shift of three.....	74
Table 5.2: Comparison of memoized algorithm speed-up for each algorithm with reuse table size of 8000	74
Table 5.3: Comparison of memoized algorithm output SNR (dB) for each algorithm at input shift of three	77
Table 5.4: Comparison of memoized algorithm output SNR (dB) for each algorithm with reuse table size of 8000.....	77
Table 5.5: Comparison of algorithm results with and without application of memoization; application with reuse table size of 8000 and input shift of 3.....	78
Table 5.6: Resulting speed-up for convolution algorithm after memoization is applied. Speed-up is sorted based on input shift and table size.....	84
Table 5.7: Output SNR (dB) for convolution algorithm after memoization is applied. SNR is sorted based on input shift and table size.....	84

Table 5.8: Resulting hit rate for convolution algorithm when memoization is applied. Hit rate is sorted based on input shift and table size.....	84
Table 5.9: Processing time for convolution algorithm when memoization is applied. Processing time is sorted based on input shift and table size.....	85
Table 5.10: Resulting speed-up for Tahir algorithm after memoization is applied. Speed-up is sorted based on input shift and table size.....	86
Table 5.11: Resulting output SNR (dB) for Tahir algorithm after memoization is applied. SNR is sorted based on input shift and table size.....	86
Table 5.12: Resulting hit rate for Tahir algorithm when memoization is applied. Hit rate is sorted based on input shift and table size.....	86
Table 5.13: Processing time for Tahir algorithm when memoization is applied. Processing time is sorted based on input shift and table size.....	87
Table 5.14: Resulting speed-up for Trajkovic corner detection algorithm after memoization is applied. Speed-up is sorted based on input shift and table size.....	88
Table 5.15: Resulting output SNR (dB) for Trajkovic corner detection algorithm after memoization is applied. SNR is sorted based on input shift and table size.....	88
Table 5.16: Resulting hit rate for Trajkovic corner detection algorithm when memoization is applied. Hit rate is sorted based on input shift and table size.....	89
Table 5.17: Processing time for Trajkovic corner detection algorithm when memoization is applied. Processing time is sorted based on input shift and table size.....	89

List of Figures

Figure 2.1: Input image (left) and resulting output from convolution algorithm (right) [5].....	20
Figure 2.2: Calculating change in image intensity in a given direction [19].....	20
Figure 2.3: Input image (left) and resulting output image when applying Trajkovic corner detection algorithm (right) [19].....	23
Figure 2.4: Input image (right) and resulting output image when applying Tahir algorithm (right).....	25
Figure 2.5: Sample images from image database [5].....	27
Figure 3.1: Flow-chart for theoretical memoization.....	30
Figure 3.2: Flow-chart for general software implementation of memoization.....	38
Figure 4.1: Proposed Implementation of Window Memoization in Software.....	51
Figure 4.2: a) top-left, graph of hit rate vs. table size for multiply and divide key generation functions. b) top-right, graph of speed-up vs. table size for multiply and divide key generation functions. c) bottom-left, graph of difference in hit rate between multiply and divide key generation functions vs. table size. d) bottom-right, graph of difference in speed-up between multiply and divide key generation functions vs. table size.....	57
Figure 5.1: Graph of memoized algorithm speed-up for each algorithm at input shift of three.....	74
Figure 5.2: Graph of memoized algorithm speed-up for each algorithm with reuse table size of 8000.....	75
Figure 5.3: Graph of memoized algorithm output SNR (dB) for each algorithm at input shift of three.....	77
Figure 5.4: Graph of memoized algorithm output SNR (dB) for each algorithm with reuse table size of 8000.....	78
Figure 5.5: Convolution algorithm with and without memoization. (left to right, top to bottom) original input, output with no memoization, slowest output - shift 8 and cache size 256000, fastest output - shift 2 and cache size 1000, output with most error - shift 1 and cache size 256000, output with least error - shift 8 and cache size 1000, output - shift 3 and cache size 8000.....	79

Figure 5.6: Tahir algorithm with and without memoization. (left to right, top to bottom) original input, output with no memoization, slowest output - shift 8 and cache size 256000, fastest output - shift 1 and cache size 8000, output with most error - shift 1 and cache size 256000, output with least error - shift 8 and cache size 1000, output - shift 3 and cache size 8000.....	80
Figure 5.7: Trajkovic algorithm with and without memoization. (left to right, top to bottom) original input, output with no memoization, slowest output - shift 8 and cache size 256000, fastest output - shift 1 and cache size 1000, output with most error - shift 1 and cache size 256000, output with least error - shift 8 and cache size 1000, output - shift 3 and cache size 8000.....	81
Figure A.1: Graph of hit ratio vs. input shift for convolution algorithm.....	96
Figure A.2: Graph of SNR vs. input shift for convolution algorithm.....	96
Figure A.3: Graph of speed-up vs. table size and input shift for convolution algorithm.....	97
Figure A.4: Graph of hit ratio vs. input shift for Tahir algorithm.....	97
Figure A.5: Graph of normalized SNR vs. input shift for Tahir algorithm.....	98
Figure A.6: Graph of speed-up vs. table size and input shift for Tahir algorithm.....	98
Figure A.7: Graph of hit ratio vs. input shift for Trajkovic corner detection algorithm.....	99
Figure A.8: Graph of SNR vs. input shift for Trajkovic corner detection algorithm.....	99
Figure A.9: Graph of speed-up vs. table size and input shift for Trajkovic corner detection algorithm.....	99

Chapter 1 Introduction

The introduction is broken into five sections. First, the motivation for this work is explained. Second, the objective is stated. Third, the methodology is introduced. Fourth, the contributions of this work is summarized. Fifth, the organization of the remainder of the thesis is presented.

1.1 Motivation

Many advances and breakthroughs in the fields of digital computing and multimedia have recently occurred and are continuing today. There has been a proliferation of digital multimedia content world-wide and accompanying advances in multimedia processing algorithms. Multimedia content is usually both data and processor intensive. Meanwhile, the role of multimedia content in everyday life is getting larger. As the volume of content increases so does the need to process it. New and expanded purposes for multimedia content are continually evolving. Much of the multimedia is used in soft and hard real-time systems where it is crucial to meet certain performance requirements. Applications such as medical imaging and security systems are two examples of fields where the timing is of vital importance. Anything that slows down these processes is undesirable, whereas speeding the processes up is highly desirable. In fact, the greater the speed-up, the better it is. In a soft real-time application, such as medical imaging, getting results faster can decrease the mortality rate. Thus, faster processing may be required as data processing algorithms become more complex. Similarly, in a hard real-time application, such as robotic vision, speeding up the processing can allow for the implementation of more advanced artificial intelligence algorithms. Therefore, even in applications where current algorithms are able to process data within current time constraints, they may not be able to meet the time constraints for more complex algorithms.

Traditionally, when faced with time requirements that could not be met, new hardware was designed to meet the more stringent requirements. Many data-intensive processes, such as ultrasound images are

time-sensitive and constantly require more advanced hardware in order to process all of the data in real-time. Unfortunately, upgrading hardware is not always a feasible choice. It is common to find that the required hardware does not exist, is too expensive or would take too long to develop. These situations require alternate solutions.

1.2 Objective

This research aims to show that a potential exists to create a software solution instead of a hardware one. A software solution can be expected to be cheaper and easier to implement compared to a hardware solution. The goal of this research is to show that a software optimization process, window memoization, can be used to increase the performance of real-time image processing algorithms by speeding them up. This may be of particular use in the medical imaging industry. The potential benefits of using software to increase speed instead of hardware include lower cost and faster implementation.

The major thrust of this thesis is to reduce the number of redundant calculations performed by an image processing algorithm. Redundant calculations are those calculations that are repeated. Performing a set of calculations on a given input once should be sufficient. When the same input repeats, if the previous result can be reused, the calculation does not need to be redone. Similar to image compression algorithms that take advantage of redundancy in data to reduce image size, memoization [13] [1] [18] aims to take advantage of the redundancy in calculations to save time.

1.3 Methodology

The window memoization process makes use of a reuse table to store previous calculation results for future reuse. In the first instance of a particular input, the full set of calculations for the data processing algorithm must occur. However, once this is complete, the results are stored in the reuse table so that if the same input ever recurs again, the previous result is reused and the calculation is avoided/skipped.

Previous work in the area of image data has shown that there are two major types of redundancy in images: coding redundancy and inter-pixel redundancy. Coding redundancy results from individual pixels storing data in more bits than is strictly necessary. Inter-pixel redundancy results from having similar areas (windows) of pixels in the image [13]. In window memoization, similar windows produce similar results and are candidates for reuse. To measure the inter-pixel redundancy in an image, the percentage of repeated windows can be used. Implementation of window memoization takes advantage of both types of redundancy in images. Furthermore, greater redundancy in images leads to greater potential time savings.

Using these ideas, a few concepts are introduced: hit rate, ideal memoization, tolerant memoization, and perfect window memoization. Hit rate of an image is the number of windows for which a previous result can be reused. Ideal memoization is a theoretical concept which cannot be practically implemented but can be used to provide a best-case scenario for memoization. Tolerant memoization refers to an implementation of memoization where additional time savings are introduced at the cost of also introducing error to the output response. Perfect window memoization refers to an implementation of memoization where no error is introduced.

Three image algorithms of varying complexity are analyzed in this thesis: a simple, a moderately-complex and a complex algorithm. The simple image algorithm studied is the convolution technique. The Trajkovic corner detection algorithm [25] is the moderately-complex algorithm studied. A new image processing algorithm, the Tahir Algorithm, of the complex variety is proposed and studied.

1.4 Contributions

The findings show that the lower bound on the speed-up achieved by window memoization is between 0.32 (slowdown of a factor of approximately 3) to 3.70 with a peak of about 4.86.

The results produced show that speed-ups for simple algorithms are difficult to achieve, while speed-ups for moderately and highly complex algorithms are not only possible, but can be optimized based on reuse table size and tolerance.

The contributions of this research include:

- A new image processing algorithm to filter input images
- Implementation of window memoization and its application to images of the natural class
- An evaluation technique for memoization applied to images in terms of both speed-up and error.

1.5 Organization

The outline of the remainder of this thesis is as follows: the thesis is broken into 6 chapters and one appendix. In Chapter 2, the literature survey is presented. The survey includes previous work in the field and background on the image processing algorithms implemented is presented. In Chapter 3, the theory of memoization and general implementation in software is presented. In Chapter 4, the proposed method of window memoization in software and the methodology is presented. In Chapter 5, the results from the implementation of window memoization in software are presented and evaluated. In Chapter 6, the conclusions are presented. The Appendix includes more results and graphs that were obtained.

Chapter 2 Literature Survey

In this chapter a number of different areas are explored. The first area is redundancy in image data. The second area is previous work in the area of memoization, specifically in software. The third area is local image processing algorithms. Specifically, the three local image processing algorithms that are implemented in testing the window memoization technique are explored. The fourth area is image databases, during which time, the image database used in testing the window memoization technique is introduced. The theory of memoization and its general implementation is discussed in Chapter 3. Chapter 4 includes the proposed technique and the methodology of implementation. Chapter 5 presents the results and observations.

2.1 Image Data Redundancy

Image processing can take advantage of three types of image data redundancy: coding redundancy, inter-pixel redundancy and psycho-visual redundancy [9]. Coding redundancy refers to pixels being represented by more bits than is strictly necessary. Inter-pixel redundancy refers to the similarity pixels share with their neighbors. Psycho-visual redundancy refers to the excess information stored in images that the human eye ignores or does not perceive.

2.1.1 Coding Redundancy

Coding redundancy refers to the redundancy that exists in an image due to pixels being represented by more bits than is required [9]. So then naturally, one would ask how to determine the minimum number of bits required to represent a pixel and how to determine the minimum number of bits required to represent an image. To determine the minimum number of bits needed to represent an image, the amount of information in the image must be determined. Entropy can be used as a measure of the average amount of information per pixel in an image. The entropy of an image determines the minimum number

of bits per pixel that are required to accurately represent the image. Images with high entropy require more bits per pixel whereas images with lower entropy require fewer bits per pixel. The extreme examples of images with low and high entropy are as follows: an image with equal probability of gray levels has the highest entropy, the lowest coding redundancy and thus can have the fewest bits removed, if any. An image where all the pixels have the same single gray level has the lowest entropy, the highest coding redundancy and can have the most bits removed. These are extreme examples and do not typically occur in real-life examples. For natural images, those that are used in this work, the entropy is somewhere between these two extremes. On the scale of entropy, natural images are typically closer to the latter extreme than the former extreme [24]. The entropy of an image can be calculated as follows [9]:

$$H = \sum_{i=0}^{GL-1} p_i * \log_2(p_i) \quad (2.1)$$

where H is the entropy of the image, GL is the number of gray levels in the image, p_i is the probability of the occurrence of a particular gray level, i , in the image and $\log_2$ is the logarithmic function with base two. In order to determine the coding redundancy CR of an image from the entropy H of the image, the following formula can be used[24]:

$$CR = \log_2(GL) - H \quad (2.2)$$

There are many image processing algorithms that have taken advantage of coding redundancy in their application. Image processing algorithms taking advantage of coding redundancy are most prevalent in the image compression field. Some common image compression algorithms taking advantage of coding redundancy are: Golomb coding, Huffman coding, and arithmetic coding [9]. All three of these image compression algorithms are lossless.

Coding redundancy, along with inter-pixel redundancy, **which is** explained in the next subsection, are joint factors in computational redundancy. Computational redundancy is a key part of the implementation of window memoization for local image processing algorithms, as is explained in Chapter 3.

2.1.2 Inter-pixel Redundancy

Inter-pixel redundancy occurs as a result of the correlation between neighboring pixels [13]. Inter-pixel redundancy along with, the previously-discussed, coding redundancy are joint factors in computational redundancy. Computational redundancy is a key part of the implementation of window memoization for local image processing algorithms, as is explained in Chapter 3.

In real-world images, neighboring pixels are highly correlated because they usually belong to the same object, and have similar gray levels. The correlation arises because objects typically are characterized by similar coloring or texturing and have a particular structure or geometric shape. Based on the correlation existing between pixels in an image, high levels of compression can be achieved [9]. Using inter-pixel redundancy to reduce the memory requirement of images, results in space savings, with a resulting image compression ratio of 16-200 [12]. Some of the image compression techniques used to exploit the inter-pixel redundancy of images include: run-length coding, difference coding, loss-less predictive coding, LZW coding, vector coding and lossy predictive coding. The latter of which is a lossy technique while the remainder are all lossless [8]. Such high levels of compression can be achieved with minimal to no loss of accuracy proving that there is a high level of inter-pixel redundancy in natural images.

Image compression is not the only arena of image processing algorithms that take advantage of the inherent inter-pixel redundancy of images. Inter-pixel redundancy has also been taken advantage of in image segmentation techniques. Image segmentation is a class of image processing algorithms that attempt to segment images into their constituent objects. For example, in a picture where there are two dogs, image segmentation could be used to segment each dog from the background of the image. Three

types of image segmentation algorithms that take advantage of inter-pixel redundancy are: finite-element models, Spline-based models, and Fourier-descriptor-based models [26].

To measure the inter-pixel redundancy of an image, there are typically three methods that can be used: mapping transforms, Markov model, and autocorrelation. Each of these are briefly discussed here. However, none of these techniques will be used to measure the inter-pixel redundancy of images in this work, as that is outside the intended scope.

2.1.2.1 Mapping Transforms

The aim of mapping transforms is to convert an image into a new format where there is no redundancy or correlation between pixels in the image in the resultant format [9]. In the resultant image, in the new format, each pixel contains only the information that is carried solely in the corresponding pixel from the original image; all common information or information that correlates with another pixel is removed.

In mapping transforms, the inter-pixel redundancy of a particular pixel is calculated by taking into account its closest 8 (if a 3x3 window is used) or 24 (if a 5x5 window is used) surrounding neighbors. The inter-pixel redundancy between the pixel and its closest neighbors is then removed in the output image. The size of the mapping window, in the form of the number of closest neighbors used, plays a role in the mapping transform based on the weighting applied to the neighboring pixels. The mapping transform formula is as follows:

$$IR(x, y) = \sum_{i=-m_1}^{i=m_1} \sum_{j=-m_1}^{j=m_1} \alpha_{ij} I(x-i, y-j) \quad (2.3)$$

where I is the input image, x is the column of the subject pixel, y is the row of the subject pixel, IR is the information contained in the subject pixel, $2m_1+1$ is the size of the window, Σ is the sum operation and α_{ij} is the weight given to the neighboring pixel located at position (i, j) in the window.

Once the input image has been mapped to the new format, the entropy of the new image can be taken, using the entropy formula discussed previously. When comparing the entropy of the two images, it is found that, if applied correctly, the resultant image would always have less than or equal entropy as compared to the original image. In fact, for normal images, the resultant image would have significantly lower entropy than the input image [9]. The reduction in entropy can be attributed to the removal of inter-pixel redundancy in the image.

2.1.2.2 Markov Model

The Markov model makes the assumption that the probability of gray level at a particular pixel is dependent on the gray levels of the closest neighboring pixels [8]. The number of pixels that are used to determine the probability function for the pixel is known as the order of the Markov source. If all pixels in an image are assumed to be independent of each other, then the resultant image is a Markov source of order zero. The original image is a Markov source of order zero which also corresponds to an image at its highest level of entropy. A Markov source of order k considers a window of k pixels in an input image as a k -D (1-D, 2-D, etc.) grey-level vector. For a given image of Markov source order one, where there is some correlation between neighboring pixels that can be removed, then the corresponding entropy for the image is lower than that of an image of Markov source order zero. Similarly, if there is correlation between neighboring pixels, as the Markov order of an image is increased, the corresponding entropy of the output image drops. The decrease in entropy is due to the removal of inter-pixel redundancy.

