

國立交通大學

資訊學院資訊科技（IT）產業研發碩士專班

碩士論文

JPEG2000 壓縮在雙核心數位訊號處理器上的實作與最佳化研究

Implementation and Optimization of JPEG2000 Compression on Dual-core
DSP Processors

研究生：何柏瑋

指導教授：游逸平 教授

中華民國一百年七月

JPEG2000 壓縮在雙核心數位訊號處理器上的實作與最佳化研究

Implementation and Optimization of JPEG2000 Compression on Dual-core
DSP Processors

研 究 生：何柏瑋

Student：Po-Chiang Ho

指導教授：游逸平 博士

Advisor：Dr. Yi-Ping You

國立交通大學
資訊學院資訊科技（IT）產業研發碩士專班



碩士論文
A Thesis
Submitted to College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Industrial Technology R & D Master Program on
Computer Science and Engineering

July 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年七月

JPEG2000 壓縮在雙核心數位訊號處理器上的實作 與最佳化研究

學生：何柏瑋

指導教授：游逸平 博士

國立交通大學資訊學院產業研發碩士專班

摘 要

多核心是未來處理器設計的趨勢。Analog Device (ADI)在最新一代的 Blackfin 處理器—ADSP-BF561—中也採用了多核心的設計。BF561 是一顆採用微信號架構 (Micro Signal Architecture)的雙核心數位訊號處理器，此架構擅長於處理影像及各種多媒體訊息。在本篇論文中，我們從 OpenJPEG 公開原始碼計畫中移植一個 JPEG2000 壓縮的程式到 BF561 上，接著在應用程式的階層上提出並實作最佳化的方法。我們的最佳方法主要在於(一)資料地域性最佳化和(二)把工作分配到兩個核心上執行。我們挑選了 JPEG2000 壓縮中佔運算比重最大的兩個部份—DWT 和 EBCOT Tier-1—來實行我們所提出最佳化方法。此外，我們在論文中討論實驗中遇到的兩個關於編譯器的問題：其一是 GCC 內建函式對跨函式最佳化的干擾，另一是 GCC 無法有效率的產生出平行指令。在我們的實驗中，我們發現使用我們所提出的資料地域性最佳化後可以有效地提昇兩個核心的使用效率，原因是我們的最佳化幅度減少了對外界低速記憶體存取的需求。我們使用了四張標準的測試影像來評估我們最佳化的效能。我們的最佳化結果相較於原始程式在一個核心上執行並加了-O3 編譯器最佳化，可以加速影像壓縮達 1.92 至 2.04 倍左右。

關鍵詞：Blackfin，數位訊號處理器，BF561，JPEG2000，平行處理，雙核心

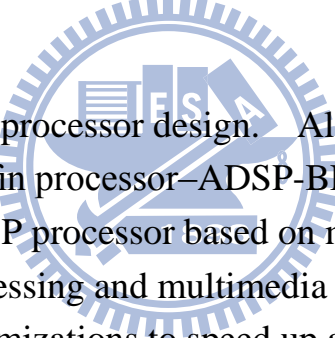
Implementation and Optimization of JPEG2000 Compression on Dual-core DSP Processors

student : Po-Chiang Ho

Advisors : Dr. Yi-Ping You

Industrial Technology R & D Master Program of
Computer Science College
National Chiao Tung University

ABSTRACT



Multi-core is the trend of future processor design. Along with this trend, Analog Device (ADI) developed their latest Blackfin processor—ADSP-BF561—with a multi-core design. BF561 is a dual-core, SMP-like DSP processor based on micro signal architecture (MSA), which is specialized for video processing and multimedia computations. In this paper we propose several software-level optimizations to speed up a JPEG2000 compression program ported from OpenJPEG project on a Blackfin BF561 processor. Two optimization methods, data locality optimization and utilization of two cores, are performed on the two heavy-loading stages of JPEG2000 compression: DWT and EBCOT Tier-1. Implementation issues such as the disturbance to compiler optimizations when using GCC attributes and inefficient generations of parallel instructions are discussed. In our experiments, we found that we can only benefit from the utilization of two cores after the data locality optimization is well performed because the data locality optimization reduces the heavy loading of accesses to low-speed SDRAM. Four popular image testbenches are used to evaluate the efficiency of our optimizations. The experiments showed that the optimizations have a speed-up of 1.92x–2.04x for the compression compared to the baseline with -O3 optimization flag running on single core.

Keywords: Blackfin, DSP, BF561, JPEG2000, parallel processing, dual-core

誌謝

首先，誠摯感謝我的指導老師游逸平教授在我碩士生涯研究上的指導與生活上的關照。老師的指導，非但讓我在學術研究及專業能力上有所收穫，也讓我養成對文章撰寫及口語表達追求嚴謹、有條不紊的態度。

感謝實驗室的同學世融在實驗設備採購上的協助。感謝學弟璨榮，翰融在研究上的討論及實驗上的協助。有你們的協力，這篇論文得以更加豐富及完整。也感謝學弟學妹：羽軒、深弘、聖偉、思捷、陸昂，有你們的歡笑及活力，讓我得以度過一天天枯燥的研究生活。

最重要的，我非常感謝父母的支持與鼓勵。這段時間甚少回家，感謝你們的體諒與關心，你們的支持讓我撐下去完成碩士的學業。

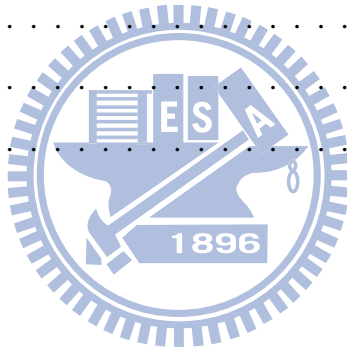
最後，感謝洋銘科技的何總監及劉副理在研究上的意見及生活上的關心。你們的賞識及支持，是我完成這篇論文的一切前提。



誌於 辛卯年夏 竹塹交大
柏瑋

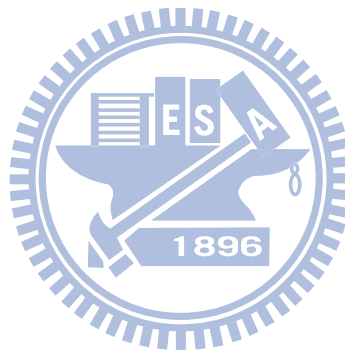
Contents

1	Introduction	1
1.1	Overview	1
1.2	Motivation	2
1.3	Problem Definition	3
1.4	Contribution	4
1.5	Thesis Organization	5
2	Related Work	6
3	JPEG2000 Overview	9
3.1	Background and History	9
3.2	JPEG2000 Compression Procedure	11
3.2.1	Pre-processing	12
3.2.2	Discrete Wavelet Transform	13
3.2.3	EBCOT Tier-1 Coding	16
3.2.4	EBCOT Tier-2 Coding	20
4	The Architecture of Analog Device BF561	21
4.1	Blackfin Core	21
4.2	Blackfin ADSP-BF561	23
4.2.1	Memory Hierarchy	24



4.2.2	DMA Support	25
5	Implementation and Optimization	28
5.1	Experiment Environment Setup	28
5.2	Software-based JPEG2000 Implementation and Profiling	29
5.3	Overview of JPEG2000 Optimizations on BF561	30
5.3.1	Data Locality Optimization	30
5.3.2	Utilization of Two Cores	33
5.4	Optimization of DWT	36
5.4.1	Data Locality Optimization	36
5.4.2	Utilization of Two Cores	40
5.4.2.1	Data Partition	40
5.4.2.2	Task Partition	42
5.4.3	DMA Optimization	45
5.5	Optimization of EBCOT Tier-1	47
5.5.1	Data Locality Optimization	47
5.5.2	Utilization of Two Cores	50
5.6	Optimization Using Inline Assembly	51
6	Evaluations and Discussions	53
6.1	Evaluations and Discussions of DWT	53
6.2	Evaluation and Discussion of EBCOT Tier-1	56
6.2.1	The Disturbance of Compiler Optimizations due to Putting Proce- dures to the L1 Instruction SRAM	60
6.3	Evaluation of Inline Assembly Optimization	61
6.4	Overall Evaluation	62
6.4.1	Data Cache V.S. Handmade Data Locality Optimization	64

6.4.2 Overall Results	64
7 Conclusion and Future Work	68
7.1 Summary	68
7.2 Future Work	69

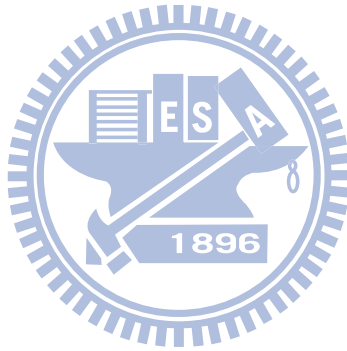


List of Figures

3.1	The procedure of JPEG2000 lossless compression.	12
3.2	(5,3) DWT (left) and inverse DWT (right).	14
3.3	An example of discrete wavelet transform: (a) the original image, (b) after 1D-DWT computation in horizontal direction, (c) after 2D-DWT computation, (d) 2-level DWT computation.	14
3.4	The ordering of high pass coefficients and low pass coefficients being generated.	16
3.5	The hierarchy of data partition of an image.	17
3.6	The scan pattern of a codeblock in one bit-plane.	18
3.7	An example to show what is “significant”.	18
3.8	The hierarchy of bit-plane coding.	18
4.1	Blackfin core architecture.	22
4.2	Block diagram of BF561 architecture.	23
4.3	Memory and bus architecture of BF561.	27
5.1	Execution time breakdown of JPEG2000 compression on the BF561 processor.	31
5.2	Master-slave model of MPEG-2 encoder on dual-core processors.	34
5.3	Pipelined model of MPEG-2 encoder on dual-core processors.	35
5.4	2D-DWT computation.	37

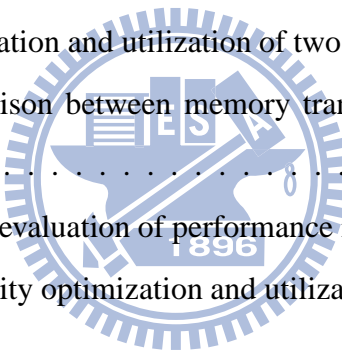
5.5	Dataflow of DWT computation performed in one line.	38
5.6	The data moving flow in a horizontal line.	39
5.7	The data moving flow in a vertical line.	40
5.8	Dataflow of DWT computation performed in one line: (a) before data locality optimization (b) after data locality optimization.	41
5.9	The data partition to two cores.	42
5.10	The memory/calculation partition to two cores	43
5.11	Latency of DMA transfer in continuous data.	46
5.12	Linking of DMA descriptors.	46
6.1	The analysis of time consumption in DWT before and after data locality optimization.	55
6.2	The analysis of execution time of DWT using mem/cal partition.	57
6.3	Speed-up of the proposed optimizations for DWT.	57
6.4	Speed-up of the proposed optimizations for EBCOT Tier-1.	59
6.5	A simple program to show the disturbance of compiler optimizations due to putting procedures to the L1 instruction SRAM.	61
6.6	The assembly codes of the simple test program in Figure 6.5. The left is the original one; the other is the disturbed one.	62
6.7	The MCT source code.	63
6.8	The assembly code generated by GCC.	63
6.9	The assembly code we reassembled.	64
6.10	The performance comparison between automatic data cache and our hand-made data locality optimization.	65
6.11	Time consumption to compress a 640x480 image.	66
6.12	The standard image testbenches.	66

6.13 The overall performance evaluation of proposed optimizations on standard image testbenches. 67



List of Tables

3.1	An example of the three coding passes being performed in every bit-plane.	19
5.1	The data allocation of EBCOT Tier-1 in the L1 data SRAM.	49
6.1	DWT: The evaluation of performance improvements of two optimizations: data locality optimization and utilization of two cores.	53
6.2	The loading comparison between memory transfer and computations in DWT.	56
6.3	EBCOT Tier-1: The evaluation of performance improvements of two opti- mizations: data locality optimization and utilization of two cores.	58



Chapter 1

Introduction

1.1 Overview

Digital images or videos need large amounts of space for storage of the contents. For the efficient utilization of memory and storage space, we need to compress them via reducing spacial or temporal redundancy. Image (video) compression is a digital signal processing technique developed to compress an image (video). The compression procedures have heavy computation and calculation loading. In a desktop environment, this is not hard because the computing power of modern CPUs often could afford the loading. However, in an embedded environment, power consumption often needs to be considered since power supply of many embedded systems come from batteries. Application-specific integrated circuit (ASIC) is a good choice for speed and power consumption, and the price are often not expensive. DSP processors may be another flexible choice since they could run software programs just like we run on desktop. Although the performance are not good as ASIC, DSP processors are convenient to change software programs to target specific applications and the performance on image compression are often better than general purpose processors.

DSP processors are microprocessors designed to perform digital signal processing, the mathematical manipulation of digitally represented signals. Digital signal processing is one of the core technologies in rapidly growing application areas such as wireless communica-

tions, audio and video processing, and industrial control [9]. Powerful ALUs and Multipliers are the basic characteristics of DSP processors and their memory access often could be parallel with mathematic calculations. Furthermore, special hardware components are designed on them for accelerating digital signal processing like subtract-absolute-accumulate (SAA), multiplier-and-accumulation (MAC), and so on.

JPEG2000 [15] is a novel image standard proposed by JPEG committee to approach the modern applications such as Internet, medical images, video conference and etc. Hence, we do some researches to examine that how JPEG2000 could benefit from the architectures of modern DSP processors.

1.2 Motivation

Moore's law tell us that the number of transistors that can be put on an integrated circuit has doubled approximately every two years. The trend has continued for more than half a century. It will stop, however, eventually on a certain level and cannot go on any more since the atomic limit. In addition, there are two serious problems while we try to put more transistors on a chip: overheat and power consumption. Therefore, processors are designed multi-cores, which means to put one more cores on one chip. Hence, how to divide calculating jobs to many cores becomes an important issue.

To follow this trend, the newest DSP processors of Blackfin family, which are developed by Analog Device (ADI), are also designed multi-core; that is ADSP-BF561 [3].

Blackfin 16/32-bit embedded processors are designed for software flexibility and scalability for convergent applications: multi-format audio, video, voice and image processing, multi-mode baseband and packet processing, control processing, and real-time security. ADSP-BF561 is configured as a symmetric multiprocessing arrangement of two Blackfin processor cores. Each is capable of operating at up to 600 MHz and has up to 2.6 MB of on-chip SRAM memory.

Why we choose Blackfin? There are some reasons make it distinctive. Blackfin architecture is named micro signal architecture (MSA); it's co-developed by Intel and Analog Device. Unlike very long instruction word (VLIW) architecture, MSA mixes powerful ALUs into RISC-like processors. This leads several advantages. First, RISC architecture is known compiler friendly. Hence, compiler designs for MSA are easier than for VLIW, which is adapted by most DSP processors. In addition, the design flow is straightforward; the two suites of development tools aren't needed. Finally, the hardware designs are more cost and power effective.

In recent years, surveillance cameras and automatic traffic recorders (ATR) are popular and widely used in our daily life. To reach better compression video quality, we need a novel video compression standard.

JPEG2000, a new compression standard for still images, is developed to overcome the shortcomings of the existing JPEG standard, which is standardized by Joint Technical Committee on Information technology of the International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC).

In JPEG2000 standard, Motion JPEG2000 has been standardized to be a part of JPEG2000. It could be used as video standard to achieve better video quality for widely uses include cinema, surveillance, ATR and so on.

For the reasons mentioned above, we try to examine how JPEG2000 software based compression could efficiently run on a dual-core BF561 to achieve good compression performance.

1.3 Problem Definition

Since we know ADSP-BF561 is a dual-core processor, and uClinux could run on both cores like symmetric multi-processor (SMP). uClinux is a lightweight version of Linux working on processors with no memory management unit (MMU) and the trunks for BF561

are developed by the community. If we have full uClinux supports on BF561, we have affluent library supports from Linux. This makes easy for us to establish our own image compressing systems. In addition, abundant resources about Linux also could be found on the Internet.

However, there are still lacks of researches and reference manuals to discuss the utilization of two cores. We need to know if jobs' partition to two cores in BF561 could as good as in general SMP. If it works well, we are convenient to move our software development procedures in a general SMP system to this SMP-like system.

For these reasons, we implement and optimize JPEG2000 on BF561. We divide JPEG2000 into several components and do the parallelization on these components.

Our JPEG2000 compression program would be expected to totally come from open-source resources. To exploit famous open-source projects from the Internet, we are not only easy to build our experimental environments but also capable to learn the source code implementations. Furthermore, they may be allowed to be commercial utilization; this depends on their release Licenses. Our JPEG2000 compression would be focused on lossless compression since it could conserve the details for flexible utilization.

Our optimization approaches would derivate from the convergence of profiling, the understanding of JPEG2000 algorithms, and hardware architectures; the optimization ordering would follow the principle: the efficient one, the prior one.

1.4 Contribution

In this paper, we implement and optimize the JPEG2000 lossless compression under SMP-like mode on Analog device BF561. Our optimizations focus on the components of JPEG2000, DWT and EBCOT Tier-1, which are the heavy loading and also potential parallel parts of the whole compression procedure. Our main contributions list in the following:

- Discussion of open-source resource supports and hardware constraints under full OS

supports on BF561 SMP-like environments.

