Analysis and Parallelization of JPEG-2000 Reference Software for General-Purpose Processors

by

Bo Fan

A thesis submitted to the Department of Electrical and Computer Engineering in conformity with the requirements for the degree of Master of Applied Science

Queen’s University
Kingston, Ontario, Canada
October 2011

Copyright © Bo Fan, 2011
Abstract

Like many other multimedia applications, image compression involves a significant amount of data processing for coding images. Sophisticated general-purpose processors with parallel architectures and advanced cache systems can be dedicated to enhancing performance for serial multimedia applications through parallelization.

This thesis describes parallelization of the JasPer reference software for the JPEG-2000 image compression standard and presents results from simulation, and from hardware execution on a multicore processor where speedups of more than 2 are obtained with 4 processors. Results from execution and cache behavior analysis are presented to establish the expected speedup and to further characterize JasPer execution.

The JasPer encoding process has been analyzed on a single processor for both simulated and hardware execution in order to obtain more insights into application behavior. On recent hardware platforms, the significant contributors to the total execution time have been identified through profiling. The granularity of parallelism for parallelizable loops have been analyzed for execution on real hardware. Cache behavior and memory access pattern have been studied closely for the simulated execution.

To facilitate parallelization, selected parallelizable loops have been transformed in order to assist the partitioning of loop iterations for parallel execution and to increase workload granularity and reduce synchronization overhead. These modifications include loop index and body transformation, and loop fusion.

A memory access pattern tracking feature has also been introduced for serial and parallel execution of a program in simulation. This feature tracks the number of memory accesses in
a particular data region during a particular interval of time in order to gain additional insights into execution behavior.

The multithreaded execution of the parallelized JasPer encoder presents a relatively balanced workload which indicates a reasonable efficiency for parallel execution. The generated images have been compared against their original images by using analytical tools to ensure the image quality and to verify correctness.
Acknowledgements

I am heartily thankful to my supervisor, Dr. Naraig Manjikian, for offering me the opportunity to be a part of his research group. I have truly appreciated his insights, guidance, and patience throughout my research. I would also like to thank Dr. Subramania Sudharsanan who served as a nominal co-supervisor for the first part of this research and who contributed to my research assistantship support.

I am grateful for the financial support provided by Natural Sciences and Engineering Research Council of Canada, Sun Microsystems, ViXS Systems, and Queen’s University. The support for equipment used in this research was provided by the Canada Foundation for Innovation and by the Ontario Innovation Trust.

I would dedicate this thesis to my grandmother for her never-ending love and encouragement, and to the loving memory of my parents, without their support and unwavering faith in me, I would never be where I am today.

Finally, my sincere thanks to my fiancee, Yunfei Wang, for her unconditional love and support.
Contents

Abstract ii

Acknowledgements iv

List of Tables viii

List of Figures x

Chapter 1: Introduction 1

1.1 Contributions ................................ 5

1.2 Outline of Thesis .............................. 6

Chapter 2: Background and Related Work 8

2.1 Overview of Image Compression ................ 9

2.2 Image Compression Standards ................ 10

2.3 JPEG-2000 Image Compression Standard ...... 11

2.4 Overview of the JPEG-2000 Standard ........ 12

2.5 Structure of the JPEG-2000 Encoder .......... 14

2.6 Reference Software Implementations of JPEG-2000 16
Chapter 3: Single-Thread Execution Analysis

3.1 Execution Profiling
   3.1.1 Test Environment
   3.1.2 Profiling Results from gprof
   3.1.3 Potential Speedup
   3.1.4 Results from SimpleScalar
   3.1.5 Hardware Execution Results
   3.1.6 Loop Iteration Count and Granularity of Parallelism

3.2 Cache Behavior Analysis
   3.2.1 The Cachegrind Modeling Tool
   3.2.2 Results from Cachegrind
   3.2.3 Results from SimpleScalar

3.3 Memory Access Tracking in SimpleScalar

3.4 Summary

Chapter 4: Single-Thread Program Transformations

4.1 Loop Index Transformation

4.2 Loop Body Transformation
Chapter 5: Parallelization and Execution Analysis

5.1 Overview

5.2 Parallelization for Simulated Execution

5.2.1 Speedup and Load Balance in Simulation

5.2.2 Tracking Memory Access Behavior for Multithreaded Execution

5.2.3 Image Comparison Results for Multithreaded Simulated Execution

5.3 Parallelization for Hardware

5.3.1 OpenMP Application Programming Interface

5.3.2 Parallelization for Code-block Processing Step

5.3.3 Parallelization for DWT

5.3.4 Parallelization for Quantization

5.3.5 Results from Multithreaded Hardware Execution

5.3.6 Image Comparison Results for Multithreaded Execution

5.4 Summary

Chapter 6: Conclusion

6.1 Future Work

Bibliography
List of Tables

Table 3.1 The complete list of test images ........................................ 26
Table 3.2 Detailed lossless encoding profiling results for cats image ........ 28
Table 3.3 Detailed lossy encoding (9/7 filtering) profiling results for cats image 30
Table 3.4 Profiling results obtained with gprof on Quad64 platform ............ 31
Table 3.5 Potential speedup calculated from gprof profiling results for cats image 32
Table 3.6 Cycle counting results for cats image in Simulation .................. 34
Table 3.7 Hardware execution results on Intel Core 2 Quad platform .......... 36
Table 3.8 Profiling results for lossless encoding from Cachegrind ............... 42
Table 3.9 Profiling results for lossy encoding from Cachegrind ................ 42
Table 3.10 Cachegrind results for cropped images from lossless encoding .... 43
Table 3.11 Cachegrind results for cropped images from lossy encoding ......... 44
Table 4.1 Comparison of total instruction fetches before and after loop fusion 63
Table 5.1 Simulated parallel execution results (in cycles) for lossless encoding of cats image ................................................................. 71
Table 5.2 Simulated parallel execution results (in cycles) for lossy encoding of cats image ................................................................. 72
Table 5.3  Speedup results for encoding cats image in simulation  

Table 5.4  Speedups for lossless encoding of cats image  

Table 5.5  Multithreaded lossless encoding results on Quad64  

Table 5.6  Multithreaded lossy encoding results on Quad64  

Table 5.7  Image comparison results from lossy compression  

ix
List of Figures

Figure 2.1 The basic structure of the JPEG-2000 encoder ............................ 14

Figure 3.1 The cats image (provided courtesy of Phil Fennessy, UK, for testing and evaluation purposes) ........................................ 25

Figure 3.2 The pcb image and its tiled version ........................................ 26

Figure 3.3 The eso0905a image (credit: ESO – reproduced with permission as indicated by the terms of use on the ESO website at http://www.eso.org) .... 27

Figure 3.4 Call tree graph with profile for lossless encoding of cats image .... 28

Figure 3.5 Call tree graph with profile for lossy encoding of cats image ....... 30

Figure 3.6 Nested loops in jpc_enc_enccblks function ............................... 37

Figure 3.7 Loop iterations in jpc_enc_enccblks function for cats image ... 38

Figure 3.8 Maximum number of code-block loop iterations ....................... 40

Figure 3.9 Data memory access pattern in 2-D ......................................... 47

Figure 3.10 Data memory access pattern for the load references ................ 49

Figure 3.11 Data memory access pattern for the store references ................ 50

Figure 3.12 Data memory access pattern for the total number of data references ... 51

Figure 4.1 Original loop header .............................................................. 55
<table>
<thead>
<tr>
<th>Figure 4.2</th>
<th>Transformed loop header</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.3</td>
<td>Original loop bodies for jpc_ft_analyze</td>
<td>57</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Transformed loop bodies for jpc_ft_analyze</td>
<td>58</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Original loop bodies for jpc_ns_analyze</td>
<td>59</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Transformed loop bodies for jpc_ns_analyze</td>
<td>60</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Original loops with pointer index variables</td>
<td>62</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Fused loop with integer index variable</td>
<td>62</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>The distribution of parallel workload for lossless encoding of cats image</td>
<td>71</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>The distribution of parallel workload for lossy encoding of cats image</td>
<td>72</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Data memory access pattern for load instructions (thread 0)</td>
<td>75</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Data memory access pattern for load instructions (thread 1)</td>
<td>76</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Data memory access pattern for load instructions (thread 2)</td>
<td>76</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Data memory access pattern for load instructions (thread 3)</td>
<td>77</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Data memory access pattern for store instructions (thread 0)</td>
<td>77</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>Data memory access pattern for store instructions (thread 1)</td>
<td>78</td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>Data memory access pattern for store instructions (thread 2)</td>
<td>78</td>
</tr>
<tr>
<td>Figure 5.10</td>
<td>Data memory access pattern for store instructions (thread 3)</td>
<td>79</td>
</tr>
<tr>
<td>Figure 5.11</td>
<td>Data memory access pattern for all instructions (thread 0)</td>
<td>79</td>
</tr>
<tr>
<td>Figure 5.12</td>
<td>Data memory access pattern for all instructions (thread 1)</td>
<td>80</td>
</tr>
<tr>
<td>Figure 5.13</td>
<td>Data memory access pattern for all instructions (thread 2)</td>
<td>80</td>
</tr>
<tr>
<td>Figure 5.14</td>
<td>Data memory access pattern for all instructions (thread 3)</td>
<td>81</td>
</tr>
<tr>
<td>Figure 5.15</td>
<td>Parallelizing loop with OpenMP in jpc_enc_encblks</td>
<td>84</td>
</tr>
</tbody>
</table>
Figure 5.16 Parallelizing loops with OpenMP in jpc_ft_analyze.......... 86
Figure 5.17 Parallelizing loops with OpenMP in jpc_ns_analyze......... 87
Figure 5.18 Parallelizing loop with OpenMP in jpc_quantize .......... 87
Figure 5.19 Speedups from lossless encoding on Quad64 ................. 92
Figure 5.20 Speedups from lossy encoding on Quad64 .................... 93
Chapter 1

Introduction

As one of the subcategories of multimedia applications, image coding for data compression involves a significant amount of processing which challenges the capability of the underlying hardware system. Image compression standards have been established for encoding and decoding images. The latest JPEG standard, JPEG-2000 [Int00], offers additional features and enhanced image compression efficiency relative to the original JPEG standard introduced in 1992. Despite higher computational demands for enhanced standards, sophisticated general-purpose multicore processors are sufficiently capable to efficiently handle the image compression entirely in the software domain. A software-based implementation for the JPEG-2000 standard is beneficial for providing an efficient and cost-effective solution for image coding. In fact, a reference implementation of the JPEG-2000 standard called JasPer [AW04] serves such a purpose. JasPer is a widely used implementation which has been incorporated into other image coding programs.

Many approaches in general-purpose processors can be used to enhance performance for serial multimedia applications, and one of them is through parallel execution. This research
seeks to improve performance for the JasPer program by reducing its execution time through parallelization. Multiprocessors in a single chip offer enhanced processing capability, where parallelized applications can be efficiently executed with the benefits of an advanced cache system. Parallel execution can reduce the execution time for computation-intensive operations in a program by utilizing task parallelism for concurrent execution.

For the specific application of image compression based on the JPEG-2000 standard, the JasPer software has task parallelism that can be exploited. Previous efforts have demonstrated the feasibility of enhancing JasPer performance through parallelization. JasPer execution on hardware was profiled and the software was parallelized by using OpenMP. However, those efforts utilized an earlier version of JasPer on a previous generation of hardware. We seek to assess execution and cache behavior of the latest version (1.900.1) of the JasPer software for both simulated and hardware execution on recent hardware in order to obtain more insights into application behavior. In addition, we transformed the original source code to enhance transformation efficiency for parallelization.

In this thesis, analysis of execution and cache behavior for JasPer is conducted initially on a single processor in order to characterize the serial execution behavior of JasPer. Candidate functions which contribute most to the execution time are then selected for further analysis. The potential for parallel speedup is assessed based on these results. Before parallelizing JasPer, the serial code, particularly code with loops that are parallelizable, has been transformed to enhance the code efficiency for parallelization. Finally, JasPer is parallelized explicitly for simulated execution, and automatically for execution on real hardware. Workload balance and the memory access patterns are analyzed to characterize the multithreaded execution. In addition, the generated images from multithreaded execution are compared to the corresponding
original images using the analytical software tools in order to verify correctness.

Multimedia applications have influenced the development of general-purpose microprocessors to a great extent [DMCY02]. Other alternative architectures, such as those implemented in Graphics Processing Unit (GPU) and Field-Programmable Gate Array (FPGA) chips, are available for multimedia applications. However, they are intended to be used for specific purposes in order to gain efficiency in performance. FPGA performance is generally superior if the algorithm involves a significant number of memory accesses, whereas GPU performance tends to be higher when there is significant data reuse [CCLH10]. A general-purpose processor possesses a clock rate which is 6-15 times higher than FPGA and 6-8 times higher than GPU [CCLH10].

There are many advantages of a software-based implementation. It provides a cost-effective way to utilize the power delivered by general-purpose processors with relatively lower implementation complexity and no additional hardware. For image compression applications, support for a variety of image formats can be easily realized in software. With operating systems and corresponding software development tools available in general-purpose processors, the application program can be effectively created and analyzed.

Image compression has been standardized to ensure the creation of equivalent representations by different image coding tools. A number of image compression standards possessing different characteristics have emerged, such as JPEG [Int92], JPEG-LS [Int99a], MPEG-4 VTC [Int99b], PNG [Wor03], and JPEG-2000 [Int00]. The JPEG-2000 standard, which is the successor of the original JPEG standard, is viewed as the most flexible solution that provides enhanced coding performance with many attractive features for lossy and lossless image compression [SCE00]. Compared to the original JPEG standard, JPEG-2000 is able to com-
press images by as much 30% more in a lossy manner and still maintain comparable image quality [SALM02] [Ebr02] [BLR05].

As one of the reference implementations defined in the JPEG-2000 Part-5 standard [Int02e], the JasPer software [AW04] provides an image coding platform based on the JPEG-2000 Part 1 standard [Int02a]. JasPer is the only reference implementation of JPEG-2000 in the C programming language. It includes an image transcoder that can handle various image formats, as well as utility programs for image comparison and viewing. The full JasPer software contains approximately 45,000 lines of raw code. Each of the codec drivers for the different image formats is organized as a separate module. Each module provides necessary functions to compress or decompress image data for a certain image format. All modules interact through a base library, which provides interfaces for manipulating image data. Because JasPer is open-source, it is widely used and is proven to be stable. It has been incorporated into a variety of commercial and non-commercial programs, such as KDE, XnView, Netpbm and many others [Ada].

Like many other reference implementations for image coding standards, the JasPer program is a serial application targeted for a single core processor. Nonetheless, the JPEG-2000 standard is designed to be flexible, where image elements can be processed concurrently. The independent data organization for manipulating the code-block, which is the smallest image element, is emphasized in the JPEG-2000 standard [CSE00]. Therefore, with the independent data organization scheme in the standard and the support from the parallel architecture in general-purpose processors, performance for JasPer can potentially be enhanced through parallelization.
1.1 Contributions

The thesis has four main contributions. Firstly, the JasPer reference implementation for JPEG-2000 has been analyzed thoroughly in order to obtain more insights into application behavior. Execution has been analyzed comprehensively, not only on real hardware as in previous work, but also in simulation. The use of simulation has enabled cache behavior analysis and memory access pattern tracking for the JasPer encoding process. Execution profiling for JasPer has been conducted in order to identify the significant contributors to the total execution time. More recent hardware platforms that offer higher intrinsic performance have been used to perform experiments as opposed to the previous-generation hardware systems used in previous work.