2.1.2.3 Auto-correlation

While the Markov source order is one measure for inter-pixel redundancy of images, auto-correlation is another. Auto-correlation is used to measure the inter-pixel redundancy between two adjacent pixels [13]. Auto-correlation measures the correlation of a signal with a shifted version of the signal where the

signal is shifted in time or in space, for a multi-dimensional signal [17]. The auto-correlation of an input is:

$$R(n) = E[(x - \mu)(x_n - \mu)] \quad (2.4)$$

where $R(n)$ is the auto-correlation, $E[x]$ is the expected value of the input, x is the input, x_n is the input shifted by n , and μ is the mean of the input. The expected value of the input is:

$$E(x) = \sum_{(x=-\infty)}^{(x=\infty)} x p(x) dx \quad (2.5)$$

where Σ is the sum function, x is the input, and $p(x)$ is the probability density function of the input [17].

The auto-correlation technique however, uses a more generalized approach to measure the inter-pixel redundancy of a pixel with its neighboring pixels. An image can be shifted in any one of 8 directions: north, south, east, west, northwest, northeast, southwest, southeast. The autocorrelation of an image shifted by x pixels is as follows:

$$AC(n) = \frac{E[(I - \mu)(I_n - \mu)]}{E[(I - \mu)^2]} \quad (2.6)$$

where $AC(n)$ is the auto-correlation of the image as defined by the auto-correlation technique and I is the input image. Also of note, is that the denominator in the auto-correlation function is the variance of the input image [9]. $AC(1)$ is where the auto-correlation of an input image with a copy of itself shifted one pixel, is calculated. $AC(1)$ also corresponds to the inter-pixel redundancy between adjacent pixels in an input image. Khalvati [13] showed that for natural images, when adjacent pixels have close gray levels, a high $AC(1)$ results.

2.1.3 Psycho-visual Redundancy

Psycho-visual redundancy in images occurs because the human brain does not process all information that it receives from the human eye equally. When looking at an image, the eye will consider some parts, or pieces of information, in the image as being more important than others. The eye will focus on the more important parts and will give less focus on the less important parts. These lesser important parts are the portions of the image data that are psycho-visually redundant. From an image compression standpoint, little loss would be perceived if only the psycho-visually redundant areas in an image were compressed or removed [9].

2.2 Previous Work in Memoization and Computational Redundancy

In 1968 Donald Michie [18] proposed the following: "It would be useful if computers could learn from experience and thus automatically improve the efficiency of their own programs during execution. A simple but effective rote-learning facility can be provided within the framework of a suitable programming language". This idea is the building block for memoization. While Michie presented a method to increase efficiencies through software, memoization is not limited to only software; memoization can also be implemented in hardware.

Previous work in the fields of memoization and computational redundancy can be split into two general areas: those that deal with software and those that deal with hardware. This chapter presents the previous work in the area of software memoization only.

For the most part, previous work in the fields of memoization and computational redundancy have been generic and designed for general use. There has been limited work in finding specific solutions for specific applications, such as applying memoization to local image processing algorithms.

An important underlying concept for both software and hardware memoization techniques is data locality. Data locality from a hardware perspective is different from a software perspective. Data locality

is discussed in more detail as it applies to software in the next subsection. The second subsection deals with the previous work in the area of software implementation of memoization.

2.2.1 Data Locality

An important feature to discuss when looking at a software implementation of memoization is data locality. From a software perspective, data locality refers to the probability that a particular calculation will be repeated. Repeated calculations lead to computational redundancy [22]. Software implementations of memoization aim to reduce this form of data locality.

For software implementations of memoization, there are two forms: bottom-up and top-down. The top-down approach is called memoization and the bottom-up approach is called dynamic programming [13].

2.2.1.1 Top-down approach

When memoization is applied, using the top-down approach (memoization), to a software algorithm an analysis is first done to see which steps can benefit from being memoized. The analysis is necessary because the memoization technique is designed to take frequently used calculations by the memoized process and to store the results in memory. When a recalculation is required, rather than recalculating the result, it is recalled from memory. This skipping of calculations can lead to time savings if the amount of time required to retrieve the result from memory is less than the amount of time required to recalculate the result. If, however, retrieving the result from memory takes longer than it would to recalculate a result, then applying memoization to that step or possibly even the algorithm as a whole, would not be effective.

In the top-down approach, the underlying algorithm is not changed. Rather, some steps in the algorithm may be skipped and a previous result may be reused. If memoization is applied to a software algorithm, then the only change to the original program flow is that some steps are added into the

algorithm so that when there is a complex series of calculation required, prior to performing the calculations, a reuse table is checked to see if the calculation was previously performed. If a previous matching calculation can be found, the result is reused. If a previous matching result cannot be found, then the calculation is performed and a decision is made whether to insert the results from the new calculation into the reuse table.

2.2.1.2 Bottom-up approach

When memoization is applied, using the bottom-up approach (dynamic programming), to a software algorithm, the algorithm is broken down into sub-problems. It is assumed that each of these sub-problems has an optimal solution. It is further assumed that combining the optimal solutions for all of the sub-problems will lead to an optimal solution for the problem as a whole. In this way, a solution is literally found from the bottom-up. The sub-problems are called overlapping sub-problems. What is interesting to note is that for most algorithms, the sub-problems that require solutions, frequently recur [13]. Memoization in dynamic programming attempts to avoid repeating redundant sub-computations.

In the top-down approach, calculations are initiated based on the input received. However, in the bottom-up approach, sub-computations are all executed at least once, ignoring whether it is actually required. For most inputs, like those of the natural image class, only a small fraction of the sub-computations are actually required so memoization using the top-down approach would produce faster results. The bottom-up approach, however, "can benefit from the regular pattern of accesses to reuse memory to reduce the memoization overhead time and the required memory size [13]". In Chapter 4 the results from the memoization implementation are analyzed and it is found that the results are heavily dependent on memoization overhead time and that it may possibly be better to use the bottom-up approach, at least when inputs are received for the first time. Once multiple inputs are received and the reuse tables begin filling, it may become more practical to use the top-down approach.

2.2.2 Memoization in Software

Michie [18] was the first to propose the concept of a computer learning through software in order to speed up the time it takes to complete a repetitive process. Michie proposed a system where the software would remember previous results and recall the result rather than recalculate when the same problem was encountered again. Michie proposed using a 'rote' (we will call it a reuse table or cache) to store the results from previous calculations. Prior to calculating the result for an input, the rote would be consulted, and if a match was found, the previous result would be reused. If no match was found, the calculation would then occur as per normal. In Michie's system, every entry in the rote would be checked prior to determining that a match did not exist. This method of checking for a previous matching calculation is not particularly efficient and when the rote gets large, it would take longer to find a match than to merely recalculate the output.

Hughes [11] proposed a technique to improve on Michie's work, so that the new input does not have to exactly match the old input. Rather, in addition to exact matches, if the new input is stored at the same address in the reuse table as a previous input, then it is considered a match. This added step allows for increased hit rates without significantly sacrificing memory and without greatly expanding the reuse table.

Pugh [21] proposed a new technique to improve the caching and eviction of data. He proposed a technique to look at both the calculation time and the number of hits an entry received when deciding whether to evict an entry. His implementation was able to show an improvement by a factor of two over a standard eviction policy.

Richardson [22] proposed a memoization technique that in addition to trying to memoize redundant calculations, it also eliminated trivial calculations. Richardson categorized a trivial operation as one of the following: multiplication, division and square root. Multiplication is trivial where at least one of the two inputs is 0, -1 or +1. Division is trivial where the absolute value of the inputs are equal, or where the

numerator is zero. The square root operation is trivial where the input is either zero or one. Richardson found trivial calculations accounted for up to 67% of all calculations for some algorithms. Using his technique, he was able to speed-up calculations by up to 47.8%.

Ding [7] proposed a function reuse technique to be used with handheld devices in order to reduce both the power consumption and the time required to compute results. He proposed breaking the code into code segments. He identified which code segments were the most frequently used, and decided to implement memoization on those code segments. He was able to achieve speed-ups ranging from 1.18 to 4.25 while also achieving a reduction in power usage, ranging from 5.9% to 55.1%.

Alvarez [4] proposed an innovation to the classic memoization technique. He proposed to relax the reuse requirements by using tolerant memoization rather than the standard strict match requirements typically used. In his technique, it is not necessary for an input to exactly match a previous input; it has to merely be sufficiently similar in order to be reused. Using a similar but not identical previous input to derive an output can introduce some error to the output. However, because the match requirement is not as strict, he is able to increase the hit rate of inputs and thus increase the speed up. So, at a cost of introducing error to the result, he was able to produce even greater speed-ups. For multimedia applications, he was able to justify adding small amounts of error to the output because as long as the error remains small, the human brain is unable to perceive any difference.

Acar [1] proposed a technique to reduce the error introduced by tolerant memoization by adapting the output to the variant input. By combining the concepts of both memoization and adaptivity he was able to increase the effectiveness of both. Acar implemented his technique on the quick sort and insertion sort algorithms.

In another work, Acar [2] proposed a framework to give a programmer control over the memoization parameters when memoization is applied to an algorithm. Acar's framework allows a programmer to apply memoization to algorithms selectively. In Acar's framework, he proposes a library for use with

programming languages. The framework also allows for statistical analysis of the memoization application on the underlying algorithm. The framework was not implemented, but the expectation was that it could be implemented in any purely-functional programming language.

Khalvati [13] proposed a software memoization technique that further refines the technique of Alvarez. While Alvarez provides a general technique to use in multimedia applications, Khalvati's technique is specifically geared to local image processing algorithms. Khalvati's technique is called window memoization. His technique includes hash tables with corresponding hash functions for the cache. Because his technique is specialized, it was able to perform better than the general techniques discussed previously.

Khalvati [15] proposed an innovation to the window memoization technique to combine it with opposition-based learning. Using his new technique, he was able to increase speed-up by an additional 5% over the speed-up achieved by using only window memoization. However, Khalvati's technique is confined to algorithms that produce binary outputs and cannot be used for algorithms that are not binary.

Acar [3] proposed a technique for imperative self-adjusting computation. His technique is an adaptation of the memoization technique that is designed to take advantage of the extra redundancy that exists in data inputs that change slowly over time.

Most algorithms are static and their control path stays constant through-out their implementation. Dynamic algorithms on the other hand can have their control path change as they operate. Harper [10] proposed an adaptation for memoization where the reuse table would store the control flow path rather than the actual output for a dynamic algorithm.

Pfeffer [20] expands on applying memoization to a dynamic algorithm. He proposes a memoization technique for sampling that recursively uses the look-up table when calculating an output. In his technique the look-up table can be used even when calculating the output for an input that will be placed

into the look-up table. Even though this technique serves to bias the look-up table, for a sampler, the biasing of the reuse table actually serves to reduce the error in the output.

2.3 Local Image Processing Algorithms

In the implementation of memoization, the technique is tested on various local image processing algorithms. In order to be able to better generalize results, the class of local image processing algorithms have been broken into three parts based on algorithm complexity: low-complexity, medium-complexity and high-complexity. One algorithm from each category was selected and the memoization technique was applied to it. The results were then analyzed, and are presented in Chapter 5. The three algorithms selected are as follows. The low-complexity algorithm studied is the convolution technique. The medium-complexity algorithm studied is the Trajkovic corner detection algorithm. The high-complexity algorithm studied is a new algorithm that we propose in this thesis, in subsection 2.3.3.

This section is broken into three subsections, one for each algorithm to be studied. The first subsection is devoted to the convolution algorithm, the second to the Trajkovic corner detection algorithm and the third to the new algorithm that we propose.

2.3.1 Convolution Algorithm

The convolution image processing algorithm is one of the simplest image processing algorithms applicable to an image. The process takes a pre-defined mask of size 3x3 and applies it to the image. A mask is a two dimensional matrix.

The convolution method involves taking the mask and sliding it along the input image. As the mask is slid along the image, the mask elements are multiplied to the corresponding pixels in the image. The multiplied values are then summed. The summed value is trimmed as required and the result is set as the amplitude for the pixel in the output image corresponding to the middle element of the mask. Trimming

the output value consists of setting the value to zero if the output value is below zero and setting it to the maximum (i.e. 255 for an 8-bit number) if the output value exceeds the maximum.

The formula for the convolution algorithm is as follows:

$$O(x, y) = \max(0, \min(255, \sum_{i=-1}^{i=1} \sum_{j=-1}^{j=1} M(i, j) * I(i+x, j+y))) \quad (2.7)$$

where x is the row of the selected pixel in the image, y is the column of the selected pixel in the image, O is the output image, I is the input image, M is the applied mask, i is the row of the mask element, j is the column of the mask element, Σ is the two dimensional sum operation, $\min$ is the minimum function and $\max$ is the maximum function. If the pixel requested by the sum operation is outside the bounds of the image, then the value of the closest pixel within the bounds of the image is used.

The mask that was selected is the combined horizontal and vertical gradient operators as shown in Tables 2.1, 2.2 and 2.3, respectively:

-1	0	1
-1	0	1
-1	0	1

Table 2.1: Horizontal Mask Operator

-1	-1	-1
0	0	0
1	1	1

Table 2.2: Vertical Mask Operator

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

Table 2.3: Combined Mask Operator

The horizontal gradient operator can be used to detect edges in the horizontal direction (from the left to the right). The vertical gradient operator can be used to detect edges in the vertical direction (from top to bottom). Combining both of the masks allows us to detect edges in both the horizontal and vertical directions.

As a sample, if the 5x5 input image from Table 2.4 were used and the mask were to be applied at position (1,2) in the image, the pixel at (1,2) in the output image would have an untrimmed value of 36. Since 36 is not greater than 255 nor is it less than zero, the value would not be trimmed and the output image would have a value of 36 at position (1,2). On the other hand, if the mask were to be applied at position (3,5) in the image, the pixel at (3,5) in the output image would have an untrimmed value of -37. Since -37 is less than zero, the value would be trimmed to zero and the output image would have a value of zero at position (3,5). Applying this mask to the whole image, the output from Table 2.5 results:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Table 2.4: Sample input image (5x5)

18	21	21	21	18
33	36	36	36	33
33	36	36	36	33
33	36	36	36	33
18	21	21	21	18

Table 2.5: Output from convolution algorithm for sample input image from Table 2.4

Applying the combined mask operator, from Table 2.3, to the actual input seen on the left of Fig 2.1, the output seen on the right in Fig 2.1 results.

Figure 2.1: Input image (left) and resulting output from convolution algorithm (right) [5]

2.3.2 Trajkovic Corner Detection Algorithm

Figure 2.2: Calculating change in image intensity in a given direction [19]

The Trajkovic corner detection algorithm is used to detect corners in an input image using the approach that "corners are points where the change of image intensity is high in all directions" [19], [25]. Thus, in order for the algorithm to classify a particular pixel as a corner, it requires that the change in pixel intensity is high in each direction. Figure 2.2 shows the general idea of how to calculate the change in image intensity in a given direction. The figure shows a circle with a line going through the center (nucleus, C). The two points where the circle is intersected by the line are P and P'. The difference between P and C and the difference between P' and C are combined to determine the change in image intensity in the particular direction. Using these three points, the following equation to calculate the corner response results:

$$R = \min((P - C)^2 + (P' - C)^2) \quad (2.8)$$

where R is the corner response, $\min$ is the minimum operation, and C is the image intensity at point C , the center of the circle. P is the image intensity at point P , which is the start of an arbitrary line segment bisecting the circle, and passing through the center of the circle, at point C . P' is the image intensity at point P' , which is the end of the arbitrary line segment bisecting the circle, and passing through the center of the circle. The corner response function requires checking for changes in image intensity in multiple directions. There are two implementations of the Trajkovic corner detection algorithm, when using a 3x3 window. The first implementation is the four neighbor approach, which checks for changes in image intensity in two directions: up-down and left-right. The second implementation is the 8 neighbor approach and it checks for changes in image intensity in four directions: upper-left to bottom right, up-down, upper-right to bottom-left, and left-right. The implementation of the Trajkovic corner detection selected for use in testing memoization is the 8-neighbor approach. Using a 3x3 window, one can define the pixels as shown in Table 2.6:

D	B	E
A	C	A'
E'	B'	D'

Table 2.6: Trajkovic corner detection algorithm 8-neighbour definition

where C is the center pixel, or nucleus of the circle. D and D' are the points corresponding to the upper-left to bottom right (diagonal one) direction. B and B' are the points corresponding to the up-down (vertical) direction. E and E' are the points corresponding to the upper-right to bottom-left (diagonal two) direction. A and A' are the points corresponding to the left-right (horizontal) direction.

The resulting Trajkovic corner detection algorithm, is summarized in the pseudo-code on Table 2.7 [19], [25]. Applying the Trajkovic corner detection algorithm to the actual input seen on the left in Figure 2.3, results in the output seen on the right in Figure 2.3.

1) Input - image I

2) Image I is scaled to lower resolution - image J

3) For each pixel in J, calculate the simple corner response in each of four directions.

$$3a) \quad R_a = (A - C)^2 + (A' - C)^2$$

$$3b) \quad R_b = (B - C)^2 + (B' - C)^2$$

$$3c) \quad R_d = (D - C)^2 + (D' - C)^2$$

$$3d) \quad R_e = (E - C)^2 + (E' - C)^2$$

$$3e) \quad R = \text{minimum}(R_a, R_b, R_d, R_e)$$

3f) If $R > \text{threshold}_1$, then it is flagged as a possible corner otherwise it is not a corner.