- Implementation and evaluation of data locality optimization by using high-speed L1 data SRAM
- Implementation and evaluation of jobs' partition to dual cores.
- Implementation and evaluation of the effectiveness of inline assembly optimization on JPEG2000.

1.5 Thesis Organization

This thesis is organized as follow. In chapter 2, the related work is introduced. In chapter 3, we describe the overview of JPEG2000. In chapter 4, we describe the architecture of Blackfin BF561, the target platform of this work, especially on the memory architecture and DMA supports. In chapter 5, we detailed discuss the implementations and our optimization methods of JPEG2000 on BF561. In chapter 6, the experimental results is presented and the problems we encountered is discussed. The chapter 7 concludes the work and presents future work.

Chapter 2

Related Work

There are several researches about JPEG2000. Majif Rabbain and Rajan Johsi gave a very good overview of JPEG2000 [22]; it's a good beginning to understand JPEG2000. David Taubman and Michael Marcellin have deeply discussed the theory of digital signal processing techniques used in JPEG2000 [25]. Timku Acharya and Ping-sing Tsai detailedly explained the specifications of JPEG2000 [2]. They focused on the specifications and implementations. In addition, many good examples are included. It is a very good reference to understand the implementation details of JPEG2000.

There are also many studies about JPEG2000 software implementations on different processors. H. Muta *et al.* did implementation and parallelizations of JPEG2000 compression on Cell/B.E [19]. They speeded up the JPEG2000 encoding by parallelizations using SPEs on the Cell/B.E. In addition, they did the system level parallelizations by using Cell/B.E blade servers. P. Meerwald *et al.* evaluated parallelizations of JPEG2000 using OpenMP and JAVA threads on SMP Intel Pentium II Xeon running at 500 MHz [17]. The tile parallelization was abandoned here due to the artifact effects. The JAVA implementation was from JJ2000 and the OpenMP was adapted in the C implementation of Jasper. The parallelization results showed that they could avoid cache missing greatly if the image was read from the vertical directions. In addition, EBCOT Tier-1 was encoded by parallel codeblocks. Azkarate-Askasua Mikel built a JPEG2000 compression system in a multi-

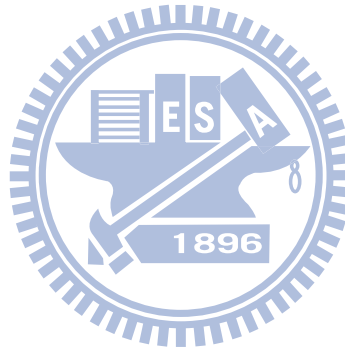
processor system on FPGA using the commercial system-level-design tool [7]. They used OpenJPEG library to be the JPEG2000 implementation and divided it into several parts in order to map them on to the design tool. System-level-design tools are used to reduce the efforts of developers and speed up the time to market.

Discrete wavelet transform (DWT), which is an important component of JPEG2000 compression, suffers memory uncontinuous reading problems while using software implementations. Dividing the image into pseudo small tiles is the solution used in [19]. However, in order to avoid edge effects, they have to make the tiles overlapping. This work needs large efforts. Putting the vertical lines together and then performing DWT to them using JAVA threads is the solution used in [17]. However, we don't have a JAVA environment and Linux threads cannot be scheduled to the other core on BF561. In our solution, we analyze the model of uncontinuous memory accesses and transform these accesses to be the jobs of DMA controllers. Then, DMA controllers help transform these data to be continuous data and put them in the high speed SRAM for fast accesses. Our work can efficiently eliminate the slow accesses to external SDRAM.

In addition, there are several with respect to Blackfin platforms. Michael G. *et al.* put data in shared L2 SRAM of BF561 and performed the data processing from both cores [8]. They showed that to put the data in the SRAM could only benefit from the stream programming model. The model means that the two cores do different jobs. Jun-Wei Gao and Ke-Bin Jia established a H.264 based video surveillance system with real-time compression on a BF561 platform [13]. They briefly described five methods to optimize the h.264 encoder: (1) allocating storage space, (2) issuing parallel instructions, (3) using special video instructions, (4) utilizing hardware loop, and (5) choosing a suitable assembly instruction. Hee Seo and Seon Wook Kim improved OpenMP performance on BF561 by moving shared data into shared L2 SRAM and further moving private data into L1 SRAM [23]. They focused on the fork/join model and put the data into L1 data SRAM as possible as they can;

only shared variables stayed in shared L2 SRAM. They showed that the power consumption could be reduced by directly measurement using external sourcemeters. C.H. Chen showed that well-optimized Blackfin assembly code could achieve high performance improvement compared to unoptimized one [11]. The assembly of Blackfin architecture can be parallelized under some restrictions. They used the feature to reassemble the assembly to speed up the discrete wavelet transform in JPEG2000.

The related work mentioned above include many aspects of researches. Some of these work mentioned that how they optimized their implementations on the Blackfin platform. These work can be references for us to avoid going the wrong ways in our researches.



Chapter 3

JPEG2000 Overview

In this chapter, we will introduce the basic concepts of JPEG2000 and explain why it is special and different from traditional JPEG.

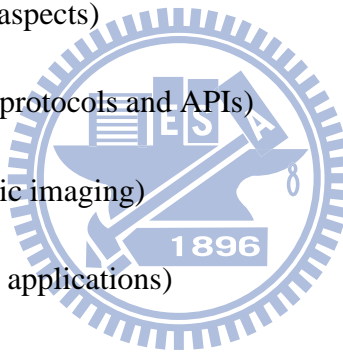
3.1 Background and History

The well-known JPEG standard is developed by JPEG (Joint Photographic Experts Group) committee, which is founded in 1986 under the joint auspices of ISO and ITU-T, and has become the most popular image compression standard in past twenty more years. Almost every image or video instrument supports JPEG standard. Despite the great success of the JPEG image compression system, it has several shortages that become increasingly apparent as the need for image compression is extended to emerging applications such as medical imaging, digital libraries, Internet multimedia transmission, and so on.

In March 1997 a call for proposals was issued to the new standard—JPEG2000. In November 1997, more than 20 algorithms were evaluated [22]. Finally it included many classic algorithms and became a “*big*” standard. Nowadays, JPEG 2000 refers to twelve parts of the standard [15]:

- **Part 1** Core coding system (intended as royalty and license-fee free — NOT patent-free)

- **Part 2** Extensions (adds more features and sophistication to the core)
- **Part 3** Motion JPEG2000
- **Part 4** Conformance
- **Part 5** Reference software (Java and C implementations are available)
- **Part 6** Compound image file format (document imaging, for pre-press and fax-like applications, etc.)
- **Part 7** has been abandoned
- **Part 8** JPSEC (security aspects)
- **Part 9** JPIP (interactive protocols and APIs)
- **Part 10** JP3D (volumetric imaging)
- **Part 11** JPWL (wireless applications)
- **Part 12** ISO Base Media File Format (common with MPEG-4)



Part 1 (the core) is now published as an International Standard , five more parts (2-6) are complete or nearly complete, and four new parts (8-11) are under development.

While the standard is well defined, why we need JPEG2000? There must be some reasons to persuade us to use the new standard. There are several new features show that why JPEG2000 could be the compression standard of the next generation [2]:

1. **Superior low bit-rate performance**—JPEG2000 offers good performance in very low bit-rates compared to traditional JPEG.
2. **Large dynamic range of the pixels**—JPEG2000 is the only standard could conduct the pixel values more than 16-bit precision; it is up to 38 bits;

3. **Lossless and lossy compression**—JPEG2000 provides lossless compression with progressive decoding. Applications such as digital libraries/databases and medical imagery can benefit from this feature.
4. **Protective image security**—the open architecture of the JPEG2000 standard makes easy the use of protection techniques of digital images such as watermarking, labeling, stamping or encryption.
5. **Region-of-interest coding**—in this mode, regions of interest (ROIs) can be defined. These ROIs can be encoded and transmitted with better quality than the rest of the image.
6. **Robustness to bit errors**—the standard incorporates a set of error resilient tools to make the bit-stream more robust to transmission errors.

Because of the good design of JPEG2000, it could be used in a variety of applications from professional medical images, Internet, wireless transmission, to low-end consumer electronics.

3.2 JPEG2000 Compression Procedure

In this section, we discuss about the JPEG2000 Part1 standard, the core of JPEG2000. We focus on the procedure of lossless compression of JPEG2000 since the lossless compression could reserve more details for flexible utilization. The main components of the coding procedure could be divided into four parts: *pre-processing*, *discrete wavelet transform*, *EBCOT Tier-1* and *EBCOT Tier-2*, as shown in Figure 3.1. We will discuss these components in the following subsections.

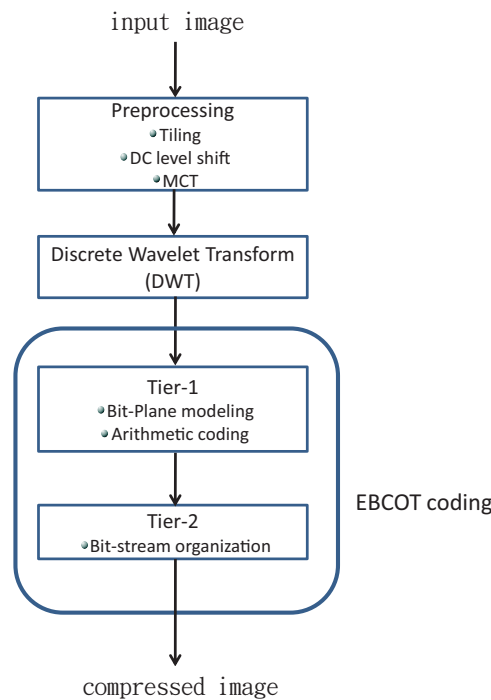


Figure 3.1: The procedure of JPEG2000 lossless compression.

3.2.1 Pre-processing

The pre-processing state includes three passes: tiling, Direct Current (DC) level shift and color transformation. In the first pass, tiling, we may partition the whole image into several independent “*tiles*”, and these tiles could be encoded by the independent parameters in the following procedures. This is useful when the compression hardware system has limited memory. The tiling size theoretically could be any size but often 512x512 or bigger up to the whole image size since small tiling size would lead to obvious edge effects [2].

After tiling, we perform DC level shift to shift pixel values from unsigned value to signed value in order to make the pixel values more balanced in the distance to “*zero*”; this leads more “*zero*” while quantization is performed and the compression ratio could be higher. Finally, we make color transformation called Multi-component Transformation (MCT) to transfer the color space of the image from RGB color space to YUV color space.

There are two kinds of MCT in JPEG2000 specification, which are Reversible Color Transformation (RCT) and Irreversible Color Transformation (ICT). RCT is applied in reversible coding and ICT is used in irreversible coding.

3.2.2 Discrete Wavelet Transform

The purpose of Discrete Wavelet Transform (DWT) is the same with discrete cosine transform in traditional JPEG but in different coding system. It tries to divide high frequency parts and low frequency parts of an input image so that we could adapt different strategies in the following steps to increase compression ratio. The “low frequency” could be realized that the values of two adjacent pixels of an image are similar. If the pixel values of a small region are similar, this region would be “smooth” as we view. The low frequency parts occupy the majority of a common natural image. On the other hand, “high frequency” implies that there may exist a shape, edge, or line or conceal more details.

The technique of DWT in JPEG2000 is based on filters. There are one high pass filter and one low pass filter in it. Low pass filter reserves low frequency data, which occupy most parts of an general natural image. On the other hand, high pass filter reserves high frequency data.

Two kinds of DWT filter are included in JPEG2000 standard: (9,7) and (5,3). The number “9” means the length of low pass filter is 9 and the number “7” means the length of high pass filter is 7. Since we focus on reversible coding, we just examine the (5,3) filter, which is designed for reversible coding, in the following discussion.

The (5,3) DWT and its opposite version, *inverse DWT*, are illustrated in Figure 3.2 [22]. The left site of the Figure 3.2 is DWT (forward) and the right one is inverse DWT. Input sequences $x(n)$ are conducted by low pass filter $h_0(n)$ and high pass filter $h_1(n)$ and then followed by sub-sampling of factor 2 to get output data; we call these output data “DWT coefficients”. On the other hand, these DWT coefficients could be reconstructed to original input data by the inverse symmetric operation: inverse DWT.

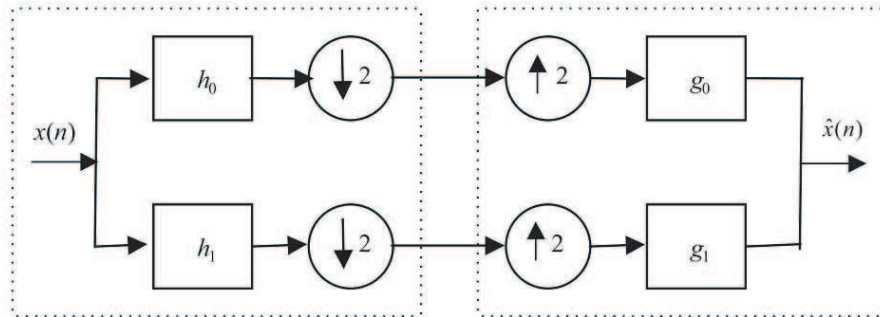


Figure 3.2: (5,3) DWT (left) and inverse DWT (right).

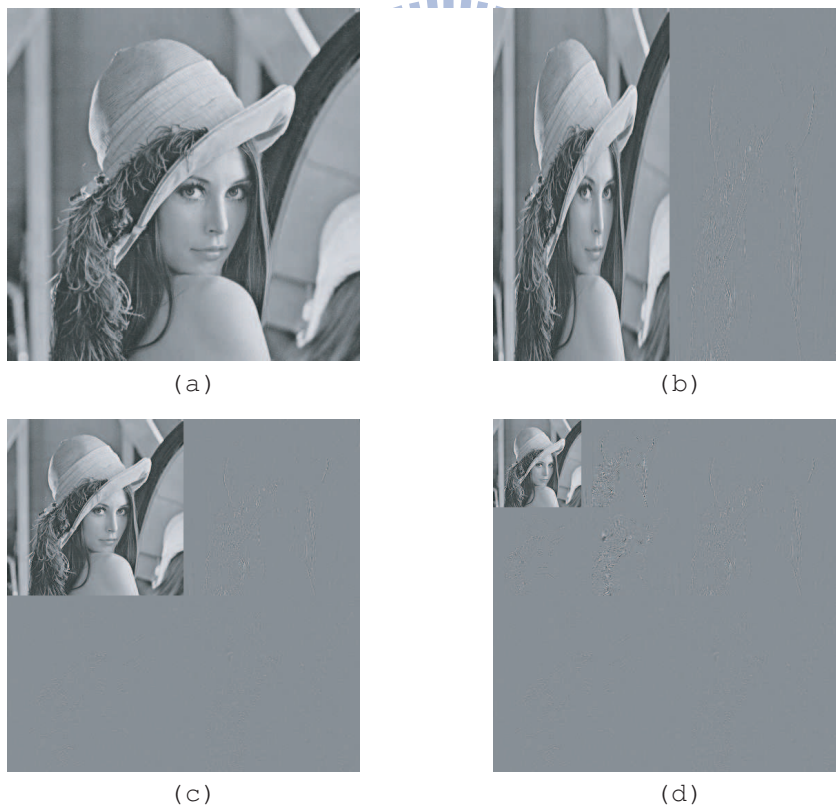


Figure 3.3: An example of discrete wavelet transform: (a) the original image, (b) after 1D-DWT computation in horizontal direction, (c) after 2D-DWT computation, (d) 2-level DWT computation.

The DWT computation of JPEG2000 is 2D-DWT; it means we do the DWT on an image from column by column to row by row. The ordering could be inverse from row by row to column by column. The effects on an image before and after 2D-DWT could be seen in Figure 3.3. Figure 3.3(a) is the original classic test patent: 512x512 gray-scale Lena. Figure 3.3(b) shows that the input image is separated to low frequency in left side and high frequency in right side after 1-D horizontal DWT computation; then we do the DWT computation to separate high and low frequency data in vertical direction, as shown in Figure 3.3(c). Furthermore, We could perform a two-level DWT for the low frequency data, as shown in Figure 3.3(d); it's level 2.