Secondly, a variety of source code transformations are detailed for JasPer. The modifications that are applied include loop index and body transformations which facilitate the partitioning of loop iterations for parallel execution. Loop fusion has also been used to increase workload granularity and reduce synchronization overhead.

Thirdly, with the availability of multicore processors, the parallel execution of JasPer has been characterized for hardware and simulated execution. The parallelization process has included steps to identify, optimize, and transform the parallelizable portions in JasPer. Parallelization candidates have been identified through profiling. The potential for speedup has been assessed in simulation. On hardware, speedups of more than 2 are obtained with 4 processors. Reasonable parallel execution efficiency has been confirmed by the relatively balanced workload for the parallel threads. Based on the profiling results, the significant contributors to the total execution time that could conceivably be parallelized have been addressed effectively.

Finally, a useful practical contribution is the development of simulation support to track
memory access patterns in order to gain additional insights into execution behavior. This capability has been integrated into the SimpleScalar simulator to track the memory access pattern for any application program whose execution is simulated. Using the data obtained from this tracking, it is possible to create a graphical representation for the access patterns in a particular memory region during a particular interval of time, based on the number of data load and store references for these memory accesses. The memory access pattern graphs can then be used to characterize the execution and understand the cache performance. Particularly for the multithreaded execution, a good understanding of the memory access pattern ensures that the programmer optimizes performance for a program in an appropriate way.

1.2 Outline of Thesis

The remainder of this thesis is organized as follows. Chapter 2 describes image compression and established standards for compressed image formats. The structure of the JPEG-2000 standard and the JasPer reference implementation are also discussed. To aid in verification of correctness for modified software, image comparison metrics are discussed in order to quantify the differences between the images. Chapter 3 describes results from hardware execution and simulation for the JasPer encoding program on a single processor. Execution profiling and cache behavior are considered, along with memory access behavior. Parallelizable portions in the JasPer software are identified, and the expected speedup is established from the execution profiling results. Chapter 4 describes code transformations that are applied to the original JasPer source code. These transformations included loop index transformation, loop body transformation, and loop fusion. Chapter 5 describes the parallelization of the serial
JasPer image encoder and presents results obtained from simulation and on real hardware. The software is explicitly parallelized for simulated execution and automatically parallelized for execution on real hardware. The parallel workload is examined in order to ensure that a balanced distribution is achieved. Memory access patterns for multithreaded execution are also analyzed. Finally, Chapter 6 concludes the thesis with a summary and suggestions for future work.
Chapter 2

Background and Related Work

This chapter provides background related to image compression and established standards for compressed image formats. Since its introduction in 1992, the original JPEG standard has proven its value for lossy image compression. The subsequent JPEG-2000 standard, however, offers numerous additional features for enhanced image compression efficiency, scalability, and interoperability. JPEG-2000 supports both lossless and lossy image compression within a single unified coding framework. The JPEG-2000 features are demonstrated in the reference software implementations. One of these reference implementations is discussed in detail in the subsequent sections of this chapter. Our research is largely based on analyzing this reference implementation, and transforming it for performance enhancement. When performing image compression, imperfections might be introduced into compressed image data. In order to quantify the imperfections for generated images, analytical methodologies for image comparison are used. Finally, several previous papers targeting performance enhancement for image compression are also examined. Selected work is discussed in terms of its achieved speedup.
2.1 Overview of Image Compression

Digital images vary in their sizes and qualities. As high-definition image and video capturing devices become more prevalent, high-resolution digital images are used primarily. These images are able to store more fine details, but they have large sizes which require additional storage space and longer transmission time. Compressing image data is necessary for the sake of saving storage space and enabling rapid image transmission.

An image is numerically represented as a two-dimensional sequence of pixels which is the smallest addressable element in the image. A pixel contains a number of samples from the original image. In a typical greyscale image, every pixel contains exactly one sample which consists of values from 0 to 255 when using 8 bits for intensity. A typical color image, on the other hand, can be represented and displayed by a sequence of 24-bit pixels which consist of 3 samples for Red, Green, and Blue (RGB). Therefore, a color image can be seen as a superposition of three color component planes. RGB information can also be encoded by using YCrCb where Y represents the luminance and Cr and Cb are the red-difference and blue-difference chroma components. Both RGB and YCrCb are the commonly used color models to represent a color image numerically.

The main goal for image compression is to represent an image using as few bits as possible. In order to achieve this goal, the source image has to be encoded into a compressed data format. The compressed image data can be decoded to obtain the reconstructed image. Generally, there are two modes of operation for a typical image compression standard, namely, lossless and lossy [TM02].

In lossless mode, the compressed image can be perfectly reconstructed to a representation
which is identical to the original image. Lossless compression is important for certain applications, for example, in medical imaging or technical drawing, as all fine details must be preserved.

In lossy mode, some information from the original image will be lost permanently and cannot be restored. Lossy compression aims to reduce the amount of data used to represent an image. Lossy compression is a trade-off in which the compression achieves desired reduction in compressed data size at the cost of reducing the image quality. The compressed data size relative to the uncompressed data size is known as the compression ratio. This ratio can be chosen based on the image size and desired quality. Lossy compression affects image quality. Nonetheless, a considerable amount of image data can be discarded in a lossy compression before the imperfections in the resulting image become significant. Human interpretation is generally insensitive to insignificant degradations in the image. A lossy compressed image can often be virtually indistinguishable from its original uncompressed image, depending on the encoder program and compression parameters.

2.2 Image Compression Standards

Since the mid-1980s, experts from the International Telecommunication Union (ITU) and International Organization for Standardization (ISO) have been establishing an international image compression standard of the Joint Photographic Experts Group (JPEG). The original JPEG standard has become an International Standard for the compression of images since 1992. Formally, the original JPEG standard is defined as ISO/IEC 10928-1, or Recommendation T.81 in International Telecommunication Union - Telecommunication Standardization Sector (ITU-T).
After being used for almost two decades, the original JPEG standard has proven a valuable tool for lossy compression of digital images.

### 2.3 JPEG-2000 Image Compression Standard

Over time, many new applications that use image compression have emerged. These applications, such as medical imaging and satellite image archiving, require more compression capabilities and computational power for image coding. Hence, the underlying image compression standard needs to satisfy all of these new requirements. The original JPEG standard is considered to no longer fulfill the requirements for advanced image compression [KJ08]. Even before more recent acceptance of this view, the JPEG committee began to draft a new image compression standard.

In March 1997, the development of a new standard for the compression of still images, the JPEG-2000 standard [Int00], was officially initiated. Formally, the JPEG-2000 standard is defined as Joint Technical Committee (JTC) 1.29.14, also known as ISO/IEC 15444-1 or ITU-T Recommendation T.800 [Int02a]. It became International Standard ISO/IEC 15444-1 in December 2000. The JPEG-2000 standard uses advanced compression techniques based on the wavelet transform to compress images. The JPEG-2000 standard supports both lossy and lossless compression of image data.

The JPEG-2000 standard addresses various weaknesses of the earlier JPEG standard such as poor low bit-rate image compression or the lack of single unified framework for lossy and lossless compression. JPEG-2000 also provides new features for enhanced image compression efficiency, scalability, and interoperability in network and mobile environments. It contains
numerous additional features, such as progressive recovery of an image, enhanced error re-
silience, region-of-interest coding, and random access to a selected part of an image without
requiring the entire code stream to be processed. It offers superior image compression qualities
and uses a more flexible file format.

Like the original JPEG standard, the JPEG-2000 standard is offered for use on a royalty-
free basis. This decision is important for the standard to become widely accepted in the field.
Due to its excellent coding performance and many attractive features, the JPEG-2000 standard
has a large potential application base. It can better serve image archiving, Internet, digital
photography, medical imaging, scanning, remote sensing, mobile devices, E-commerce, and
desktop publishing [Ada05].

2.4 Overview of the JPEG-2000 Standard

The JPEG-2000 standard contains various parts.

- Part 1 [Int02a] is the core coding system which contains the minimum functionality of
  the JPEG-2000 standard. This part is also known as the baseline codec of JPEG-2000.

- Part 2 [Int02b] defines more features and various extensions to Part 1 of the JPEG-2000
  standard.

- Part 3 [Int02c] describes what is known as “Motion JPEG-2000” which is used to repre-
  sent motion sequences of JPEG-2000 images.

- Part 4 [Int02d] outlines methodologies for testing conformance to JPEG-2000 Part 1.
• Part 5 [Int02e] describes two reference software implementations of JPEG-2000 Part 1 (Java and C implementations are outlined).

• Part 6 [Int03b] defines the compound image file format for storing multi-page documents.

• Part 7 has been abandoned. It provided guidelines for minimum support functions related to JPEG-2000 Part 1.

• Part 8 [Int06] addresses security issues for the JPEG-2000 standard. Such issues are important for applications to generate, consume, and exchange JPEG-2000 secured bit-streams.

• Part 9 [Int04] describes client-server interactive protocols for supporting image data transmission over communication networks.

• Part 10 [Int07a] defines extensions for coding three-dimensional floating-point data. This part is still under development.

• Part 11 [Int07b] defines methods and tools for efficiently transmitting JPEG-2000 imagery over error-prone wireless networks.

• Part 12 [Int03a] defines a common base file format which can be used for both JPEG-2000 and MPEG-4.

Most of the parts described above have been or will be published as International Standards. The JPEG-2000 standard is continuously evolving and being extended to meet new requirements for advanced image compression.
2.5 Structure of the JPEG-2000 Encoder

The JPEG-2000 standard defines the structure of a codec which contains an encoder and a decoder for compressing and decompressing the image data. We are particularly interested in the encoder because it is usually more computationally complex than the decoder. Also, the encoding process is the most important step for image compression because it is the prerequisite for studying the decoding process. But most importantly, the encoding process is the reverse operation of the decoding process. Therefore, we only focus our attention on discussing the JPEG-2000 encoder.

The JPEG-2000 standard specifies the organization of data for a digital image in terms of components. Each image component can then be split into a number of sub-bands using the discrete wavelet transform (DWT) [Ada05]. Each sub-band is further decomposed into code-blocks which can be encoded independently.

The JPEG-2000 encoder in Figure 2.1 consists of several basic processing steps.

- In the Preprocessing step, the values of the input sample data are adjusted to be centered about zero.

- The Intercomponent Transform performs either an irreversible or a reversible color transform based on the user’s requirement. The irreversible color transformation involves operations with real numbers. In a computer system, real numbers are represented as
floating-point numbers. This representation causes a precision loss due to the fixed number of significant digits. This is the reason why the transformation is irreversible. In contrast, the reversible color transformation performs an integer-to-integer conversion with no loss of precision. Both of these transformations essentially map image data from RGB to YCrCb color space because YCrCb components are typically less statistically dependent than RGB components. Therefore, YCrCb is more suitable for the independent transformation of each color component in the DWT.

- In the Intracomponent Transform stage, individual image components undergo wavelet transformation operations where each component is split into various frequency bands called sub-bands.

- The Quantization process is a lossy process which reduces the dynamic range of coefficients for the transformed image components to a specified level. Doing so reduces the output image file size, but it also reduces the quality of the image reconstructed from the compressed data.

- The Tier-1 Encoder partitions each sub-band into code blocks which will be coded independently.

- The Tier-2 Encoder is responsible for bit rate allocation and packet reordering. It collects the output from the Tier-1 encoder to create image data packets, and outputs the final code stream.

- The Rate Control is responsible to achieve the desired bit rate in encoding. It can be realized by choosing the desired step size in quantization, or by selecting an optimal
2.6 Reference Software Implementations of JPEG-2000

The specification of the JPEG-2000 standard includes a baseline codec (coder/decoder) description and two reference software implementations of the baseline codec called JasPer [AW04] and JJ2000 [JJ2]. JasPer is implemented in C, and JJ2000 is implemented in Java.

JasPer is an open-source project which provides a software-based reference implementation of JPEG-2000. JasPer is the implementation selected for this research effort. The details of this reference implementation will be discussed in the following section.

2.6.1 JasPer Software

The JasPer Project [AW04] provides open-source software as a reference implementation based on the JPEG-2000 Part 1 standard. The JasPer software is one of the reference implementations of the JPEG-2000 Part-1 specified in the JPEG-2000 Part-5 standard. This software includes an image transcoder that can handle various image formats, as well as utility programs for image comparison and viewing. The software is written in the C programming language and consists of approximately 45,000 lines of raw code. The JasPer software is expandable, Any additional support for new image formats can be easily added to the tool suite [AW04]. Because it is open-source, JasPer has been incorporated into a variety of commercial and non-commercial programs, such as KDE, XnView, Netpbm, and many others [Ada].

The JasPer software implements the steps in Figure 2.1. In JasPer, the Discrete Wavelet Transform (DWT) in the Intracompoment Transform and the code-block processing step in the
Tier-1 Encoder are two significant contributors to execution time, and therefore they are candidates for performance enhancement. The code-block processing step for the Tier-1 encoder is based on extensive bit-level processing for a large number of independent code-blocks. The discrete wavelet transform (DWT) within the Intracomponent Transform processing step is realized using a lifting scheme [AW04] which is an efficient algorithm that performs the wavelet computation.

### 2.6.2 JJ2000 Software

JJ2000 [JJ2] is the other reference implementation defined in the Part 5 of JPEG-2000 standard. It is implemented in Java. It was developed by Canon Research, EPFL, and Ericsson. Unfortunately, the website of the JJ2000 project is no longer accessible, and any detailed information about this project cannot be obtained.

### 2.7 Image Quality Assurance

Image comparison serves the purpose of quantifying the differences between images. Imperfections might be introduced when performing image compression. Therefore, the image quality has to be measured in an appropriate way. Using visual comparison alone for generated images may not be acceptable.

In fact, the comparison can be done by using the analytical comparison tool provided with JasPer toolkit. We have also created a comparable custom image comparison tool in C++. Compressed images are compared to the original image by using the image comparison tools. Both the analytical tools produce the same comparison results, as described later in this thesis.
The tools use various metrics to measure the quality of the reconstructed image from a lossy compression. Two metrics are commonly used to assess the reconstructed image. The mean squared error (MSE) is a ratio that describes the amount of distortion introduced by the compression process. For two images $I_1$ and $I_2$ with the same dimensions $N_1 \times N_2$, the MSE is defined in Equation 2.1, where $I_1[x,y]$ is the color value of the pixel at the two-dimensional coordinates given by $x$ and $y$.

$$\text{MSE} = \frac{1}{N_1 N_2} \sum_{x=0}^{N_1-1} \sum_{y=0}^{N_2-1} (I_1[x,y] - I_2[x,y])^2$$ \hspace{1cm} (2.1)

The peak signal-to-noise (PSNR) is simply the equivalent reciprocal measure to MSE [TM02]. It is often expressed in terms of the logarithmic decibel scale. The PSNR value can be calculated by using Equation 2.2, where $B$ is the number of bits used to represent each sample, and $(2^B - 1)$ is the maximum possible pixel value for the image.

$$\text{PSNR} = 10 \log_{10} \left( \frac{(2^B - 1)^2}{\text{MSE}} \right)$$ \hspace{1cm} (2.2)

For example, if $B = 8$ bits, the maximum possible pixel value is 255.