4) For each pixel flagged as a possible corner in J, calculate the simple corner response in each of four directions at the corresponding pixel locations in I.

$$4a) \quad R_a = (A - C)^2 + (A' - C)^2$$

$$4b) \quad R_b = (B - C)^2 + (B' - C)^2$$

$$4c) \quad R_d = (D - C)^2 + (D' - C)^2$$

$$4d) \quad R_e = (E - C)^2 + (E' - C)^2$$

$$4e) \quad R = \text{minimum}(R_a, R_b, R_d, R_e)$$

4f) If $R > \text{threshold}_2$, then the pixel is left as flagged as a possible corner otherwise it is not a corner.

5) For each remaining pixel flagged as a possible corner, apply the inter-pixel corner response function otherwise set the corresponding pixel in the corner map to zero.

$$5a) \quad C_{(\text{interpixel})} = \begin{matrix} R & \text{if for all } i=1,2,3,4 & \text{either } B_i \geq 0 \vee A_i + B_i \leq 0 \\ \min(r_i - B_i^2 / A_i) & \text{for all } i=1,2,3,4 & \text{where } B_i < 0 \wedge A_i + B_i > 0 \end{matrix}$$

where

$$r_1 = R_a, r_2 = R_b, r_3 = R_c, r_4 = R_d$$

$$B_1 = (B - A)(A - C) + (B' - A')(A' - C)$$

$$B_2 = (D - B)(B - C) + (D' - B')(B' - C)$$

$$B_3 = (E - D)(E - C) + (E' - D')(E' - C)$$

$$B_4 = (A' - E)(E - C) + (A - E')(E' - C)$$

$$A1 = r_2 - r_1 - 2 * B_1$$

$$A2 = r_3 - r_2 - 2 * B_2$$

$$A3 = r_4 - r_3 - 2 * B_3$$

$A4 = r_1 - r_4 - 2 * B_4$ <p>5b) If $C_{(interpixel)} > threshold_2$ then the pixel is a possible corner and the corresponding pixel in the corner map is assigned the $C_{(interpixel)}$ value otherwise it is set to zero.</p> <p>6) Non-maximal suppression is performed on the corner map to find local maxima. The remaining non-zero locations correspond to the corners detected in the input image.</p>

Table 2.7: Pseudo-code summary of the Trajkovic corner detection algorithm [19], [25]

Figure 2.3: Input image (left) and resulting output image when applying Trajkovic corner detection algorithm (right) [19]

2.3.3 Proposed Image Processing Algorithm (Tahir algorithm)

In this subsection, a new image processing algorithm of the complex variety is presented. The subsection is broken into three parts. First, there is a discussion on the motivation for the algorithm. Second the algorithm is proposed. Third, the resulting output from the proposed technique is shown.

2.3.3.1 Motivation

If a simple low pass filter is applied to a normal image with impulse noise, the impulse noise is removed. When applied to an image, the low pass filter serves to move a given pixel's intensity toward the middle range of the intensity table, i.e. for an image with 256 grey levels; the average pixel value is moved towards 127. However, in some situations, this change, in driving the average pixel intensity to the

middle, may cause a situation where the image is distorted. Taking the example of a medical ultrasound image where the majority of the image is quite dark and only a few areas of the image have pixels with higher intensities. In this situation, applying a low-pass filter to remove impulse noise may lead to undesirable image distortion. If however, an algorithm were to exist that would give more weight to the lower grey-levels rather than equal weighting to all grey-levels when filtering, then that algorithm would be better suited to removing impulse noise from a medical ultrasound image.

2.3.3.2 Proposed Algorithm

Taking into account the motivation, a low pass filter that averages the pixels in a given window, but gives more weight to lower intensity pixels is proposed. The algorithm takes the square of the average of the square root of the intensity of each pixel in the window as shown in Eq 2.9.

$$O(x, y) = \left(\sum_{i=-M_1}^{i=M_1} \sum_{j=-M_1}^{j=M_1} \frac{\sqrt{I(i+x, j+y)}}{M_1^2} \right)^2 \quad (2.9)$$

where x is the row of the selected pixel in the image, y is the column of the selected pixel in the image, O is the output image, I is the input image, i is the row of the mask element, j is the column of the mask element, Σ is the two dimensional sum operation, $\sqrt{x}$ is the square root function performed on an input x and the size of the window used is square with sides of length $2M_1+1$ pixels. If the pixel requested by the sum operation is outside the bounds of the image, then the value of the closest pixel within the bounds of the image is used.

2.3.3.3 Output From Proposed Algorithm

Using the actual input seen on the left in Figure 2.4, and applying the proposed low-pass filter algorithm results in the output seen on the right in Figure 2.4.

Figure 2.4: Input image (right) and resulting output image when applying Tahir algorithm (right)

The main characteristic of the algorithm output that is desired from a low-pass filter is that the high frequency components of the input image be subdued. The output image in Figure 2.4 shows the high frequency components of the input image are subdued. However, there is an additional benefit that the output image is also darker than the input image. This occurs in the output because the algorithm is able to give more weight to pixels in the input window that have a lower grey-level as compared to the weight given to pixels in the input window with a higher grey-level. Also, it is important to note that the change in weighting is not based on a particular threshold; rather it is a gradual change over the whole range of grey-levels.

This shows, that if the algorithm is applied to a medical image, the effect of impulse noise could be significantly subdued. Thus it is indeed possible to create a low-pass filtering algorithm designed to selectively target higher grey-level components in addition to the normally targeted high-frequency component of an input.

2.4 Set of Test Images

In this section, the set of images used for the testing of the implementation of the memoization technique is discussed and a select few images from the set are shown.

The set of images use in the implementation of the memoization technique comes from the Berkeley segmentation dataset and benchmark [5]. The set contains 300 images of the natural class. All the pictures are color and of identical size, 321 by 481 pixels. Some of the pictures have a landscape orientation while others have a portrait orientation. Figure 2.5 shows some selected images from the set.

2.5 Summary

A number of different areas were explored in this chapter. The first area explored was redundancy in image data. Three forms of redundancy in images can be exploited by image processing algorithms: coding redundancy, inter-pixel redundancy and psycho-visual redundancy. Coding redundancy results from pixels being represented by more bits than necessary [9]. Inter-pixel redundancy results from the correlation between neighboring pixels [13]. Psycho-visual redundancy results from the human visual system not treating all visual information received, equally [9].

The second area explored was previous work in the area of coding redundancy reduction and memoization. Previous work in the field has proceeded in two directions: hardware and software. In hardware, there have been a number of proposals for chip design, but none have proceeded beyond that level. In software, proposals for memoization and reduction in computational redundancy have been implemented and some success has been achieved.

The third area explored was local image processing algorithms. The three local image processing algorithms were explored. The convolution technique, the Trajkovic corner detection algorithm and the Tahir algorithm, an algorithm we proposed, were discussed. The three image processing algorithms were used to test the proposed implementation of window memoization.

The fourth area explored was image databases. The image database presented contains 300 images of the natural class and is what the proposed implementation of window memoization was tested with.

Figure 2.5: Sample images from image database [5]

Chapter 3 Memoization

In this chapter, memoization is approached from two different directions. It is approached from the theoretical stand-point and the application stand-point. In the first section, the theoretical portion of memoization is presented and discussed. In the second section, the software implementation of memoization is presented and discussed. The third section provides a summary. The goal for this chapter is two-fold. The first goal is to explain the theory of memoization and its ideal implementation. The second goal is to present how memoization can be applied in the real world. In Chapter 4, the proposed memoization technique is presented along with the methodology of implementation. The results are presented in Chapter 5.

3.1 Theory of Memoization

In this section, the theory of memoization is presented and discussed. This section is broken into 6 sub-sections. The first sub-section deals with the theoretical process for memoization. The second sub-section deals with the explanation of ideal memoization. The third sub-section deals with the concept of tolerant memoization. In the fourth sub-section, computational redundancy is discussed. In the fifth sub-section, memoization as targeted towards image processing algorithms is discussed. In the last sub-section, the theory of memoization is summarized. The goal for this section is to explain the theory of memoization and its ideal implementation.

Memoization is a process that is designed to take advantage of the computational redundancy that exists in the application of a given algorithm. The concept proposes to reuse results from previous calculations when a previous input is repeated. Thus the memoization process proposes to exchange a savings in processing time for a cost of increased memory usage. For time-sensitive applications, the **time** savings introduced may be critical to achieving timely results. For calculations, where considerable

calculation is required to determine an output from the input, a significant time saving can be achieved by retrieving previous results from memory instead of recalculating.

3.1.1 Memoization Process

The core part of the memoization process is built on a rather simple principle: if a calculation for an input is done once, it should never have to be repeated again on a future identical input. Certain intermediate steps are required in order to follow through on this principle. Some of these steps are as follows: the output from a particular input must be saved in a cache, an incoming input must be compared to the cache to see if that input has been processed previously, and if the incoming input has been seen then the output must be retrieved from the cache. A cache is a place to store data for frequent quick access to speed up an application [13] [14]. Fig 3.1 summarizes the steps in the memoization technique in a flow-chart.

3.1.1.1 Memoization Steps

In the following, a look will be taken at each of the steps from Fig 3.1 and a brief explanation will be provided:

- **Get Input** – All processes require inputs. This is the step where the memoization process acquires its input.
- **Check Cache** – The input must be compared to the cache. If a match can be found in the cache, a hit occurs. If a match cannot be found in the cache, a miss occurs. The matching input must be identical to the original input (ideal memoization) or similar to the original input (tolerant memoization). Ideal memoization is discussed in the next subsection and tolerant memoization is discussed in the subsection that follows it.
- **Retrieve Result from Cache** – In the event of a hit, the resulting output from the matching input in the cache can be reused and is therefore retrieved from the cache.

- **Perform Calculation** – In the event of a miss, the normal calculations from the memoized algorithm must be performed on the input.
- **Store Result in Cache** – After the calculations are complete, the input and its resulting output are stored in the cache for future reuse.
- **Return Output** – All processes provide an output and this step is where the memoization process returns its output.

Figure 3.1: Flow-chart for theoretical memoization

3.1.1.2 Hits and Misses

The first time a unique symbol is encountered the cache does not hold a matching symbol so the results have to be calculated, this type of miss always occurs. Any miss occurs when a matching symbol cannot be found in the cache and the results need to be calculated [14]. When the symbol is encountered in subsequent windows, a matching symbol can be found in the cache and the previous results can be reused, this is also called a hit. A hit occurs when a matching symbol is found in the cache and the

previous results can be reused [14]. Misses can occur if a symbol is encountered for the first time or if the matching symbol has been evacuated from the cache. Cache eviction is discussed in the next section under the heading, dealing with crashes.

Next the number of hits for a particular image and the number of misses for an image are analyzed. In memoization, a hit is desirable while a miss is not. The hit rate of an image refers to the number of symbols out of all symbols that produce a hit. A higher hit rate corresponds to more hits and fewer misses. In memoization, a higher hit rate is better because it corresponds to more skipped calculations. Thus when more calculations get skipped, the time savings increase.

3.1.2 Ideal Memoization

In ideal memoization, the cache is able to hold all the unique symbols produced by images permanently. Once a unique symbol is produced, the results are calculated and stored in the cache along with the corresponding symbol. The cache is never cleared. This allows previous results to be used for subsequent matching symbols.

In ideal memoization, input hit rates increase as more inputs are applied to the memoized algorithm. The first input is applied to an empty cache. Subsequent inputs are applied to a cache that is partially filled. Theoretically, the cache would eventually become populated with the entire universe of possible inputs and the hit rate for all subsequent inputs would be 100%; maximal speed-up would then be achieved.

Ideal memoization represents a theoretical concept because in actual practice it is impractical to create a cache that can store all possible symbols permanently. Practical applications of memoization use a cache, or reuse table of a defined, finite size. This is discussed in the section on memoization in software.

3.1.3 Tolerant Memoization

Tolerant memoization is an innovation of the ideal memoization concept that can be introduced to a particular implementation of memoization in order to achieve greater speed-up at the cost of possibly introducing some error into the output [13] [21] [4] [14]. Tolerant memoization is compared to ideal memoization both in theory and application. The theoretical portion of the analysis is in this section, and the application portion of the analysis is in the next chapter.

In ideal memoization, every unique input produces a unique symbol. However, as previously discussed, when applying memoization, it is impractical to create a cache that can store all inputs. As such, in a real-world application of memoization, a point would never be reached where the cache holds the entire universe of possible inputs. Thus, maximal speed-up by means of 100% hit rate cannot be achieved for all inputs in an implementation of memoization.

Therefore, it becomes more important to design a system to maximize the hit rate. There are a number of approaches that can be used to increase the hit rate. In tolerant memoization, the approach used is to relax the requirement that every unique window must produce a unique symbol. In tolerant memoization, it is possible that multiple inputs produce the same symbol. However, different inputs do not always produce the same output. Therefore, it is possible that error can be introduced to the output. Nevertheless, if the symbol producing process is designed so that only similar inputs produce the same symbol, it is possible that the amount of error introduced to the eventual output is negligible [13] [1].

When it comes to visual images the human eye is not capable of detecting small amounts of error [4]. Thus, if the amount of error introduced to an image is small enough, the eye would not be able to detect any difference.

Combining the two previous ideas, one arrives at the conclusion that, for an image, if the amount of error introduced to the output can be kept small, it is possible to implement a system where the hit rate is

increased, leading to greater time savings while leaving the output image virtually unchanged to the human eye.

Thus in tolerant memoization, at a cost of introducing a small amount of error that is undetectable to the human eye, additional time savings are achieved by relaxing the requirement from ideal memoization that unique inputs must produce unique symbols to allow similar inputs to produce the same symbols.

3.1.4 Computational Redundancy

When memoization is applied to images, there are two types of redundancy that arise; coding redundancy and inter-pixel redundancy. Coding redundancy is the portion of computational redundancy that exists in the input image itself because more bits are used per pixel than strictly required. Inter-pixel redundancy is the portion of computational redundancy that exists in the repetition of calculating output for the same input multiple times. Inter-pixel redundancy is dependent on the correlation between neighboring pixels in an image [13].

Khalvati [13] showed in his work that both coding and inter-pixel redundancy of an image has a positive correlation or 'positive relationship' with reusability of an image. Reusability of an image corresponds to the probability of finding multiple instances of a window through an image. Khalvati [13] showed in his work that the computational redundancy of an image has a positive relationship with the hit rate of an image. Computational redundancy of an image is a combination of both the coding redundancy and inter-pixel redundancy of an image. The hit rate of an image is the ratio of input windows for which memoization can be applied so that the normal algorithm calculations required to process the window can be avoided to the total number of input windows. Khalvati [13] showed in his work that there is a positive relationship between the reusability of an image and the computational redundancy of the image. Thus, for local image processing algorithms, the computational redundancy of an image can lead to the existence of similar windows across an image. These similar windows can be exploited by the memoization technique to improve performance.

The percentage of inputs producing duplicate symbols that exist in the cache is the computational redundancy of that input. Thus computational redundancy is a way to express the percentage of calculations that are redundant. For each unique symbol produced, the underlying algorithm would only need to be applied once, at which point it would be added to the cache. On every subsequent occurrence of the symbol, recalculating the result would be redundant, and the previous result residing in the cache could be reused. Computational redundancy can be expressed in the following equation:

$$CR = \frac{(n-s)}{n} = 1 - \frac{s}{n} \quad (3.1)$$

where CR is the computational redundancy (in ratio form), s is the total number of unique symbols in an input and n is the total number of inputs and total number of symbols in an input. The fewer the number of symbols produced by an input, the more computationally redundant the input. Conversely, if every symbol produced is unique, then the computational redundancy in an input is zero and every calculation is required.

3.1.5 Window Memoization

In window memoization, the inputs to the memoization process are windows of pixels. Each window is made up of $M \times M$ pixels. In order to compare the two windows, M^2 compare operations have to take place. However, if a single compare operation could compare two windows to each other, that would be more efficient. To compare two windows to each other with a single compare, a representative symbol for each input would need to be derived. Then if the symbols produced from each window were the same, the windows would be considered a match. If the symbols were not the same, then it would be determined that the corresponding windows are different. Rather than recreating the symbol for the existing windows in the cache, it would be more time-saving to store the symbol for the original input instead of the original input along with the corresponding result.

In order for the symbol generating process to be useful in ideal memoization, each unique window must always produce the same unique symbol. Every occurrence of a particular window should always produce the same symbol. Also, two differing windows should not produce the same symbol. This way there is a one to one relationship between window and symbol. Every unique symbol comes from a unique window and every unique window creates a unique symbol.

The percentage of windows producing duplicate symbols that exist in the cache is the computational redundancy of that image. The fewer symbols produced by an input image, the more computationally redundant the image. Conversely, if every symbol produced is unique, then the computational redundancy in an image is zero and every calculation is required. This, however, is very rare for typical images. In typical images, the number of unique symbols is less than the number of total symbols [13].

3.1.6 Summary

Memoization is a technique used to speed up the processing time of an algorithm by exchanging processing time with increased use of memory. A cache to store previous results is used. When a new input is received, the corresponding output is calculated and stored to the cache. When that input is repeated, corresponding calculations can be skipped because the result stored in the cache can be retrieved instead. When calculations are avoided, it is possible to decrease the amount of time required to run an algorithm. The time savings achieved are known as the speed-up. Theoretically, maximum speed-up can be achieved when an infinite cache is used. However, a cache of infinite size is not practical in a real-world application.