The traditional DWT needs complex convolution computations and is not adapted in JPEG2000 standard. JPEG2000 adapts a lifting-based DWT [12], which reduces significant memory footprint and computing complexity compared with traditional DWT. Furthermore, it could run in place; this means no more other memory space is needed during the computation, and the input data and the output data use the same memory space. The lifting-based DWT is based on two steps: *prediction* and *updating*. The Equation 3.1 shows that how to make the *prediction* calculation. $\{s^0\}$ and $\{d^0\}$ means even and odd values of input sequence, respectively. $\{d^1\}$ refers to the output of high pass coefficients. The *updating* procedure is shown in Equation 3.2; the output of low pass coefficients $\{s^1\}$ are obtained by specific calculation of modified coefficients $\{d^1\}$ and input data $\{s^0\}$. The subscript “i” means the input number. The concept could be expressed by Figure 3.4 [22]. We could see the ordering that high pass coefficients and low pass coefficients are interleaved generated.

$$d_i^1 = d_i^0 - \frac{1}{2}(s_i^0 + s_{i+1}^0) \quad (3.1)$$

$$s_i^1 = s_i^0 + \frac{1}{4}(d_{i-1}^1 + d_i^1) \quad (3.2)$$

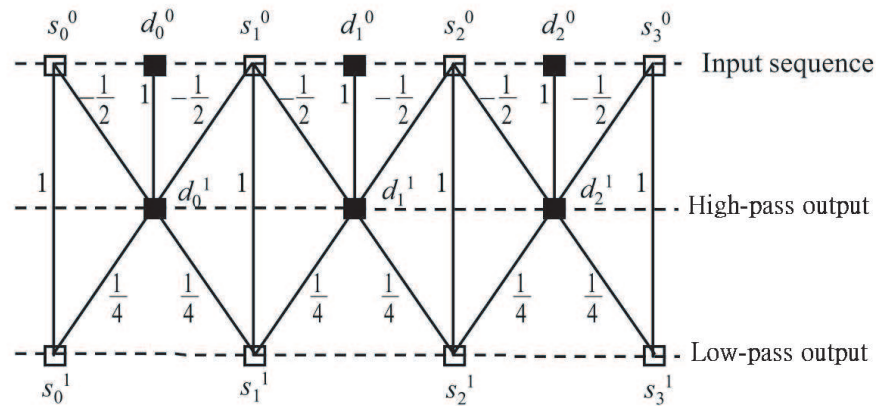


Figure 3.4: The ordering of high pass coefficients and low pass coefficients being generated.

3.2.3 EBCOT Tier-1 Coding

After DWT computation, the JPEG2000 compression enters the entropy coding, *Embedded Block Coding with Optimal Truncation (EBCOT) coding*. EBCOT coding is divided into two steps: *Tier-1* and *Tier-2*. Tier-1 coding divides the DWT coefficients to several non-overlapping blocks and then encodes each of the blocks independently; we call these blocks “codeblocks”. Besides codeblocks there are several blocks defined hierarchically for efficient coding in Tier-2. The whole data partition scenario could be illustrated in Figure 3.5. We see that the image is separated into sub-bands; then each sub-band is divided into several precincts; then each precinct is divided into several codeblocks. The codeblock size could be any size but the power of 4. However, the size often is 32x32 or 64x64 because the performance is better [2].

Since the basic coding element is codeblock, a codeblock is encoded in the elements of “bit-plane”. The three coding passes are performed to encode the bit-level data in a bit-plane; the encoding ordering in a bit-plane is followed by scanning of 4 subsequent bits as shown in Figure 3.6 [16]. The bit-plane coding is started from the most significant bit (MSB) to least significant bit (LSB) of the coefficients in this codeblock. Actually, it starts from which any bit in this bit-plane is significant. The “significant” means the first

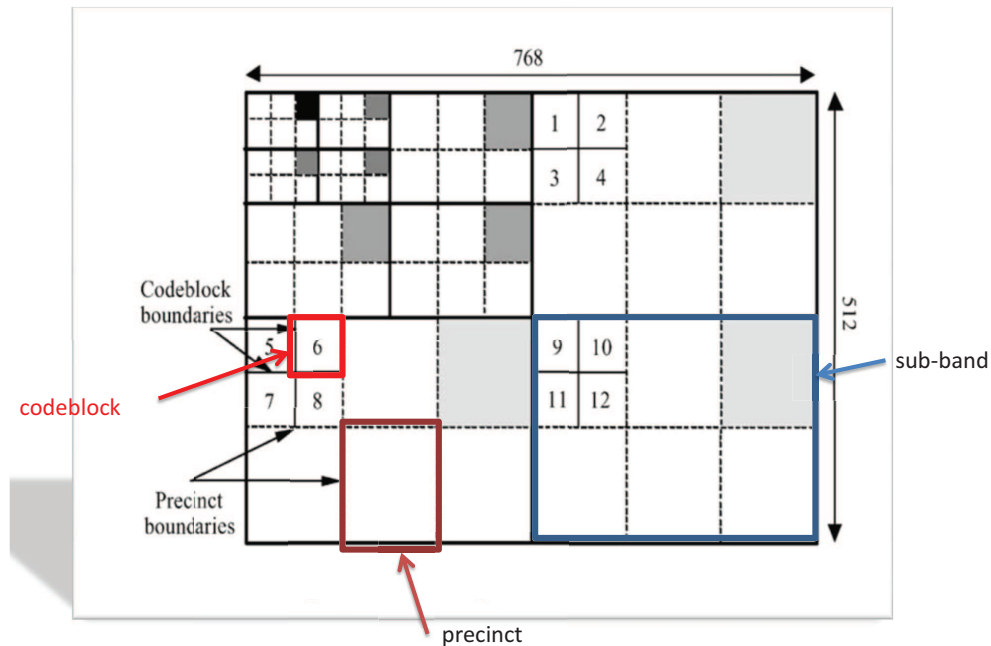


Figure 3.5: The hierarchy of data partition of an image.

non-zero bit of a coefficient, which may be 32 bits or any others. Figure 3.7 [16] is an example to show what is “significant”. The figure 3.8 [16] shows that the hierarchy of the coding elements in a codeblock. We could see that the least basic element is “a bit”.

The three coding passes performed in a bit-plane are:

- **Significant Propagation Pass (SPP):** This is the first coding pass used in one bit-plane except the first bit-plane of the codeblock. This coding pass is adapted if this bit is a preferred bit, which means eight of its adjacent bits are already in significant state.
- **Magnitude Refinement Pass (MRP):** This coding pass is applied after the first “1” bit of this coefficient has been encoded and the bit now is 1.
- **Cleanup Pass (CUP):** This coding pass is used when the bit is not encoded in SPP

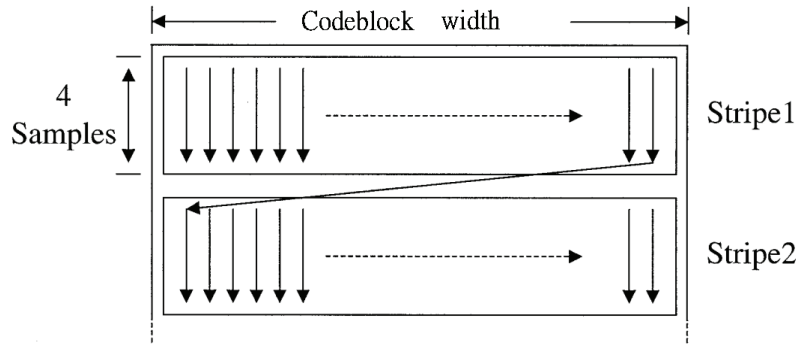


Figure 3.6: The scan pattern of a codeblock in one bit-plane.

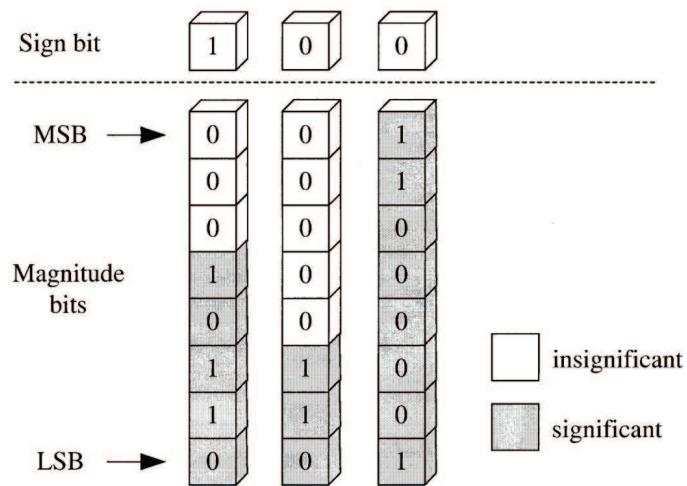


Figure 3.7: An example to show what is “significant”.

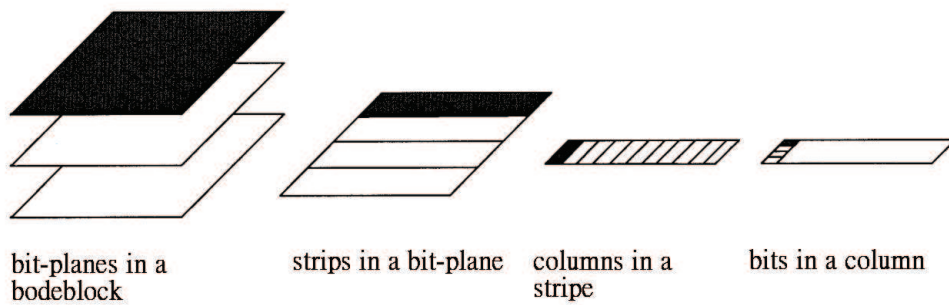


Figure 3.8: The hierarchy of bit-plane coding.

	coefficient value
coding pass	10 1 3 -7
cleanup	1+ 0 0 0
significance refinement cleanup	0 0 0 1-
significance refinement cleanup	1 0 1+ 1 1
significance refinement cleanup	0 1+ 0 1 1

Table 3.1: An example of the three coding passes being performed in every bit-plane.

and MRP except the first bit-plane. The first bit-plane starts coding from CUP.

Table 3.1 shows an example that how the 4 coefficients are encoded in every bit-plane. The every bit-plane is encoded via three coding passes and the a bit is encoded in one of the coding passes. Where a bit should be coded is following a sequence of conditional adjustments. The adjustments include four coding operations. When to use these operations bases on some conditions are satisfied. The four coding operations are:

- **Zero Coding (ZC):** ZC encodes a bit according to that if the neighbors of the bit are already in significant state. If one's neighbors are already in significant state, it is very likely to be significant.
- **Sign Coding (SC):** SC records the sign information of a coefficient and its adjacent 4 coefficients (right, left, up, down).
- **Magnitude Refinement Coding (MRC):** MRC is applied after a coefficient is already in significant state; in other words, there its first non-zero bit has already been coded.
- **Run-Length Coding (RLC):** RLC is used to encode the consecutive four bits in a vertical scanning pattern; how many bits should be encoded depends on where the first non-zero bit exists.

Since we know every bit-plane is coded in three passes with four operations, this complicated mechanism will not be detailed discussed. The detailed procedures can be found in [2].

After the three coding passes are generated, these coding passes are encoded using binary arithmetic coding, *MQ-coder*. Arithmetic coding is a superior efficient coding architecture compared to traditional Huffman coding in JPEG and can tackle binary input data. It rescales probability interval when a input datum is coming in according to the appearing probability of the datum. The arithmetic coding applied in JPEG2000 is MQ-coder. MQ-coder is a kind of adaptive arithmetic coding; it means that the encoding site changes its probability prediction synchronizing with the decoding site. The probability prediction changes following the input data with look-ups to a fixed constant table. MQ-coder divides the probability interval into two sub-intervals: more probable symbol (MPS) and less probable symbol (LPS). The two sub-intervals indicate that the input symbol, 1 or 0, which is more probable to happen. If the input symbol is in the LPS interval, the output codeword will be updated according to the estimation table.

3.2.4 EBCOT Tier-2 Coding

The purpose of Tier-2 coding is that how to efficiently organize the encoded data of Tier-1. The main works of Tier-2 are to represent the layer and block summary information for each codeblock. A layer consists of consecutive bit-plane coding passes from each codeblock in a tile, including all the sub-bands of the components in the tile. The block summary information consists of lengths of compressed code words of the codeblock, the most significant magnitude bit-plane at which any sample in the codeblock is non-zero, and the truncation point between the bit-stream layers among others [2]. Then, these information are coded by Tag Tree Coding and then put into the bit-stream. These information are important information for the reference of decoding cite.

Chapter 4

The Architecture of Analog Device BF561

In this chapter, the core architecture of Analog Device's Blackfin processor and its dual-core version—*BF561* will be introduced.

4.1 Blackfin Core

Blackfin processors are a new breed of 16-/32-bit embedded processor designed specifically to meet the computational demands and power constraints of today's embedded audio, video and communications applications. Based on the Micro Signal Architecture (MSA) jointly developed with Intel Corporation, Blackfin processors combine a 32-bit RISC-like instruction set and dual 16-bit multiply accumulate (MAC) signal processing functionality with the ease-of-use attributes found in general-purpose microcontrollers. This combination of processing attributes enables Blackfin processors to perform equally well in both signal processing and control processing applications—in many cases deleting the requirement for separate heterogeneous processors. This capability greatly simplifies both the hardware and software design implementation tasks.

As shown in Figure 4.1, Blackfin core contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit

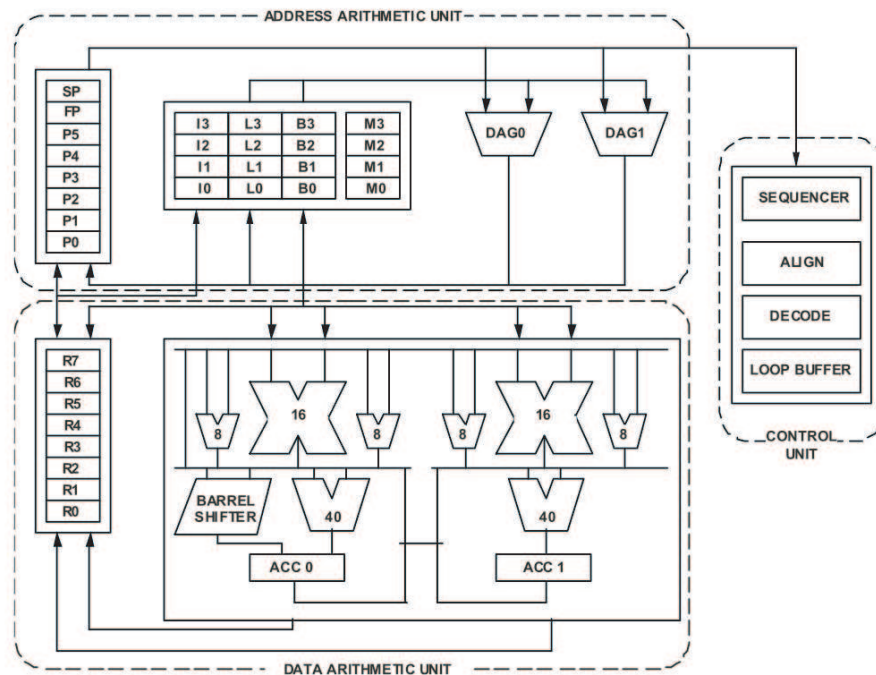


Figure 4.1: Blackfin core architecture.

shifter, along with the functional units. The computational units process 8-, 16-, or 32-bit data from the register file. The compute register file contains eight 32-bit registers. When performing compute operations on 16-bit operand data, the register file operates as 16 independent 16-bit registers. All operands for compute operations come from the multiported register file and instruction constant fields. Each MAC can perform a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. Signed and unsigned formats, rounding, and saturation are supported. The ALUs perform a traditional set of arithmetic and logical operations on 16-bit or 32-bit data. Many special instructions are included to accelerate various signal processing tasks. These include bit operations such as field extract and population count, divide primitives, saturation and rounding, and sign/exponent detection. The set of video instructions includes byte alignment and packing operations, 16-bit and 8-bit adds with clipping, 8-bit average operations, and 8-bit subtract/absolute value/accumulate (SAA) operations. Also provided are the compare/select and vector search instructions.

For some instructions, two 16-bit ALU operations can be performed simultaneously on register pairs [4].

4.2 Blackfin ADSP-BF561

ADSP-BF561 is a member of Blackfin processor family of products targeting consumer multimedia applications. At the heart of this device are two independent enhanced Blackfin processor cores that offer high performance and low power consumption while retaining their ease-of-use and code-compatibility benefits. As shown in Figure 4.2, the two Blackfin cores are connected via buses, which is a complicated bus system. In addition to L1 instruction SRAM and L1 data SRAM, there is a L2 SRAM works around half speed compared to L1 SRAM and it could be accessed by both cores.

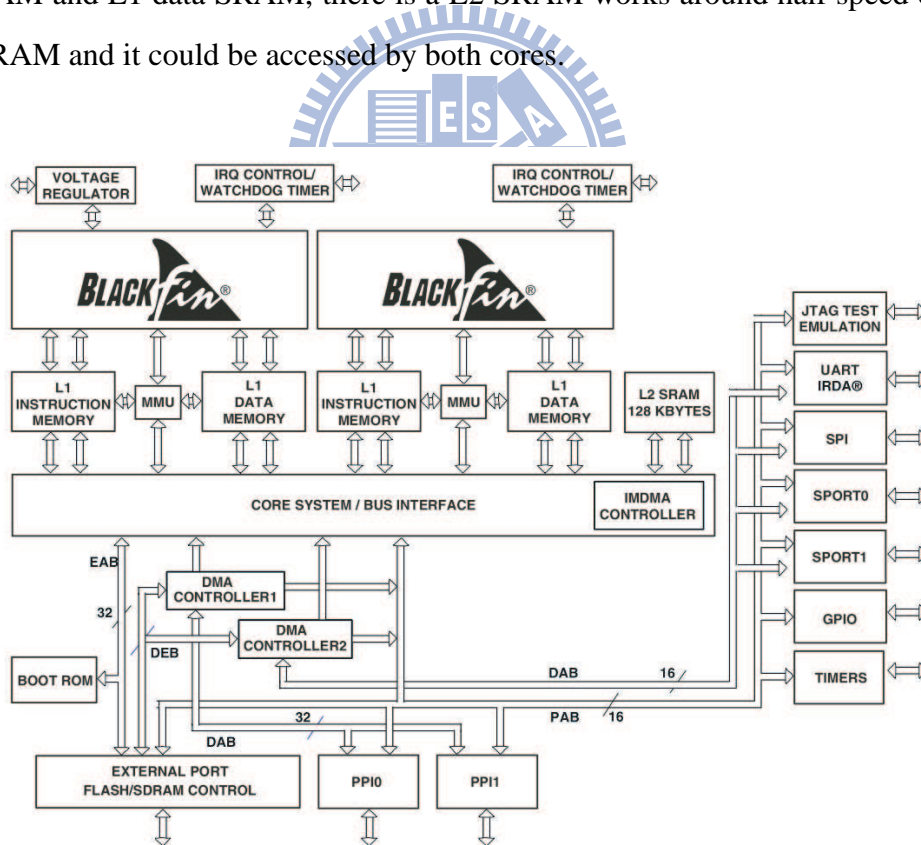


Figure 4.2: Block diagram of BF561 architecture.