The ideal values for a perfect (lossless) image reconstruction are 0 for MSE and infinity for PSNR. For lossy image compression, the generally accepted criterion of $\text{PSNR} > 30 \text{ dB}$ is used to distinguish a good image reconstruction [TM02].

### 2.8 Related Work

The DWT and code-block processing steps of the JasPer software are two significant contributors to execution time, hence they are candidates for performance enhancement. Previous efforts had demonstrated various approaches to enhance performance for the JasPer reference
Cook and Delp [CD95] proposed tiling of an image, and distributing the tiles to different processors for parallel execution. Although JPEG-2000 supports image tiling in low-memory environments, tile-based parallelization potentially impairs image quality as the number of tiles and processors increases [MNU02a]. This is due to the fact that the wavelet transform in JPEG-2000 is performed in each individual block independently.

Other earlier work has described for parallelizing JasPer. Norcen et al. [MNU02a][NU05] parallelized an earlier version of the JasPer software. In their work, two previous-generation hardware platforms were used for experimentation; the SGI Power Challenge multiprocessor platform which has 20 IP25 RISC CPUs running at 195MHz, and the SGI Origin 3800 system which has 128 MIPS RISC R12000 CPUs, running at 400MHz. The Lena image which has a dimension of 4096 × 4096 was used in the tests. They had parallelized what accounted for 84.6% of the execution time in the JasPer encoder and experimented on the SGI Power Challenge. A speedup of 3.34 was achieved from the parallel execution on 10 processors. Furthermore, they claimed that the DWT constitutes 69.9% of total execution time, but this reflects the occurrence of cache misses as we will describe below. On the Origin 3800 system, 79.8% of JasPer code had been parallelized and executed. A speedup of 2.72 was achieved on 10 processors.

However, in their work, they used an earlier version of JasPer on a large number of previous-generation processors with smaller caches having lower associativity. We use fewer, faster processors with enhanced cache capacity and associativity to experiment with the latest version (1.900.1) of the JasPer software [Ada].

Additionally, the 9/7 lossy transformation was primary focus in the previous efforts. We
have chosen to also focus our attention on using the 5/3 transformation for the lossless image compression. One reason for also choosing lossless image compression is that we have found that lossless encoding has somewhat better performance than the lossy encoding to compress the same image. Additionally, lossless compression is important for certain applications where all fine details in the image must be preserved.

In the previous paper by Norcen et al. [NU05] [MNU02b], the occurrence of cache misses has received considerable attention. Their results showed that during the DWT process, when the image width is a power of two and the filter length is greater than 4, cache misses in the vertical filtering process increased execution time significantly. They believe this is due to an entire column in the image being mapped onto a single cache set. Therefore, they provided a solution to enhance the data locality in cache for the vertical filtering by reordering the corresponding memory access. They conducted the experiments for their parallelized JasPer with the cache enhancement. On the SGI Power Challenge system, 66.5% of the execution time had been parallelized, and resulted a speedup of 2.22 on 10 processors. The modification to address cache misses reduces execution time considerably so that the DWT constitutes 33.6% of the total execution time. On Origin 3800, a speedup of 2.23 was achieved on 10 processors with 65.1% of the execution time parallelized in JasPer. For both cases, the parallelizable portions of JasPer were reduced, and the experimental speedups were also decreased.

In our research, we will spend some efforts to examine the cache issue for the vertical filtering, and demonstrate in a later chapter that this is no longer the case for the latest version of the JasPer software. Moreover, we have studied JasPer closely to characterize its cache behaviors for both direct-mapped and associative cache systems. To better explain the cache behavior, we have gone further to explore the memory access patterns.
2.9 Summary

This background chapter discussed the need for image compression and established standards for compressed image formats. The structure of the JPEG-2000 standard and the JasPer reference implementation have been described. The image comparison metrics have been discussed in order to quantify the differences between the images. Finally, previous efforts related to the performance enhancement of image compression software have also been presented.
Chapter 3

Single-Thread Execution Analysis

This chapter describes results of hardware execution and simulation for the JasPer encoding program on a single processor before describing the code transformation and parallelization of the software in the subsequent chapters. Profiling is used to assess the contribution of different portions of the software to the total execution time. This profiling is done on hardware and in simulation. Parallelizable portions in the JasPer software are identified using knowledge of the JPEG-2000 standard and some insights obtained in previous work. These portions are inspected in order to characterize the independent workload. Profiling results are used to predict speedup. The results will also be used to guide our subsequent parallelization decisions. For the cache behavior analysis, the sensitivity of the miss rate to cache parameters in simulation is investigated. Our observation for the cache problems reported in previous efforts is stated. Furthermore, the memory access patterns are studied in order to obtain more insights into the low cache miss rates that were observed in the cache profile.

This chapter is organized as follows. Section 3.1 describes the profiling of JasPer execution by using the gprof tool on real hardware, and by using SimpleScalar in simulation.
Section 3.2 provides results of analyzing cache behavior for JasPer. Section 3.3 describes the additional memory access tracking feature in SimpleScalar that enables our investigation of the memory access behavior.

### 3.1 Execution Profiling

This section presents results of execution analysis by using profiling on a single processor. On real hardware, the GNU `gprof` tool is used to characterize the execution of the encoding process in original JasPer. In simulation, a modified version of the SimpleScalar tool is used for profiling. Based on the profiling results obtained from both hardware and simulated executions, speedups are predicted.

#### 3.1.1 Test Environment

For execution and cache profiling of the JasPer encoding process on hardware, the following two computer systems are used. For brevity, each system is assigned a label which will be used for identification in the later discussion.

- The *Quad64* system consists of a 2.4-GHz Intel Core 2 Quad Q6600 processor with 4 Gbytes of DDR2-800 RAM, running 64-bit Debian Linux. Each processor has an L1 cache capacity of 32 Kbytes (8-way associative), and the total L2 cache capacity is 8 Mbytes (8-way associative).

- The *Duo32* system consists of a 1.86-GHz Intel Core 2 Duo E6320 processor which has 64-byte line size, 32-Kbyte capacity (8-way associative) in L1, and 4-Mbyte capacity
(8-way associative) in L2. This platform has 3 Gbytes of DDR2-800 RAM and a 32-bit Ubuntu Linux operating system.

On our Quad64 and Duo32 platforms, we use the latest version (4.4.5) of the gcc compiler to compile the source code. The gcc compiler has the ability to optimize the generated code to varying degrees based on a specified optimization level. The \texttt{-O2} and \texttt{-O3} optimization levels are commonly used in compiling source code. However, some preliminary tests show that the JasPer code with \texttt{-O3} optimization level does not significantly enhance the program performance in terms of the execution time. Furthermore, the compiler takes longer to compile the code with the \texttt{-O3} optimization level. Therefore, in all our experiments, we decided to use the \texttt{-O2} optimization level to compile the JasPer program because this optimization can provide increased performance of the generated code without facing the space-speed tradeoff [Freb].

In our experiments, lossless and lossy image encodings have been done. For lossless encoding, JasPer employs 5/3 filtering for a five-level DWT via lifting, and for lossy encoding, JasPer employs 9/7 filtering for a five-level DWT via lifting. However, the 9/7 filtering has to be explicitly enabled by specifying the option \texttt{mode=real}. In addition, we set the \texttt{rate} option to 1 for lossy encoding. A lower \texttt{rate} results in reduced execution time and reduced image quality.

A suite of test images with different attributes are used for testing, as summarized in Table 3.1. All test images are in the Portable Any Map (PNM) format, which represents a collection of image formats that were designed for exchanging images between platforms. The collection includes the Portable Pix Map (PPM), the Portable Gray Map (PGM), and the Portable Bit Map (PBM) formats. They all contain uncompressed image data. JasPer natively supports
Figure 3.1: The cats image (provided courtesy of Phil Fennessy, UK, for testing and evaluation purposes)

the PNM format. Therefore, we chose to use this image format for convenience. The cats image in Figure 3.1 was obtained from the JasPer software website [Ada]. This image is in greyscale which contains exactly one component. It has two large black color regions, which are useful for testing the encoder’s capability to compress image data with large areas of same color. The pcb image is a photograph of a large printed-circuit board containing multiple FPGA chips in the Computer Architecture Laboratory at Queen’s University. It includes many similar small objects, and its background has smooth color transitions. It is used in full size, in quarter size, and in tiled form comprising four instances of the quarter-sized version (see Figure 3.2). Finally, eso0905a in Figure 3.3 is an astronomical telescope image (available from http://www.eso.org) which contains fine details and a significant amount of subtle color transitions.
Figure 3.2: The pcb image and its tiled version

Table 3.1: The complete list of test images

<table>
<thead>
<tr>
<th>Image name</th>
<th>Resolution</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats.pnm</td>
<td>3072 x 2048</td>
<td>6.00</td>
</tr>
<tr>
<td>pcb_large.pnm</td>
<td>3264 x 2448</td>
<td>9.26</td>
</tr>
<tr>
<td>pcb_small.pnm</td>
<td>1632 x 1224</td>
<td>5.71</td>
</tr>
<tr>
<td>pcb_tiled.pnm</td>
<td>3264 x 2448</td>
<td>22.9</td>
</tr>
<tr>
<td>eso0905a.pnm</td>
<td>8408 x 8337</td>
<td>200.6</td>
</tr>
<tr>
<td>eso0905a_small.pnm</td>
<td>4204 x 4169</td>
<td>18.77</td>
</tr>
</tbody>
</table>

3.1.2 Profiling Results from gprof

The GNU gprof tool [Frea] can profile execution of software by collecting statistics at runtime. It provides an efficient way to analyze the execution time contribution of each individual function in an application program. The gprof profiling result contains two tables which are used for presenting the flat profile and the call graph. The flat profile shows the time that a program spent in each function, and the number of calls to each function. The call table describes the call tree of the program, where the caller and callee are listed with the number of calls.

All of the gprof profiling experiments on hardware were conducted on the Quad64 sys-
Figure 3.3: The eso0905a image (credit: ESO – reproduced with permission as indicated by the terms of use on the ESO website at http://www.eso.org)

tem. The detailed analysis for JasPer execution to encode the cats image is described first, followed by a discussion of compressing the entire test image suite.

Table 3.2 summarizes profiling results for lossless encoding of the cats image. We observed that the code-block processing (cblk) and DWT consumed 55.32% and 16.31% of the total execution time, respectively. Figure 3.4 elaborates on the summary results shown in Table 3.2. The shaded ovals for functions show their contributions to the total execution time in
Table 3.2: Detailed lossless encoding profiling results for cats image

<table>
<thead>
<tr>
<th>Proc. step</th>
<th>Function name</th>
<th>% time</th>
<th>time (sec)</th>
<th># of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>cblk</td>
<td>jpc_enc_enccblk</td>
<td>28.37</td>
<td>0.4</td>
<td>1,538</td>
</tr>
<tr>
<td></td>
<td>jpc_encclnpass</td>
<td>9.22</td>
<td>0.13</td>
<td>5,351</td>
</tr>
<tr>
<td>55.32%</td>
<td>jpc_mqenc_codelps</td>
<td>12.77</td>
<td>0.18</td>
<td>6,426,130</td>
</tr>
<tr>
<td></td>
<td>jpc_mqenc_codemps2</td>
<td>4.96</td>
<td>0.07</td>
<td>5,998,542</td>
</tr>
<tr>
<td>DWT</td>
<td>jpc_qmfb_split_colgrp</td>
<td>6.38</td>
<td>0.09</td>
<td>372</td>
</tr>
<tr>
<td>16.31%</td>
<td>jpc_ft_fwdlift_colgrp</td>
<td>5.67</td>
<td>0.08</td>
<td>372</td>
</tr>
<tr>
<td></td>
<td>jpc_ft_fwdlift_row</td>
<td>2.13</td>
<td>0.03</td>
<td>3,968</td>
</tr>
<tr>
<td></td>
<td>jpc_qmfb_split_row</td>
<td>2.13</td>
<td>0.03</td>
<td>3,968</td>
</tr>
</tbody>
</table>

Execution of the encoding process

......

\[\text{jpc\_enc\_enccblk} \quad 28.37 \%
\]

\[\text{jpc\_enc\_encchlp} \quad 9.22 \%
\]

\[\text{jpc\_mqenc\_codelps} \quad 12.77 \%
\]

\[\text{jpc\_mqenc\_codemps2} \quad 4.96 \%
\]

......

\[\text{jpc\_ft\_analyze} \quad \]

\[\text{jpc\_qmfb\_split\_colgrp} \quad 6.38 \%
\]

\[\text{jpc\_ft\_fwldlift\_colgrp} \quad 5.67 \%
\]

\[\text{jpc\_qmfb\_split\_colres} \quad \]

\[\text{jpc\_ft\_fwldlift\_colres} \quad \]

\[\text{jpc\_qmfb\_split\_row} \quad 2.13 \%
\]

\[\text{jpc\_ft\_fwldlift\_row} \quad 2.13 \%
\]

unshaded ovals for functions with no percentage given

have a negligible fraction of the total execution time

Figure 3.4: Call tree graph with profile for lossless encoding of cats image

percentage form.

For lossy compression of the cats image, profiling results are summarized in Table 3.3,
which has been split into three groups. The code-block processing step (cblk), DWT, and quantization (qntz) consumed 33.54%, 27.44%, and 9.76% of the total execution time, respectively. Figure 3.5 presents the call tree graph where the percentage in shaded ovals for functions indicates their contribution to the total execution time.

The significant time contributors to the total execution time which are shown in Table 3.2 and Table 3.3 are identified from the profiling results for encoding the cats image. Other functions contributing small fractions of execution time in the profiling results are not considered; even if there is parallelism in such functions, the potential contribution to the overall speedup would not be significant.

Comparing Table 3.2 and Table 3.3, there are more calls in the code-block processing step for lossless encoding than for lossy encoding. In DWT, the number of calls are identical, but different functions are invoked in JasPer. The quantization step is only applicable for lossy encoding, which constitutes less than 10% of total execution time.