In a real-world application a finite cache of fixed size, called a reuse table, is used. For every input, the reuse table must be checked to see if a matching input exists. In order to speed up the checking process, symbols are used. Every unique input is converted to a corresponding unique symbol. Inputs and symbols have a one-to-one relationship.

In order to achieve further speed-up, the requirement to have a one-to-one relationship between inputs and symbols is relaxed so that multiple similar inputs can produce the same symbol. This relaxation of the input to symbol criteria is called tolerant memoization. Tolerant memoization allows for increased speed-up at a cost of introducing error to the output. Provided that the error can be contained to a small amount, this is not a problem because the human eye is not capable of detecting small amounts of error [4].

When tolerant memoization is applied to local image processing algorithms, rather than treating the input window as a combination of multiple input pixels, the window can be converted to a single symbol and treated as a single input. Treating a whole window as a symbol instead of individual pixels allows for further speed gains to be achieved.

3.2 Software Implementation of Memoization

In this chapter, memoization is approached from two different directions: the theoretical and the application stand-points. The previous section dealt with the theoretical portion of memoization. In this section, the software implementation of memoization is presented and discussed. This section is broken into 5 sections. The first sub-section outlines the general steps required to implement memoization in software. The second sub-section deals with how to check the cache for matches. The third sub-section deals with the concept of an eviction policy. In the fourth sub-section, tolerant memoization is presented. In the fifth sub-section, the general implementation of memoization in software is summarized. The goal for this section is to outline the general implementation of memoization and to explain some of the nuances in the general implementation.

3.2.1 Software Memoization Steps

In this section, the general steps required to implement memoization in software are presented. Fig 3.2 shows the flowchart of the memoization process in software. Every step of the memoization process in

the flow chart is discussed below. There are some key differences between ideal memoization discussed previously and the implementation of memoization, which are discussed as they arise.

The steps in the software implementation of the memoization technique from Fig 3.2 are explained next:

Check if Cache Exists: this step is necessary to ensure that a cache exists for the memoization to act on.

Create Cache: this step initializes the memoization algorithm that is applied to the original algorithm. It is only required in the first usage of memoization on an algorithm. In application, this step would not be required for any usage of an algorithm after the initial usage as every subsequent usage would be able to use the table of stored results of previous usages of the algorithm. However, as this thesis serves to establish a lower limit in terms of speed-up, a new cache is created every time the algorithm is run in order to capture the time required for the first usage of memoization.

Get Input: just as in the Get Input step from the previous section, all processes require inputs and this is the step where a software implementation of memoization process acquires its input.

Check Cache: this step is where the memoization algorithm checks to see if this exact input (ideal memoization), or a reasonably similar input (tolerant memoization), has previously been processed. This step is the heart of the memoization process and most of the implementation and thesis deals with this portion of memoization. This step is discussed in its own subsection, which follows next.

Hit: the calculation is skipped and the output is retrieved from the hash table. This step is where all of the time savings from memoization are realized.

Miss: in the event of a miss, the calculation is performed on the input window.

Check for space in cache: if a miss occurs, this means that a new input has been found and its corresponding output is calculated. The new input and corresponding output may be stored in the cache, as long as the cache has adequate space to store it.

Store result in cache: If there is space in the cache to store the input, then the input and corresponding output are stored in the cache.

Eviction Policy: If the cache has no space for the input, then the eviction policy would come into play. The eviction policy used is discussed later.

Figure 3.2: Flow-chart for general software implementation of memoization

Return Output: this step is common to all algorithms whether or not memoization is applied. If tolerant memoization is used, then it is possible that the output from the memoized algorithm would be different from the non-memoized algorithm. This difference is the error introduced by the algorithm and is discussed later.

3.2.2 Check Cache

This step is broken into multiple sub-steps, as shown in Table 3.1. The key components from the steps in Table 3.1 are discussed in detail next:

1) Generate Symbol
2) Make Hash Key
3) Check Hash Table at Hash Key Index
4) Compare symbol at hash key index to symbol generated in step one
5a) If the symbols are different or if the hash index is unoccupied, it is determined that the input has not been seen before (miss)
5b) If the symbols are the same, it is determined that the input has been seen before (hit)

Table 3.1: Sub-steps required to check reuse table for previous results matching current input

3.2.2.1 Generating Symbols

A symbol is a representation for an input. In perfect memoization, each unique input would result in a unique symbol. In tolerant memoization on the other hand, similar inputs result in the same symbol. Similarity of inputs is judged by comparing the N most significant bits of each pixel in the input window. As discussed earlier, tolerant memoization allows us to trade increased speed-up and less processing time at the cost of increased error. As such, in practical application, reducing N would lead to an increase in hit-rate. However, increasing N would also lead to increased error.

In order to generate a symbol for an input window, the pixels in the window are scanned by column and by row (top-left to bottom-right). For each pixel, the N most significant bits are used to create the symbol. The symbol is created by sequencing the bits from each pixel in the scanning order. Thus, the N most significant bits from the top-left pixel in the window would form the N most significant bits in the symbol and the N most significant bits from the bottom-right pixel in the window form the N least significant bits in the symbol. Table 3.2 shows the steps required to create a symbol from an input window:

1. Input window, win
2. For each input window, do:
a. Initialize symbol:
$sym = 0$
b. For each pixel in the window, starting from the top-left moving to bottom-right, do:
$sym = (sym \ll N) \vee (pix \gg 8 - N)$
where:
sym corresponds to the output symbol,
N corresponds to an integer number between 1 and the bit-size of an input pixel
$a \ll N$ corresponds to a left logic shift of N bits for input a
$a \gg N$ corresponds to a right logic shift of N bits for input a
pix corresponds to the selected pixel
$a \vee b$ corresponds to the logical OR operation between inputs a and b

Table 3.2: Generating a symbol

In order to implement symbol generation in software, it can be necessary to accommodate a symbol that is larger than a standard long character. If this is the case, the symbol is stored as an array of variables of type long.

3.2.2.2 Hash Table

In the software implementation of memoization, the cache is known as a hash table. A hash table is used to store the results from previous inputs for reuse with future inputs. The inputs must be stored in the table using a particular strategy or mapping scheme. There are three commonly used mapping schemes

for hash tables: direct-mapped, fully associative, and set-associative [23]. Each of these three strategies is briefly discussed.

In the direct-mapped scheme, a particular input is always mapped to a particular output. Because the range of inputs is larger than the range of outputs, there are multiple inputs that would result in the same output, so this is a many-to-one mapping scheme [13].

In the fully associative scheme, any input can be mapped to any output resulting in an any-to-any mapping scheme. In order to check if a particular input exists in the hash table, the entire table must be checked. This is a time-consuming process and does not fit well with the memoization methodology.

The set-associative scheme is a hybrid of the previous two schemes. The first step is that the input gets mapped to a range of possible entries, similar to the direct-mapped scheme. Then the input gets assigned to one of the entries in the range. If an entry is free it would get assigned there. If all entries are used, then an eviction policy would be used. This is again a time-consuming process and does not fit well with the memoization methodology.

When dealing with hash tables of finite size, hash tables that are not capable of holding the entire universe of possible inputs, a situation can result where different symbols produce the same hash key when the hash function is applied. When two different symbols both result in the same hash key, this is called a crash. Crashes are undesirable as they force the computer to recalculate previous results. The opposite of a crash is a hit. A hit occurs when an input symbol is found in the hash table. A hit is desirable because when a hit occurs, the processor does not have to calculate the results for the input and can reuse the previous result. Therefore it is desirable to try and maximize the number of hits that occur. Hash functions are discussed in the hash functions subsection.

3.2.2.2.1 Dealing with Crashes

There are generally two solutions to solving the problem of crashes in software implementation of hash tables: open addressing and chaining [13]. However, both of these solutions introduce extra processing

time that, if implemented for memoization would reduce the time savings. When it comes to memoization, there is a third solution that is possible. Because any given input can still be applied to the original algorithm, it is not mandatory to store previous results. As such, it is possible to discard old unused results if desired. This implies that for the memoization technique it is possible to use an eviction policy. An eviction policy is the strategy used to decide whether to discard the results stored at a particular hash entry.

The idea behind the eviction policy is that when a crash occurs, one of two things happen: either the symbol and its corresponding data currently residing in the hash table are discarded and the new symbol and its associated data are stored at the same hash entry, or the existing data at the hash entry is kept and the new symbol and its associated data are discarded. The method used to decide which of these two actions to take after a given crash is called the eviction policy.

3.2.2.2 Hash Table Size

In the ideal situation, the size of the reuse table would be such that it could accommodate the entire universe of possible inputs. However, since the amount of memory required to hold a reuse table of that size would be very large, an implementation of memoization with a table holding the entire universe of inputs is not practical. As such, a software implementation of memoization would have to use a much smaller reuse table. This implementation of the memoization algorithm tested the algorithm with reuse tables of the following sizes: 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k and 256k.

3.2.2.3 Hash Functions

Because the memoization technique used in software makes use of a hash table, each symbol needs to be operated on by a hash function in order to arrive at a hash key. The hash function is used to map incoming symbols to a particular hash entry on the hash table.

As previously discussed, when applied to memoization, hash functions should also try to result in higher hit rates for the input so that the hash table is reused as often as possible.

As the hash function is being used for memoization purposes, one of the goals for the hash function is to be fast. It would be counterproductive to create a memoization technique to increase the speed of algorithms and then use a hash function that is slow. Therefore hash functions with long processing times are undesirable.

Thus there are two main features that one must examine when selecting a hash function to use for the memoization technique: ability to maximize the number of hits and ability to minimize the time required to create a hash key.

3.2.2.3.1 Generating a Key (Hash Function)

There are two hash functions that are commonly used: division method and multiplication method [6]. Both of these key generation methods are examined below.

3.2.2.3.2 Division Method

The idea behind the division method is to map a symbol into one of the entries on a hash table by using the remainder of the symbol when divided by the size of the reuse table. Thus, the division method for creating a key is:

$$hk = \text{mod}(sym, RT_{size}) \quad (4.1)$$

where hk is the resultant hash key, mod is the modulo function, sym is the input symbol and RT_{size} is the size of the reuse table. The modulo function takes two inputs, the numerator and the denominator and performs the division operation. The fractional output of the quotient is extracted and re-multiplied by the denominator to result in a whole number that corresponds to the remainder of the original division operation.

To improve the division method it is possible to select a reuse table size that results in a better spread of symbols, thus minimizing the number of possible crashes. Because the remainder operator, which is closely related to the division operator, is used in the division method, to improve the results of the division method, RT_{size} must not be a power of two. Choosing a prime number which is not too close to a power of two would be a good choice for RT_{size} [13].

3.2.2.3.3 Multiplication method

The idea behind the multiplication method is to map a symbol into one of the entries on a hash table by multiplying the symbol by a fraction between zero and one, extracting the fractional component, then multiplying it by the size of the reuse table and finally using the floor operation to get rid of any remaining fractional numbers. Thus the multiplication method does not use any processor intensive operations, like division or modulo. The multiplication method for creating a key is:

$$hk = \text{floor}(RT_{size} \cdot \text{sym} \cdot A - \text{sym} \cdot A) \quad (4.2)$$

where hk is the resultant hash key, floor is the floor function, sym is the input symbol, RT_{size} is the size of the reuse table, A is a fractional number between zero and one and $\cdot$ is the multiplication function. The floor function rounds the input down to the largest whole number that is less than or equal to the input.

Because the multiplication operation is used, there is no constraint on the value of RT_{size} . However, in order to create additional time savings, a power of two can be used in order to convert the multiplication operation into a bit shift operation [13].

3.2.3 Eviction Policy

In a real-world application of memoization, the cache or reuse table is of a fixed finite size. As the table is fixed in size and not infinite, this limits the maximum hit rate that can be achieved. Higher hit rates are

more desirable than lower hit rates as higher hit rates lead to more skipped calculations and that translates to more time savings and high speed-up. One way to increase hit rates is to relax the symbol generating criteria, this is called Tolerant Memoization and it was discussed in the previous subsection. Another way to increase the hit rate is to design the cache so that it holds the symbols for the windows that have the highest probability of occurring in a given image. In order to optimally design such a cache, it would be necessary to possess a priori knowledge of incoming images. As this is not practical in most real-world applications, an alternate strategy to decide what symbols should be stored in the cache must be used.

The strategy used to decide what symbols to keep and what symbols to discard from a cache is known as the eviction policy. When a new symbol is generated and there is no space in the cache to keep it, a decision must be made as to whether to discard the newly generated symbol and corresponding result, or to discard a symbol and result that already exists in the cache. In order to decide which symbol to discard, there are a few strategies beyond the ideal one requiring a priori knowledge of the input. There are three eviction policies commonly used in literature [9]: random, a priori and first-in-first-out (FIFO).

The random eviction policy, as the name implies, randomly determines which input to discard. There is no weight given to how often a particular input is reused. A priori policies take advantage of information about the input that is known before the input is received. An eviction policy is then designed to take advantage of the information and to maximize the hit rate. This of course, requires an intimate knowledge of the input prior to receiving it, which is not usually possible. The third policy is FIFO. In this policy, the older input is always discarded and the new input is always kept. Again, this policy does not take into account how often a previous input has been reused. The strategy used to decide which symbol to discard in the proposed implementation is explained in the next chapter.

3.2.4 Tolerant Memoization in Software

The idea for tolerant memoization was originally proposed by Hughes [11] as an innovation to the ideal memoization proposed by Michie [18]. Introducing tolerant memoization to a software implementation of memoization allows one to achieve greater speed-up at a cost of possibly introducing some error into the output. Tolerant memoization is compared to perfect memoization below both in theory and application.

3.2.4.1 Perfect Memoization vs. Tolerant Memoization

The 'check reuse table' step from the software memoization section, can allow for the algorithm to reuse a previous input which is similar but not identical to the actual input. This is called tolerant memoization. If a previous input were only to be reused on condition that it is identical to an actual input, this would be known as perfect memoization. Tolerant memoization allows for the memoization algorithm to increase speed-up because the hit ratio (ratio of inputs as compared to all inputs that cause a 'hit' to register) increases. The cost for this increase in speed is error. The more tolerant the memoization process the more potential for error that is introduced. This is shown later.

3.2.4.2 Tolerance

In the ideal situation, with a reuse table spanning the entire universe of possible inputs, input tolerance is not required. However, when dealing with a reuse table of limited size, significant speed-up can be achieved by effectively mapping multiple inputs to the same reuse entry. The number of inputs mapped to a particular entry is another way of looking at the tolerance of the input. In the following implementation of memoization, perfect and tolerant memoization are tested. In the tolerant versions, one through 8 bits of tolerance were tested on 8 bit inputs.

The goal for memoization is to decrease the time required to run an algorithm by increasing the speed of the algorithm. In order to determine whether the memoization process is effective at this, two factors must be examined. The first factor is actual speed-up. This factor is closely related to the amount of time

saved by running the memoization algorithm and is discussed shortly. The second factor that must be analyzed is the amount of error introduced by the memoization algorithm. While each of these factors is important individually, it is also necessary to look at both of these factors together. If while memoizing an algorithm, significant speed-up is achieved but the output has too much error then the memoization is not practical. Conversely, if there is no error introduced, but memoizing an algorithm causes the actual processing time to increase, then the memoization is not practical and one would be better served by not implementing memoization on the algorithm.

3.2.5 Summary

Memoization is a technique used to speed up the processing time of an algorithm by exchanging processing time with increased use of memory. In a real-world application, a finite cache of fixed size called a reuse table is used. For every input, the reuse table must be checked to see if a matching input exists. In order to speed up the checking process, symbols are used. Every unique input is converted to a corresponding unique symbol. Inputs and symbols have a one-to-one relationship.

A hash table is used to store previous inputs and corresponding outputs. Three mapping schemes for hash tables were discussed. To decide where to store an input in the hash table, a hash function is used; two hash functions were discussed. When a new input is received, the corresponding output is calculated and stored in the hash table if there is space. If there is no space in the reuse table, the eviction policy is used to decide which input to discard.

Tolerant memoization in software allows one to map multiple similar inputs to a single hash entry in the hash table. This allows one to further increase speed-up at a cost of possibly introducing some error to the output. However, the amount of error introduced to the output must be controlled and cannot be allowed to become too large.

3.3 Summary

Memoization is a technique used to speed up the processing time of an algorithm by exchanging processing time with increased use of memory. Memory, in the form of a cache is used to store previous results. When a new input is received, the corresponding output is calculated and stored to the cache. When an input is repeated, the result is retrieved from the cache and the corresponding calculations can be skipped. Skipping calculations can lead to a decrease in the amount of time required to run an algorithm. The time savings achieved are known as the speed-up. Theoretically, maximum speed-up can be achieved when an infinite cache is used. However, a cache of infinite size is not practical in a real-world application.

In a real-world application a finite cache of fixed size, called a reuse table, is used. The reuse table is a hash table storing the previous inputs in a particular form and their corresponding results. For every input, the reuse table must be checked to see if a matching input exists. In order to speed up the checking process, symbols are used. Every unique input is converted to a corresponding unique symbol. Inputs and symbols have a one-to-one relationship.

Three mapping schemes for hash tables were discussed: direct-mapped, fully associative, and set-associative [23]. To decide where to store an input in the hash table, a hash function is used; two hash functions were discussed: the multiplication method and the division method. When a new input is received, the corresponding output is calculated and stored in the hash table if there is space. If there is no space in the reuse table, the eviction policy is used to decide which input to discard.