4.2.1 Memory Hierarchy

Blackfin products support a modified Harvard architecture in combination with a hierarchical memory structure shown in Figure 4.2. Generally speaking, a hierarchical memory architecture means there exists multi-level memory blocks and they run under different speeds from fast to slow. The memory block near the processor core often works on the highest speed and we call it Level 1 (L1) memory. Following the principle, the follower is L2, L3,... memory. A hierarchical memory structure is designed for cost and power effective.

Level 1 (L1) memory of Blackfin BF561 operates at the full processor speed with little or no latency. At the L1 level, the instruction memory holds instructions, the data memory holds data, and a dedicated scratchpad data memory stores stacks and the information of local variables.

L1 instruction SRAM consists of 32Kb SRAM, of which 16Kb can be configured as a four-way set-associate cache. If we configure it as a general instruction SRAM, it could be put not only instructions but also data. However, the data put in the instruction SRAM can be moved only by DMA and the core can not take the data from L1 instruction SRAM directly.

L1 data SRAM consists of two banks of 32Kb each. Half of each bank is always configured as SRAM while the other half can be configured as SRAM or a two-way set associate cache. In addition, there exists a block of 4Kb L1 scratchpad SRAM, which runs at the full speed but is only accessible as a data SRAM and cannot be configured as a cache memory.

For safe memory access, the Memory Management Unit (MMU) provides memory protection for individual tasks that may be operating on the core and can protect system registers from unintended access.

The ADSP-BF561 dual cores share an on-chip L2 memory system, which provides

high speed SRAM access with somewhat longer latency than the L1 memory banks. The L2 memory is a unified instruction and data memory and can hold any mixture of code and data required by the system. It could be only configured as SRAM and cannot configured as a cache. On the other hand, it could be set to *cache-able* to data cache; this means it could be cached by the data cache. The total L2 SRAM size in BF561 is 128Kb.

The L1 instruction SRAM and data SRAM could be broken into 4Kb sub-banks, which can be accessed independently by the DMA and the core simultaneously.

External (off-chip) memory is accessed via the External Bus Interface Unit (EBIU). This 32-bit EBIU provides a glueless connection to as many as four banks of synchronous DRAM (SDRAM) and as many as four asynchronous memory devices including flash memory, EPROM, ROM, SRAM, and memory-mapped I/O devices. The PC133-complaint SDRAM controller can be programmed to interface to up to 512 MBs of SDRAM.

4.2.2 DMA Support

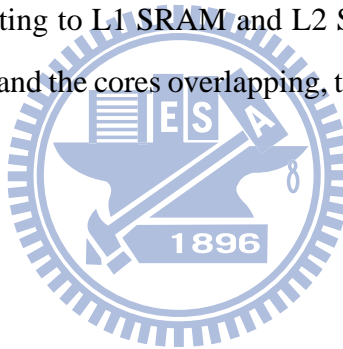
To see the architecture of ADSP-BF561, we could easily be attracted by the two DMA controllers. DMA is well known for efficient data movement, and exists not only in general CPUs but also in DSP processors. The advantage of the DMA devices in BF561 is that the buses are independent while connecting to internal L1 SRAM and L2 SRAM. This is special because most DMA devices in other processors are designed connecting to the main bus and share the bus access with processor cores and other devices connecting to the bus; that's why we say "cycle stealing". However, "cycle stealing" doesn't exist in BF561 due to the independent DMA accesses; this means the utilization of DMA on BF561 could promote higher performance.

Since we say DMA accesses to internal L1, L2 SRAM could benefit from independent buses, the access to external SDRAM is all controlled by EBIU. This seems to make no big difference between core access and DMA access. However, the DMA access could be more efficient since it works under burst read/write.

For different purposes, the DMAs on BF561 can be categorized to three functions:

- **Peripheral DMA (DMA):** It is used to transfer data between peripheral devices and internal L1, L2 SRAM
- **Memory DMA (MDMA):** It is used to transfer data between external SDRAM and internal L1, L2 SRAM.
- **Internal Memory DMA (IMDMA):** It is used to transfer data between internal L1/L2 SRAM.

The Figures 4.3 shows the bus architectures of Blackfin BF561. we could see that there are independent buses connecting to L1 SRAM and L2 SRAM. If we can manipulate the accesses by the DMA devices and the cores overlapping, the performance can be promoted.



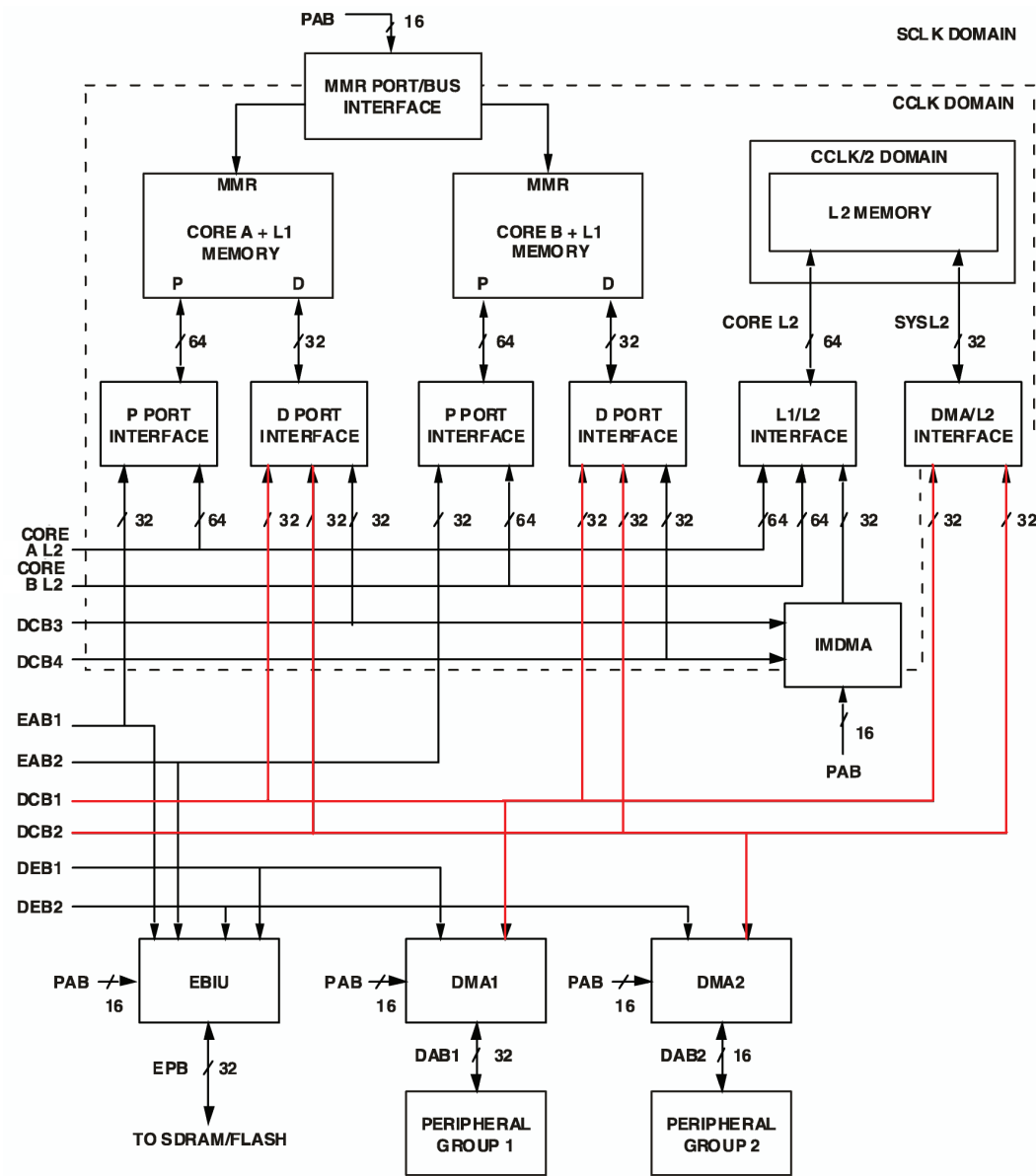


Figure 4.3: Memory and bus architecture of BF561.

Chapter 5

Implementation and Optimization

5.1 Experiment Environment Setup

There are several kinds of developing tools for us to develop our programs on BF561. The official integrated tool is Visual DSP++, which is a integrated developing environment (IDE) like ARM Developer Suite (ADS) in ARM-based environments. For more complex applications, they also developed a lightweight real-time kernel called VDK, which has many libraries for real-time applications for developers.

Instead of official tools we have another choice: GNU open-source project. In this project, we could use uClinux and GCC toolchains on Blackfin system; all the toolchains and uClinux are well supported by the community. uClinux is a lightweight version of Linux to support non-MMU processors.

We choose the open-source GNU project for our experimental environment for two reasons. First, an open-source environment is more proper for academic researches. Second, if we have Linux kernel support on BF561, we theoretically could transplant the codes from any other Linux-based platform and could exploit the library supports from Linux kernel; this is very convenient for us to develop our applications quickly since resources for Linux-based systems are easy to find on the Internet.

For dual-core BF561, uClinux could run on only one core or both cores. If uClinux runs on one core, the other core is treated as a device and could run programs through

driver supports. In addition to running on one core, uClinux also could run on both cores; it is called “*SMP-like*” mode.

Why we say it’s “*SMP-like*” is that BF561 lacks of hardware cache coherency mechanism; a “*real*” SMP must have hardware supported cache coherency mechanism. Hence, cache coherency should be done by software mechanism when needed. This implicates three significant features [5]:

- caches must be in write-through mode,
- more overhead is introduced due to software coherency mechanism, and
- all threads of a process are restricted to be executed on the same core.

Another problem is that the L1 SRAM owned by one core cannot be accessed directly from the other core so that L1 SRAM cannot be used in the kernel. Because it will cause kernel panic while the kernel threads running on one core try to access the kernel resources put in L1 SRAM of the other core. This would reduce the optimization potential because we cannot put critical system calls in the L1 SRAM to optimize Linux kernel. The developments of user space applications also have to be taken care that the user process runs on a specific core if we try to put the data or instruction codes in the L1 SRAM.

We finally configure the uClinux as SMP-like mode because a full Linux supported environment gives us a consistent environment to develop applications. There are no needs to load programs to the other core by special drivers.

5.2 Software-based JPEG2000 Implementation and Profiling

There are several projects working on open-source JPEG2000 codec. The most famous are Jasper [18] and OpenJPEG [20].

Jasper is developed and maintained by its main author, Michael Adams, who is affiliated with the Digital Signal Processing Group (DSPG) in the Department of Electrical and Computer Engineering at the University of Victoria. It is developed for the implementation of JPEG-2000 Part-1 standard (i.e., ISO/IEC 15444-1) and itself is a part of JPEG-2000 Part-5 standard (i.e., ISO/IEC 15444-5).

OpenJPEG implements not only Part-1 standard but also many other features like JP2 (JPEG2000) and MJ2 (Motion JPEG2000) file formats, JPEG2000 Interactive Protocol, and so on. It's developed and maintained by Communications and Remote Sensing Lab, in the Universit catholique de Louvain (UCL).

With the comparison of two implementations, we choose OpenJPEG for our implementation for two reasons: the source code is easy to trace and the code partition is clear.

Since the source code of OpenJPEG is well written and portable, it's not too hard to port the code onto our platform. The uClinux is also easy to configure to SMP-like mode.

Figure 5.1 shows the execution time breakdown of JPEG2000 compression on BF561; the input image is a 640x480 color image taken from OpenJPEG official site and the profiling is subject to default setting: DWT level $n=5$, codeblock $b=64 \times 64$, lossless. We see that EBCOT Tier-1 and DWT dominate the JPEG2000 compression; the two components occupy 92% loading of the whole time. Our optimizations will be focused on these two parts because they are not only the hotspot of JPEG2000 compression but also potentially parallel parts.

5.3 Overview of JPEG2000 Optimizations on BF561

5.3.1 Data Locality Optimization

After we finish kernel and JPEG2000 porting to our BF561 environment, where do we start to optimize JPEG2000? As we know, image processing often divides an image (data) into several blocks and in concept the block is a 2-D array. However, memory accesses are

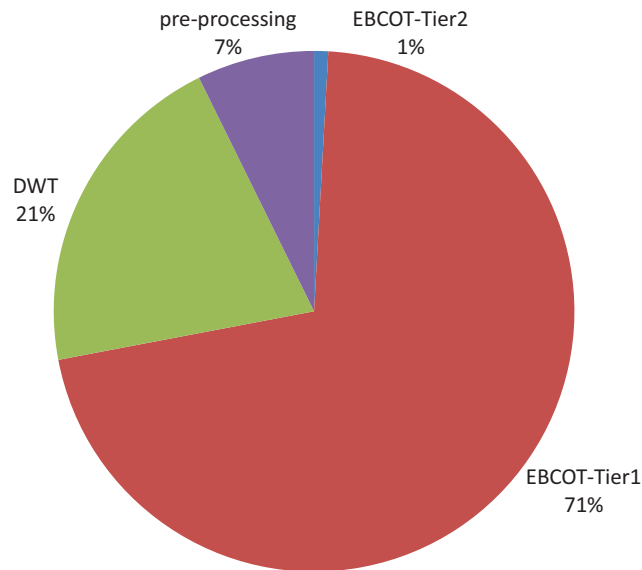


Figure 5.1: Execution time breakdown of JPEG2000 compression on the BF561 processor.

practically 1-D; hence, it will have bad performance if we don't carefully arrange the data in proper location. A big problem is the cache-miss problem. There are many researches in management of data locality in different design levels such as system-level, application-level or compiler level [26] [6] [1] [10].

As we discussed in Section 4.2.1, there are a L1 instruction SRAM, a data SRAM and a L2 SRAM companied with two DMA devices on the BF561 architecture. Now we focus on the data SRAM. Some of the L1 data SRAM can only be configured as general data SRAM rather than a cache, and therefore there is no cache-miss problem. The data SRAM works as fast as the core. This data SRAM is a precious resource for us to do the data optimization. For convenience, we simplify the term "general data SRAM" to be "data SRAM".

The best scenario for the utilization of the L1 data SRAM is that we can put all data in it to achieve best performance. However, this often doesn't happen due to the limited SRAM size. Hence, we only can move some of them into L1 SRAM; these may include

parts of the input data, output buffer, temporary data, constant data and so on.

On the other hand, we configure the parts that can be configured as a cache to be a cache because we know that this state-of-the-art mechanism could efficiently promote the performance without any software overhead. This configuration is a good choice for general utilization. However, the utilization of general SRAM depends on application developers. Hence, the utilization of SRAM is an emphasis of our optimization.

DMA is a technique designed for data moving and now almost exists in every modern CPU. There are also DMA devices in BF561 and the amount is two. Different to many other SOC and CPU designs, the two DMA devices in BF561 have independent buses and can access the SRAM in one sub-bank while Blackfin core is accessing another. Each of them has 16 channels, 4 of which could be used as Memory DMA (MDMA); it means that we could use them to move data among L1 SRAM, L2 SRAM, and external memory.

As a result, we can move data into L1 data SRAM by DMA before they are needed; then we move out these data after the processing is completed. Furthermore, it will be the best if the data moving can be overlapped with the accesses from processor cores.

The hotspot instruction codes also can be put in the instruction SRAM like we do in data. For the utilization of the instruction SRAM, GCC supports compiler intrinsics for us to put specific procedures into L1 instruction SRAM. For instance, we can simply use `__attribute__((l1_text))` to put one procedure into the L1 instruction SRAM while we are writing source code. It is put after the definition of the procedure we want to put in the L1 instruction SRAM. The following is an example to show how to use the intrinsic:

```
void foo(int a) __attribute__((l1_text));
```

The function `foo(int a)` will be allocated in the L1 instruction SRAM and the linker will maintain the linking information for the call to `foo`.

5.3.2 Utilization of Two Cores

After the discussion of SRAM, we talk about the two cores of BF561. If we could put parts of the calculating jobs onto the other core to be processed simultaneously, the performance will be promoted significantly. It is widely known that there are two ways to partition calculating jobs to multi-cores: task partition and data partition. Task partition means that many cores run different codes and the data are processed through these cores like a pipeline. Data partition means that many cores run the same code and the data are partitioned to these cores to be processed.