Profiling results obtained for encoding the entire test image suite are shown in Table 3.4. We compare lossless compression that uses 5/3 filtering with the lossy 9/7 filtering where rate is set to 1. The execution times are comparable, with somewhat smaller times from 5/3 filtering. There are, however, notable differences in the fractions of execution time spent in the code-block (cblk) processing and DWT steps. For both processing steps, their fractions range from 72% to 84% for lossless encoding, from 45% to 59% for lossy encoding, relative to the total execution time. The quantization (qntz) step only exists in lossy encoding, it contributes from 10.7% to 16.3% to the total execution time.
Table 3.3: Detailed lossy encoding (9/7 filtering) profiling results for cats image

<table>
<thead>
<tr>
<th>Proc. step</th>
<th>Function name</th>
<th>% time</th>
<th>time (sec)</th>
<th># of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>cblk</td>
<td>jpc_enc_encclblk</td>
<td>20.73</td>
<td>0.34</td>
<td>1,538</td>
</tr>
<tr>
<td>33.54%</td>
<td>jpc_encclnpass</td>
<td>7.32</td>
<td>0.12</td>
<td>4,420</td>
</tr>
<tr>
<td></td>
<td>jpc_mqenc_codelps</td>
<td>2.44</td>
<td>0.04</td>
<td>4,418,577</td>
</tr>
<tr>
<td></td>
<td>jpc_mqenc_codemps2</td>
<td>3.05</td>
<td>0.05</td>
<td>4,312,483</td>
</tr>
<tr>
<td>DWT</td>
<td>jpc_qmfb_split_colgrp</td>
<td>4.27</td>
<td>0.07</td>
<td>372</td>
</tr>
<tr>
<td>27.44%</td>
<td>jpc_ns_fwdlift_colgrp</td>
<td>16.46</td>
<td>0.27</td>
<td>372</td>
</tr>
<tr>
<td></td>
<td>jpc_ns_fwdlift_row</td>
<td>6.1</td>
<td>0.1</td>
<td>3,968</td>
</tr>
<tr>
<td></td>
<td>jpc_qmfb_split_row</td>
<td>0.61</td>
<td>0.01</td>
<td>3,968</td>
</tr>
<tr>
<td>qntz 9.76%</td>
<td>jpc_quantize</td>
<td>9.76</td>
<td>0.16</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 3.5: Call tree graph with profile for lossy encoding of cats image
Table 3.4: Profiling results obtained with gprof on Quad64 platform

<table>
<thead>
<tr>
<th>Image file</th>
<th>Dimensions</th>
<th>Size (Mb)</th>
<th>Lossy 9/7 profile (%)</th>
<th>Lossless 5/3 profile (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>blk</td>
<td>DWT</td>
</tr>
<tr>
<td>cats.pnm</td>
<td>3072×2048</td>
<td>6.00</td>
<td>36.7</td>
<td>23.4</td>
</tr>
<tr>
<td>pcb_small.pnm</td>
<td>1632×1224</td>
<td>5.71</td>
<td>36.2</td>
<td>10.7</td>
</tr>
<tr>
<td>pcb_large.pnm</td>
<td>3264×2448</td>
<td>9.26</td>
<td>35.4</td>
<td>13.5</td>
</tr>
<tr>
<td>pcb_tiled.pnm</td>
<td>3264×2448</td>
<td>22.9</td>
<td>46.9</td>
<td>7.9</td>
</tr>
<tr>
<td>eso0905a_small.pnm</td>
<td>4204×4169</td>
<td>18.77</td>
<td>28.8</td>
<td>16.6</td>
</tr>
<tr>
<td>eso0905a.pnm</td>
<td>8408×8337</td>
<td>200.6</td>
<td>34.4</td>
<td>23.3</td>
</tr>
</tbody>
</table>

### 3.1.3 Potential Speedup

Speedup quantifies the performance improvement of a parallel program over its serial version. For a multithreaded application program, the measured speedup can be calculated based on the actual execution time for both serial and parallel execution as defined in Equation 3.1.

\[
S(p) = \frac{\text{Elapsed time for executing task sequentially on 1 processor}}{\text{Elapsed time for executing task in parallel on } p \text{ processors}} \tag{3.1}
\]

The expected speedup can also be predicted mathematically. Suppose that \( f_s \) represents the fraction of execution time that must be executed sequentially, and \( f_p \) is the fraction of the original sequential task that can be executed in parallel on \( p \) processors. The speedup can be formally predicted by using Amdahl’s Law [Amd67], which is defined in Equation 3.2.

\[
S(p) = \frac{1}{f_s + f_p/p} \tag{3.2}
\]
Table 3.5: Potential speedup calculated from gprof profiling results for cats image

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Potential Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lossless encoding</td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>2.2</td>
</tr>
</tbody>
</table>

The sum of \( f_p \) and \( f_s \) equals 1 that represents the entire task.

As stated in the JPEG-2000 standard described in Chapter 2, image data are organized independently for encoding in the code-block processing step and DWT. Earlier work has parallelized these portions. Based on our experimental results presented in Table 3.2 and Table 3.3, the parallelizable portions constitute 72% of total execution time for lossless encoding and 71% of total execution time for lossy encoding. The potential speedups estimated by using Amdahl’s Law are shown in Table 3.5.

In Table 3.5, only 2 and 4 threads are chosen to predict the speedup due to the nature of our Quad64 platform. The Quad64 system contains a quad-core processor which is manufactured by combining two dual-core processors together. Therefore, 2 and 4 threads are used for performing all experiments on our Quad64 platform.

### 3.1.4 Results from SimpleScalar

SimpleScalar [ALE02] is a popular simulation tool set that has been developed for computer architecture research. It provides an efficient and flexible infrastructure for implementing a particular model of a hardware architecture to assess performance. It supports a RISC-style instruction set. The original release of SimpleScalar includes uniprocessor models which sup-
port fast functional simulation, cache simulation, and detailed superscalar simulation. In our research, a version of SimpleScalar with multiprocessor enhancements [Man01] is used. It models the state of multiple processors and provides a runtime library with routines for thread creation and synchronization for application programs. Because it is based on the original functional simulation model, an ideal single-cycle latency is assumed for all simulated instructions. No caches are simulated. The configuration of the underlying hardware platform does not influence the simulation results, hence the simulations can be performed on any computer on which the SimpleScalar software is installed.

In order to profile the execution of JasPer in simulation, we analyze the time fractions spent in parallelizable portions relative to the total simulated instructions. Because each instruction takes one cycle to run in SimpleScalar, these time fractions can be determined by counting the number of instructions that have been executed for these portions relative to the total number of instructions.

We have modified both the JasPer software and the multiprocessor SimpleScalar simulation code to accomplish the desired cycle-count profiling for JasPer. Two functions were added to the SimpleScalar runtime library in order to invoke system calls to start or end the cycle counting feature. Each library function includes a system call instruction. The SimpleScalar simulator recognizes system call instructions and uses information provided in simulated registers to determine which service to perform. For starting and stopping the cycle-count feature, the current simulation time is recorded. The number of cycles spent in a selected interval of execution is the difference between these two counts. In JasPer, two calls to the new runtime library functions are inserted in the desired locations. The number of cycles is reported between these two calls.
Table 3.6: Cycle counting results for *cats* image in Simulation

<table>
<thead>
<tr>
<th>Encoding type</th>
<th>Unit</th>
<th>Number of cycles</th>
<th>% of total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lossless</td>
<td>cblk</td>
<td>2,395,122,056</td>
<td>52.57%</td>
</tr>
<tr>
<td>5/3 filtering</td>
<td>dwt</td>
<td>405,155,244</td>
<td>8.89%</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>4,556,033,559</td>
<td>100%</td>
</tr>
<tr>
<td>Lossy</td>
<td>cblk</td>
<td>1,819,186,900</td>
<td>30.63%</td>
</tr>
<tr>
<td>9/7 filtering</td>
<td>dwt</td>
<td>1,033,279,228</td>
<td>17.40%</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>5,938,715,821</td>
<td>100%</td>
</tr>
</tbody>
</table>

For the *cats* image, Table 3.6 shows the simulated time spent in the code-block processing step (53%) and the DWT (9%) relative to the total simulated execution time in lossless encoding. Using Amdahl’s Law, the ideal speedups using 2 and 4 threads are 1.44 and 1.86, respectively. In lossy encoding, the fractions for simulated time spent in the code-block processing step and DWT are 31% and 17% relative to the total simulated execution time. Therefore, the ideal speedups using 2 and 4 threads are 1.32 and 1.56, respectively. These speedups are somewhat lower because the code-block processing and DWT steps constitute 48% to the total simulated execution time for lossy encoding rather than 62% of total simulated execution time for lossless encoding. Therefore, the speedup for lossy encoding is smaller. In simulation, the quantization step does not significantly contribute to the total execution time, therefore, its execution is not analyzed.
3.1.5 Hardware Execution Results

On our Quad64 platform, lossless and lossy encoding tests for the entire collection of test images have been conducted. Performance of the JasPer software is assessed by measuring execution time using the Linux `time` command. For each test, three statistics are reported by the `time` command in Table 3.7, namely the `real`, `usr`, and `sys` times. The `real` time is the actual elapsed time for the execution of the entire program. The `usr` contribution is the time used by executing the program code and library code which are outside the operating system kernel. The `sys` contribution is the time consumed by system calls which are invoked by the program. It measures the time spent within the kernel, as opposed to library code which is running in the user space. The `usr` and `sys` times do not count time spent for I/O activities. On an unloaded system, the sum of the `usr` and `sys` values is often approximately equal to the `real` time for executing the program. However, if the sum is significantly less the `real` time, it normally means that the execution involves I/O activities.

Although the Quad64 system is unloaded for all of our experiments, there are still some variations in the execution time due to other activities such as invocation of system processes and servicing of interrupts. In order to compensate for these effects, averages were calculated from multiple executions.

The hardware execution results which are averaged over 20 runs are shown in Table 3.7. The time for lossless encoding is generally better than lossy encoding. In fact, in the inter-component transformation stage, integer coefficients are employed in lossless encoding rather than the floating-point coefficients used in lossy encoding. This difference is one factor that can contribute to the slightly reduced execution time for lossless encoding. Other factors may
Table 3.7: Hardware execution results on Intel Core 2 Quad platform

<table>
<thead>
<tr>
<th>Image file</th>
<th>Dimensions</th>
<th>Size (Mb)</th>
<th>Lossy 9/7 exec. time (sec)</th>
<th>Lossless 5/3 exec. time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats.pnm</td>
<td>3072×2048</td>
<td>6.00</td>
<td>1.98 1.86 0.08</td>
<td>1.80 1.68 0.08</td>
</tr>
<tr>
<td>pcb_small.pnm</td>
<td>1632×1224</td>
<td>5.71</td>
<td>1.88 1.75 0.08</td>
<td>1.65 1.52 0.08</td>
</tr>
<tr>
<td>pcb_large.pnm</td>
<td>3264×2448</td>
<td>9.26</td>
<td>5.75 5.33 0.34</td>
<td>5.57 5.16 0.34</td>
</tr>
<tr>
<td>pcb_tiled.pnm</td>
<td>3264×2448</td>
<td>22.9</td>
<td>8.09 7.61 0.36</td>
<td>7.00 6.52 0.37</td>
</tr>
<tr>
<td>eso0905a_small.pnm</td>
<td>4204×4169</td>
<td>18.77</td>
<td>12.81 11.90 0.79</td>
<td>12.95 12.04 0.79</td>
</tr>
<tr>
<td>eso0905a.pnm</td>
<td>8408×8337</td>
<td>200.6</td>
<td>80.96 61.25 3.60</td>
<td>80.38 62.07 3.42</td>
</tr>
</tbody>
</table>

include the fewer computational steps performed in the lossless 5/3 filtering than the lossy 9/7 filtering.

3.1.6 Loop Iteration Count and Granularity of Parallelism

A parallelizable loop with a large number of iterations usually indicates a greater granularity of parallelism. Such a granularity of parallelism can better offset synchronization overheads, and hence offer the potential to achieve a speedup closer to the ideal. On the other hand, parallelizing a loop with a small number of iterations may have an adverse impact on performance because overheads become more significant if the per-iteration workload is modest.

To explore the granularity issue, we consider the \texttt{jpc\_enc\_enccblks} function in the code-block processing step of JasPer. As shown in Figure 3.6, the \texttt{jpc\_enc\_enccblks} function contains loops nested in five levels. Each loop retrieves its number of iterations from a pointer-based expression. There are three loops in the innermost level, and they all iterate
int jpc_enc_encblks(jpc_enc_t *enc)
{
    for (tcmpt = tile→tcompts; tcmpt != endcomps; ++tcmpt) // Components
    {
        ... for (lvl = tcmpt→rlvls; lvl != endlvls; ++lvl) // Resolution Level
        {
            ... for (band = lvl→bands; band != endbands; ++band) // Bands
            {
                ... for (prcno = 0, prc = band→prcs; prcno < lvl→numprcs; ++prcno, ++prc) // Precincts
                {
                    ... for (cblk = prc→cblks; cblk != endcblks; ++cblk) ... // Code-block
                }
            }
        }
    }
}
...

Figure 3.6: Nested loops in jpc_enc_encblks function

through the number of code-blocks. In order to collect the loop iteration counts at each nesting level, the JasPer software was modified to achieve this task. The iteration counts are reported for the number of components, the number of resolution levels, the number of bands, the number of precincts, and the number of code-blocks during lossless encoding with single-threaded execution.

The loop iteration counts for encoding the cats image are presented in Figure 3.7. The horizontal axis of the graph represents the successive iterations of the loop for the resolution level. For each iteration of this loop, the iteration counts in the more deeply nested loops are shown, namely, in the loops for bands, precincts, and code-blocks. Because the number of components is a constant 1 for the cats image, the corresponding outermost loop is omitted in Figure 3.7. The code-block loop, which is the innermost loop, shows an increase in its number of iterations as the containing resolution-level loop is executed. The count increases
exponentially each time, as shown, along the vertical axis which is in logarithmic scale, and the count eventually reaches a relatively large number. Iteration counts for other loops are small. Therefore, these loops are not individually suitable for parallelization. Because greater granularity is available in the innermost loop, it is the best candidate for parallelization.

The loop iteration counts from the `jpc_enc_enccblks` function have also been assessed for the other test images. The maximum number of iterations occurs in the innermost loop for all test images. The `jpc_enc_enccblks` function is invoked for every image component. Whereas a greyscale image has one component, a typical RGB color image has three components. Three figures similar to Figure 3.7 would therefore be produced for each image
component. Hence, these results are not shown in detail. Instead, the maximum count for the innermost loop is determined for each image.

These maximums are presented in Figure 3.8, and they are proportional to the image size. Ideally, an image with a large number of innermost loop iterations would offer the best performance after the parallelization. However, in practice, other overheads may dictate the overall speedup. For example, encoding of our largest image eso0905a involved considerable disk I/O activities that increased total execution time, as evidenced by the difference in the last row of Table 3.7. Such intervals of time limit the achievable speedup, as they are part of the factor $f_s$ in Amdahl’s Law of Equation 3.2.

### 3.2 Cache Behavior Analysis

In this section, the cache behavior of JasPer is studied closely on real hardware and in simulation. In hardware execution, the cache behavior of lossless encoding is analyzed on our Duo32 system. In simulation, SimpleScalar is used to model a cache hierarchy and to profile the lossless encoding process. The sensitivity of the miss rate to cache parameters in simulation has been investigated and results show that even without associativity, the miss rate is relatively low. In order to obtain more insight into such cache behavior, SimpleScalar is modified to track the memory access patterns.

#### 3.2.1 The Cachegrind Modeling Tool

The Cachegrind tool [Dev] can profile cache behavior of an application program at runtime. It is included in the Valgrind tool suite [Dev]. Cachegrind detects the cache configuration for the
Figure 3.8: Maximum number of code-block loop iterations
underlying system. It can models the L1 and L2 caches that exist on the real hardware to obtain the statistics of cache misses, memory references, and instruction execution for each line of the source code. It is also able to provide the provide aggregated results in the form of the global and function-level counts.

Cacheegrind does not account for all factors affecting performance. These factors include kernel activity, other process activity, as well as virtual-to-physical address mappings. Nonetheless, it provides useful insights for the realistic cache behaviors. Cacheegrind is mainly used in our research for investigating cache behaviors of JasPer and to justify our observations related to cache problems reported in previous work.

3.2.2 Results from Cachegrind

Lossless and lossy encoding of our collection of test images with JasPer has been profiled in Cachegrind on our Duo32 system with its 8-way associative caches. Table 3.8 and Table 3.9 summarize the cache profiling results for lossless and lossy encoding, respectively. The miss rates are determined by comparing the corresponding instruction fetch misses and data read/write misses relative to the total number of instructions and data accesses. Additionally, the L2 miss rate is calculated relative to all data accesses. All encoding experiments show low cache miss rates regardless of the image sizes and resolutions.