In order to achieve further speed-up, the requirement to have a one-to-one relationship between inputs and symbols is relaxed so that multiple similar inputs can produce the same symbol. This relaxation of the input to symbol criteria is called tolerant memoization. Tolerant memoization allows for increased speed-up at a cost of introducing error to the output. Provided that the error can be contained to a small amount, this is not a problem because the human eye is not capable of detecting small amounts of error

[4]. Tolerant memoization in software allows one to map multiple similar inputs to a single hash entry in the hash table.

When tolerant memoization is applied to local image processing algorithms, rather than treating the input window as a combination of multiple input pixels, the window can be converted to a single symbol and treated as a single input. Treating a whole window as a symbol instead of individual pixels allows for further speed gains to be achieved.

Chapter 4 Proposed Implementation of Memoization and Challenges Encountered

In this chapter the proposed implementation of window memoization is presented. This chapter is broken into 6 sections. In section one, the general steps for the proposed implementation are presented. This is followed by a brief explanation of each step. Section two expands on the specific innovations in the technique. In section three, the methodology of the experiment is provided. The next couple of sections discuss a framework to calculate the speed-up and the error introduced by any implementation of memoization. In section four, there is a discussion of the time requirements for running the memoization technique on an algorithm, an explanation of speed-up and its calculation. In section 5, there is a discussion on error and how to calculate it. Specifically, two different forms of error calculation are discussed: mean pixel error and signal-to-noise ratio (SNR). The calculation of speed-up from section four, and the measurement of error constitute the two aspects that the evaluation of memoization is based on. In section 6, a summary of the chapter will be provided. The goal for this chapter is to explain the proposed implementation of memoization, how it was implemented and how to evaluate the effectiveness of any implementation of the technique. The next chapter will present the results from this technique.

4.1 Memoization in Software:

In this section, the general steps in the proposed implementation of window memoization in software are presented. Fig 4.1 shows the flowchart of the steps of the proposed window memoization technique in software. Every step of the memoization process in the flow chart is discussed below. There are some key differences from the general software implementation of memoization as presented in Chapter 3 and the steps outlined in this section. These differences are explored in the next section.

Figure 4.1: Proposed Implementation of Window Memoization in Software

Read Image: this step provides the input for the underlying image processing algorithm, and is a required step for all image processing algorithms. When window memoization is applied, the necessity

for an input is not removed, so an input is still required. Thus, for the memoization implementation, this step is the input to the memoization algorithm.

Create Hash Table: this step initializes the memoization algorithm that is applied to the original algorithm. As was the case in Section 3.2, it is only required in the first iteration of memoization on an algorithm.

Create Window: this step is also common to local image processing algorithms whether or not memoization is applied. As all three image processing algorithms do not operate on the image as a whole, but rather on windows of the original image, this step is required to make each window that the image processing algorithm is applied to.

Check Reuse Table: this step is where the memoization algorithm checks to see if this window, or a reasonably similar window, has previously been processed. The proposed process uses the fundamentals of tolerant memoization presented previously. As was the case in Section 3.2, this step is the heart of the memoization process. As per the Check Reuse Table step in Section 3.2, it too is broken into multiple sub-steps.

Miss: in the event of a miss, the calculation is performed on the input window. If the hash index is unoccupied, the input symbol and output are saved and the hit count is set to one. If the hash index is occupied, then the eviction policy would come into play. The eviction policy used is discussed later.

Hit: the calculation is skipped and the output is retrieved from the hash table. The hit count for the index is also incremented. This step is where all of the time savings from memoization are realized.

End of Input: this step is common to local image processing algorithms whether or not memoization is applied. Processing occurs on the whole window, once the algorithm has calculated the output for the last pixel in the image, the algorithm finishes and the output is returned.

Increment Input Counter: this step is common to algorithms whether or not memoization is applied. If the last pixel in the image has not been reached, then the input has not been fully processed. The algorithm is instructed to move on to the next pixel in the input.

Output: this step is also common to local image processing algorithms whether or not memoization is applied. With the tolerant memoization used in the proposed implementation, it is possible that the output from the memoized algorithm would be different from the non-memoized algorithm. This difference is the error introduced by the algorithm and is discussed later.

4.2 Specific Innovations in the Proposed Implementation

In the proposed implementation of window memoization in software, there are a number of innovative ideas that are incorporated. Each of these ideas will be discussed in the following subsections.

4.2.1 Generating Symbols

In Table 3.2, the general method to generate a symbol was discussed. There is a more efficient method to generate symbols than the method used in Chapter 3. In the proposed implementation the second method to generate symbols is used because it is more efficient and thus it leads to decreased calculations and a lower time requirement for window memoization.

To improve the symbol generation technique, one can reduce the amount of time required to build a symbol by using the overlapping windows technique. Since all three of the image processing algorithms use a window that slides across the image from top-left to bottom right (by row and then by column), it is possible to design a symbol generation function that takes advantage of this. The difference between one window and the next, if they are both located on the same row, is one column. With a window size of 3x3 this means 6 of the 9 pixels in the two windows are identical. It is thus possible to reuse the previous symbol in calculating the next symbol in a row. This can be accomplished by following the steps in Table 4.1.

1. Input window, win
2. For each input window, do:
 - a. If window is the first window in the row, then:
 - perform steps from Table 3.2
 - b. Else:
 - take previous symbol
 - For each pixel in the rightmost column in the window, do:

$$sym = (sym \ll N) \vee (pix \gg 8 - N)$$

where:

sym corresponds to the output symbol,

N corresponds to an integer number between 1 and the bit-size of an input pixel

$a \ll N$ corresponds to a left logic shift of N bits for input a

$a \gg N$ corresponds to a right logic shift of N bits for input a

pix corresponds to the selected pixel

$a \vee b$ corresponds to the logical OR operation between inputs a and b

Table 4.1: Generating symbol using the overlapping windows

Thus the overlapping window technique for symbol generation removes the bits corresponding to the left-most column from the previous input window. Then the previous symbol is left shifted so as to move the remaining columns over one column to the left. Lastly, the N most significant bits from each pixel in the right-most column of the window are added in. This technique allows for the saving of 2/3 of the processing time to create symbols for windows that are not made from the leftmost column of pixels in the input image.

4.2.2 Hash Table

In Chapter 4, three different mapping schemes for hash tables were presented. Both the fully associative scheme and the set-associative scheme do not fit well with the memoization methodology so they were not selected. This leaves us with just the direct-mapped scheme to use in the software implementation of memoization. When implementing the direct-mapped scheme with hash tables of finite size, the hash tables are not capable of holding the entire universe of possible inputs, and crashes can occur.

4.2.2.1 Hash Table Size

In the ideal situation, the size of the reuse table would be such that it could accommodate the entire universe of possible inputs. However, since the amount of memory required to hold a reuse table of that size is very large, an implementation of memoization with a table holding the entire universe of inputs is not practical. As such, a software implementation of memoization would have to use a much smaller reuse table. This implementation of the memoization algorithm tested the algorithm with reuse tables of the following sizes: 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k and 256k.

4.2.3 Eviction Policy

As crashes are undesirable it is beneficial to minimize their occurrences and as hits are desirable, it is beneficial to maximize their occurrences. A properly designed eviction policy can help achieve this goal. At the same time, the eviction policy cannot be too time consuming as the whole memoization process is designed to speed-up the underlying algorithm. Both of these points were taken into account when designing the selected eviction policy.

The selected eviction policy is an attempt to make a hybrid policy between the first-in-first-out (FIFO) and the a priori eviction policies in literature.

The eviction policy used is as follows: if the hit count for the index is one, then the existing data would be overridden. If the hit count for the index is greater than one, then the existing data is kept and the new data is discarded. A hit count of one indicates that the index has not been previously reused and thus qualifies to be considered as an 'old unused result' while a hit count of greater than one indicates that the index has been previously reused and does not qualify as an 'old unused result' and cannot be discarded.

4.2.4 Hash Functions

In order to decide on the hash function to use, it was decided that both methods presented in Chapter 3 should be tested and the results should be compared. Based on those results, the hash function to be used in the remainder of the thesis should be selected. As such, both the division method and the multiplication method were tested. For the multiplication method, the golden ratio was selected for the A value.

4.2.4.1 Comparing Hash Functions

As stated earlier, when comparing the hash functions, multiplication and division methods, it is necessary to compare them on two levels: hit rate and key calculation time. Using these criteria, both methods were implemented and compared. The methods were then compared based on hit rate and on total speed-up. The processing time to create the keys was not directly measured, but indirectly compared through the total time required for each function.

To compare the two hash functions, they were both implemented in the memoization framework and were run with a bit shift of three and with table sizes ranging in size from 1000 to 256000 slots (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000 and 256000) as per the Hash Table Size in the Hash Table section. The memoization was run on the Tahir algorithm. The results are shown in the Figure 4.2.

Figure 4.2: a) top-left, graph of hit rate vs. table size for multiply and divide key generation functions. b) top-right, graph of speed-up vs. table size for multiply and divide key generation functions. c) bottom-left, graph of difference in hit rate between multiply and divide key generation functions vs. table size. d) bottom-right, graph of difference in speed-up between multiply and divide key generation functions vs. table size.

As seen in Fig 4.2A and Fig 4.2C, the multiplication method for creating key generally achieves similar or faster results than the division method. This shows that when memoization is implemented, the process with the multiplication key generally required less time than the process with the division key. This is seen through the resulting higher speed-ups with the multiplication method as compared to the division method.

As seen in Fig 4.2B and Fig 4.2D, the multiplication method also results in higher hit rates than the division key method. The higher hit rate for the multiplication method can be attributed to having a lower

crash rate than the division method because the multiplication key method is better able to spread the symbols through the reuse table rather than clumping the reuse slots as does the division key method.

Since the multiplication method generally results in both a higher speed-up and a higher hit rate than the division method, it is better suited to memoization. Thus it was determined that the multiplication method was better suited to the memoization technique and only it was used in the remainder of the testing of memoization in software.

4.2.5 Tolerant Memoization

In order to test the amount of tolerance that can be used by tolerant memoization while maintaining the error at a reasonable level, the amount of information taken from each pixel in the window was varied. The information used varied from 1 to 8 bits of data from the 8-bit pixels. Where the number of bits used corresponded to the number of most significant bits selected from each pixel.

4.3 Methodology

The next step after presenting the proposed technique is to implement it. This section deals with the implementation of the proposed memoization technique. The specific methodology of implementation, testing and evaluation is discussed. In order to test the validity and effectiveness of a proposed technique, it must be implemented and tested against a particular benchmark. There are a number of possible problems that can arise in implementation, and they will be briefly discussed. The software implementation of the proposed technique must take place on a particular platform; this platform must be selected. The implementation must be done using a particular method; the method of implementation must therefore also be selected. The implementation will need to be tested and evaluated; the method of evaluation must be determined. Each of these issues are discussed, with the selected solution presented with an explanation and justification.

4.3.1 Platform

Selecting a platform for a particular purpose can lead to biasing of the results. As such, it would be desirable to select a platform that is as average and general as possible. It would not be prudent to pick a specialized platform. The platform was selected because it is an average PC system and is not designed for a specific purpose, rather it is for general computing use. The platform selected is a Windows based PC system. The operating system was Microsoft Windows XP Professional with Service Pack 3. The processor used in the PC had the following specifications: Intel ® Dual Core T2500, CPU: 2.00GHz, 2.00GHz, 2MB cache size. The computer had 2424 MB of DDR2 SDRAM at 266MHz. The hard-drive in the PC was a Fujitsu MHV2080AH which has the following specifications: 2.5", 80GB, 5400RPM, 8MB Cache.

The implementation was written completely from scratch in the C++ programming language. For linear flow of the program, the implementation is written as a single-thread process and was not multi-threaded. The program was developed using the CodeBlocks IDE and was compiled using the GNU GCC compiler without applying any optimization techniques.

4.3.2 Implementation Method

The method of implementation of the proposed memoization technique was to initially create a framework program on which the various image processing algorithms can be run. In order to create a fair comparison, the framework would have to treat both memoized and non-memoized algorithms equally. The framework created was designed specifically to support the running of the image processing algorithms. The framework was also designed to monitor this running and to calculate and store the resulting statistics. The framework would need to be able to track the timing and accuracy for each image processing algorithm when it was run both with and without memoization. In addition, the framework would need to track each run of the memoized algorithm as the parameters were altered. Therefore, the framework was created and designed to handle all these factors.

Each image processing algorithm was first run on the framework without implementing memoization in order to determine a baseline for the amount of time required to run the algorithm and also to determine the actual desired output. Next, each image processing algorithm was run on the framework with memoization implemented based on the proposed technique. The memoized algorithms were run on the framework multiple times as the memoization parameters were adjusted. For each combination of parameters, the algorithm was run 5 times. The average time for the 5 runs was calculated and stored. Each run consisted of running the image processing algorithm on the entire database of 300 natural images. The parameters varied were input shift and reuse table size. Input shift was varied from 1 to 8 most significant bits. Reuse table size was set to the following values: 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, 256000.

4.3.3 Testing and Evaluation Technique

After determining how to implement the proposed memoization technique, it becomes necessary to test and evaluate the technique in general and the implementation in particular. For the proposed memoization technique, the speed-up achieved and the amount of error introduced to the output were calculated in order to evaluate the effectiveness of the technique. The hit rate for each algorithm with each combination of memoization parameters were also evaluated.

4.3.3.1 Speed-up

In order to calculate the lower-bound for the speed-up achieved for each algorithm with each image and at each combination of parameters, the following were taken into consideration. To calculate the speed-up of a particular algorithm, two things need to be known: the amount of time required to run the algorithm without memoization and the amount of time required to run the algorithm with memoization applied. It then becomes necessary to determine what is included in an algorithm's processing time. Clearly, any time required to run the algorithm must be taken into account. For the non-memoized algorithm, the

variable time component was used. Only the time to run the algorithm after the image was read and before the image was stored was counted. For the memoized algorithm, as expected, the variable-time component of the memoization technique was included in the memoization time. However, only some of the fixed-time component of the algorithm with and without application of the memoization technique was accounted for. As with the non-memoized algorithm, the time to read and write the images to memory were not taken into account. On the other hand, the time to create and delete the memoization cache was taken into account.

The time to read and write the input and output images were not accounted for because typically an image processing algorithm is not used alone. Rather, an image processing algorithm is typically part of a chain of algorithms that are combined in order to achieve a particular result. Usually, an image is not read immediately before and stored immediately after a particular image algorithm is applied to it. Rather, it is much more common that the image is read before the chain of algorithms is applied and stored after the application of the chain. The assumption is made that the memoized image processing algorithm is only one algorithm from that chain, and therefore it is not necessary to account for the time to read and write the image from memory.

Normally, the memoization cache would need to be created just prior to the first iteration of an algorithm and would need to be deleted after the final iteration of an algorithm. In order to find the lower bound of the speed-up from memoization, the assumption was made that a particular image is both the first and last image that a chain of image processing algorithms, that the selected algorithm is a part of, is being applied to. Therefore, the amount of time required to create and delete the memoization cache was accounted for.

The decision to take into account the time to create and delete the cache will lead to a lower resulting speed-up than seen in previous implementations of memoization. Previous implementations of memoization do not account for the lower-bound speed-up that is achieved with memoization, they

calculate the long-term speed-up that would be achieved [13]. The testing and evaluation technique presented here, however, is designed to calculate the lower-bound speed-up in order to find what the worst-case speed-up would be for an algorithm and what kind of cost would be seen the first time memoization is applied to an algorithm.

4.3.3.2 Error

In order to calculate the error introduced by a particular implementation of memoization with a particular combination of parameters, two things need to be known: the output for the algorithm when memoization is not applied and the output of the algorithm when the memoization is applied. The output for the algorithm when memoization is not applied is the desired output. The output for the algorithm when memoization is applied is the actual output. As will be explained in Section 5, the difference between the desired output and the actual output is the error. In the implementation of memoization, the output for the image processing algorithm without memoization applied was stored in memory. The output from the algorithm with memoization was then compared to that output and the difference was calculated. The final SNR for the output was calculated as per Section 5.

4.3.3.3 Hit Rate

In order to calculate the hit rate for each run of the memoized algorithms with the different combination of parameters, the total number of hits, misses and crashes were monitored. Every time an input window produced a matching window, a hit was recorded. Every time an input window produced a non-matching window, a crash was recorded. Every time an input window resulted in a key pointing to an empty slot in the cache, a miss was recorded. The **total** number of hits was then compared to the total number of hits, misses and crashes to calculate the hit rate.

Comparing hit rate for a particular implementation of memoization does not directly give you the resulting speed-up or error, the two key aspects that must be examined when evaluating an

implementation of memoization. It is nonetheless important to take it into account and to analyze it. In tolerant memoization, increasing the hit rate is a double-edged sword. The hit rate is a key contributing factor to both error and speed-up. All things being equal, a higher hit rate leads to a higher speed-up. In tolerant memoization, based on the tolerance selected, a higher hit rate will also lead to increased error. Therefore, based on the effect of hit rate on both speed-up and error, it is important to analyze what the hit rate for the implementation of memoization with the specified parameters is.

4.4 Time Requirements and Speed-up

Before looking at the results of the memoization algorithm, speed-up and error must be discussed. In this section speed-up is discussed. The next section discusses the error. In the following sections, the actual results of the memoization implementation are presented.