Similar to the principle, David J. Katz and Rick Gentile, the members of Analog Devices' Embedded Processor Application Group, use MPEG-2 as an example to show the two partition ways on Blackfin BF561 [14]. The first, as shown in Figure 5.2, is a *master-slave* model; it's similar to "data partition". In this model, the coding process is mainly controlled in master core and it spills some data to be processing in the other core. The advantage of this model is that we don't need to change codes a lot; the development procedure is just similar to the development in one core. However, synchronization overhead is needed and the slave core would not be fully loaded. As the example shown in Figure 5.2, some components of the MPEG-2 compression are parallelized to both cores and some are not. Whether the components can be parallelized may depend on their algorithms. When running the unparallelized components, the slave core is in idle state. In addition, the synchronizations are needed after some components in order to make sure that the data for their next components are ready.

The other programming model is a *pipelined* model; it's similar to "task partition" and some people call it "stream partition". As shown in Figure 5.3, the compression procedure is divided into several sub-procedures and then these sub-procedures are dispatched to two cores. If the loading of two cores are balanced enough, the idle states happening in master-slave model don't happen here. However, the whole developing procedure needs to be

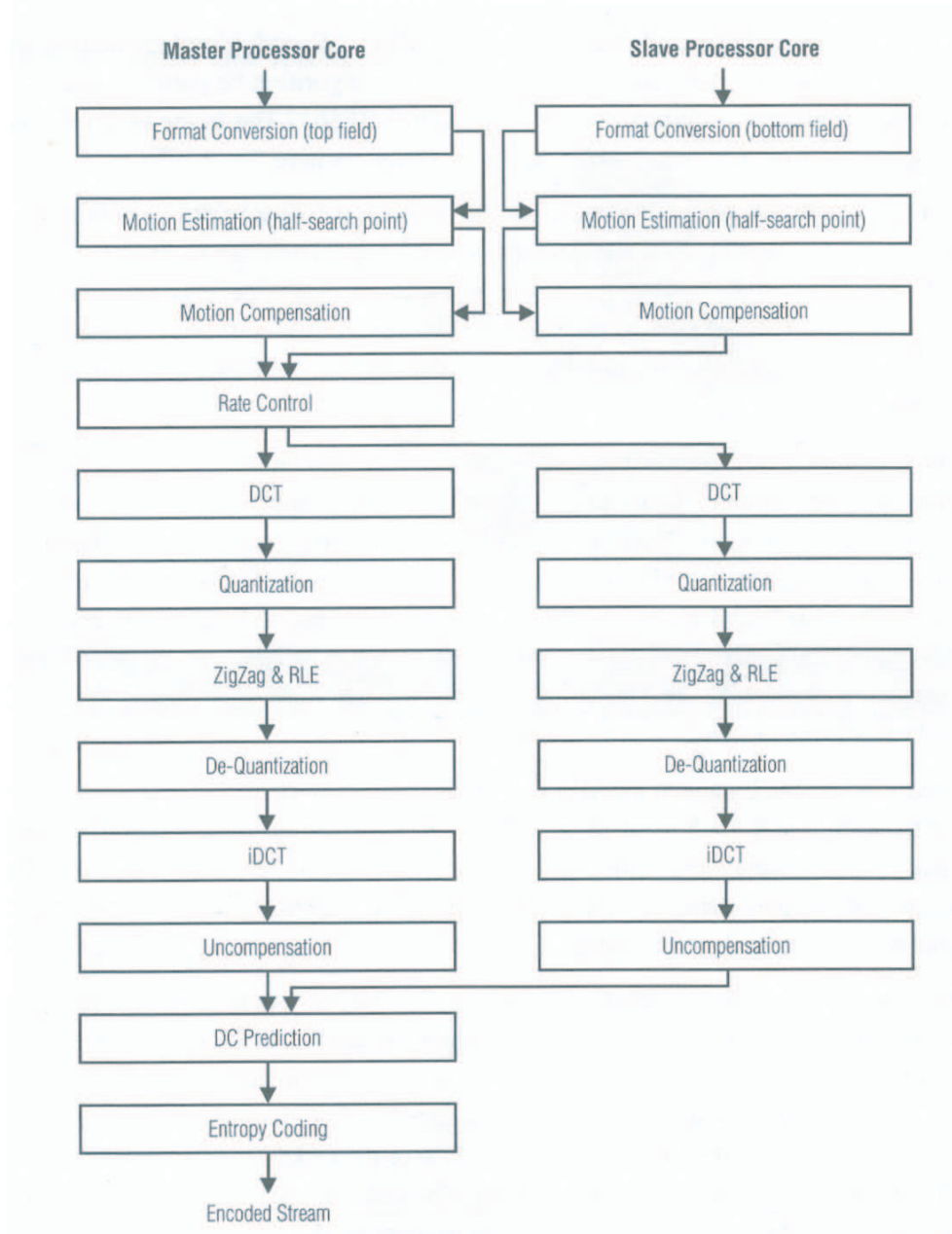


Figure 5.2: Master-slave model of MPEG-2 encoder on dual-core processors.

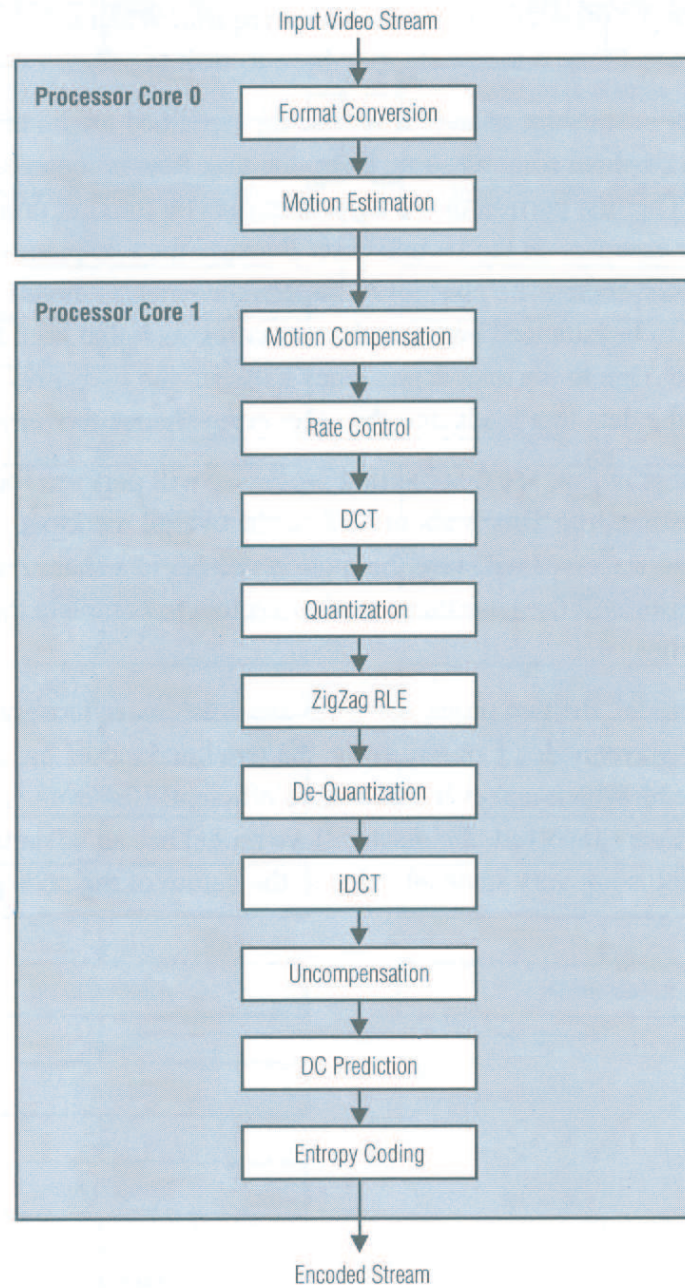


Figure 5.3: Pipelined model of MPEG-2 encoder on dual-core processors.

changed more and is not straightforward compared to which in master-slave model.

To consider these two models we choose *master-slave* model for several reasons: first, it's more scalable while the amount of hardware cores is changed; second, we could easily increase or decrease the loading of the slave core if we need to assign other jobs to the slave core; finally, JPEG2000 is hard to make balanced job partitions according to the profiling results we made, which are presented in Chapter 5.1.

5.4 Optimization of DWT

5.4.1 Data Locality Optimization

As we described in Section 3.2.2, JPEG2000 uses 2-D DWT computation to transform input image to high frequency and low frequency parts. The 2-D DWT computation is shown in Figure 5.4; we perform DWT calculation on the input image line by line in the horizontal and vertical direction, respectively.

Let's take a close look at the dataflow of the DWT computation in Figure 5.5. Before we perform one-line DWT calculation, we need to move the line data into a buffer for the processor core to do the calculation. Thanks to the well designed (5,3) lifting-based DWT, it is a "*in-place*" calculation and we only need one buffer. In general case, processor itself can do the data moving well and data cache can cache the subsequent data for potential uses. Hence, it is easy to take the following data for processing in the high speed cache memory if our data are continuous in the memory; in image processing, it means that the data are from horizontal direction. However, this would suffer problems while reading from vertical direction. Furthermore, it is wasted if we just ask the processor core to do the data moving; it should focus on calculating jobs.

In general memory device, data are practically located and moved in 1-D mode even though the high-level description is in 2-D mode. For this reason, we change our view from 2-D to 1-D to see how data are moved into and out of the buffer. Figure 5.6(a) shows

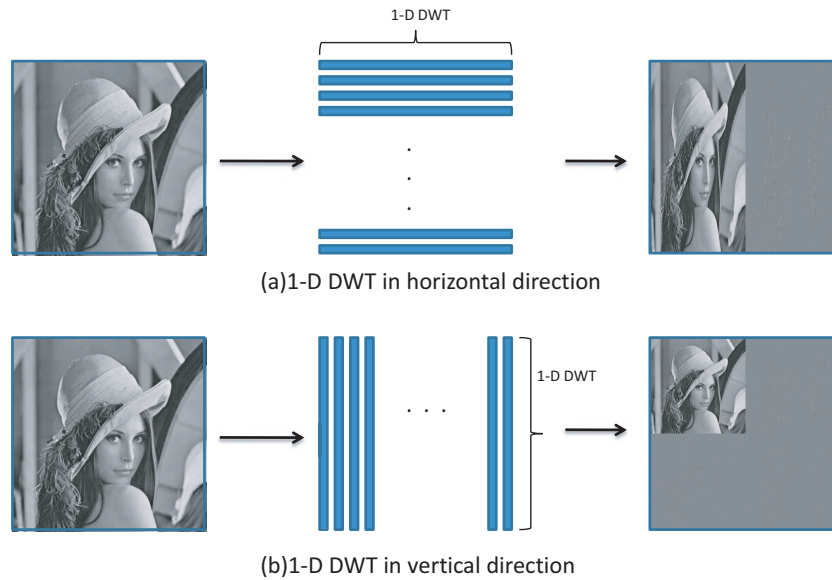


Figure 5.4: 2D-DWT computation.

the data moving scenario that how data are moved into the buffer from candidate line data in the horizontal direction. We could see that it is continuous reading while data are read to the buffer; this is the best model that cache can perform well.

Since the data is filled into the buffer, DWT computation can be performed to the data in this buffer. As discussed in Section 3.2.2, the DWT computation produces DWT coefficients and the low frequency and high frequency coefficients are regularly interleaved. After DWT computation, while the data are moved back, we have to separate the low frequency coefficients and high frequency coefficients and put them back to the correct location. How data are moved back is shown in Figure 5.6(b). We see that high frequency and low frequency coefficients are centralized to the start and the middle of the original line data, respectively.

In the vertical direction, however, the candidate line data are not continuous. As shown

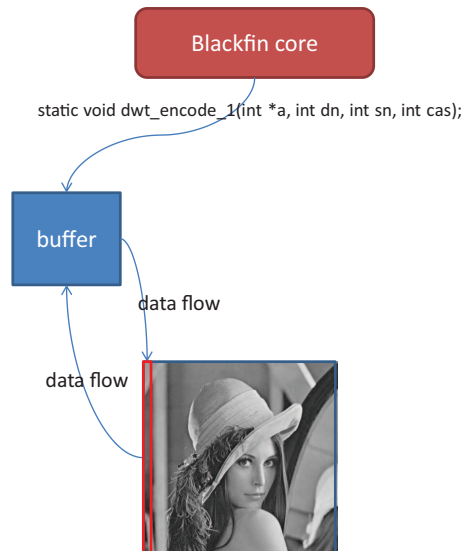


Figure 5.5: Dataflow of DWT computation performed in one line.

in Figure 5.7(a), the data read from candidate line data are periodically separated by a fixed stride; this is bad for cache to handle. On the other hand, similar to the data restoration in the horizontal direction, we need to put the interleaved low frequency coefficients and high coefficients back to the correct location. Where the data should be put back is shown in Figure 5.7(b).

Through the observation and analysis, the actions of data moving, including data moving into and out of the buffer, which are performed in the horizontal and vertical directions, can all be configured to be the jobs of DMA. The main reason about why DMA can perform these data moving is that these data moving are regular. Suppose one “data moving” consists of moving of several data elements, if the elements of the source data are regularly placed in a fixed stride and their target location are also at a fixed stride, we call the data moving “regular” and it can be performed by DMA.

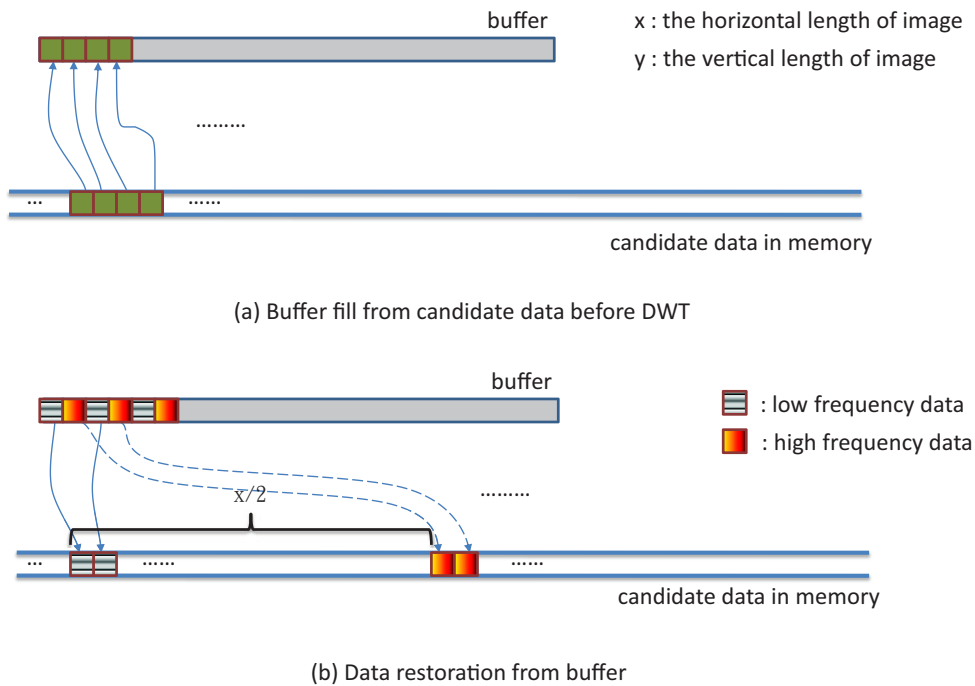


Figure 5.6: The data moving flow in a horizontal line.

As a result, we can use DMA to move data into and out of the buffer and we just put the buffer into the L1 data SRAM to be accessed in high speed clock rates. The dataflows before and after our optimization are illustrated in Figure 5.8. We add the cache into the figure to show the specialty of our optimization. We can see that our optimization bypass the cache mechanism.

Because of the frequent invocations of DMA operations, a low latency system call to configure DMA controllers is essential. For this reason, we write a lightweight system call instead of standard Linux I/O control driver and put it in the L1 instruction SRAM. In addition, thanks to the problem that L1 instruction SRAM cannot be accessed by the other core, the DMA system call is cloned to the L1 instruction SRAM of both cores in order to be accessed from both cores.