As mentioned in the Section 2.8 of Chapter 2, the occurrence of cache misses received great attention in previous work. We tried to reproduce this issue in our experiment, but we did not see considerable cache misses. We suspect that the cache misses issue has been addressed in the latest version of JasPer. It is also possible that the issue is less significant because of the advanced cache systems with larger cache sizes and higher associativity in more recent
In order to further investigate the previously reported issue related to cache misses, we have created a situation where the image width is a power of two. We cropped the `pcb_large` im-
Table 3.10: CacheGrind results for cropped images from lossless encoding

<table>
<thead>
<tr>
<th></th>
<th>pcb_large</th>
<th>eso0905a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2048 × 1024</td>
<td>2176 × 1088</td>
</tr>
<tr>
<td></td>
<td>8192 × 8192</td>
<td>8088 × 8188</td>
</tr>
<tr>
<td>Instr. fetch</td>
<td>4,894,909,783</td>
<td>5,537,275,998</td>
</tr>
<tr>
<td></td>
<td>152,712,148,716</td>
<td>150,474,036,753</td>
</tr>
<tr>
<td>L1 instr. fetch miss</td>
<td>26,148</td>
<td>27,370</td>
</tr>
<tr>
<td></td>
<td>222,729</td>
<td>214,384</td>
</tr>
<tr>
<td>L2 instr. fetch miss</td>
<td>2,813</td>
<td>2,734</td>
</tr>
<tr>
<td></td>
<td>5,396</td>
<td>6,328</td>
</tr>
<tr>
<td>Data Read Access</td>
<td>1,841,071,706</td>
<td>2,084,899,498</td>
</tr>
<tr>
<td></td>
<td>51,220,682,469</td>
<td>50,480,074,784</td>
</tr>
<tr>
<td>L1 data read miss</td>
<td>10,913,664</td>
<td>14,184,142</td>
</tr>
<tr>
<td></td>
<td>498,790,283</td>
<td>518,310,923</td>
</tr>
<tr>
<td>L2 data read miss</td>
<td>5,418,901</td>
<td>3,685,725</td>
</tr>
<tr>
<td></td>
<td>360,438,882</td>
<td>172,898,091</td>
</tr>
<tr>
<td>Data Write Access</td>
<td>674,761,427</td>
<td>765,286,361</td>
</tr>
<tr>
<td></td>
<td>20,963,959,203</td>
<td>20,661,062,549</td>
</tr>
<tr>
<td>L1 data write miss</td>
<td>2,942,471</td>
<td>3,514,632</td>
</tr>
<tr>
<td></td>
<td>127,649,031</td>
<td>125,442,388</td>
</tr>
<tr>
<td>L2 data write miss</td>
<td>2,330,192</td>
<td>1,909,495</td>
</tr>
<tr>
<td></td>
<td>94,070,725</td>
<td>59,974,816</td>
</tr>
<tr>
<td>L1 miss rate</td>
<td>0.55%</td>
<td>0.62%</td>
</tr>
<tr>
<td></td>
<td>0.87%</td>
<td>0.90%</td>
</tr>
<tr>
<td>L2 miss rate</td>
<td>0.31%</td>
<td>0.20%</td>
</tr>
<tr>
<td></td>
<td>0.63%</td>
<td>0.33%</td>
</tr>
</tbody>
</table>

age to obtained 2 images with different dimensions (2048 × 1024 and 2176 × 1088). Profiling results shown in Table 3.10 and Table 3.11 for lossless and lossy encoding, respectively, reveal a low cache miss rate.

To stress the cache capacity even more, we cropped our largest image eso0905a to obtain two images with different dimensions (8192 × 8192 and 8088 × 8188) and profiled their lossless and lossy encoding processes. Profiling results in Table 3.10 and Table 3.11 show increased miss rates due to the considerable image size. Nonetheless, the miss rates are still low.
### Table 3.11: Cachegrind results for cropped images from lossy encoding

<table>
<thead>
<tr>
<th></th>
<th>pcb_large</th>
<th>eso0905a</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2048 × 1024</td>
<td>2176 × 1088</td>
<td>8192 × 1024</td>
<td>8088 × 1088</td>
<td></td>
</tr>
<tr>
<td>Instr. fetch</td>
<td>4,590,354,433</td>
<td>5,228,051,092</td>
<td>152,712,148,735</td>
<td>150,474,036,772</td>
<td></td>
</tr>
<tr>
<td>L1 instr. fetch miss</td>
<td>10,859</td>
<td>11,523</td>
<td>222,729</td>
<td>214,384</td>
<td></td>
</tr>
<tr>
<td>L2 instr. fetch miss</td>
<td>2,820</td>
<td>2,811</td>
<td>5,419</td>
<td>6,353</td>
<td></td>
</tr>
<tr>
<td>Data Read Access</td>
<td>1,515,552,544</td>
<td>1,727,232,145</td>
<td>51,220,682,469</td>
<td>50,480,074,784</td>
<td></td>
</tr>
<tr>
<td>L1 data read miss</td>
<td>14,214,530</td>
<td>17,390,662</td>
<td>498,694,431</td>
<td>517,830,874</td>
<td></td>
</tr>
<tr>
<td>L2 data read miss</td>
<td>8,077,136</td>
<td>4,279,307</td>
<td>360,438,139</td>
<td>172,903,181</td>
<td></td>
</tr>
<tr>
<td>Data Write Access</td>
<td>612,092,950</td>
<td>698,706,880</td>
<td>20,963,959,206</td>
<td>20,661,062,552</td>
<td></td>
</tr>
<tr>
<td>L1 data write miss</td>
<td>3,384,326</td>
<td>4,042,973</td>
<td>127,679,255</td>
<td>125,380,553</td>
<td></td>
</tr>
<tr>
<td>L2 data write miss</td>
<td>2,321,182</td>
<td>2,090,542</td>
<td>94,070,830</td>
<td>59,974,759</td>
<td></td>
</tr>
<tr>
<td>L1 miss rate</td>
<td>0.83%</td>
<td>0.88%</td>
<td>0.87%</td>
<td>0.90%</td>
<td></td>
</tr>
<tr>
<td>L2 miss rate</td>
<td>0.49%</td>
<td>0.26%</td>
<td>0.63%</td>
<td>0.33%</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2.3 Results from SimpleScalar

SimpleScalar has also been used in simulation for lossless encoding to model a cache hierarchy. Profiling results for processing the `cats` image shows that there are a larger number of instructions using SimpleScalar than for the Cachegrind profile. The reason is that SimpleScalar uses a RISC-style instruction set, which leads to having more instructions than the Intel CISC-style instruction set. Moreover, the simpler RISC instruction set indeed has more instructions, but somewhat fewer data accesses due to having a larger register set.

In SimpleScalar, various data cache organizations have been modeled using the original
sim-cache simulator which supports associativity and the alternative sim-mpcache simulator from the multiprocessor version of SimpleScalar which only supports direct mapping.

In fact, we aim to investigate the sensitivity of miss rate to cache parameters, primarily for the L2 cache, as misses there have higher cost to access the main memory. By using the original sim-cache simulator, an 8-way associative organization comparable to how the Intel E6320 processor of our Duo32 platform is configured. Then, the L2 size is reduced while keeping the associativity and L1 size unchanged. Consequently, the L2 miss rate relative to total data accesses ranged from 0.30% for 4-Mbyte L2, and up to 0.70% for 32-Kbyte L2. Despite a somewhat different total number of instructions and data accesses, the L2 miss rate from simulation with a RISC-style instruction set is comparable to that of a CISC-style instruction set.

When conducting experiments using the sim-mpcache simulator, all caches are direct-mapped. The L1 and L2 capacities, at first, are set to sizes that are comparable to the Intel Core 2 Duo E6320 processor of our Duo32 platform. Then, the size of the L2 cache is reduced while keeping the size of L1 unchanged. As a result, the L2 miss rate relative to the total data accesses ranged from 0.93% for 4-Mbyte L2, and up to 2.1% for 32-Kbyte L2. We have also used the original sim-cache simulator to vary associativity for both L1 and L2 caches with the same sizes as in our Duo32 platform. It turns out that the L2 miss rate ranged from 0.30% for 8-way associativity, and up to 0.93% for 1-way associativity (direct-mapped). Misses and miss rates from sim-cache and sim-mpcache are comparable for this direct-mapped organization.
3.3 Memory Access Tracking in SimpleScalar

In order to obtain more insights on the low L2 miss rates even without cache associativity, we have initially modified the SimpleScalar simulator to track accesses made within adjacent segments of the overall data memory space (excluding the stack) during consecutive intervals of the overall execution time. At the end of simulated execution, a binary indication is provided for each memory segment and time interval as to whether or not at least one reference was made.

The initial memory access tracking results in Figure 3.9 for lossless encoding of the cats image show that the data memory allocation grows to more than 90 Mbytes to process a 6-Mbyte image file. Each marker on the graph represents a 1.15-Mbyte memory space along the vertical axis within a time interval of $2^{23}$ cycles on the horizontal axis. The access distribution pattern on the graph reveals that most of the memory is not accessed during any given interval of time. Instead, the data accesses are concentrated in certain regions of the memory.

The two-dimensional graph of memory access behavior tracking in Figure 3.9 only shows whether a particular region in the memory is accessed during a particular time interval. The number of references that has been made to that region is not shown. To better appreciate the memory access pattern, SimpleScalar has been further modified to count the number of data references that have been made to the specific memory region during an interval of time. The number of data references for load and store instructions are reported separately.

The memory access tracking results for lossless encoding the cats image is presented in Figure 3.10 as a three-dimensional mesh where the height of the graph indicates the number of data references to a certain region during a particular time interval of the execution. For the 2.1
each point covers $2^{23}$ cycles in time along the horizontal axis, and 1.15 Mbytes of memory space along the vertical axis

Figure 3.9: Data memory access pattern in 2-D
billion data references made over the course of simulated time along the horizontal axis, more than 90 Mbytes of memory are allocated to process the 6-Mbyte cats image.

Figure 3.10 is partitioned into three regions along the horizontal axis. They are labeled as pre-processing, memory-allocation, and post-processing which will be used in discussing the memory access pattern figures. A significant amount of preparatory work is done in the pre-processing region before encoding the cats image, as observed in Figure 3.10. There are many load references in neighboring memory regions. In the memory-allocation region, the amount of memory grows substantially, but the number of load data references is less significant. In any given time interval, only a subset of all of the memory is accessed. In the post-processing region, a significant number of memory accesses are spread along the memory-extent axis. A large number of clustered memory accesses are observed in pre-processing and memory-allocation regions. These patterns suggest potentially higher data reuse within the regions that are accessed, which helps to explain the low cache miss rate.

The number of load data references shown in Figure 3.10 is larger than the number of store data references in Figure 3.11. Figure 3.12 presents the total amount of data references that are made during encoding; it is a combined view of Figure 3.10 and Figure 3.11.

3.4 Summary

In this chapter, we have profiled the JasPer encoder on a single processor for insight into its behavior and performance. We have used hardware-based profiling as in previous work, but we have gone further to also use simulation for additional detailed profiling that is not affected by various sources of overhead. Moreover, we have analyzed both lossy and lossless
Figure 3.10: Data memory access pattern for the load references
Figure 3.11: Data memory access pattern for the store references
Figure 3.12: Data memory access pattern for the total number of data references
encoding behavior in the JasPer software whereas previous work has largely considered lossy encoding. Of particular interest are the portions of the JasPer software that are parallelizable; these portions have been identified based on knowledge of the JPEG-2000 standard and insights provided by previous work. Potential speedup has been estimated from the profiling results. The workload in parallelizable portions varies based on the number of loop iterations. This variation in the workload affects actual parallel execution efficiency. A larger number of loop iterations usually implies a greater granularity of parallelism, which can be utilized to obtain a speedup that is closer to the ideal. Moreover, the cache behavior of the JasPer encoding program was analyzed for both simulated and hardware execution. It was confirmed that even without cache associativity, the L2 miss rate is relatively low. Finally, the memory access patterns have been studied closely in order to obtain more insights into the low L2 cache miss rates. A large number of clustered memory accesses are observed which suggest potentially higher data reuse that helps to explain the low cache miss rate.
Chapter 4

Single-Thread Program Transformations

This chapter describes code transformations that have been applied to the original JasPer source code with a particular emphasis on loops. Based on the analysis in Chapter 3 that considered functions containing parallelizable loops, the modifications that are applied include loop index transformation, loop body transformation, and loop fusion.

Loops using a pointer as the index variable are transformed to have an integer index variable. Integer index variables are generally easier to be interpreted by the programmer and less error prone in programming. Furthermore, it also assists the loop parallelization because the distinct subsets of the loop iterations are required to be distributed to different processors for parallel execution. Loop bodies are then transformed to make image data array accesses dependent on the value of the loop index variable, which can facilitate the partitioning of loop iterations. The calculation of loop bounds is necessary for partitioning the loop iterations. For this purpose, a relation between array subscript expressions and the loop index variable ensures that the image data array can be indexed correctly by each subset of loop iterations.

In addition, a sequence of three parallelizable loops are identified and fused in order to
reduce synchronization overhead. All these code transformations are aimed to facilitate the
transformation for parallelizing JasPer. As confirmed experimentally, the transformations de-
scribed in this chapter do not have any adverse effects on performance.

4.1 Loop Index Transformation

This section describes the loop index variable transformation for the parallelizable loops in the
original JasPer source code, where the original loop headers use pointers as index variables.
When partitioning contiguous blocks of iterations into subsets, the calculation of loop bounds
is involved. Using integer variables to index the array elements in such a computation is ob-
viously easier. It also makes the source code easier to read and maintain. Because the data
organization consists of contiguous arrays, it is possible to use integer variables for the loop
indexes. Parallelizing a loop requires the distribution of contiguous blocks of iterations to dif-
ferent processors for parallel execution. Using an integer variable makes loop parallelization
straightforward, either explicitly or automatically using directives. The loops of interest are
therefore rewritten to use integer index variables.

In the code-block processing step, many of the original loops use pointers as index vari-
ables, as shown in Figure 4.1. The loops of particular interest are the parallelizable innermost
loops, as shown in Figure 3.6. After transforming the loop index to an integer variable in its
loop header, the pointer value needed in the loop body is easily obtained in the manner shown
in Figure 4.2. In fact, the compiler resolves the array subscripting during compilation and op-
timizes the generated assembly code with \texttt{-O2} optimization level as discussed in Chapter 3.
The generated assembly code is equivalent for both pointer and integer indexed loops.
for (cblk = prc->cblks; cblk != endcblks; ++cblk)
{
    ...
}

Figure 4.1: Original loop header

for (i = 0; i < prc->numcblks; i++)
{
    cblk = &prc->cblks[i]; //retrieve pointer value
    ...
}

Figure 4.2: Transformed loop header

4.2 Loop Body Transformation

The original source code for DWT uses pointers in its loop bodies to retrieve individual image data items from a contiguous array. The base address of the array is stored in a pointer variable, and the array is traversed by advancing the pointer. The loop index variable controls the maximum number of iterations for the loop. However, it has nothing to do with the statements inside the loop. This form of array access using a pointer variable presents challenges in partitioning loop iterations. In fact, the partitioning of loop iterations involves retrieving the corresponding array elements. Because loop bounds for the subset of iterations cannot be determined solely by the loop index variable, the elements in the image data array cannot be retrieved in the desired manner.