In order to compare and calculate the speed-up, the time required by the memoized algorithm to complete must be compared to the time required by the algorithm to complete without memoization. The time required by the algorithm to complete when memoization is not applied is the baseline. A brief discussion on the amount of time required by the underlying algorithm to process an image follows. There is time required to make a window, T_{window} . There is time required to process the window and calculate the resulting output, T_{calc} . Both of these steps are required at each pixel. Therefore the time required by the baseline algorithm is:

$$T_{baseline} = (T_{window} + T_{calc}) \cdot C \cdot R \quad (4.1)$$

where $T_{baseline}$ is the baseline time, C is the total number of columns of pixels in the image, R is the total number of rows of pixels in the image.

To fully understand speed-up and its importance as it relates to memoization, one needs to look at the time requirements of the memoization algorithm and to analyze where the time is used. The memoization algorithm has multiple steps and each step has a cost involved in performing it. There is a fixed amount

of time needed to perform algorithm specific tasks. There is also a variable amount of time that is dependent on the number of pixels in the file and whether or not a hit occurs. The time required for a hit and a miss are different, as outlined next:

If there is a hit, then the following times are required: time required to make the input window (T_{window}), time required to make the input symbol and corresponding key (T_{sym}), and time required to check the reuse table to see if a previous input matches (T_{check}). Thus total time required for a hit (T_{hit}) is:

$$T_{hit} = T_{window} + T_{sym} + T_{check} \quad (4.2)$$

If there is a miss, then the following times are required: time required to make the input window (T_{window}), time required to make the input symbol and corresponding key (T_{sym}), time required to check the reuse table to see if a previous input matches (T_{check}), time required to perform the calculation (T_{calc}), and time required to store the result in the lookup table as per the eviction policy (T_{store}). Thus total time required for a miss (T_{miss}) is:

$$T_{miss} = T_{window} + T_{sym} + T_{check} + T_{calc} + T_{store} \quad (4.3)$$

The fixed time requirements come from the following steps: time required to make the reuse table, if none exists (T_{create}), and time required to delete the reuse table after the algorithm is completed if no further calculations are required (T_{delete}). Adding these fixed time costs to the algorithm time allows for the calculation of the lower bound for the speed-up. Most of the time when a memoized algorithm is run, the reuse table already exists and does not need to be created or deleted. In terms of processing time, the most time intensive part of the process (creating and deleting of dynamic memory) is not required. In previous implementations of memoization, the assumption was made that the tables already existed. This

artificially created a significantly larger speed-up time and did not provide an accurate representation for the true worst-case time cost of memoization.

The total time required by the algorithm is a combination of all the above factors. The memoization time can be expressed as:

$$T_{memo} = T_{variable} + T_{fixed} \quad (4.4)$$

where T_{memo} is the total time required to run the memoized algorithm, $T_{variable}$ is the variable portion of the memoization algorithm and T_{fixed} is the fixed portion of the memoization algorithm.

The variable component of the total time is:

$$T_{variable} = (T_{hit} \cdot HR + T_{miss} \cdot MR) \cdot R \cdot C \quad (4.5)$$

where HR is the hit rate and MR is the miss rate. Also, hit rate and miss rate are related as follows:

$$1 = HR + MR \quad (4.6)$$

As such, $T_{variable}$ can be expressed as:

$$T_{variable} = (T_{hit} \cdot HR + T_{miss} \cdot (1 - HR)) \cdot R \cdot C \quad (4.7)$$

Using Eq 4.2 and Eq 4.3, $T_{variable}$ can be broken down into the following:

$$T_{variable} = (T_{window} + T_{sym} + T_{check}) \cdot HR \cdot R \cdot C + (T_{window} + T_{sym} + T_{check} + T_{calc} + T_{store}) \cdot (1 - HR) \cdot R \cdot C \quad (4.8)$$

after simplifying Eq 4.8, the result is:

$$T_{variable} = (T_{window} + T_{sym} + T_{check}) \cdot R \cdot C + (1 - HR)(T_{calc} + T_{store}) \cdot R \cdot C \quad (4.9)$$

Thus, the variable component of the total time required by the memoization algorithm is dependent on the window generation time, the symbol generation time, the time to check the reuse table and the hit rate.

The fixed components of the memoization technique are dependent on T_{create} and T_{delete} , which are the time required to create and delete the hash table respectively.

$$T_{fixed} = T_{create} + T_{delete} \quad (4.10)$$

Both T_{create} and T_{delete} are dependent on the hardware and platform used, so they are not discussed in the software implementation. We merely acknowledge that they are present and contribute to the total time required by the memoization algorithm. It was found that these two components end up being the largest time sinks in the memoization algorithm, and once their effect is mitigated (the algorithm is run at least once previously) the resultant speed-up would be much greater. However, this is beyond the scope of this work and is not discussed further here.

The difference in processing time between application of an image processing algorithm in a memoized environment and a non-memoized environment is used to determine the speed-up [14].

$$S = \frac{t_{baseline}}{t_{memo}} \quad (4.11)$$

where S is the speed-up, $t_{baseline}$ is the amount of time used by the algorithm when the memoization technique is not applied and t_{memo} is the amount of time used by the algorithm when the memoization technique is applied.

Taking Eq 4.11 along with Eq 4.1, Eq 4.4 and Eq 4.9 one can derive the following equation:

$$S = \frac{(T_{window} + T_{calc})C \cdot R}{(T_{window} + T_{sym} + T_{check})R \cdot C + (1 - HR)(T_{calc} + T_{store})R \cdot C + T_{fixed}} \quad (4.12)$$

if one assumes that T_{fixed} is negligible in the long run, then:

$$S = \frac{(T_{window} + T_{calc})C \cdot R}{(T_{window} + T_{sym} + T_{check})R \cdot C + (1 - HR)(T_{calc} + T_{store})R \cdot C} \quad (4.13)$$

$$S = \frac{T_{window} + T_{calc}}{(T_{window} + T_{sym} + T_{check}) + (1 - HR)(T_{calc} + T_{store})} \quad (4.14)$$

Eq 4.13 shows that to achieve maximum speed-up, T_{sym} , T_{check} , and T_{store} need to be minimized and hit rate (HR) needs to be maximized. Both T_{window} and T_{calc} are dependent on the underlying algorithm and cannot be changed without directly impacting the algorithm itself, so they are not discussed. Previously, an explanation on how one can select the symbol generation function so that T_{sym} would be as small as possible was given. Additionally, there was a discussion on how the hash function and hash table were selected so that T_{check} would be as small as possible. T_{store} is dependent on the processor and the platform used. It is not something that a software implementation of memoization has any control over, so it is not discussed. The last factor is the hit rate. In order to reduce the variable time for the memoization algorithm, hit rate needs to be maximized. The practical benefit of this is seen in the later section on speed-up.

In literature, there is an assumption that T_{fixed} is negligible in the long run and this is true because the table only needs to be created once and after that point in time, it exists permanently. However, the assumption does not hold true for the first iteration. Thus the fixed component of the memoization technique only plays a factor the first time an algorithm is memoized. On each subsequent iteration of the algorithm, the table would exist and would not need to be remade. Additionally, T_{fixed} is dependent on T_{create} and T_{delete} and these are themselves dependent on the hardware. This thesis is based on software implementation, so they are not discussed. However, it is sufficient to mention that they are present and contribute to the total time required by the memoization algorithm. It was found that these two components end up being the largest time sinks in the memoization algorithm, and once their effect is mitigated (the algorithm is run at least once previously) the resultant speed-up would be much greater. However, this is beyond the scope of this work and is not discussed further.

4.5 Error

Before looking at the results of the memoization algorithm, one must discuss both speed-up and error. In the previous section speed-up was discussed. In this section, error is discussed. In the following sections, the achieved results are presented.

Error is the difference between the desired outcome and the actual outcome. As it relates to memoization, the desired outcome is the output from the underlying algorithm prior to having the memoization technique applied. The actual outcome is the output that results from the algorithm after the memoization technique is applied. In memoization, the absolute error is used and not relative error. Absolute error can be calculated as follows:

$$E = |O_{expected} - O_{actual}| \quad (4.15)$$

where E is the absolute error, $|x|$ is the absolute operator applied to input x , $O_{expected}$ is the expected output, or the output that results when the memoization technique is not applied and O_{actual} is the actual output, or the output that results when the memoization technique is applied.

There are two forms of error that are explored. In the first subsection, the first form of error, mean pixel error, is explored. In the second subsection, the second form of error, signal to noise ratio (SNR), is explored.

4.5.1 Mean Pixel Error

The mean pixel error is the sum of the absolute error from each pixel in the output image divided by the total number of pixels in the image. A lower number is desirable and a zero indicates no error is present in the output.

$$E_{mean} = \frac{\sum_{x=1}^{X=C} \sum_{y=1}^{Y=R} |O_{expected}(x, y) - O_{actual}(x, y)|}{C \cdot R} \quad (4.16)$$

where E_{mean} is the average absolute pixel error, C is the total number of columns of pixels in the image, R is the total number of rows of pixels in the image and $|x|$ is the absolute operator applied to input x .

4.5.2 SNR

The signal to noise ratio (SNR) is the ratio of signal power to noise power. The higher the SNR, the better. The ideal amount of noise is zero which leads to the maximum ratio. The SNR can be calculated in a number of ways. The definition of SNR is [16]:

$$SNR = \frac{P_{signal}}{P_{noise}} \quad (4.17)$$

where P_{signal} is the signal power and P_{noise} is the noise power. In order to calculate the SNR using this formula, both signal power and noise power must be known. If both signal and noise power are not known, then they need to be calculated. Analog signal power can be calculated as follows [16]:

$$P_{signal} = \lim_{T \rightarrow \infty} \left(\frac{1}{T} \right) \cdot \int_{t=-\frac{T}{2}}^{t=\frac{T}{2}} |x(t)|^2 dt \quad (4.18)$$

where $x(t)$ is the signal.

However, for the given situation, the images that are being dealt with are discrete in nature. As such, the discrete signal power needs to be calculated. The formula to calculate discrete-time signal power is [16]:

$$P_{signal} = \lim_{N \rightarrow \infty} \frac{1}{2N+1} \sum_{n=-N}^{n=N} |x(n)|^2 \quad (4.19)$$

The discrete-time signal power formula is for a single dimensional signal. However, the images used are two-dimensional. Thus, the signal power formula needs some expansion. The signal power for an image is calculated as follows:

$$P_{signal} = \frac{1}{C \cdot R} \sum_{i=1}^{i=C} \sum_{j=1}^{j=R} |x(i, j)|^2 \quad (4.20)$$

where $x(i, j)$ is the output when no memoization is applied, C is the total number of columns in the image, and R is the total number of rows in the image.

In order to calculate the noise power, the noise must be known. The noise is also known as the error, so Eq 4.15 can be used.

Using Eq 4.8 and Eq 4.20 one can also calculate the noise power, using the following formula:

$$P_{noise} = P_{error} = \frac{1}{C \cdot R} \sum_{i=1}^{i=C} \sum_{j=1}^{j=R} |e(i, j)|^2 \quad (4.21)$$

where $e(i, j)$ is the error signal, C is the total number of columns in the image, and R is the total number of rows in the image.

Using Eq 4.20 along with Eq 4.17 and Eq 4.21, the SNR calculation for the image can be refined to:

$$SNR = \frac{\frac{1}{C \cdot R} \sum_{i=1}^{i=C} \sum_{j=0}^{j=R} |x(i, j)|^2}{\frac{1}{C \cdot R} \sum_{i=1}^{i=C} \sum_{j=0}^{j=R} |e(i, j)|^2}$$

$$SNR = \frac{\sum_{i=1}^{i=C} \sum_{j=1}^{j=R} |x(i, j)|^2}{\sum_{i=1}^{i=C} \sum_{j=1}^{j=R} |e(i, j)|^2} \quad (4.22)$$

where $x(i, j)$ is the output when no memoization is applied, $e(i, j)$ is the error signal, C is the total number of columns in the image, and R is the total number of rows in the image.

SNR can be expressed in two ways, as a raw number or using a logarithmic scale, i.e. decibels. Eq 4.15 shows the SNR of the memoized image as a raw number. To express the SNR in the logarithmic form, the following equation can be used [16]:

$$SNR_{dB} = 10 \cdot \log \frac{P_{signal}}{P_{noise}} = 10 \cdot \log P_{signal} - 10 \cdot \log P_{noise} \quad (4.23)$$

where SNR_{dB} is the signal to noise ratio expressed in decibel form, $\log$ is the logarithmic function in base 10, P_{signal} is calculated as per Eq 4.20 and P_{noise} is calculated as per Eq 4.20.

Eq 4.23 is used in calculating the resulting SNR from the memoization implementation.

4.6 Summary

The proposed implementation of the memoization technique was presented. The implementation is an adaptation of the general memoization technique in software presented in Chapter 3 but with a number of specific innovations that are designed to improve its effectiveness. The proposed innovations include: a more efficient method to generate symbols, size and type of hash table to use, a new eviction policy, hash function used and the amount of tolerance used. The symbols are generated using the overlapping windows technique. The hash table is used with a direct-mapped mapping scheme and ranges in size from 1000 to 256000 entries. The eviction policy was designed to combine the principles of a first-in-first-out policy along with an eviction policy that is customized based on the input probabilities that are determined at run-time. The hash function selected was the multiplication method because it generally produces higher hit-rates and higher speed-up than the alternative division method. The amount of tolerance allowed ranges from 1 to 8 most significant bits from each input pixel in an input window.

In order to reduce the bias introduced in implementation, the proposed technique was implemented on a general non-specialized platform. The implementation was programmed in C++. In order to analyze and record the statistics of the algorithm outputs with the various inputs, a framework for the implementation was developed. To test the proposed technique, the database of 300 images of the natural class was inputted. Each image had all three algorithms applied to it. The algorithms were each applied 5 times with each combination of parameters. The results were averaged and stored.

To evaluate the effectiveness of the proposed technique, the speed-up and the error introduced to the output were measured. The specific method to calculate the speed-up and the error introduced were explained in Sections 4 and 5. Speed-up was calculated based on the amount of time it took the algorithm to run both with and without memoization applied. Error was calculated based on the difference in resulting outputs when the algorithms are applied with and without memoization.

Chapter 5 Results and Observations

In this chapter the results from the implementation of the memoization technique are presented. Section one includes the speed-up results. Section two includes the error results. Section three includes a comparison and analysis of resulting speed-up and error. Section four includes a comparison of the results achieved using the proposed technique with results achieved using the Khalvati [13] technique. In section 5 general observations and a summary of the results are provided.

5.1 Resulting Speed-up

The speed-up resulting from applying the memoization technique to the three image processing algorithms is highly dependent on the complexity of the algorithm. The image processing algorithm with the most complexity, the Tahir algorithm, saw the greatest resulting speed-up whereas the algorithm with the lowest complexity, the convolution method, saw the lowest resulting speed-up. Speed-up for the three algorithms ranged from 0.32 to a peak of 4.72, where a speed-up of 0.32 is equivalent to a slowing down by a factor of approximately 3.

Table 5.1 and Fig 5.1 tabulate and graph the resulting speed-up for all three algorithms when an input shift of three is used. Table 5.2 and Fig 5.2 tabulate and graph the resulting speed-up for all three algorithms when memoization is applied with a reuse table size of 8000.

Table Size	Convolution	Trajkovic	Tahir
1000	0.746	0.967	1.427
2000	0.753	0.941	1.568
4000	0.759	0.901	1.753
8000	0.756	0.839	1.903
16000	0.750	0.787	2.117
32000	0.728	0.758	2.180
64000	0.697	0.731	2.192
128000	0.665	0.679	2.171
256000	0.637	0.618	2.121

Table 5.1: Comparison of memoized algorithm speed-up for each algorithm at input shift of three

Figure 5.1: Graph of memoized algorithm speed-up for each algorithm at input shift of three

Input Shift	Convolution	Trajkovic	Tahir
1	0.85	1.13	3.69
2	0.82	1.02	2.96
3	0.76	0.84	1.9
4	0.63	0.7	1.2
5	0.62	0.66	1.03
6	0.62	0.65	1.01
7	0.6	0.64	0.91
8	0.57	0.61	0.85

Table 5.2: Comparison of memoized algorithm speed-up for each algorithm with reuse table size of 8000

Figure 5.2: Graph of memoized algorithm speed-up for each algorithm with reuse table size of 8000

While the speed-up is heavily dependent on the underlying algorithm, as expected from Eq. 4.13 it is also highly affected by the input shift used. The amount of input shift corresponds to the amount of information used from each pixel in the input window. The least amount of input shift used, one bit, corresponded to the highest speed-up. The most input shift used, 8 bits, corresponded to the lowest speed-up. The decrease in speed-up from input shift can be attributed to two major factors: the resulting hit-rate and to the size of the symbol. The greatest speed-up is achieved when the hit-rate is highest. Lower speed-up is achieved when the hit rate is lower. The input shift has an inverse relationship with the hit rate. Thus, as the input shift increases, the hit rate and corresponding speed-up decrease. When using a larger input shift, the resulting symbol size is also larger. Larger symbols are more resource intensive than smaller symbols, where symbol size is determined by the number of bytes in the symbol.

The third factor that is analyzed that affects the speed-up achieved by the memoization algorithm is the size of the reuse table. The size of the reuse table has the effect of increasing the hit rate. This should lead to a corresponding increase in speed-up. However, the opposite occurs. Larger reuse table size leads to lower speed-up. The reason for the adverse affect on algorithm speed-up with larger reuse table sizes is the T_{fixed} portion of the time requirements for memoization from Eq 4.4 has a significantly larger role in the speed-up at than the $T_{variable}$ portion. The time requirements for creating the reuse table can

be mitigated by creating the reuse table after the first iteration of the algorithm, but then also keeping the table in memory so that it does not have to be remade on every subsequent iteration of the algorithm.