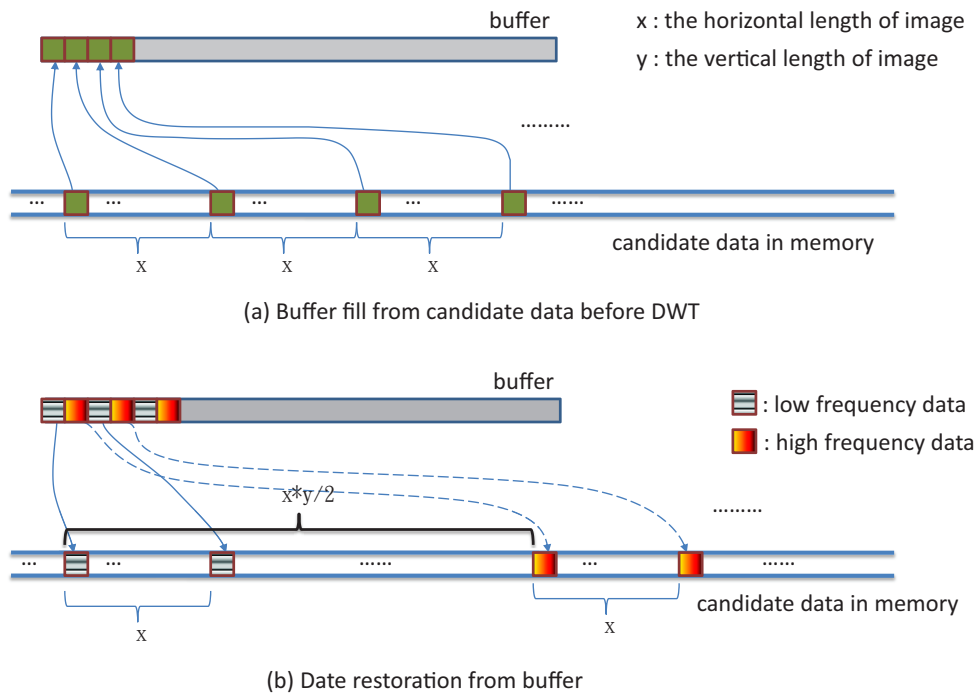


Figure 5.7: The data moving flow in a vertical line.

5.4.2 Utilization of Two Cores

5.4.2.1 Data Partition

After the discussion of optimization using DMA and internal SRAM, we discuss how to partition the calculation jobs to the other core. As we mentioned in Section 5.3, we use data partition to spread the half of the data to the other core to speed up the calculation. The fact that L1 SRAM cannot be accessed by the other core would still be a problem at this moment. This enforces us to bind the user process to one of two cores; this means that we should enforce the Linux kernel to schedule the process on only one core. This could be achieved by system call `int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)`. Another problem is that on BF561 a thread can only run on one core with its process due to the lack of hardware cache coherency. As a result, we have to fork a new process and bind it to the other core to help us share calculations. The new process is

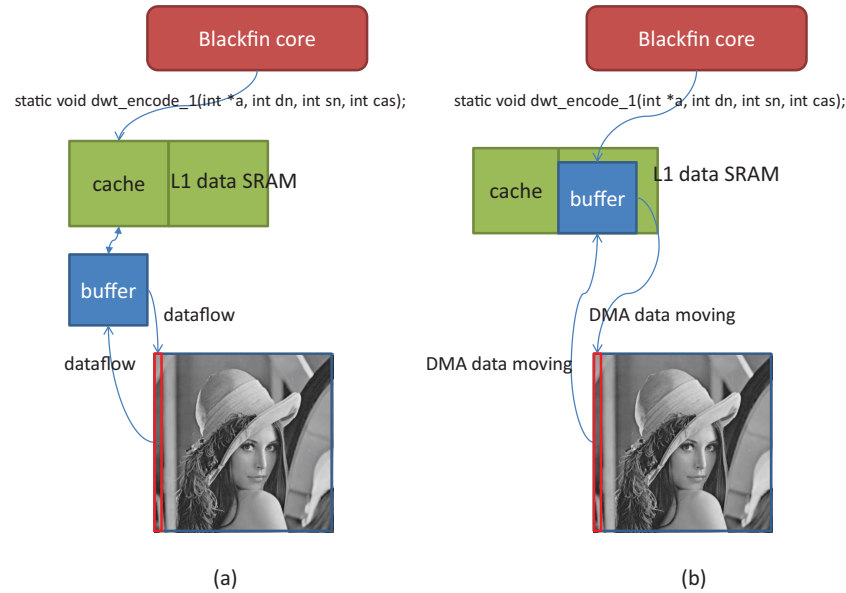


Figure 5.8: Dataflow of DWT computation performed in one line: (a) before data locality optimization (b) after data locality optimization.

generated after performing *vfork()* and *exec()* families.

The only parameter needed to pass to the new process is the address of the tile address (which is the start address of the image in our scenario); it is put in shared L2 SRAM. L2 SRAM is now used as a shared memory for us to communicate between two cores. As we discussed, the DWT performed in each line is independent; hence, we divide the line data of the same direction at the same level into two groups: half front parts and half back parts, and perform DWT on different cores as shown in Figure 5.9. Until the computations of dispatched jobs at both cores are finished, the next stage, which may refer to different direction or the next level, are not allowed to start; this means that the synchronization is needed here to make sure both cores finish their jobs. We use shared variables for synchronization; they are placed in shared L2 SRAM.

However, we find that the data partition to two cores is inefficient. The main reason is that the loading of data transfer is heavy—more discussions will be given in Section 6.1.

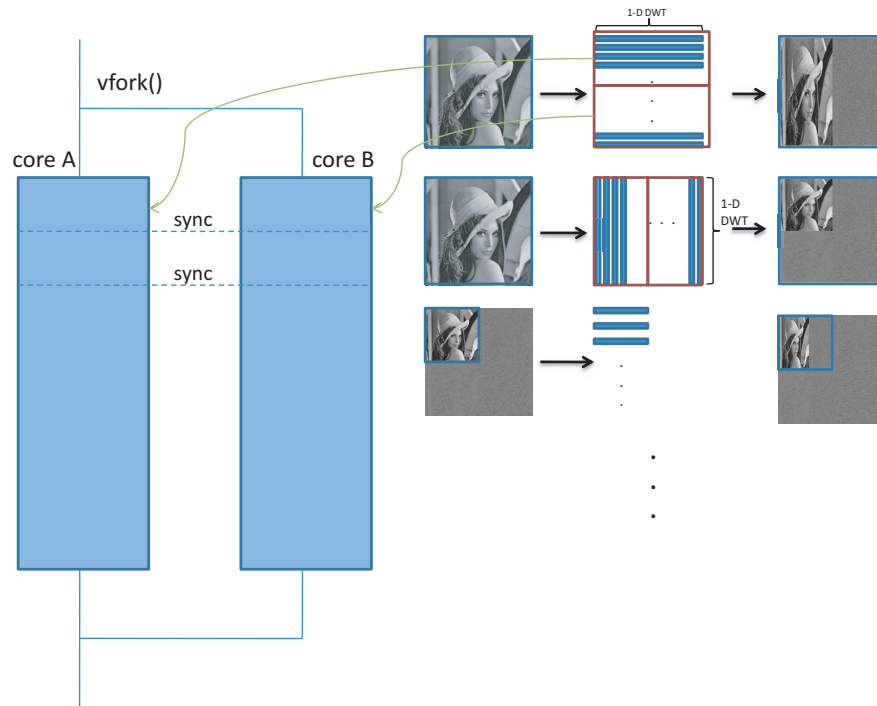


Figure 5.9: The data partition to two cores.

Hence, we propose another method to partition jobs of DWT. This is discussed in the following subsection.

5.4.2.2 Task Partition

Due to the heavy loading in data transfer, we try to partition DWT computations in another way (we name two cores as CoreA and CoreB for discussion); we try to partition jobs between memory transfers and DWT computations themselves. We ask CoreA to focus on DMA control; CoreA is responsible to control DMA to move the data into L1 data SRAM of CoreB, and CoreB focuses on the DWT computation but needn't to care about the data transfer. As discussed in Section 4.2.1, the data transfer using DMA control and core access could be overlapped; this makes the memory/calculation partition possible. The scenario can be shown in Figure 5.10. We finally sum up our optimizations with Algorithm 1 and Algorithm 2, which show what CoreA and CoreB do, respectively. As we discussed above,

CoreA moves data and CoreB does the computations. For the cooperations of two cores, we set a synchronization mechanism to make sure that each DWT computation starts after the completion of moving out of the old data and moving in of the new data. The experimental results presented in Section 6.1 shows that the performance of task partition is better than that of data partition for the DWT process.

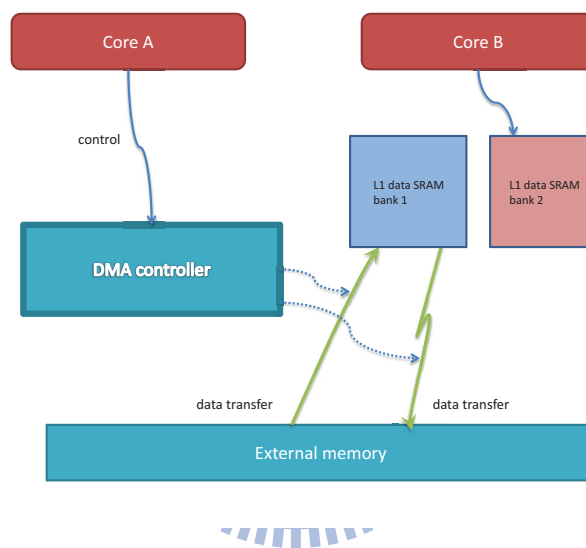


Figure 5.10: The memory/calculation partition to two cores .

Algorithm 1: DWT: The workload of CoreA

Input: three components of the input image: Y, U, V**Output:** DWT coefficients

```

for each component of the input image do
  for each resolution level do
    for each sub-band do
      for each 2 vertical lines of the input image do
        move the old data out;
        move the first line data into Bank-1 of data SRAM of CoreB;
        sync_1; //notify CoreB data in Bank-1 are ready;
        move the old data out;
        move the second line data into Bank-2 of data SRAM of CoreB
        sync_2; //notify CoreB data in Bank-2 are ready;
      end
      for each 2 horizontal lines of the input image do
        move the old data out;
        move the first line data into Bank-1 of data SRAM of CoreB
        sync_1; //notify CoreB data in Bank-1 are ready;
        move the old data out;
        move the second line data into Bank-2 of data SRAM of CoreB
        sync_2; //notify CoreB data in Bank-2 are ready;
      end
    end
  end
end

```

Algorithm 2: DWT: The workload of CoreB

Input: three components of the input image: Y, U, V**Output:** DWT coefficients

```

sync_1; //wait Bank-1 data ready;
DWT_1D(bank-1); //perform DWT put in Bank-1;
sync_2; //wait Bank-2 data ready;
DWT_1D(bank-2); //perform DWT put in Bank-2;

```

5.4.3 DMA Optimization

Because of the frequent invocations of DMA system calls, how to reduce the call latency becomes an important issue. In this subsection, we discuss how to optimize DMA system calls.

The data transfer will start after the DMA is well configured; the configuration is done by user programs through a system call. If we return the system call just after the completion of configuration, we can do other things while data transferring. We just have to keep in mind that we should compute the data after the data are really ready. The completion of data transfer can be known by polling the completion register of DMA.

Figure 5.11 shows the latency comparison between the system call of DMA configuration and data transfer. We can see that whatever the sizes of data to be transferred are, a fixed period of time for configuration is needed. This fact implies that we have to make the data size to be transferred in one DMA system call as large as possible. This makes it more possible to do other things while the data is transferring.

Another optimization strategy is that we can use *descriptor-mode* DMA. The basic method to configure DMA controllers is to fill the parameters into the control registers of DMA controllers directly. However, every time when we need the next transfer, we have to activate the system call once. The DMA controllers of BF561 provide a mechanism to reduce the number of DMA system calls: *descriptor mode*. “Descriptor” means a data structure put in the memory, which describes the DMA configuration. Then we just tell the DMA controllers the address of the descriptor and the DMA controllers will fetch the descriptor and configure itself; hence, we can quickly return back to our user program from the system call. Furthermore, there is a special field in the descriptor structure which keeps the address pointing to the next descriptor. This means we can put many configurations in the memory and link them together if we could know the configurations of following DMA operations, as shown in Figure 5.12. The DMA controllers automatically perform

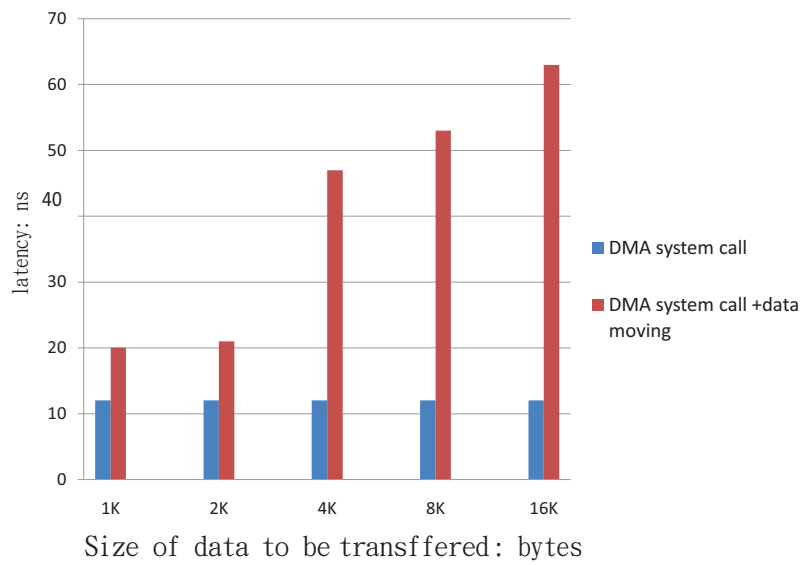


Figure 5.11: Latency of DMA transfer in continuous data.

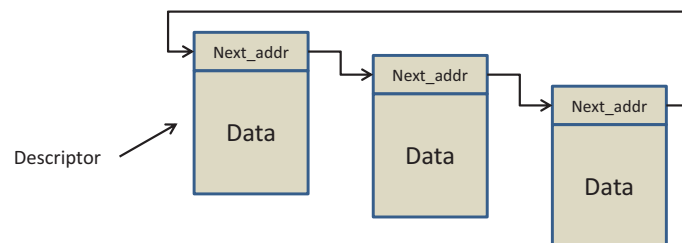


Figure 5.12: Linking of DMA descriptors.

these DMA transfers.

Because of the convenience described above, we optimize our DMA system calls using the descriptor-mode DMA. Originally, we need three system calls for one DWT computation: data moving in, low frequency data moving out and high frequency data moving out. Instead of direct DMA register filling, we write three descriptors for data moving and link them together. Then, these data moving will be automatically performed with only one system call and the system call is quickly returned because it only passes the address of the first descriptor to the DMA controller. Our experimental results presented in Section 6.1 shows that the total data transfer time can be aggressively reduced due to the latency of DMA configuration is reduced.

5.5 Optimization of EBCOT Tier-1

5.5.1 Data Locality Optimization

As we discussed in Section 3.2.3, we know the basic coding unit in Tier-1 is *codeblock* and every codeblock is coded independently. This gives us very good beginning to think the optimization methods. We try the same strategy to put the data in the L1 SRAM to improve performance.

Before the optimization, how do we decide codeblock size? Theoretically the codeblock size can be any number which is the power of 4. However, 32x32 and 64x64 are often chosen according to practical experience [2]; OpenJPEG also only supports these two sizes. The choice of the different sizes will affect the essential buffer size for codeblock data and total final codeblock numbers. According to our experiments, the coding time of EBCOT Tier-1 under 32x32 and 64x64 is approximately the same; however, the coding time in EBCOT Tier-2 under 32x32 is almost three times than under 64x64. The main reason is that there are much more codeblocks that Tier-2 has to arrange into the final bit-stream. As a result, we adapt 32x32 for the codeblock size in Tier-1 coding.

Unlike DWT computation, the data for EBCOT Tier-1 computation are complex and many constant data and intermediate buffers are needed. We detailedly trace the OpenJPEG implementation of EBCOT Tier-1 and collect the data structures essential for the coding. The essential data for Tier-1 coding can be categorized to several types according to their functionality : input data (DWT coefficients), output buffer (the codewords generated by MQ-coder), structures for handling EBCOT Tier-1 coding, and essential constant data. To sum up the essential memory space for putting these data structures, we need to adjust some of the data SRAM from a cache to a general data SRAM.

As we discussed in Section 4.2.1, the data SRAM is composed of two banks: bank-1 and bank-2. The half of bank-1 and the half of bank-2 can be configured as a cache, respectively. There is no apparent difference in performance under our experimental scenario while we configure the half of bank-2 as a cache or as a data SRAM.

After the adjustment of configurations, we can put all essential data structures into the L1 data SRAM. We carefully arrange them explicitly in the L1 data SRAM and sum up them as listed in the Table 5.1. The address of each data structure is free chosen. Only the size is taken care. The overlapping of DMA operations and core accesses is not considered here because that the space of SRAM is limited and the loading of data moving is not heavy.

After the data arrangement is completed, we discuss the utilization of the L1 instruction SRAM. The total instruction codes of EBCOT Tier-1 are too bulky to be put in the instruction SRAM. Hence, we have to choose some of them to put in it. As we discussed in Section 3.2.3, MQ-coder is used to encode each of the three encoding passes after each of them is generated. This module is a hotspot in Tier-1. Hence, we put the whole codes of MQ-coder in the L1 instruction SRAM to achieve high performance with the compiler intrinsics supported by GCC.

Table 5.1: The data allocation of EBCOT Tier-1 in the L1 data SRAM.