For loop parallelization, parallel functions are typically created for each loop that is to executed by multiple processors. These functions accept arguments to enable distribution of distinct subsets of iteration spaces to different processors. These arguments include upper
and lower bounds specific to each processor which are used to partition the loop iterations. Therefore, the array accesses in the DWT loop bodies must solely depend on the loop index variable.

DWT in JasPer distinguishes between lossless and lossy encoding operations. For lossless compression, the function \texttt{jpc\_ft\_analyze} is employed for the discrete wavelet transform. Both vertical and horizontal filtering loops in the original source code shown in Figure 4.3 use a pointer variable, namely \texttt{startptr}, which points to the image data array. At the end of each iteration, the \texttt{startptr} pointer is incremented to where the next block of elements starts. However, the loop index variable is not involved in the array accesses; it only controls the maximum number of loop iterations.

In order to establish a relation between the loop index variable and the array accesses in the loop body, these loops are transformed as shown in Figure 4.4. The loop index variable is used in the loop body as an offset for the base pointer \texttt{startptr} to retrieve the elements from the image data array.

For lossy compression, the function \texttt{jpc\_ns\_analyze} is used, which is similar to the function \texttt{jpc\_ft\_analyze} used in lossless encoding. The original source code shown in Figure 4.5 presents a similar way for referencing an image data array as described above. After the transformation, the array accesses in the loop body depend on the loop index variable, as shown in Figure 4.6.
int jpc_ft_analyze(jpc_fix_t *a, int xstart, int ystart,
                int width, int height, int stride)
{
    ...

    startptr = &a[0];

    // vertical filtering loop
    for (i = 0; i < maxcols; i += JPC_QMFB_COLGRPSIZE)
    {
        jpc_qmfb_split_colgrp(startptr, numrows, stride, rowparity);
        jpc_ft_fwdlift_colgrp(startptr, numrows, stride, rowparity);
        startptr += JPC_QMFB_COLGRPSIZE;
    }
    ...

    startptr = &a[0];

    // horizontal filtering loop
    for (i = 0; i < numrows; ++i)
    {
        jpc_qmfb_split_row(startptr, numcols, colparity);
        jpc_ft_fwdlift_row(startptr, numcols, colparity);
        startptr += stride;
    }
    ...
}

Figure 4.3: Original loop bodies for jpc_ft_analyze
int jpc_ft_analyze(jpc_fix_t *a, int xstart, int ystart,
                    int width, int height, int stride)
{
    ...  

    startptr = &a[0];
    for (i = 0; i < maxcols; i += JPC_QMFB_COLGRPSIZE)
        {  
            //use the value in the loop index (i) as an offset
            jpc_qmfb_split_colgrp((startptr + i), numrows,  
                                stride, rowparity);
            jpc_ft_fwdlift_colgrp((startptr + i), numrows,  
                                stride, rowparity);
        }
    ...

    startptr = &a[0];
    for (i = 0; i < numrows; ++i)
        {  
            //use the value in the loop index (i) as an offset
            jpc_qmfb_split_row((startptr + i * stride),  
                               numcols, colparity);
            jpc_ft_fwdlift_row((startptr + i * stride),  
                               numcols, colparity);
        }
    ...

}

Figure 4.4: Transformed loop bodies for jpc_ft_analyze
int jpc_ns_analyze(jpc_fix_t *a, int xstart, int ystart,
      int width, int height, int stride)
{
  ...

    startptr = &a[0];

    //vertical filtering loop
    for (i = 0; i < maxcols; i += JPC_QMFB_COLGRPSIZE)
    {
        jpc_qmfb_split_colgrp(startptr, numrows, stride, rowparity);
        jpc_ns_fwdlift_colgrp(startptr, numrows, stride, rowparity);
        startptr += JPC_QMFB_COLGRPSIZE;
    }
    ...

    startptr = &a[0];

    //horizontal filtering loop
    for (i = 0; i < numrows; ++i)
    {
        jpc_qmfb_split_row(startptr, numcols, colparity);
        jpc_ns_fwdlift_row(startptr, numcols, colparity);
        startptr += stride;
    }
    ...
}

Figure 4.5: Original loop bodies for jpc\_ns\_analyze
```c
int jpc_ns_analyze(jpc_fix_t *a, int xstart, int ystart,
                   int width, int height, int stride)
{
    ...

    startptr = &a[0];
    for (i = 0; i < maxcols; i += JPC_QMFB_COLGRPSIZE)
    {
        //use the value in the loop index (i) as an offset
        jpc_qmfb_split_colgrp(startptr + i, numrows,
                              stride, rowparity);
        jpc_ns_fwdlift_colgrp(startptr + i, numrows,
                              stride, rowparity);
    }
    ...

    startptr = &a[0];
    for (i = 0; i < numrows; ++i)
    {
        //use the value in the loop index (i) as an offset
        jpc_qmfb_split_row((startptr + i * stride),
                           numcols, colparity);
        jpc_ns_fwdlift_row((startptr + i * stride),
                           numcols, colparity);
    }
    ...
}
```

Figure 4.6: Transformed loop bodies for `jpc_ns_analyze`
4.3 Loop Fusion

Loop fusion [KM94] is a well-known transformation that involves merging loops at the same nesting level that have identical bounds and iteration counts. The loop bodies are combined with a single loop header. Doing so can reduce loop overhead and the number of memory accesses, and improve data locality in the cache. The increased granularity from fusing parallelizable loops also reduces synchronization overhead.

In the code-block processing step, loop fusion involves aggregating a sequence of three loops in the function `jpc_enc_encblk`, as shown in Figure 4.7, for the sake of convenience and efficiency. The three loops in question are parallelizable. Loop fusion for these three loops is considered because the overhead from separate loop parallelization is likely higher. The workload is small in each iteration of the first two loops, but the last loop has more work per iteration. Fortunately, the work done in each of the loops is independent, so it is feasible to fuse them and therefore increase parallel efficiency by requiring only one synchronization point instead of three.

The transformation of loop index variable is applied before the loop fusion, as discussed in Section 4.1. The pointer value needed in the loop body is easily obtained from the resulting integer loop index variable in the manner shown in Figure 4.8. The fused loop in Figure 4.8 offers increased granularity, which reduces synchronization overhead.

4.3.1 CacheGrind Profiling for the Loop Fusion

In order to ensure that the loop fusion does not affect JasPer execution, we have compared the total number of instructions executed with and without the loop fusion. Both greyscale and
for (cblk = prc->cblks; cblk != endcblks; ++cblk)
{
    cblk->numbps = JAS_MAX(jpc_firstone(mx) + 1
                        - JPC_NUMEXTRABITS, 0);
}

for (cblk = prc->cblks; cblk != endcblks; ++cblk)
{
    cblk->numimsbs = band->numbps - cblk->numbps;
}

for (cblk = prc->cblks; cblk != endcblks; ++cblk)
{
    if (jpc_enc_enc_cblk(enc, cblk->stream, tcmpt, band, cblk))
    {
        return -1;
    }
}

Figure 4.7: Original loops with pointer index variables

for (i = 0; i < prc->numcblks; i++)
{
    cblk = &prc->cblks[i]; //retrieve pointer value
    cblk->numbps = JAS_MAX(jpc_firstone(mx) + 1
                        - JPC_NUMEXTRABITS, 0);
    cblk->numimsbs = band->numbps - cblk->numbps;
    if (jpc_enc_enc_cblk(enc, cblk->stream, tcmpt, band, cblk))
    {
        return -1;
    }
}

Figure 4.8: Fused loop with integer index variable
Table 4.1: Comparison of total instruction fetches before and after loop fusion

<table>
<thead>
<tr>
<th></th>
<th>cats.pnm</th>
<th>pcb_small.pnm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original JasPer</td>
<td>4,164,066,836</td>
<td>5,622,923,082</td>
</tr>
<tr>
<td>Original JasPer with fused loops</td>
<td>4,164,068,248</td>
<td>5,622,924,050</td>
</tr>
<tr>
<td>Extra instructions to original JasPer</td>
<td>1,412</td>
<td>968</td>
</tr>
</tbody>
</table>

Color images are used for these tests.

Lossless encoding of the cats and pcb_small images has been profiled with Cachegrind on the Core32 platform discussed in Chapter 2. The profiling results in Table 4.1 reveal that the transformation does not dramatically increase the total number of instructions. There are 1412 more instructions executed for the cats image, and 968 more instructions executed for the pcb_small image. With over 4 billion instruction references during the encoding process, the additional instructions are negligible.

### 4.4 Summary

All of the code transformations that have been discussed in this chapter are largely intended to facilitate the parallelization for JasPer, either explicitly or automatically by a compiler. Parallelizable loops have undergone loop index transformation, loop body transformation, and loop fusion, which together will aid in the loop bounds calculation, loop iteration partitioning, and synchronization overhead reduction. These loop transformations do not have any adverse effects on encoding performance based on our experimental results. Rather, they increase the granularity of parallelism and reduce synchronization overhead.

Chapter 5 will describe the code parallelization for JasPer following the code transforma-
tions that have been done in this chapter. With these transformations, distinct subsets of the loop iterations can be easily distributed to different processors for parallel execution.
Chapter 5

Parallelization and Execution Analysis

This chapter describes the parallelization of the serial JasPer image encoder and presents results obtained from simulation and on real hardware. After applying transformations involving the loops of interest as discussed previously in Chapter 4, JasPer is parallelized explicitly for simulated execution and automatically for execution on real hardware.

In simulation, a version of SimpleScalar with multiprocessor enhancements is used for parallelizing the image compression code of JasPer. The parallelizable loops are explicitly transformed in order to assess the potential for speedup. On hardware, general-purpose systems with multi-core processor chips are used to examine the performance of the parallelized JasPer software. We rely on the compiler for our Quad64 hardware platform discussed in Chapter 2 to automatically transform the code, driven by directives that label the parallelizable loops.

On hardware, caches and various sources of overhead dictate the actual speedup. In contrast, the simulation results are useful to set appropriate expectations. Speedups are obtained from the experimental results and they are analyzed in terms of the profiling results and workload distribution.
Generated images from multithreaded execution are compared to the original image using the analytical software tool provided with JasPer and a comparable custom image comparison tool developed for this research. The comparison results are repeatable, which confirms the correctness of the parallelization despite the nondeterministic parallel execution on real hardware.

5.1 Overview

Multiprocessors offer potential performance enhancement through parallel execution. When a task contains a considerable amount of parallelism, the workload can be distributed to different processors for parallel execution.

The degree of parallelism in a program can be identified through execution profiling. Loops containing independent iterations are desirable for parallelization due to the ease of transformation. A straightforward parallelization of parallelizable loops involves distributing contiguous blocks of iterations to different processors, with synchronization to ensure loop completion so that the original program semantics are respected. The source code can then be transformed accordingly for the target environment to enable parallel execution.

As discussed in Chapter 3, the JasPer encoding process has been profiled by using the gprof profiler on our Quad64 system. The profiling results in Section 3.1.2 show that the parallelizable portions constitute 72% and 71% of total execution time for lossless and lossy encoding, respectively. Through further experiments discussed in Chapter 3, we observed that the amount of independent loop iterations in the code-block processing and DWT steps are significant. The absence of dependences allows the workload to be distributed to different
processors for parallel execution. In both the code-block processing and DWT steps, the computational workload is defined in loops whose iterations are independent.

The code transformation for parallelization must be done explicitly for simulated execution due to the GNU C compiler for SimpleScalar not supporting directives for automatic parallelization. The transformation can, however, be done automatically for execution on real hardware because the GNU C compiler for our hardware platform is able to generate the necessary code to perform the JasPer encoding process in parallel.

5.2 Parallelization for Simulated Execution

This section describes the parallelization techniques used to transform the JasPer image encoder for simulated execution. With all relevant loops using integer index variables and three loops fused in the manner described in the Chapter 4, parallelizable code is transformed explicitly for simulated parallel execution to assess potential speedup. A multiprocessor version of SimpleScalar is used [Man01]. This version provides a functional multiprocessor model with single-cycle latency for all operations, and a run-time library to support thread creation and synchronization for the application program. Explicit code transformation is employed because the GNU C compiler for SimpleScalar does not support directives for automatic parallelization.

The explicit transformation involves introducing new C functions to perform the computation expressed in the original parallelizable loops. These functions accept arguments to enable distribution of distinct subsets of iteration spaces to different processors. At runtime, the main thread creates $N - 1$ additional threads at the beginning of execution. These threads wait on a
barrier. Meanwhile, the main thread performs serial computation, until it reaches the original position of the first parallel loop. At this point, the main thread prepares information for the other threads to indicate which loop function to invoke and the associated iteration space. It then causes the waiting threads to exit the barrier. All $N$ threads, including the main thread, concurrently call the specified loop function to complete their respective iterations. The threads synchronize to ensure loop completion, and then the main thread continues with sequential execution while the other threads wait again on a barrier for their next invocation.

From the profiling results obtained in Section 3.1.4, parallelizable loops are exploited in the code-block processing and DWT steps for further performance enhancement. The code-block processing step is shared by both the lossless and lossy encoding algorithms. The DWT encoding algorithm, however, is distinguished by what type of filtering is used. For lossless encoding, 5/3 filtering with reversible integer coefficient transformation is employed in the $jpc_{ft\_analyze}$ function. For lossy encoding, 9/7 filtering with irreversible floating-point coefficient transformation is employed in the $jpc_{ns\_analyze}$ function.

Based on the explicit code transformation strategy discussed above, the body of each parallelizable loop is transformed to a function with arguments. These arguments are used to enable distribution of distinct subsets of iteration spaces to different processors. Thread management and synchronization are done as needed to ensure that the original semantics of the program are respected.

5.2.1 Speedup and Load Balance in Simulation

This section presents the speedup results obtained from simulated parallel execution of JasPer. As one of the advantages in the simulation environment, various overheads introduced by real
hardware are eliminated. Therefore, the potential speedup of the parallelization can be assessed. Also in this section, the parallel execution phases are analyzed and characterized in terms of load balance. The in-depth analysis is provided only for the cats image. Nonetheless, others images are tested and the similar behaviors are observed for their speedup and load balance. Our largest image, however, presents a challenge in the simulated execution because its image size causes the allocated storage to exceed the capacity of the simulator.

We have simulated the parallel execution of the explicitly transformed code described in the preceding section using the multiprocessor version of SimpleScalar. SimpleScalar uses native host code execution to handle low-level operating system service requests such as input/output and memory allocation from the simulated application. System time is therefore not measured. But, simulated user time in library calls is included in the profile results. Speedups have been calculated from the simulated execution times in cycles.

For lossless encoding of the cats image, the simulated profiling results shown in Section 3.1.4 reveal that the code-block processing and DWT steps consume 53% and 9% of the total execution time, respectively. These percentages are calculated out of 4.5 billion simulated instructions with idealized single-cycle latency. Consequently, the upper bound on speedup is $1/(1-0.62) = 2.4$ and the expected speedups for 2, 4, and 8 processors are 1.4, 1.9, and 2.2, respectively. The corresponding experimental speedups are 1.4, 1.8, and 2.1, which are calculated from the ideal execution times in cycles shown in Table 5.1.