5.2 Resulting Error

The error resulting from applying the memoization technique to the three image processing algorithms is highly dependent on the hit-rate. The portions of the hit-rate that most affects the error are the reuse table size and the input shift.

The input shift has by far the greatest effect on the error introduced by the memoization technique. The introduction of tolerant memoization allows us to create greater speed-up at a cost of introducing error into the resulting output. For the Tahir algorithm and the Trajkovic corner detection algorithm, the introduction of the error is readily apparent when the input shift is two bits or less. At an input shift of three bits and higher, the resulting error is quite small. For the convolution algorithm, the error introduced is significant up to three bits of input shift and is quite small from four bits and higher. Error is considered to be small if the resultant SNR is greater than 40 dB.

The reuse table size has an adverse effect on the error when tolerant memoization is used. This is expected as the trade-off being made when allowing tolerant memoization is speed-up at a cost of introducing error. The error introduced by varying the reuse table size in tolerant memoization is less significant than the error introduced by the actual input shift. For the convolution algorithm, Tahir algorithm, and Trajkovic corner detection algorithm larger reuse table size can increase the error by up to about 3%, 8% and 1% respectively.

Table 5.3 and Fig 5.3 tabulate and graph the output SNR for the memoized version of all three algorithms when an input shift of three is used. Table 5.4 and Fig 5.4 tabulate and graph the output SNR for all three algorithms when memoization is applied with a reuse table size of 8000.

Table Size	Convolution	Trajkovic	Tahir
1000	31.57	86.48	66.33
2000	29.71	81.21	64.57
4000	27.89	77.03	62.86
8000	26.78	74.60	61.82
16000	25.60	73.28	60.68
32000	25.16	72.72	60.20
64000	24.83	72.43	59.94
128000	24.63	72.25	59.81
256000	24.54	72.19	59.72

Table 5.3: Comparison of memoized algorithm output SNR (dB) for each algorithm at input shift of three

Figure 5.3: Graph of memoized algorithm output SNR (dB) for each algorithm at input shift of three

Input Shift	Convolution	Trajkovic	Tahir
1	1.13	54.61	19.3
2	12.06	57.54	41.56
3	26.78	74.6	61.82
4	48.41	105.17	84.07
5	60.6	108.15	101.41
6	63.24	106.19	98.65
7	83.86	103.9	119.02
8	120	120	120

Table 5.4: Comparison of memoized algorithm output SNR (dB) for each algorithm with reuse table size of 8000

Figure 5.4: Graph of memoized algorithm output SNR (dB) for each algorithm with reuse table size of 8000

5.3 Comparison and Analysis of Speed-Up vs. Error

In order to compare the time savings and the error introduced by memoizing an algorithm, a baseline must be established. The memoized output must then be compared to the baseline in terms of both processing time and the difference in output (error). Comparing the results, it is clear that error and speed are generally inversely related. This makes sense intuitively, as error is introduced when a previous result is reused but simultaneously, time is saved because fewer calculations are required.

Algorithm	Elapsed time without Memoization	Elapsed time with Memoization	Speed Up	Reuse Rate	SNR (dB)
Convolution	0.1722	0.2286	0.756	67.24%	26.78
Trajkovic	0.0438	0.0563	0.839	67.58%	74.60
Tahir	0.7376	0.4171	1.903	67.24%	61.82

Table 5.5: Comparison of algorithm results with and without application of memoization; application with reuse table size of 8000 and input shift of 3

When applying memoization to an algorithm, if maximal speed-up is desired, then a significant amount of error may be introduced to the output on all iterations. On the initial iteration, the speed-up for the Tahir Algorithm and the Trajkovic corner detection algorithm were significant. On subsequent iterations we expect to see even greater speed-up, but testing this is beyond the scope of this thesis.

Figure 5.5: Convolution algorithm with and without memoization. (left to right, top to bottom) original input, output with no memoization, slowest output - shift 8 and cache size 256000, fastest output - shift 2 and cache size 1000, output with most error - shift 1 and cache size 256000, output with least error - shift 8 and cache size 1000, output - shift 3 and cache size 8000

Figure 5.6: Tahir algorithm with and without memoization. (left to right, top to bottom) original input, output with no memoization, slowest output - shift 8 and cache size 256000, fastest output - shift 1 and cache size 8000, output with most error - shift 1 and cache size 256000, output with least error - shift 8 and cache size 1000, output - shift 3 and cache size 8000

Figure 5.7: Trajkovic algorithm with and without memoization. (left to right, top to bottom) original input, output with no memoization, slowest output - shift 8 and cache size 256000, fastest output - shift 1 and cache size 1000, output with most error - shift 1 and cache size 256000, output with least error - shift 8 and cache size 1000, output - shift 3 and cache size 8000

On the other hand, when applying memoization, if the implementation is selected such that no error is introduced, then the memoization serves to slow down the algorithm on its first iteration. Subsequent iterations for a sufficiently complex algorithm should see a speed-up, but testing this is beyond the scope of this thesis.

Ideally, if it is allowable for some error to be introduced to the output, then it is possible to introduce significant speed-up on even the first iteration for both the Tahir algorithm and the Trajkovic corner detection algorithm.

For the convolution algorithm a medium input shift, of at least three, is required to keep the error down. With a medium input shift, the speed-up is low and is detrimental to the overall speed of the algorithm. Thus the convolution algorithm is not well-suited to have the memoization algorithm applied.

For the Trajkovic corner detector algorithm, an input shift of only one or two will result in only a small amount of error introduced to the output. At these input shifts, an average speed-up of up to 17% can be realized on the first iteration of the algorithm. However, once the reuse table is established and subsequent iterations for the algorithm occur, the speed-up would be significantly higher.

For the Tahir algorithm, significant speed-up can still be achieved with small error on the first iteration of the algorithm. A speed-up on the order of 3.25, or 70% can be achieved with an input shift of two or three bits. Subsequent iterations would result in even greater speed-ups. At this small input shift, the error introduced to the output is still quite small.

The inputs and outputs for the different algorithms with a variety of pictures from the database, when memoization is applied and when it is not applied are shown in Figures 5.5, 5.6 and 5.7. Figure 5.5 shows the relevant pictures for input '10075' when the convolution algorithm is applied. Figure 5.6 shows the relevant pictures for input '376020' when the Tahir algorithm is applied. Figure 5.7 shows the relevant pictures for input '100098' when the Trajkovic corner detection algorithm is applied. The image processing algorithms are first applied without the memoization technique. The subsequent pictures show

the output of the algorithm when the memoization technique is applied with different parameters. Specifically, outputs are selected based on the extreme outputs and then the sample output characteristics discussed previously. The output with the specific characteristics were selected: fastest output time, slowest output time, most error introduced and least error introduced. Lastly, the resulting output when the memoization technique is applied with an input shift of 3 and reuse table size of 8000 is also shown.

Table 5.6 shows the average resulting speed-up, Table 5.7 shows the average resulting error, expressed in dB, Table 5.8 shows the average resultant hit rate and Table 5.9 shows the average processing time for the convolution algorithm as the memoization parameters were varied. The memoization parameters varied were input shift and reuse table size, as explained earlier. Analyzing the results from Tables 5.6, 5.7, 5.8 and 5.9 the effects of memoization on the convolution algorithm become clear. Table 5.6, shows that speed-up is negatively correlated with input shift and with reuse table size and Table 5.7 shows that SNR is highly positively correlated with input shift and slightly negatively correlated with reuse table size. Table 5.8 shows hit rate is negatively correlated with input shift and positively correlated with reuse table size and Table 5.9 shows that processing time is positively correlated with both input shift and reuse table size. Based on Equation 5.14 one would expect that increased hit rate, as seen in Table 5.9 should lead to a corresponding increased speed-up but the results on Table 5.6 show that this is not the case. The cause for this is the amount of time required to create and delete the cache. Thus the effect of the time requirements for memory allocation and de-allocation is quite significant. As such, these time requirements should be taken into account when calculating processing time and corresponding speed-up.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	0.856	0.797	0.746	0.652	0.644	0.642	0.629	0.598
2000	0.858	0.802	0.753	0.648	0.638	0.635	0.622	0.591
4000	0.855	0.806	0.759	0.643	0.632	0.629	0.616	0.584
8000	0.853	0.817	0.756	0.633	0.620	0.618	0.602	0.569
16000	0.851	0.819	0.750	0.613	0.599	0.599	0.580	0.540
32000	0.846	0.810	0.728	0.584	0.562	0.559	0.539	0.497
64000	0.837	0.800	0.697	0.545	0.507	0.499	0.478	0.444
128000	0.817	0.787	0.665	0.509	0.457	0.439	0.415	0.387
256000	0.778	0.762	0.637	0.476	0.416	0.389	0.362	0.320

Table 5.6: Resulting speed-up for convolution algorithm after memoization is applied. Speed-up is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	1.13	14.38	31.57	55.15	69.07	75.32	93.63	Infinite
2000	1.13	13.66	29.71	52.07	65.48	70.17	89.58	Infinite
4000	1.13	13.05	27.89	49.95	62.80	66.48	86.42	Infinite
8000	1.13	12.06	26.78	48.41	60.60	63.24	83.86	Infinite
16000	1.13	11.57	25.60	47.33	58.83	60.80	81.32	Infinite
32000	1.13	11.51	25.16	46.41	57.75	58.77	79.10	Infinite
64000	1.13	11.47	24.83	45.90	56.67	56.89	77.18	Infinite
128000	1.13	11.45	24.63	45.47	55.56	55.40	75.34	Infinite
256000	1.13	11.43	24.54	45.19	54.79	54.33	73.94	Infinite

Table 5.7: Output SNR (dB) for convolution algorithm after memoization is applied. SNR is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	99.90%	71.78%	44.03%	20.03%	11.04%	7.69%	3.29%	1.16%
2000	99.90%	77.35%	51.82%	25.52%	14.36%	10.62%	4.41%	1.20%
4000	99.90%	82.49%	60.75%	30.62%	17.78%	14.09%	5.63%	1.24%
8000	99.90%	91.88%	67.24%	35.34%	20.78%	17.66%	6.73%	1.26%
16000	99.90%	95.98%	75.27%	38.08%	23.07%	21.01%	7.62%	1.27%
32000	99.90%	96.65%	78.74%	40.82%	24.35%	24.01%	8.34%	1.28%
64000	99.90%	97.07%	81.06%	42.51%	25.49%	26.48%	9.02%	1.30%
128000	99.90%	97.38%	82.54%	43.92%	26.74%	28.37%	9.60%	1.30%
256000	99.90%	97.52%	83.24%	44.77%	27.83%	29.75%	10.03%	1.31%

Table 5.8: Resulting hit rate for convolution algorithm when memoization is applied. Hit rate is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	0.2013	0.2163	0.2312	0.2645	0.2676	0.2684	0.2738	0.2882
2000	0.2012	0.2150	0.2293	0.2660	0.2702	0.2716	0.2770	0.2915
4000	0.2015	0.2141	0.2276	0.2684	0.2729	0.2742	0.2799	0.2950
8000	0.2020	0.2109	0.2286	0.2726	0.2784	0.2792	0.2863	0.3028
16000	0.2023	0.2105	0.2304	0.2820	0.2880	0.2882	0.2973	0.3195
32000	0.2037	0.2127	0.2377	0.2965	0.3078	0.3093	0.3204	0.3471
64000	0.2057	0.2156	0.2489	0.3191	0.3416	0.3464	0.3612	0.3893
128000	0.2108	0.2192	0.2623	0.3436	0.3812	0.3956	0.4163	0.4459
256000	0.2215	0.2264	0.2747	0.3697	0.4216	0.4499	0.4796	0.5412

Table 5.9: Processing time for convolution algorithm when memoization is applied. Processing time is sorted based on input shift and table size.

Table 5.10 shows the average resulting speed-up, Table 5.11 shows the average resulting error, expressed in dB, Table 5.12 shows the average resultant hit rate and Table 5.13 shows the average processing time for the Tahir algorithm as the memoization parameters were varied. The memoization parameters varied were input shift and reuse table size, as explained earlier. Analyzing the results from Tables 5.10, 5.11, 5.12 and 5.13 the effects of memoization on the Tahir algorithm become clear. Table 5.10 shows speed-up is negatively correlated with input shift and with reuse table size. Table 5.11, shows that SNR is highly positively correlated with input shift and slightly negatively correlated with reuse table size. Table 5.12 shows hit rate is negatively correlated with input shift and positively correlated with reuse table size. Table 5.13 shows that processing time is positively correlated with both input shift and reuse table size. Based on Equation 5.14 one would expect that increased hit rate, as seen in Table 5.13 should lead to a corresponding increased speed-up but as with the convolution algorithm, seen earlier, this is not the case. The cause is the same, and the resultant conclusion is the same: the time requirements for memory allocation and de-allocation should be taken into account when calculating processing time and corresponding speed-up for an implementation of the memoization technique. It is interesting to note though that for the Tahir algorithm, with a low input shift, the effect of the time requirements for memory allocation can be overcome, and an increased hit rate does in fact correspond to a higher speed-up.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	3.694	2.076	1.427	1.048	0.966	0.940	0.901	0.867
2000	3.698	2.270	1.568	1.101	0.988	0.957	0.904	0.862
4000	3.693	2.471	1.753	1.151	1.014	0.985	0.908	0.857
8000	3.689	2.964	1.903	1.196	1.031	1.008	0.907	0.846
16000	3.674	3.212	2.117	1.211	1.039	1.025	0.900	0.831
32000	3.651	3.235	2.180	1.213	1.021	1.025	0.879	0.806
64000	3.611	3.222	2.192	1.194	0.989	0.999	0.844	0.774
128000	3.528	3.197	2.171	1.172	0.956	0.962	0.802	0.733
256000	3.356	3.112	2.121	1.142	0.922	0.916	0.755	0.669

Table 5.10: Resulting speed-up for Tahir algorithm after memoization is applied. Speed-up is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	19.30	44.13	66.33	90.29	109.20	110.49	128.61	Infinite
2000	19.30	43.45	64.57	87.53	105.93	105.40	124.85	Infinite
4000	19.30	42.89	62.86	85.54	103.38	101.76	121.60	Infinite
8000	19.30	41.56	61.82	84.07	101.41	98.65	119.02	Infinite
16000	19.30	41.10	60.68	83.29	99.98	96.34	116.70	Infinite
32000	19.30	41.06	60.20	82.57	99.02	94.36	114.64	Infinite
64000	19.30	41.03	59.94	82.14	98.11	92.65	112.92	Infinite
128000	19.30	41.02	59.81	81.76	97.19	91.24	111.19	Infinite
256000	19.30	41.00	59.72	81.53	96.50	90.23	109.93	Infinite

Table 5.11: Resulting output SNR (dB) for Tahir algorithm after memoization is applied. SNR is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	99.90%	71.78%	44.03%	20.03%	11.04%	7.69%	3.29%	1.16%
2000	99.90%	77.35%	51.82%	25.52%	14.36%	10.62%	4.41%	1.20%
4000	99.90%	82.49%	60.75%	30.62%	17.78%	14.09%	5.63%	1.24%
8000	99.90%	91.88%	67.24%	35.34%	20.78%	17.66%	6.73%	1.26%
16000	99.90%	95.98%	75.27%	38.08%	23.07%	21.01%	7.62%	1.27%
32000	99.90%	96.65%	78.74%	40.82%	24.35%	24.01%	8.34%	1.28%
64000	99.90%	97.07%	81.06%	42.51%	25.49%	26.48%	9.02%	1.30%
128000	99.90%	97.38%	82.54%	43.92%	26.74%	28.37%	9.60%	1.30%
256000	99.90%	97.52%	83.24%	44.77%	27.83%	29.75%	10.03%	1.31%

Table 5.12: Resulting hit rate for Tahir algorithm when memoization is applied. Hit rate is sorted based on input shift and table size.

Table Size \ Input Shift		1	2	3	4	5	6	7	8
1000		0.2019	0.3791	0.5532	0.7253	0.7793	0.7991	0.8292	0.8609
2000		0.2017	0.3450	0.5061	0.6945	0.7641	0.7868	0.8274	0.8658
4000		0.2019	0.3146	0.4533	0.6674	0.7469	0.7685	0.8242	0.8709
8000		0.2022	0.2573	0.4171	0.6452	0.7369	0.7545	0.8260	0.8820
16000		0.2030	0.2342	0.3737	0.6384	0.7335	0.7471	0.8331	0.8986
32000		0.2043	0.2321	0.3618	0.6396	0.7472	0.7512	0.8532	0.9262
64000		0.2065	0.2328	0.3596	0.6523	0.7741	0.7753	0.8904	0.9648
128000		0.2114	0.2345	0.3636	0.6680	0.8046	0.8093	0.9385	1.0180
256000		0.2223	0.2409	0.3726	0.6875	0.8375	0.8540	0.9976	1.1179

Table 5.13: Processing time for Tahir algorithm when memoization is applied. Processing time is sorted based on input shift and table size.