Procedure name	Address	Size (byte)	Function description
opj_t1_t *t1	0xff800000	40 (0x28)	The struct to handle the EBCOT-Tier1 coding
struct opj_mqc *mqc	0x ff800028	104 (0x68)	The struct to handle the MQ-coder
t1->data	0xff904000	16384 (0x4000)	To put the DWT coefficients, which is the input data of EBCOT-Tier1
t1->flags	0xff800090	8712 (0x2208)	The variables to record the bit status using in bit-plane coding
lut_ctxno_zc	0ff802298	1024 (0x400)	Related table for bit-plane coding
lut_ctxno_sc	0xff802698	256 (0x100)	Related table for bit-plane coding
lut_spb	0xff802798	256 (0x100)	Related table for bit-plane coding
lut_nmsedec_sig	0xff802818	128 (0x80)	Related table for bit-plane coding
lut_nmsedec_sig0	0xff802898	128 (0x80)	Related table for bit-plane coding
lut_nmsedec_ref	0xff802918	128 (0x80)	Related table for bit-plane coding
lut_nmsedec_ref0	0xff802998	128 (0x80)	Related table for bit-plane coding
opj_mqc_state_t mqc_states	0xff802f78	1504 (0x5e0)	The probability prediction table for MQ-coder
unsigned char *output_data	0xff900010	8197 (0x2005)	The output data of EBCOT-Tier1

5.5.2 Utilization of Two Cores

Now we discuss the utilization of two cores. Since we know that every codeblock is coded independently, we dispatch these codeblocks to two cores to compute without synchronization until their dispatched jobs are finished. The implementation details are similar to what we do in DWT; we use *vfork()* and *exec()* mechanisms and pass the parameters to the other core via shared variables put in shared L2 SRAM. The parameters need to pass to the other core are the address of tile, tile encoding handler, and a high level encoding handler. Finally, we sum up our optimizations using pseudo codes. Suppose the two cores are CoreA and CoreB, Algorithms 3 and 4 show the high level pseudo codes of Tier-1 coding on CoreA and CoreB, respectively. The jobs of two cores are similar. The only difference is that CoreA encode odd codeblocks and CoreB encode even codeblocks.

Algorithm 3: The workload of CoreA

Input: DWT coefficients

Output: MQ-coder codewords

allocate the structures for handling the Tier-1 coding;

```

for each component of the input image do
  for each resolution level do
    for each sub-band do
      for each precinct do
        for each odd codeblock do
          move the DWT coefficients into the L1 data SRAM;
          encode codeblock;
          move out the codewords generated by MQ-coder;
        end
      end
    end
  end
end

```

Algorithm 4: The workload of CoreB

Input: DWT coefficients**Output:** MQ-coder codewords

allocate the structures for handling the Tier-1 coding;

```

for each component of the input image do
  |
  | for each resolution level do
  | |
  | | | for each sub-band do
  | | | |
  | | | | | for each precinct do
  | | | | | |
  | | | | | | | for each even codeblock do
  | | | | | | | |
  | | | | | | | | | move the DWT coefficients into the L1 data SRAM;
  | | | | | | | | | encode codeblock;
  | | | | | | | | | move out the codewords generated by MQ-coder;
  | | | | | | | | end
  | | | | | | | end
  | | | | | | end
  | | | | | end
  | | | end
  | | end
  | end
end

```

5.6 Optimization Using Inline Assembly

Inline assembly is a technique for embedding assembly code into high level source codes like C language. Why we need this technique is that sometimes compiler may generate inefficient codes. We could write an inline function written in assembly to replace a critical code fragment in order to achieve better performance—hand-written assembly code usually have better performance than compiler-generated code. Inline assembly is commonly used in Linux kernel; some hotspot system calls, ISRs or hardware related critical codes are directly written in assembly to achieve the best performance. Do we need to “reassemble” assembly codes during the developments of user applications? In most scenarios, we don’t think this is necessary.

As we know, RISC architecture is friendly for compiler designs. In addition, the researches for compiler optimizations on RISC architecture have been very mature. We often

don't "reassemble" assembly codes by hand because it's a bulky work and we often might obtain poor performance than compiler do. High level programmers just need to focus on their application developments.

However, Blackfin core is a little different compared to general RISC architecture; mathematic calculations and memory accesses can be performed simultaneously on Blackfin.

Because of the complex architecture of Blackfin core, the compiler designs for Blackfin architecture are a little difficult than for RISC general architecture. Hence, we are more likely to reassemble assembly codes to run faster than compiler do. In the JPEG2000, MCT, which is dedicated for color space transformation as described in Section 3.2.1, is chosen to do the inline assembly optimization since it runs a small piece of codes thousands of times. This gives us a big chance to obtain high performance improvements.



Chapter 6

Evaluations and Discussions

In this chapter, we will evaluate the performance of our optimizations presented in Chapter 5 and do some discussions to the results. The image Bretagne1.bmp taken from the official website of OpenJPEG is used for evaluation. In addition, four popular image benchmarks will be adapted to evaluate our optimizations. All our experiments are completed on the development board: ADSP-BF561 EZ-KIT Lite, running on the uClinux operating system which is configured in SMP-like mode.

6.1 Evaluations and Discussions of DWT

Table 6.1: DWT: The evaluation of performance improvements of two optimizations: data locality optimization and utilization of two cores.

Data locality optimization	Performance improvement
L1 data SRAM with DMA	65%
L1 instruction SRAM	-2%

Utilization of two cores	Performance improvement
Data partition	8%
Data partition with data locality optimization	-4%
Task Partition (mem/cal partition)	30%

This section we evaluate the performance of our optimizations to DWT, which are proposed in Section 5.4. The first is the data locality optimization and the other is the utilization of two cores.

Table 6.1 shows the performance improvements of the two optimizations with the baseline configuration of -O3 optimized DWT running on single core.

As shown in the upper part of Table 6.1, we obtain 65% performance improvement using the data locality optimization. We now further investigate how our optimizations reduce the time consumption. As discussed in Section 5.4, we move the data into the buffer, and then do the 1-D DWT computation. Finally, we move back the data from the buffer.

Now we examine the time consumption of DWT in three parts: data transfer in the vertical directions, data transfer in the horizontal directions and DWT computations themselves. As shown in Figure 6.1, before our data locality optimization, the data transfer in the vertical directions occupies the most of time due to mass cache misses. However, this latency is apparently reduced after our data locality optimization has been applied. This implies that we improve severe cache-miss problems. In addition, our optimization can slightly reduce the computations themselves as well as horizontal data moving.

On the other hand, there is no improvement while we put the DWT computation code in the L1 instruction SRAM. In fact, the performance looks worse (2% degradation) if we do it. The reason is that the intrinsics GCC support under this development environment could't well support using L1 instruction SRAM; this will be discussed in the Section 6.2.1.

After the discussion of L1 optimization, we evaluate the utilization of two cores. The bottom part of Table 6.1 shows that the performance of the data partition is not good whether the data locality optimization has been performed first or not. The reason is that the loading of data transfer is too heavy. As shown in Table 6.2, we find that the whole DWT computation procedure spends almost 4/5 of time in data transfer. Even after our

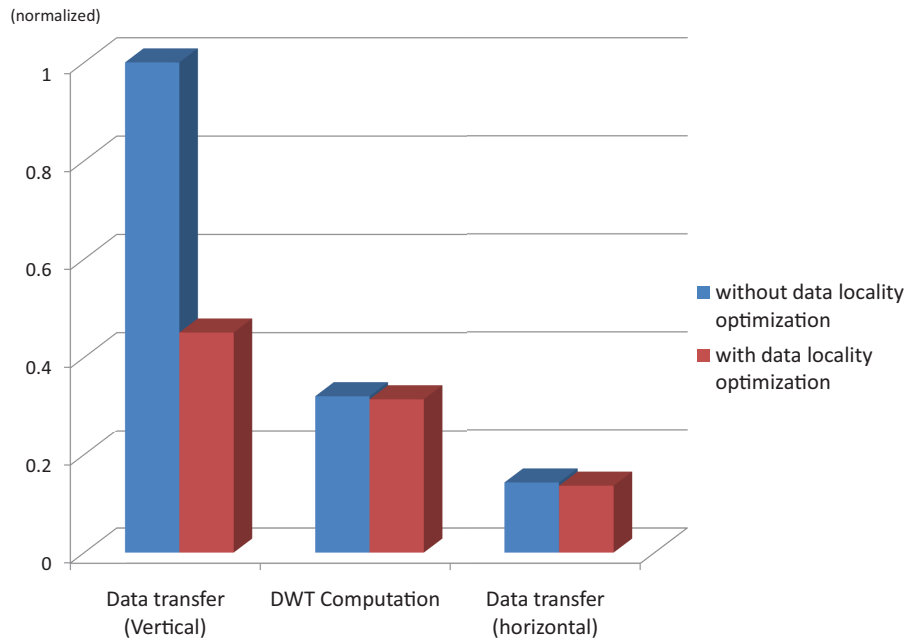


Figure 6.1: The analysis of time consumption in DWT before and after data locality optimization.

optimization, the data transfer still occupy approximately 2/3 of time. We think this is an important point why we cannot benefit from data partition to two cores. Another problem about why we cannot speed up the process is the synchronization problem. Every time when synchronization is needed, we have to wait the slow core to finish its jobs and then we can keep on going. Because of the heavy loading of data transfer, *memory transfer/calculation (mem/cal) partition*, which is proposed in Section 5.4.2.2, is adapted to promote the utilization of two cores.

Figure 6.2 shows that the time consumption of two cores working under mem/cal partition. Due to the inconsistent completing time of two cores' jobs, waiting for the synchronization is necessary. This analysis can be a reference for further optimizations. We see that the main idle time is when CoreB waits for CoreA to complete the data transfer in the vertical direction. After our optimization using descriptor-mode DMA, the latency of data transfer can be aggressively reduced about 10% compared to traditional register-mode

DMA.

Table 6.2: The loading comparison between memory transfer and computations in DWT.

	Data Transfer	DWT computation
vertical	68.6%	11.3%
horizontal	9.8%	10.3%

(a)without DMA and L1 optimization

	Data Transfer	DWT computation
vertical	50%	17.9%
horizontal	15.1%	17%

(b)with DMA and L1 optimization

The experimental results of the incremental optimizations are illustrated in Figure 6.3. The best speed-up can be achieved via the combinations of following optimizations: task partition (mem/cal partition) to two cores, L1 data SRAM optimization and using the descriptor-mode DMA. It can achieve totally up to 2.45x compared to the baseline, which is the original OpenJPEG implementation with -O3 optimization under single core.

6.2 Evaluation and Discussion of EBCOT Tier-1

In this section we evaluate the performance of our optimizations to EBCOT Tier-1, which are proposed in Section 5.5. The first is the data locality optimization and the other is the utilization of two cores.

Table 6.3 shows the performance improvements of the two optimizations with the baseline configuration of -O3 optimized EBCOT Tier-1 running on single core.

We see that the performance improvement of data locality is around 12%, which is not as good as what we got in DWT. According to our analysis of the coding algorithm of EBCOT Tier-1, the bottleneck of Tier-1 coding is the bit-level computations and complex

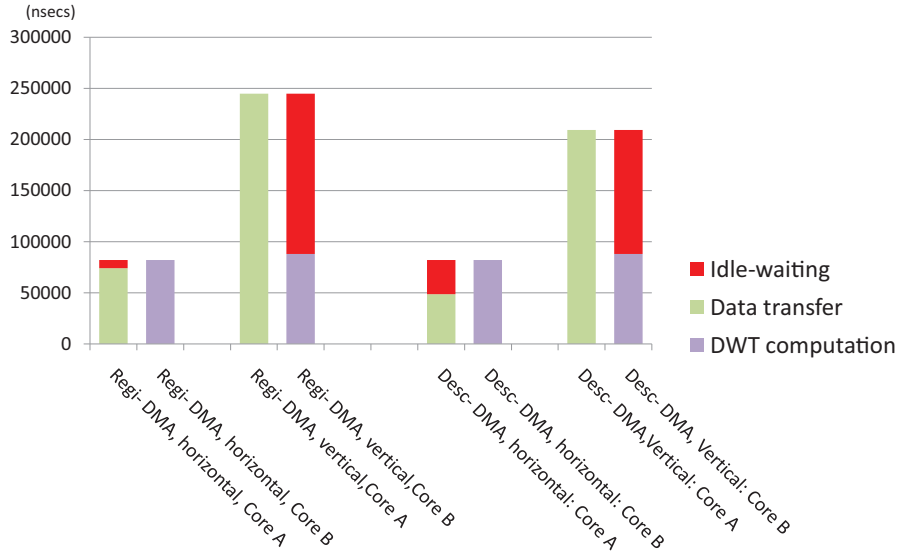


Figure 6.2: The analysis of execution time of DWT using mem/cal partition.

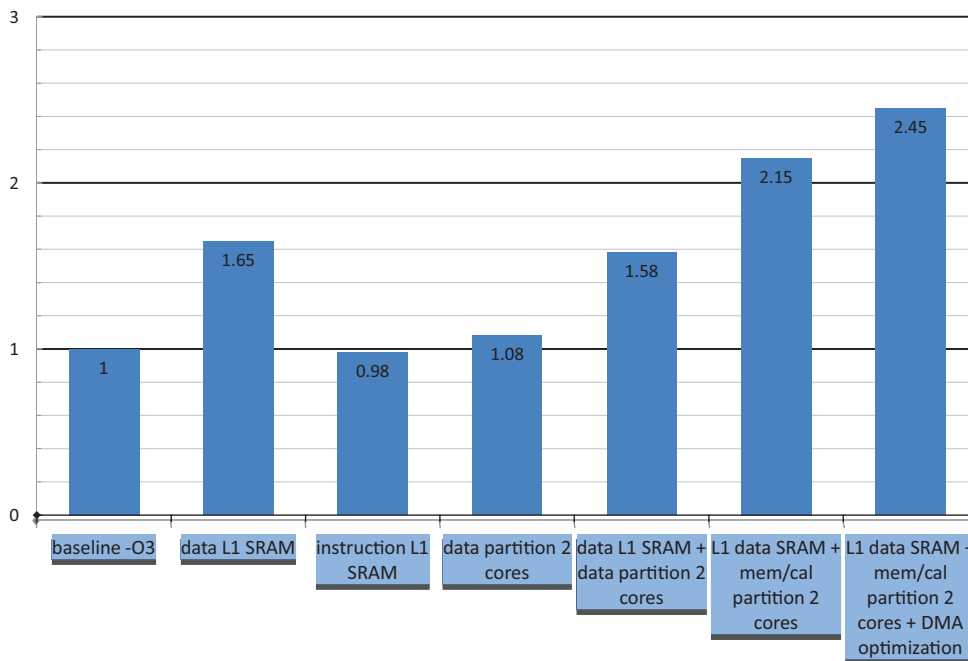


Figure 6.3: Speed-up of the proposed optimizations for DWT.

Table 6.3: EBCOT Tier-1: The evaluation of performance improvements of two optimizations: data locality optimization and utilization of two cores.

Data locality optimization	Performance improvement
L1 data SRAM with DMA	12%
L1 instruction SRAM	-8%

Utilization of two cores	Performance improvement
Data partition	7.4%
Data partition with data locality optimization.	71%

control dependencies.

Bit-level calculation is a problem not only to DSP processors but also to general processors. To speak in general, “processors” are not designed to conduct bit-level data; they are designed to tackle data in “variable” level. This is difficult for us to do aggressive optimizations. In addition, MQ-coder also needs to do the comparisons and jump operations frequently. These two reasons infer that why the data locality optimization is not so obvious than in DWT.

On the other hand, we see that we cannot benefit from using the L1 instruction SRAM. The performance is worse than we don’t do it. The main reason is that the toolchains from the board support package (BSP) only support compiler intrinsics to put “a procedure” into the L1 instruction SRAM. This action would violently disturb the inter-procedural optimizations of the compiler. Our experiments show that almost none of procedures can benefit from being put in the L1 SRAM. Hence, this optimization is abandoned. How compiler optimizations are disturbed is discussed in the Section 6.2.1.

As to two cores’ partition, as we discussed in Section 5.5, we dispatch the independent codeblocks to both cores to promote the performance. As shown in Table 6.3, we found an interesting phenomenon. We found that the performance promotion of two core’s utiliza-

tion with data locality optimization is much more than that without data locality optimization. Table 6.3 shows that there is only 7.4% of performance improvement if we would not perform the data locality optimization first. However, we get 71% of performance improvement if we did that first. This can be explained that the data locality optimization reduces the heavy loading of external memory accesses. As a result, the L1 data optimization is not only an “additional” optimization but also a “must” optimization. This fact makes us to take care that data locality optimization always has to be done before consideration of utilization of two cores.

The experimental results of the incremental optimizations are illustrated in Figure 6.4. We see that we can only apparently improve the performance by combining the data locality and data partition to two cores. The overall speed-up is around 1.89x.