Similarly, for lossy encoding of cats image, the simulated profiling results indicate that the code-block processing and DWT steps consume 31% and 17% of the total execution time, respectively, out of 5.9 billion simulated instructions. The increased total number of instructions coincides with our expectation because lossy encoding takes longer to execute on real
hardware for the same image than lossless encoding. Furthermore, the lossy 9/7 lifting algorithm for DWT performs more computations than the lossless 5/3 lifting algorithm. Consequently, the upper bound on speedup is \( \frac{1}{1-0.48} = 2.1 \) and the expected speedups for 2, 4, and 8 processors are 1.3, 1.6, and 1.7, respectively. The corresponding experimental speedups are 1.3, 1.5, and 1.7, which are calculated from the ideal execution times in cycles shown in Table 5.2.

Table 5.3 summarizes the speedup results for lossless and lossy encoding of the cats image in simulation. Lossless encoding has somewhat better performance than the lossy encoding in compressing the same image. The expected and measured speedups are close to each other because of the ideal single-cycle latency for simulated instructions in SimpleScalar.

The distribution of the workload can also affect the performance of a parallel program. A balanced workload is essential for the efficiency of a parallel execution with good utilization of all computing resources. The independent loop iterations are able to be distributed evenly across the modest number of processors used in the parallel phases of the computation because the granularity of the parallelizable loops is good. Indeed, the parallelization has been efficiently done in those phases, at least in simulation.

Figure 5.1 and Figure 5.2 show the distribution of the parallel workload across 2, 4, and 8 threads for lossless and lossy compression of the cats image, respectively. The main thread (thread 0) performs all serial computations in each case, hence, it has a much larger number of simulated instructions. In both figures, the number of simulated instructions for 8-thread execution are not exactly identical for each of parallel threads. Other parallel executions are also characterized by this somewhat imperfect load distribution. Nonetheless, these parallel threads possess a reasonably balanced workload which indicates a generally efficient parallel execu-
Table 5.1: Simulated parallel execution results (in cycles) for lossless encoding of cats image

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>total</strong></td>
<td>4556103975</td>
<td>3156563161</td>
<td>2772312976</td>
<td>2505408350</td>
<td>2368270602</td>
<td>2277317745</td>
<td>2265445651</td>
<td>2153525837</td>
</tr>
<tr>
<td><strong>pid 0</strong></td>
<td>4556103975</td>
<td>3104428061</td>
<td>2568039732</td>
<td>2313396440</td>
<td>2170582971</td>
<td>2111994330</td>
<td>2001191785</td>
<td>1992245857</td>
</tr>
<tr>
<td><strong>pid 1</strong></td>
<td>-</td>
<td>1451736490</td>
<td>1021356772</td>
<td>773095754</td>
<td>559872083</td>
<td>455959154</td>
<td>378344148</td>
<td>315452205</td>
</tr>
<tr>
<td><strong>pid 2</strong></td>
<td>-</td>
<td>-</td>
<td>966791640</td>
<td>732840855</td>
<td>635997867</td>
<td>523868785</td>
<td>450562168</td>
<td>384745804</td>
</tr>
<tr>
<td><strong>pid 3</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>736894420</td>
<td>573336669</td>
<td>497528662</td>
<td>395387984</td>
<td>378367561</td>
</tr>
<tr>
<td><strong>pid 4</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>616453176</td>
<td>479252724</td>
<td>378855683</td>
<td>365824974</td>
</tr>
<tr>
<td><strong>pid 5</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>487637686</td>
<td>433435358</td>
<td>361358410</td>
</tr>
<tr>
<td><strong>pid 6</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>518496368</td>
<td>364383192</td>
</tr>
<tr>
<td><strong>pid 7</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>393922896</td>
</tr>
</tbody>
</table>

Figure 5.1: The distribution of parallel workload for lossless encoding of cats image

Moreover, the relatively balanced workload also explains the good agreement between the expected and experimental speedup results in simulation.
Table 5.2: Simulated parallel execution results (in cycles) for lossy encoding of *cats* image

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5938807618</td>
<td>4521771799</td>
<td>4126893259</td>
<td>3861904343</td>
<td>3737917661</td>
<td>3620618362</td>
<td>3508259206</td>
<td>3434161664</td>
</tr>
<tr>
<td>pid 0</td>
<td>5938807618</td>
<td>4455414772</td>
<td>3905757147</td>
<td>3640794467</td>
<td>3502093220</td>
<td>3434161664</td>
<td>3335978890</td>
<td>3318146249</td>
</tr>
<tr>
<td>pid 1</td>
<td>-</td>
<td>1483451718</td>
<td>1039502391</td>
<td>790652075</td>
<td>573388221</td>
<td>471323424</td>
<td>374054355</td>
<td>315624911</td>
</tr>
<tr>
<td>pid 2</td>
<td>-</td>
<td>-</td>
<td>993638918</td>
<td>752513757</td>
<td>643122420</td>
<td>532024575</td>
<td>452868945</td>
<td>392595801</td>
</tr>
<tr>
<td>pid 3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>754982397</td>
<td>586201546</td>
<td>507550815</td>
<td>411231976</td>
<td>386101629</td>
</tr>
<tr>
<td>pid 4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>634130801</td>
<td>499293880</td>
<td>397403693</td>
<td>376130839</td>
</tr>
<tr>
<td>pid 5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>494587195</td>
<td>432894309</td>
<td>369422244</td>
</tr>
<tr>
<td>pid 6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>534544915</td>
<td>371999366</td>
</tr>
<tr>
<td>pid 7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>408995723</td>
</tr>
</tbody>
</table>

Figure 5.2: The distribution of parallel workload for lossy encoding of *cats* image
Table 5.3: Speedup results for encoding *cats* image in simulation

<table>
<thead>
<tr>
<th></th>
<th>Number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Lossless</td>
<td>Expected</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
</tr>
<tr>
<td>Lossy</td>
<td>Expected</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
</tr>
</tbody>
</table>

5.2.2 Tracking Memory Access Behavior for Multithreaded Execution

Chapter 3 examined the memory access behavior for single-threaded execution of lossless encoding of the *cats* image. For multithreaded execution, it is also useful to inspect the memory access behaviors on per-thread basis. Therefore, SimpleScalar has been further modified from what was described in Chapter 3 to monitor the memory access patterns for the parallel execution. The number of data references for load and store operations are reported for each thread.

Lossless encoding of the *cats* image with parallelized JasPer has been conducted on four simulated processors in SimpleScalar. Figure 5.3 to Figure 5.6 show the load data references for thread 0 to 3 respectively. The number of data references is indicated by the vertical axis. For the 2.1 billion data references that extend along the horizontal axis, more than 90 Mbytes of memory are allocated to process the 6-Mbyte *cats* image.

Based on the nature of the image compression algorithm, Figure 5.3 is partitioned into three regions along the horizontal axis. They are labeled as pre-processing, memory-allocation, and post-processing, which will be used in discussing the memory access pattern figures.
In the pre-processing region, the main thread (thread 0) performs a significant amount of preparatory work for encoding the *cats* image, as observed in Figure 5.3. There are many load references in neighboring memory regions. The number of load data accesses are considerable, but the memory consumption is low in the pre-processing region. The memory grows substantially in the memory-allocation region, but the number of load data references is less significant. In any given time interval, only a subset of all of the memory is accessed. Other parallel threads (threads 1 to 3) show relatively similar memory access patterns in the memory-allocation region. In the post-processing region, many memory accesses are flatly spread along the memory-extent axis.

Likewise, for the store data references shown in Figure 5.7 to Figure 5.10, similar memory access patterns are observed. However, there are fewer store references than the load references. Furthermore, no store references are observed in the pre-processing and post-processing regions for parallel threads (threads 1 to 3).

The total number of simulated data references are shown in Figure 5.11 to Figure 5.14, which combine the load and store reference graphs correspondingly. They show similar memory access patterns as described for the load data reference graphs. The main thread (thread 0) performs all serial computations in each case. Hence, it has a much larger number of simulated instructions and simulated data accesses.

### 5.2.3 Image Comparison Results for Multithreaded Simulated Execution

Generated images from simulated multithreaded execution in SimpleScalar are compared to the original image using the analytical software tool provided with JasPer. We ensure that the outputs meet the generally accepted criterion of $\text{PSNR} > 30 \text{ dB}$ for a good image reconstruction.
Figure 5.3: Data memory access pattern for load instructions (thread 0)
Figure 5.4: Data memory access pattern for load instructions (thread 1)

Figure 5.5: Data memory access pattern for load instructions (thread 2)
CHAPTER 5. PARALLELIZATION AND EXECUTION ANALYSIS

Figure 5.6: Data memory access pattern for load instructions (thread 3)

Figure 5.7: Data memory access pattern for store instructions (thread 0)
Figure 5.8: Data memory access pattern for store instructions (thread 1)

Figure 5.9: Data memory access pattern for store instructions (thread 2)
Figure 5.10: Data memory access pattern for store instructions (thread 3)

Figure 5.11: Data memory access pattern for all instructions (thread 0)
Figure 5.12: Data memory access pattern for all instructions (thread 1)

Figure 5.13: Data memory access pattern for all instructions (thread 2)
The comparison results for lossless encoded images that have been generated from the parallelized JasPer in SimpleScalar all have PSNR$=\infty$ and MSE$=0$, regardless of the number of threads that are used. For the lossy encoded images, the comparison results show PSNR$=49.81$ dB and MSE$=0.68$, regardless of the number of threads that are used. The PSNR values for lossy encoded images are well beyond the PSNR$>30$ dB criterion for a good image reconstruction.

### 5.3 Parallelization for Hardware

With all relevant loops using integer index variables and three loops fused in the manner described in Chapter 4, JasPer is transformed for parallelization on our Quad64 platform.
OpenMP parallel directives are used to direct the compiler to automatically transform the parallelizable loops in JasPer. On our Quad64 system, the gcc compiler is able to generate the necessary code to execute the encoding process in parallel. From the experimental results obtained from the multithreaded execution, speedups are calculated and analyzed. The image comparison is also done in order to ensure the quality of the generated images from the multithreaded execution.

5.3.1 OpenMP Application Programming Interface

OpenMP [Ope11] is a standard for parallel programming on shared-memory multiprocessor systems. It supports loop-level parallel programming as well as parallel regions and sections. OpenMP directives offer a higher level of abstraction that hides the details of creating a multithreaded parallel application [FAQ03].

OpenMP provides relatively straightforward techniques to create new parallel applications, and it also permits the incremental development of parallel programs from existing sequential applications. Using OpenMP, it is possible to convert a sequential application into a parallel application easily in a stepwise fashion without significantly changing the structure of the existing program. Parallel application developers can use OpenMP programming language constructs to express parallelism, work sharing, and synchronization.

The automatic parallelization of JasPer with OpenMP relies on the programmer annotating the code with directives that are recognized by a compiler. Our Quad64 hardware platform has a version of the GNU C compiler with this capability. Thus, the OpenMP parallel directives are used to direct the automatic transformation of the parallelizable loops in JasPer. Loops with integer index variables are easily handled by the compiler, as described in Chapter 4. The
compiler generates the necessary code to create and synchronize threads, and to distribute the computation in the original loops to different processors.

The parallel execution of the original computation from each loop created by OpenMP is managed in a manner comparable to that described for the explicit transformation of the code. OpenMP distributes subsets of loop iterations to multiple processors for independent execution. A barrier synchronization will be performed by OpenMP after finishing all the distributed processing work. In fact, it is the compiler’s responsibility to ensure the completeness and correctness of thread creation and synchronization.

5.3.2 Parallelization for Code-block Processing Step

The code-block processing step is a good candidate for performance enhancement through parallelization, based on the gprof profiling results presented in Section 3.1.2. For the cats image, profiling shows that 55.32% and 33.54% of total execution time were spent in this step for lossless and lossy encoding, respectively. Both encoding modes share the same code-block processing step, therefore, the same parallelization strategy is applicable to both cases.

Based on the analysis of loop iteration counts for the jpc_enc_encblk function discussed in Section 3.1.6, the innermost for loop has the greatest granularity. The OpenMP directives are therefore inserted before this loop to direct the automatic transformation for parallelization, as shown in Figure 5.15. Variables v and mx are privatized because each thread must have its own copy of these variables to calculate a local maximum without presenting any errors.
int jpc_enc_enccblks(jpc_enc_t *enc)
{
    ... for (tcmpt = tile->tcmpts; tcmpt != endcomps; ++tcmpt)
    { ... for (lvl = tcmpt->rlvls; lvl != endlvls; ++lvl)
        { ... for (band = lvl->bands; band != endbands; ++band)
            { ... for (prcno = 0, prc = band->prcs;
                prcno < lvl->numprcs; ++prcno, ++prc)
                { ... #pragma omp parallel for private(v, mx)
                    for (ii = 0; ii < prc->numcblks; ii++)
                        { ... jpc_enc_enccblk(enc, cblk->stream, tcmpt,
                            band, cblk);
                        }
                    }
                }
            }
        }
    }
    ...
}
5.3.3 Parallelization for DWT

The lossless and lossy operations are distinguished in DWT. For lossless encoding, the function \texttt{jpc\_ft\_analyze} is invoked which employs the 5/3 lifting algorithm, whereas for lossy encoding, the function \texttt{jpc\_ns\_analyze} is used to employ the 9/7 lifting algorithm. Nonetheless, both functions are coded in a similar fashion. They contain vertical and horizontal filtering loops which are parallelizable. The vertical filtering acts on each column of the image data, and the horizontal filtering processes the data for each row of an image.

The parallelization techniques used for transforming the DWT code are exactly the same. With appropriate loop transformations that have been done in Chapter 4, OpenMP directives are added for these loops, as shown in Figure 5.16 and Figure 5.17 for the 5/3 and 9/7 filtering algorithm, respectively.

5.3.4 Parallelization for Quantization

The quantization step is only applicable to lossy encoding, and it resides in the \texttt{jpc\_quantize} function. The process contributes a small fraction 10\% of the total execution time for encoding the \texttt{cats} image, as indicated in Section 3.1.2. Therefore, a significant speedup is not expected to be obtained from parallelizing the quantization process.

The parallelization for the quantization step is straightforward. The outermost for loop is parallelized by using the OpenMP \texttt{parallel} directive, as shown in Figure 5.18.
int jpc_ft_analyze(jpc_fix_t *a, int xstart, int ystart,
                   int width, int height, int stride)
{
    ...
    startptr = &a[0];

#pragma omp parallel for
    for (i = 0; i < maxcols; i += JPC_QMFB_COLGRPSIZE)
    {
        ...
    }
    ...
    startptr = &a[0];

#pragma omp parallel for
    for (i = 0; i < numrows; ++i)
    {
        ...
    }
    ...
}

Figure 5.16: Parallelizing loops with OpenMP in \texttt{jpc\_ft\_analyze}
int jpc_ns_analyze(jpc_fix_t *a, int xstart, int ystart,
    int width, int height, int stride)
{
    ...
    startptr = &a[0];

    #pragma omp parallel for
    for (i = 0; i < maxcols; i += JPC_QMFB_COLGRPSIZE)
    {
        ...
    }
    ...
    startptr = &a[0];

    #pragma omp parallel for
    for (i = 0; i < numrows; ++i)
    {
        ...
    }
    ...
}

Figure 5.17: Parallelizing loops with OpenMP in jpc_ns_analyze

void jpc_quantize(jas_matrix_t *data, jpc_fix_t stepsize)
{
    ...