Table 5.14 shows the average resulting speed-up, Table 5.15 shows the average resulting error, expressed in dB, Table 5.16 shows the average resultant hit rate and Table 5.17 shows the average processing time for the Trajkovic corner detection algorithm as the memoization parameters were varied. The memoization parameters varied were input shift and reuse table size, as explained earlier. Analyzing the results from Tables 5.14, 5.15, 5.16 and 5.17 the effects of memoization on the Trajkovic corner detection algorithm become clear. As Tables 5.4 and 5.9 showed for the convolution algorithm and Tahir algorithm respectively, Table 5.14 shows for the Trajkovic corner detection algorithm that speed-up is negatively correlated with input shift and with reuse table size. Table 5.15, shows that SNR is highly positively correlated with input shift and slightly negatively correlated with reuse table size. As Tables 5.6 and 5.10 showed for the convolution algorithm and Tahir algorithm respectively, Table 5.16 shows that hit rate is negatively correlated with input shift and positively correlated with reuse table size. Table 5.17 shows that processing time is positively correlated with both input shift and reuse table size. As with the convolution and Tahir algorithms, generally, increasing the hit rate by using a larger cache does not lead to a corresponding increase in speed-up. The time requirements for memory allocation and deallocation outweigh any time savings achieved by skipping calculations. It is interesting to note though that for the Trajkovic corner detection algorithm, while it is possible to select parameters to achieve either

a high SNR or a speed-up greater than 1 the memoization technique, on the first iteration, at least, is not capable of providing both, as it could with the Tahir algorithm.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	1.172	1.056	0.967	0.869	0.852	0.845	0.832	0.814
2000	1.151	1.045	0.941	0.839	0.818	0.808	0.795	0.777
4000	1.130	1.038	0.901	0.776	0.747	0.748	0.727	0.715
8000	1.130	1.023	0.839	0.701	0.661	0.653	0.635	0.614
16000	1.091	0.998	0.787	0.620	0.565	0.546	0.526	0.502
32000	1.054	0.985	0.758	0.571	0.501	0.472	0.444	0.416
64000	1.005	0.958	0.731	0.542	0.463	0.421	0.396	0.357
128000	0.930	0.887	0.679	0.507	0.426	0.388	0.360	0.320
256000	0.825	0.790	0.618	0.465	0.390	0.356	0.331	0.292

Table 5.14: Resulting speed-up for Trajkovic corner detection algorithm after memoization is applied. Speed-up is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	54.61	60.27	86.48	107.53	104.99	104.95	103.74	Infinite
2000	54.61	59.08	81.21	110.52	105.34	104.83	103.89	Infinite
4000	54.61	58.29	77.03	109.86	106.84	105.48	103.88	Infinite
8000	54.61	57.54	74.60	105.17	108.15	106.19	103.90	Infinite
16000	54.61	57.20	73.28	102.51	108.50	107.35	103.89	Infinite
32000	54.61	57.15	72.72	101.24	109.30	108.19	103.88	Infinite
64000	54.61	57.11	72.43	100.72	109.43	108.61	104.50	Infinite
128000	54.61	57.10	72.25	100.46	109.07	108.62	104.50	Infinite
256000	54.61	57.09	72.19	100.13	108.97	108.67	104.50	Infinite

Table 5.15: Resulting output SNR (dB) for Trajkovic corner detection algorithm after memoization is applied. SNR is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	98.76%	79.69%	54.04%	26.52%	15.34%	12.47%	4.85%	0.75%
2000	98.76%	82.59%	60.11%	29.26%	17.01%	14.01%	5.57%	0.76%
4000	98.76%	85.17%	64.30%	31.26%	18.34%	15.15%	6.16%	0.77%
8000	98.76%	89.87%	67.58%	32.86%	19.35%	16.23%	6.57%	0.78%
16000	98.76%	91.88%	71.00%	33.97%	20.08%	17.20%	6.91%	0.79%
32000	98.76%	92.13%	71.67%	34.85%	20.53%	17.96%	7.13%	0.80%
64000	98.76%	92.28%	72.05%	35.15%	20.81%	18.42%	7.27%	0.80%
128000	98.76%	92.36%	72.29%	35.39%	20.99%	18.63%	7.37%	0.81%
256000	98.76%	92.41%	72.37%	35.51%	21.09%	18.73%	7.42%	0.81%

Table 5.16: Resulting hit rate for Trajkovic corner detection algorithm when memoization is applied. Hit rate is sorted based on input shift and table size.

Table Size \ Input Shift	1	2	3	4	5	6	7	8
1000	0.0399	0.0443	0.0485	0.0538	0.0549	0.0553	0.0561	0.0574
2000	0.0409	0.0449	0.0499	0.0558	0.0573	0.0581	0.0587	0.0601
4000	0.0416	0.0451	0.0521	0.0604	0.0628	0.0625	0.0644	0.0655
8000	0.0414	0.0458	0.0563	0.0676	0.0712	0.0719	0.0740	0.0769
16000	0.0429	0.0471	0.0605	0.0774	0.0848	0.0874	0.0905	0.0957
32000	0.0445	0.0477	0.0633	0.0845	0.0973	0.1031	0.1086	0.1175
64000	0.0467	0.0489	0.0658	0.0916	0.1059	0.1181	0.1236	0.1390
128000	0.0503	0.0530	0.0712	0.0971	0.1163	0.1283	0.1373	0.1544
256000	0.0567	0.0594	0.0779	0.1061	0.1272	0.1392	0.1489	0.1710

Table 5.17: Processing time for Trajkovic corner detection algorithm when memoization is applied. Processing time is sorted based on input shift and table size.

5.4 Results Comparison with Results from Khalvati [13] Technique

Khalvati [13] implemented a version of window memoization that is very similar to the proposed technique. Because of the similarity in technique, in this section a comparison between the results from the proposed technique and the results that Khalvati achieved with his technique is presented. Khalvati tested his technique with three processors. Of the three processors, processor 2, with the following specifications, Intel(R) XEON(TM), CPU: 1.80GHz, cache size: 512 KB has the closest specifications to the processor used in the implementation of the proposed technique. The processor used in the implementation of the proposed technique, was explained in Chapter 4. Khalvati selected a particular

input shift, four, and reuse table size, 16000, to test his implementation on. Khalvati tested his technique on 6 different image processing algorithms. Of the 6 algorithms that he tested, and the three algorithms that were implemented in this thesis, there is 1 algorithm in common, the Trajkovic corner detection algorithm.

For the Trajkovic corner detection algorithm, with the processor and memoization parameters selected, Khalvati was able to achieve a speed-up of 3.73x. In the implementation in this thesis, with the same parameters, a speed-up of 0.620x, or a slow-down of 1.61x was achieved. In Khalvati's implementation, the fixed time requirements for window memoization were not factored. Whereas, in the implementation here, they were included. The significant difference in speed-up shows that it is necessary to take the fixed-time requirements into account when determining the speed-up of an algorithm. At the same time, Khalvati's results show that in the long-run significant time savings can be achieved by implementing the memoization technique.

5.5 General Observations and Summary of Results

Both the speed-up and the resulting error from applying the memoization technique to the three image processing algorithms are generally inversely related to the reuse table size and the input shift. Speed-up is highest with a small reuse table and small input shift and lowest with a large reuse table and large input shift. Error is highest with a large reuse table and a small input shift. Error is lowest at a large input shift regardless of table size. The ideal combination of reuse table size and input shift is highly dependent on the underlying algorithm.

The convolution technique is a very simple image processing algorithm. The calculations required to compute an output from an input window take less time than retrieving a result from memory. As such, the convolution technique is not well-suited to the memoization technique.

The Trajkovic corner detector algorithm and the Tahir algorithm are both good candidates for the memoization technique. When memoization is applied, even on the first iteration, when the memoization parameters are selected so that the output error is kept small, significant time savings and corresponding speed-up can be achieved.

Chapter 6 Conclusion

In this work, a software process to improve the efficiency of software algorithms was analyzed. The software process analyzed takes advantage of the computational redundancies that exist in the underlying algorithm to reduce the time it takes to process an input with a given algorithm. It was shown that it is also possible to further improve performance with a marginal loss of accuracy that cannot be perceived by the human eye.

Previous work in the field was shown to be promising but at the same time, needing improvement in some areas. One particular work in specific with results that were quite promising is the software memoization process proposed by Khalvati [13]. Khalvati took the software memoization technique and applied it to local image processing algorithms that use a window-based approach. This gives rise to the window memoization technique. He showed that images and image processing algorithms have a high degree of redundancy in two principal forms: inter-pixel redundancy and coding redundancy. This makes image processing algorithms ideal candidates for the application of the window memoization technique.

Memoization is a technique that uses a trade-off between speed and memory. In memoization, once a particular input is processed, the results are stored in a cache. Whenever the input occurs in the future, rather than recalculating, the result is retrieved from the cache thus saving the calculation and corresponding time that would otherwise be required. In theory, maximal time savings can be achieved if infinite memory is used and all possible inputs exist in the cache. Practically, it is not possible to create an infinite cache so a fixed cache of finite size is used.

Because of hardware and software constraints, cache size is not unlimited. Due to limited cache size, maximum time savings are limited. An eviction policy is key to maximizing time savings because, when the cache is small, the eviction policy greatly affects the hit rate. The hit rate is the number of inputs for which matches can be found in the cache and therefore can have their associated calculations skipped.

The eviction policy presented in the proposed technique is designed to maximize hit rate without prior knowledge of the input by evicting entries that have not been previously reused and keeping those entries that have been reused.

To further increase hit rate, tolerant memoization can be used. In tolerant memoization, the requirements for matching inputs to the cache can be relaxed so that a match does not need to require an exact copy but merely a sufficiently similar copy. Tolerant memoization typically leads to additional time savings at a cost introducing a small amount of error to the output. Since the human eye is not capable of detecting small amounts of error, tolerant memoization can be used, provided the amount of error introduced is small [13]. When the tolerance is set such that inputs are at least 50% matching, the error introduced to the output is negligible.

In window memoization, the cache is implemented in the form of a hash table. The hash table must be optimally designed to maximize speed-up. The direct-mapped implementation of a hash table was selected in order to minimize the time required to look up entries in the table.

In order to map the hash table, a hash function is required. The multiplication method generally had better results than the division method. The multiplication method is the hash function that was selected for implementation of the proposed memoization technique.

The proposed technique was applied to three local image processing techniques: a simple algorithm, convolution; a moderately-complex algorithm, the Trajkovic corner detection algorithm; and a complex algorithm, the Tahir algorithm. Upon implementing memoization for the three algorithms, it was found that the speed-up achieved by the memoization process is highly dependent on the base algorithm being memoized. At the lower bound of memoization speed-up, a simple algorithm, with relatively few calculations to convert an input window to an output pixel is slowed down when memoization is applied but both the moderately and complex image processing algorithms displayed promising speed-ups. For

both the moderately and complex image processing algorithms, tuning is required to achieve maximal speed-up.

The Tahir algorithm, a completely new algorithm that was proposed here, was the third algorithm selected. The algorithm is designed to modify the basic low-pass filter so that it is better suited to images that have pixel intensities concentrated at the lower end of the pixel intensity spectrum. The algorithm gives more weight to pixels in an input window that have lower intensity than to pixels that have higher intensity.

In order to evaluate the effectiveness of an implementation of memoization, it is prudent to find the speed-up and the error introduced at the lower bound. This is accomplished by creating the cache just prior to running an algorithm and destroying the cache just after. The time required to create and delete the cache is then included in the algorithm time. Both the resulting error and speed-up are then compared to the baseline (algorithm without memoization) to gauge effectiveness. Based on this measure, the proposed memoization technique is effective on moderately complex and highly complex algorithms. However, the technique is not effective on low-complexity algorithms.

Generally for a technique, reuse rates of less than about 15-20% lead to a speed-up of less than 1. To overcome the performance overhead caused by window memoization, the technique must be able to save a significant amount of calculations if a mask operation is skipped. For low-complexity algorithms, this is simply not possible. For higher-complexity algorithms, time savings can be achieved due to the number of calculations that can be skipped. Also, the memoization mechanism must be very efficient and fast. If the technique is slowed down, then the effectiveness is lost. When the technique is ineffective either too much error is introduced or too much time is consumed in processing the underlying algorithm.

Future work for this research could be in four major directions: since the hit rate is one of the most important factors in determining the speed-up achieved by memoization, innovative ideas to increase the hit rate need to be developed. This can include researching a hash function that better distributes

incoming symbols and research into a more intelligent eviction policy. The second direction of future work is determining **for** which algorithms memoization should be applied. Since the results from memoization are highly dependent on the complexity of the base algorithm, it becomes important to decide whether to apply memoization to a particular algorithm rather than applying it blindly by default. The third direction of future work is to analyze the effect of introducing error to fields where even a little bit of error may be critical, such as the medical and military fields. The fourth direction for future work is to evaluate the effect of changing the set of input images. Images of the natural class were used in the implementation of memoization. However, memoization is dependent on the entropy in images. It would be worth exploring the effect of using images of higher entropy or lower entropy on the algorithm speed-up and on the amount of error introduced.

Memoization is a promising field that is young and has lots of room for future work. Using memoization, significant time savings can be achieved at a relatively small cost of memory usage and error that is undetectable to the human eye. All things considered, the technique is worth looking into in more detail.

Appendix A

More Results

Figure A.1: Graph of hit ratio vs. input shift for convolution algorithm

Figure A.2: Graph of SNR vs. input shift for convolution algorithm

Figure A.3: Graph of speed-up vs. table size and input shift for convolution algorithm

Figure A.4: Graph of hit ratio vs. input shift for Tahir algorithm

Figure A.5: Graph of normalized SNR vs. input shift for Tahir algorithm

Figure A.6: Graph of speed-up vs. table size and input shift for Tahir algorithm

Figure A.7: Graph of hit ratio vs. input shift for Trajkovic corner detection algorithm

Figure A.8: Graph of SNR vs. input shift for Trajkovic corner detection algorithm

Figure A.9: Graph of speed-up vs. table size and input shift for Trajkovic corner detection algorithm

Bibliography

- [1] U. Acar, G. Blelloch and R. Harper. "Adaptive Memoization," Carnegie Mellon University Computer Science Department, 2003.
- [2] U. Acar, G. Blelloch, and R. Harper. "Selective memoization," *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, January 2003.
- [3] U. Acar, A. Ahmed and M. Blume. "Imperative Self-Adjusting Computation," *Proceedings of the 35th Annual ACM symposium on Principles of programming languages*, 2008, pp. 309-322.
- [4] C. Alvarez, J. Corbal and M. Valero. "Fuzzy memoization for floating-point multimedia applications." *IEEE Transactions on Computers*, vol.54, no.7, pp. 922- 927, Jul 2005.
- [5] P. Arbelaez, C. Fowlkes and D. Martin, "The Berkeley Segmentation Dataset and Benchmark". University of California, Berkeley Electrical Engineering and Computer Sciences Department , June 2007. [Online]. Available:
<http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/>
- [6] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein. *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press, 2001.
- [7] Y. Ding and Z. Li. "Operation reuse on handheld devices," *Languages and Compilers for Parallel Computing*, 2003, vol. 2958 pp. 273-287.
- [8] R. Easton. *Basic Principles of Imaging Science*. Rochester, NY: Chester F. Carlson Center for Imaging Science, Rochester Institute of Technology, 2005.
- [9] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*, 3rd Edition. Upper Saddle River, NJ: Prentice Hall, 2008.

- [10] Robert Harper, "Self-Adjusting Computation," *19th Annual IEEE Symposium on Logic in Computer Science*, July 2004, pp.254-255.
- [11] J. Hughes. "Lazy memo-functions," *Conference on Functional Programming Languages and Computer Architecture*, 1985, vol. 201 pp. 129-146.
- [12] A.K. Jain. "Image data compression: A review." *Proceedings of the IEEE* , vol. 69, no. 3, pp. 349-389, March 1981.
- [13] F. Khalvati. "Computational Redundancy in Image Processing." PHD Thesis, University of Waterloo, Canada, 2008.
- [14] F. Khalvati, M.D. Aagaard, and H.R. Tizhoosh. "Accelerating Image Processing Algorithms Based on the Reuse of Spatial Patterns," *Canadian Conference on Electrical and Computer Engineering*, April 2007, pp.172-175.
- [15] F. Khalvati, M.D. Aagaard, and H.R. Tizhoosh. "Opposition-Based Window Memoization for Morphological Algorithms," *IEEE Symposium on Computational Intelligence in Image and Signal Processing*, April 2007, pp.425-430.
- [16] B.P. Lathi. *Linear Systems and Signals*. 2nd Edition. New York: Oxford University Press, 2005
- [17] A. Leon-Garcia. *Probability and Random Processes for Electrical Engineering*. 2nd Edition. Reading, MA: Addison-Wesley Publishing Company, 1994.
- [18] D. Michie. "Memo functions and machine learning". *Nature*, vol. 218 pp. 19-22, April 1968.
- [19] D. Parks and J.P. Gravel. "Corner Detector." McGill University, Centre for Intelligent Machines. April 2005. [Online]. Available:<http://www.cim.mcgill.ca/~dparks/CornerDetector/trajkovic8.htm/>
- [20] A. Pfeffer. "Sampling with Memoization," *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, 2007, vol. 2, pp. 1263-1270.

- [21] W. Pugh "An improved replacement strategy for function caching," *In Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, 1988, pp. 269-276.
- [22] S. Richardson "Exploiting trivial and redundant computation," *IEEE Symposium on Computer Arithmetics*, June 1993, pp. 220-227.
- [23] J.P. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors* . McGraw-Hill Higher Education, 2005.
- [24] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis, and Machine Vision*. 2nd Edition, CL-Engineering September 1998.
- [25] M. Trajkovi and M. Hedley. "Fast corner detection," *Elsevier Science: Image and Vision Computing*, February 1998, vol. 16, pp. 75-87.
- [26] L. Yi. G. Zhijun, "A review of segmentation method for MR image," *International Conference on Image Analysis and Signal Processing*, April2010, pp.351-357.