    #pragma omp parallel for
    for (i = 0; i < jas_matrix_numrows(data); ++i)
    {
        for (j = 0; j < jas_matrix_numcols(data); ++j)
        {
            ...
        }
    }
}

Figure 5.18: Parallelizing loop with OpenMP in jpc_quantize
5.3.5 Results from Multithreaded Hardware Execution

In this section, the results of parallel execution of JasPer are discussed. Analysis for encoding of the cats image is described first, followed by the discussion of compressing the entire test image suite. The automatically transformed code described in the preceding sections has been executed on 2 and 4 processors on our Quad64 platform.

For lossless encoding of the cats image, the profiling results shown in Section 3.1.2 for hardware execution indicated that the parallelizable portion of JasPer constitutes 72% of the total execution time. Consequently, the upper bound on speedup is $1/(1-0.72) = 3.5$, and the expected speedups for 2 and 4 processors are 1.6 and 2.2, respectively. The experimental speedups are 1.3 and 1.8, which are computed from Table 5.5. Through further experimentation, we have determined the speedups for the code-block processing and DWT steps alone. The loops in the DWT step have a small workload relative to the parallelization overhead, leading to insignificant speedup. The speedup from the code-block processing step is much higher because those loops have a larger workload relative to the overhead.

Table 5.4 summarizes the speedups for lossless encoding of the cats image. For the simulated execution, the speeds obtained from cycle counting (ideal) are close to those obtained from explicitly parallelized JasPer (sim) because SimpleScalar assumes an ideal single-cycle latency for all simulated instructions. On the other hand for the hardware execution, the speeds calculated from gprof profiling results (gprof) are somewhat different than those obtained from OpenMP parallelized JasPer (omp). A certain amount of overhead is introduced while executing the multithreaded JasPer on our Quad64 system. Nonetheless, the overhead does not prevent a reasonable speedup from being achieved.
From the standpoint of the profiling results on real hardware in Section 3.1.2 for lossless encoding of the cats image, the significant contributors to the total execution time that could conceivably be parallelized have been addressed effectively. The potential for further incremental improvement in the speedup is indeed limited. As the largest remaining contributor is only 6%, even if it were parallelized on 4 processors, the expected speedup would not be significantly increased.

From the cache-related performance analysis in Section 3.2, the expected miss rate is not particularly high. Hence, there is presumably limited impact for the modification of the data organization and layout to attempt to reduce the cache miss rate. Further performance improvements may better be sought with enhancements related to dynamic memory management, which accounts for approximately 6% of execution time in hardware execution profiling, and also the possible vectorization of both serial and parallel portions of the code.

Similarly, for lossy encoding of the cats image, the parallelizable portions of JasPer constitute 71% of the total execution time, based on the gprof profiling results shown in Section 3.1.2. Consequently, the upper bound on speedup is $1/(1-0.71) = 3.4$, and the expected speedups for 2 and 4 processors are 1.5 and 2.1, respectively. The experimental speedups are

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Speedup</th>
<th>Simulated Execution</th>
<th>Hardware Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.4</td>
<td>1.4</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>1.9</td>
<td>1.8</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 5.4: Speedups for lossless encoding of cats image
Table 5.5: Multithreaded lossless encoding results on Quad64

<table>
<thead>
<tr>
<th>Image</th>
<th>1</th>
<th></th>
<th></th>
<th>2</th>
<th></th>
<th></th>
<th>4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>usr</td>
<td>sys</td>
<td>real</td>
<td>usr</td>
<td>sys</td>
<td>real</td>
<td>usr</td>
<td>sys</td>
</tr>
<tr>
<td>cats.pnm</td>
<td>1.437</td>
<td>1.286</td>
<td>0.080</td>
<td>1.084</td>
<td>1.730</td>
<td>0.096</td>
<td>0.822</td>
<td>2.280</td>
<td>0.104</td>
</tr>
<tr>
<td>pcb_small.pnm</td>
<td>1.870</td>
<td>1.693</td>
<td>0.093</td>
<td>1.320</td>
<td>2.156</td>
<td>0.095</td>
<td>0.923</td>
<td>2.427</td>
<td>0.113</td>
</tr>
<tr>
<td>pcb_large.pnm</td>
<td>6.231</td>
<td>5.642</td>
<td>0.370</td>
<td>4.783</td>
<td>7.425</td>
<td>0.419</td>
<td>3.164</td>
<td>7.251</td>
<td>0.425</td>
</tr>
<tr>
<td>pcb_tiled.pnm</td>
<td>7.695</td>
<td>6.983</td>
<td>0.371</td>
<td>5.805</td>
<td>9.152</td>
<td>0.419</td>
<td>3.606</td>
<td>8.494</td>
<td>0.462</td>
</tr>
<tr>
<td>eso0905a.pnm</td>
<td>96.429</td>
<td>71.657</td>
<td>4.428</td>
<td>73.964</td>
<td>79.318</td>
<td>4.924</td>
<td>78.054</td>
<td>82.231</td>
<td>5.508</td>
</tr>
<tr>
<td>eso0905a_small.pnm</td>
<td>13.527</td>
<td>12.232</td>
<td>0.804</td>
<td>10.165</td>
<td>15.175</td>
<td>0.869</td>
<td>7.060</td>
<td>15.319</td>
<td>0.959</td>
</tr>
</tbody>
</table>

1.1 and 1.3, which are calculated from the execution results presented in Table 5.6.

Based on the profiling results on real hardware in Section 3.1.2 for lossy encoding of the cats image, the significant contributors to the total execution time that could conceivably be parallelized have been addressed effectively. The cache-related performance analysis in Section 3.2 for lossy encoding of the cats image shows a low cache miss rate. Hence, the possibility of further improving cache performance for lossy encoding is limited.

The experiments for encoding the entire collection of test images with parallelized JasPer have been conducted on our Quad64 platform. Execution times for lossless and lossy encoding are summarized in Table 5.5 and Table 5.6, respectively. Speedups are calculated from these tables by using the (real) execution time on 1 processor relative to the execution time on 2 or 4 processors.

Figure 5.19 presents speedup results for lossless encoding in graphical form. The best speedups are for pcb_small and pcb_tiled, which were shown in Table 3.4 to have the largest fractions of time (above 70%) for the code-block processing step. The speedup is high
Table 5.6: Multithreaded lossy encoding results on Quad64

<table>
<thead>
<tr>
<th>Image</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>usr</td>
<td>sys</td>
</tr>
<tr>
<td>cats.pnm</td>
<td>1.807</td>
<td>1.675</td>
<td>0.081</td>
</tr>
<tr>
<td>pcb_small.pnm</td>
<td>1.644</td>
<td>1.507</td>
<td>0.075</td>
</tr>
<tr>
<td>pcb_large.pnm</td>
<td>5.754</td>
<td>5.287</td>
<td>0.347</td>
</tr>
<tr>
<td>pcb_tiled.pnm</td>
<td>7.349</td>
<td>6.806</td>
<td>0.357</td>
</tr>
<tr>
<td>eso0905a.pnm</td>
<td>82.352</td>
<td>66.351</td>
<td>3.951</td>
</tr>
<tr>
<td>eso0905a_small.pnm</td>
<td>11.118</td>
<td>10.164</td>
<td>0.749</td>
</tr>
</tbody>
</table>

Even when the total execution time is small. The worst speedup is for the eso0905a image, whose large size significantly increases disk I/O activity (note the difference between real time and usr time in Table 5.5 unlike the other images).

Figure 5.20 summarizes the speedup results for lossy encoding. All of the speedups are below 1.5. The large images tend to have better performance for lossy encoding because the disk I/O activity is less significant for lossy encoding. The size of large image are reduced substantially in lossy encoding, for instance, our largest image eso0905a is reduced approximately 50% relative the same image encoded losslessly. Hence, less disk I/O activity is involved and a relatively better performance is achieved.

5.3.6 Image Comparison Results for Multithreaded Execution

Generated images from multithreaded execution on our Quad64 platform are compared against the original images using the analytical software tool provided with JasPer. A comparable custom image comparison tool developed in C++ for this research is used to further ensure
Figure 5.19: Speedups from lossless encoding on Quad64
Figure 5.20: Speedups from lossy encoding on Quad64
the quality of the generated images. As expected, both tools have produced identical image comparison results in all experiments.

For lossless encoding, the output images in the JPEG-2000 format are expected to be identical to their original images without any imperfections. The comparison results for generated images from the multithreaded execution on Quad64 all have PSNR = ∞ and MSE = 0, regardless of the number of threads that are used.

For lossy encoding, the output images in the JPEG-2000 format are compared against their original images to ensure that these output images are acceptable in terms of quality. The results are expected to meet the generally accepted criterion of PSNR > 30 dB for a good image reconstruction. Table 5.7 shows the image comparison results for each image component. The cats image contains one component because it is a greyscale image. The comparison results reveal that all PSNR values are greater than 40 dB, which is well above the accepted 30-dB threshold for a good image reconstruction. In order to further verify the correctness of our parallel algorithm, we compare the JPEG-2000 image that was generated by the original serial version of JasPer against all output JPEG-2000 images that were generated from the multithreaded execution. Results show that all of these images are identical to each other which confirms the correctness of our parallel algorithm. Despite the nondeterministic parallel execution, the comparison results are repeatable, which again confirms the correctness of the parallelization for JasPer encoding.
Table 5.7: Image comparison results from lossy compression

<table>
<thead>
<tr>
<th>Image</th>
<th>Component</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats.pnm</td>
<td>1</td>
<td>49.81</td>
<td>49.81</td>
<td>49.81</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>43.23</td>
<td>43.23</td>
<td>43.23</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>45.79</td>
<td>45.79</td>
<td>45.79</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>42.69</td>
<td>42.69</td>
<td>42.69</td>
</tr>
<tr>
<td>pcb_small.pnm</td>
<td>1</td>
<td>44.79</td>
<td>44.79</td>
<td>44.79</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>46.48</td>
<td>46.48</td>
<td>46.48</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>44.19</td>
<td>44.19</td>
<td>44.19</td>
</tr>
<tr>
<td>pcb_large.pnm</td>
<td>1</td>
<td>43.23</td>
<td>43.23</td>
<td>43.23</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>45.78</td>
<td>45.78</td>
<td>45.78</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>42.68</td>
<td>42.68</td>
<td>42.68</td>
</tr>
<tr>
<td>pcb_tiled.pnm</td>
<td>1</td>
<td>44.59</td>
<td>44.59</td>
<td>44.59</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>44.98</td>
<td>44.98</td>
<td>44.98</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>41.76</td>
<td>41.76</td>
<td>41.76</td>
</tr>
<tr>
<td>eso0905a.pnm</td>
<td>1</td>
<td>45.71</td>
<td>45.71</td>
<td>45.71</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>46.33</td>
<td>46.33</td>
<td>46.33</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>44.48</td>
<td>44.48</td>
<td>44.48</td>
</tr>
</tbody>
</table>
5.4 Summary

This chapter described the parallelization of the serial JasPer image encoder and presented results obtained from simulated and hardware execution. For the simulated execution, loops of interest have been parallelized explicitly to assess the potential for speedup. A relatively balanced workload distribution has been obtained from parallelized JasPer. The expected and experimental simulated speedups were in good agreement due to this balanced workload distribution. The memory access patterns have also been examined and characterized for the simulated multithreaded execution. On real hardware, loops of interest have been parallelized with OpenMP. Based on the profiling results shown in Section 3.1.2, the significant contributors to total execution time that could conceivably be parallelized have been addressed effectively. As a result, speedups of approximately 2 has been achieved for lossless encoding on 4 processors for both simulated and hardware multithreaded execution. The speedups for lossy encoding are lower because the parallelizable loops for lossy encoding have relatively lower granularity of parallelism. Finally, the generated images from both lossless and lossy encoding have been compared against the corresponding original images to ensure the image quality.
Chapter 6

Conclusion

Image coding for data compression involves a significant amount of processing for which higher performance can be achieved with parallel execution on general-purpose multicore processors. This thesis has discussed parallelization of the JasPer software which is a reference implementation of the JPEG-2000 image compression standard. The thesis has also described additional efforts for characterizing execution, memory access, and cache behavior.

JasPer encoding has been profiled on a single processor in simulation and on real hardware. The cache behavior has been analyzed, and a low miss rate has been observed. In addition, the memory access patterns have been studied to confirm that corresponding cache behaviors are reasonable reflections of the memory activities. Candidate functions for performance enhancement have been identified through execution profiling. Because loops usually have rich parallelism, parallelizable loops in the candidate functions have been identified to be transformed for parallelization. Potential speedups have been assessed from the profiling results in order to set appropriate expectations for subsequent experiments.

Prior to parallelizing JasPer, the original source code has been transformed in order to assist
The JasPer encoder has been parallelized explicitly for simulated execution and automatically for execution on real hardware. Parallelizable portions have been identified based on the knowledge of the JPEG-2000 standard and experiments in the previous work. For simulated execution, the expected speedup has been assessed from the multithreaded execution results. A relatively balanced workload distribution has been observed which confirmed a reasonable efficiency in simulated parallel execution. In addition, the memory access patterns have been examined and characterized for the simulated parallel JasPer execution.

On real hardware, loops of interest have been parallelized by using OpenMP. The OpenMP parallel directive has been used to label the parallelizable loops in the JasPer source code in order to guide the compiler to automatically transform these loops. As a result, for processing of images where 70% or more of single-processor execution time is spent in the code-block processing and the discrete wavelet transform, execution on an Intel Core 2 Quad processor has resulted in speedups of more than 2 on 4 processors.

Finally, the generated images from both lossless and lossy encoding have been compared against their corresponding original images to ensure the image quality. All generated images for lossless encoding have been perfectly reconstructed with PSNR=∞ in all image comparison results. The comparison results for lossy encoding reveal that all PSNR values are greater 40 dB which is well above the accepted 30 dB threshold for good image reconstruction.

From the standpoint of the profiling results for JasPer execution, the significant contributors to the total execution time that could conceivably be parallelized have been addressed. Further-
more, the load balance analysis has confirmed that reasonable efficiency has been achieved in the parallel JasPer execution.

6.1 Future Work

The parallelizable portions in JasPer that contribute most of the total execution time have been addressed. Therefore, the potential for further significant improvement in the speedup from parallel execution in the form considered in this thesis is limited. Moreover, from the cache-related performance analysis in Section 3.2, the miss rate is not particularly high. Hence there is presumably limited impact from changing the data organization and layout to attempt to reduce the cache miss rate. Further performance improvement may better be sought with enhancements related to distributed image processing and efficient data transfer within each processor.

The multithreaded JasPer with OpenMP could be adapted to execute in a cluster of multicore systems for distributed processing of large images. Building on the memory access analysis described in this thesis, the workload for a large image can be partitioned for distributed encoding, as opposed to pure tiling of the image which can potentially impair image quality. By effectively managing the read-only image data and generated image data on each cluster node, including distribution and collection of such data, distributed processing in this manner can be combined with multithreaded execution within each node of the cluster.

Another possible direction for performance enhancement within an individual processor is through vectorization. Because of the amount of data movement in the encoding process, vectorization may be beneficial to transfer multiple data items in parallel. General-purpose
processors have such an ability for vector computation. One of the challenges for this approach would be resolution of data dependencies. All data dependencies must be respected in order to produce correct results.
Bibliography


101