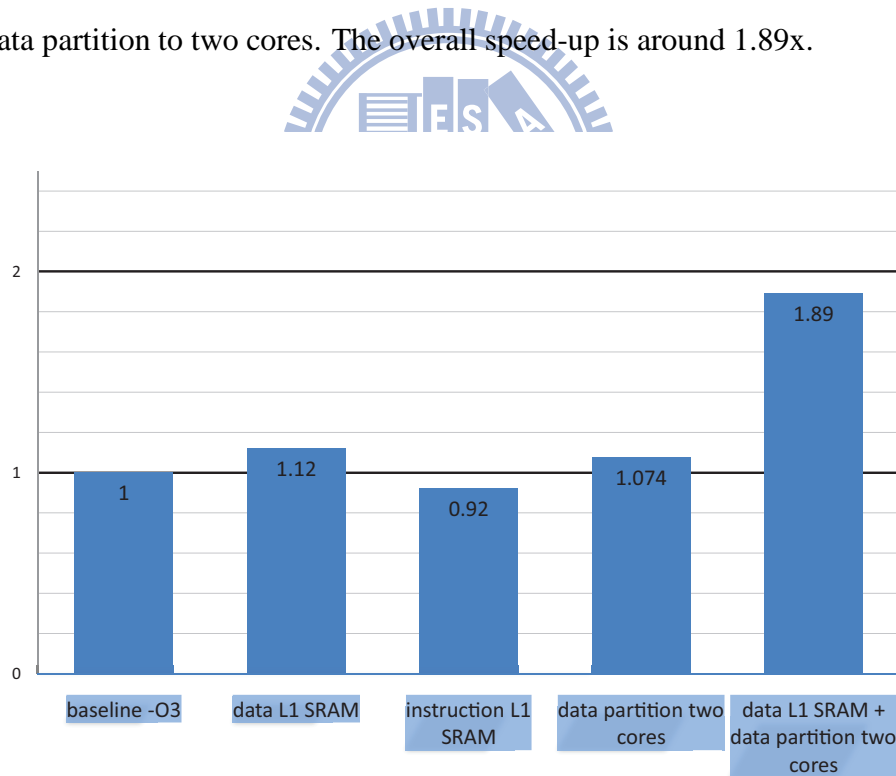


Figure 6.4: Speed-up of the proposed optimizations for EBCOT Tier-1.

6.2.1 The Disturbance of Compiler Optimizations due to Putting Procedures to the L1 Instruction SRAM

Since we got a bad performance by using GCC intrinsics to put procedures in the L1 instruction SRAM, we doubt that this optimization disturbs the inter-procedural optimizations in compilers. To confirm our doubt, we write a very simple program to show that how a GCC intrinsic, which arranges a specific procedure to be put in the L1 instruction SRAM, disturbs the inter-procedural optimizations in compilers. The program is shown in Figure 6.5. The *main()* calls the *fun_a()* and then *fun_a()* calls *fun_b()*, and the input variable *d* of value 0 will be passed to *fun_a()* as the parameter. The value will be added by one by *fun_a()* and by *fun_b()*, respectively, and then variable *c* will catch the return value; so the final result is $c = 2$. Figure 6.6 shows the assembly that compiler translated. The left side is the original code with -O3 optimization and the right side is the optimized one by adding the GCC intrinsic.

Originally, the compiler can smartly know the final results is $c = 2$; hence, the compiler just returns the result “2” to the variable *c* and eliminates the call to *fun_a()*. Furthermore, *fun_a()* itself is also been optimized to eliminate the call to *fun_b()*. However, these optimizations are all disappeared when we put *fun_a()* and *fun_b()* into the L1 instruction SRAM using the GCC built-in intrinsic: `__attribute__((l1_text))`, which is added behind the definition of a procedure. We found that the procedures put in the L1 instruction SRAM are categorized into the section: “.l1.text”. This is different from the common code section: “.text”. This might be the reason why the existing inter-procedural optimizations cannot be performed here.

Due to the severe disturbance of compiler optimizations, we find that we cannot benefit from utilizing the L1 instruction SRAM under this BSP.


```

int fun_a(int u) __attribute__((l1_text));
int fun_b(int u) //__attribute__((l1_text));

int fun_b(int u)
{
    int f;
    f = u + 1;
    return f;
}

int fun_a(int u)
{
    int f, g;
    f = u + 1 ;
    g = fun_b(f);
    return g;
}

int main()
{
    int c, d = 0;
    c = fun_a(d);
    printf("the answer is %d\n", c);
    return 0;
}

```

Figure 6.5: A simple program to show the disturbance of compiler optimizations due to putting procedures to the L1 instruction SRAM.

6.3 Evaluation of Inline Assembly Optimization

The target we select for inline assembly optimization is MCT, as discussed in Section 3.2.1. It's a color space transformation performed in the pre-processing of JPEG2000. The MCT source code is shown in Figure 6.7. It's a loop with very short codes and the calculations are also very simple . However, the loop is iterated more than 200,000 times (if the input image size is around 640x480). Hence, a small change will cause a big effect to the entire performance.

Figure 6.8 shows the assembly produced by GCC. We can see that there are no parallel executions and registers are used conservatively. We now aggressively use more registers than the compiler does. Hence, the memory operations, which include memory loading and memory restoration, are well parallelized with mathematic calculations as shown in Figure 6.9. We call the instructions which are performed simultaneously “a parallel in-

```

.text:
    .align 4
.global _fun_b;
.type _fun_b, STT_FUNC;
_fun_b:
    nop;
    LINK 0;
    R0 += 1;
    UNLINK;
    rts;
    .size _fun_b, .-_fun_b
    .align 4
.global _fun_a;
.type _fun_a, STT_FUNC;
_fun_a:
    nop;
    LINK 0;
    R0 += 2;
    UNLINK;
    rts;
    .size _fun_a, .-_fun_a
    .section .rodata.strl.4,"aMS",@progbits,1
    .align 4
.LC0:
    .string "the answer is %d\n"
.text:
    .align 4
.global _main;
.type _main, STT_FUNC;
_main:
    .fill 0, 12, 0
    LINK 12;
    R0 = [P3+.LC0@GOT17M4];
    R1 = 2 (X);
    call _printf;

.section .ll.text,"ax",@progbits
    .align 4
.global _fun_b;
.type _fun_b, STT_FUNC;
_fun_b:
    nop;
    LINK 0;
    R0 += 1;
    UNLINK;
    rts;
    .size _fun_b, .-_fun_b
    .align 4
.global _fun_a;
.type _fun_a, STT_FUNC;
_fun_a:
    LINK 0;
    R0 += 1;
    UNLINK;
    jump.l _fun_b;
    .size _fun_a, .-_fun_a
    .section .rodata.strl.4,"aMS",@progbits,1
    .align 4
.LC0:
    .string "the answer is %d\n"
.text:
    .align 4
.global _main;
.type _main, STT_FUNC;
_main:
    [--sp] = ( p5:5 );
    P5 = P3;
    P3 = [P3+_fun_a@FUNCDESC_GOT17M4];
    LINK 12;
    R0 = 0 (X);
    P1 = [P3];
    P3 = [P3+4];
    call (P1);
    R1 = R0;
    R0 = [P5+.LC0@GOT17M4];
    P3 = P5;
    call _printf;

```

Figure 6.6: The assembly codes of the simple test program in Figure 6.5. The left is the original one; the other is the disturbed one.

struction” and a parallel instruction can be composed of two or three instructions. The latency of a parallel instruction is subject to the slowest instruction. The evaluation shows that our optimization achieves about 75% of improvement compared to the unoptimized one.

6.4 Overall Evaluation

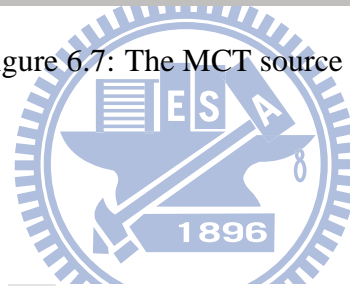
In this section, we first do the comparison between the performance of the data cache and our handmade data locality optimization. Data cache and our data locality optimization both cache temporal data in the high speed internal memory. We will evaluate that if the cache is necessary while the handmade data locality optimization is being applied. Then, we evaluate the overall performance of the proposed optimizations presented in this paper. Beside the image taken from OpenJPEG official website, another four popular standard

```

/* <summary> */
/* Foward reversible MCT. */
/* </summary> */
void mct_encode(
    int* restrict c0,
    int* restrict c1,
    int* restrict c2,
    int n)
{
    int i;
    for(i = 0; i < n; ++i) {
        int r = c0[i];
        int g = c1[i];
        int b = c2[i];
        int y = (r + (g * 2) + b) >> 2;
        int u = b - g;
        int v = r - g;
        c0[i] = y;
        c1[i] = u;
        c2[i] = v;
    }
}

```

Figure 6.7: The MCT source code.



```

1  .L3:
2      P2 = R3;
3      P1 = [P4];
4      P0 = [P2];
5      P2 = R1;
6      P5 = [P2];
7      P2 = P1 + P0;
8      P2 = P2 + (P5 << 1);
9      R0 = P2;
10     P2 = R3;
11     R0 >>>= 2;
12     P1 -= P5;
13     P0 -= P5;
14     R3 += 4;
15     [P2++] = R0;
16     P2 = R1;
17     R1 += 4;
18     [P2++] = P1;
19  .L7:

```

Figure 6.8: The assembly code generated by GCC.

```

1 LOOP_BEGIN LOOP
2     mnop                || R0 = [I0]          || R1 = [I1];
3     R5 = R0 - R1 (s)    || R2 = [I2]          || nop;
4     R4 = R2 - R1 (s)    || [I2 ++ M0] = R5  || nop;
5     R3 = R1 << 0x01     || [I1 ++ M0] = R4  || nop;
6     R3 = R3 + R0;
7     R3 = R3 + R2;
8     R3 >>>= 0x02
9     [I0++M0] = R3
10 LOOP_END LOOP

```

Figure 6.9: The assembly code we reassembled.

image testbenches are adapted to evaluate our optimizations.

6.4.1 Data Cache V.S. Handmade Data Locality Optimization

As we know, data cache automatically stores consecutively data into the high speed SRAM for the potential use. On the other hand, our data locality optimization stores the potential used data based on the understanding of the user program. We now do a comparison to compare the performance between data cache and our data locality optimization in the two parts of JPEG2000 compression: DWT and EBCOT Tier-1.

As shown in Figure 6.10, the time consumption can be reduced after the data cache is turned on. In DWT, even our handmade optimization can be better than pure data cache mechanism. However, the best performance can be achieved only by turning on the data cache and further adding our handmade optimization.

6.4.2 Overall Results

After our optimizations, the whole procedure of JPEG2000 compression can be speeded up around 1.92x. We choose a ARM-based embedded processor, Intel Xscale-PXA270, for our comparison target because when BF561 was issued and available, PXA270 was a famous and popular embedded processor. The PXA270 is based on ARMv5TE architecture and works on 520MHz; the Blackfin BF561 works on 600MHz. Our results show that after our optimizations, the performance to compress the 640x480 image, Bretagne1.bmp,

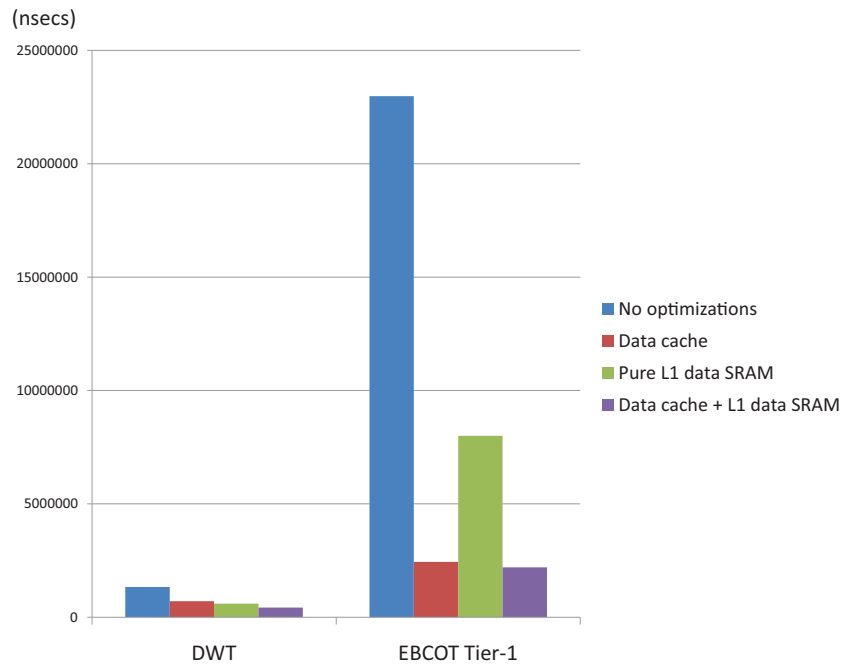


Figure 6.10: The performance comparison between automatic data cache and our hand-made data locality optimization.

which is taken from OpenJPEG official website, is around 2.45x compared to that running on the PXA270 with -O3 optimization, as shown in Figure 6.11.

In addition, we evaluate our optimizations using several standard image benchmarks: airplane, baboon, Lena and peppers, as shown in Figure 6.12 [24]; the results are shown in Figure 6.13. The experiment results show that our optimizations can be widely effective, and the speedups are around 1.92x–2.04x.

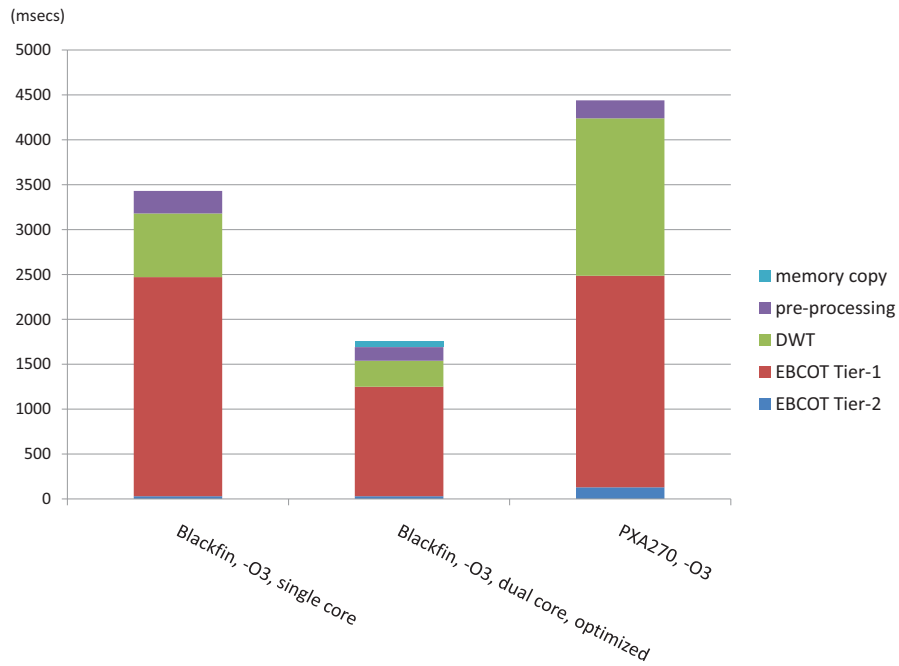


Figure 6.11: Time consumption to compress a 640x480 image.



Figure 6.12: The standard image testbenches.

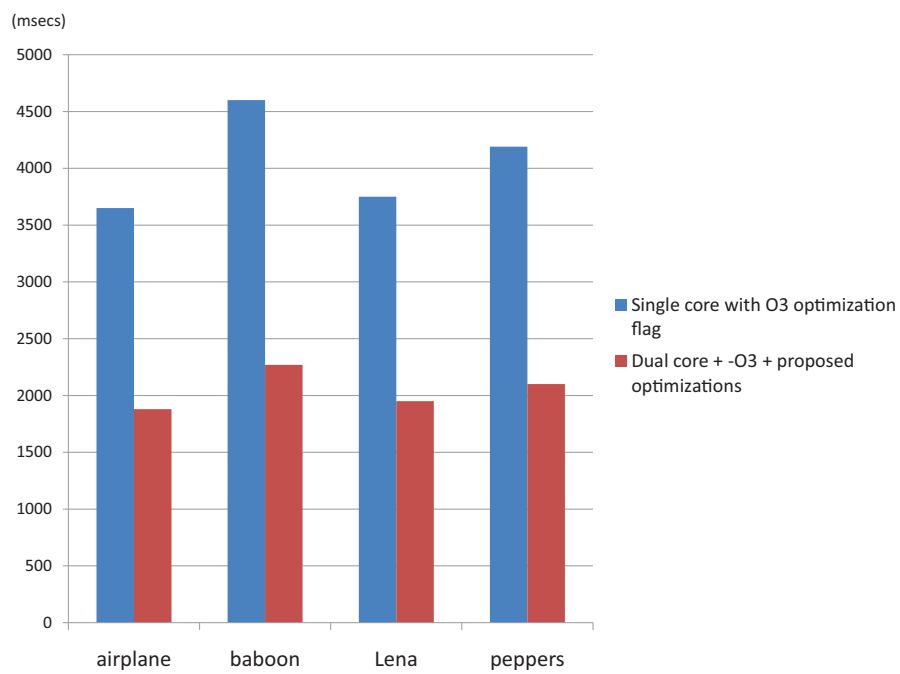


Figure 6.13: The overall performance evaluation of proposed optimizations on standard image testbenches.

Chapter 7

Conclusion and Future Work

7.1 Summary

In this paper we transplant a JPEG2000 compression program onto the Analog Device ADSP-BD561 platform and do the optimizations which focus on data locality optimization and jobs' partition to two cores. We briefly summarize the experimental results:

- Putting intermediate data in the high-speed internal L1 data SRAM can efficiently promote the performance.
- The utilization of two cores should be taken care about the ratio between memory accesses and data computations.

The two lists of summarization above are not only useful on the Blackfin BF561 but also valuable on other processors. However, the following lists are the specialty of BF561.

- Due to the lack of the hardware cache coherency mechanism, threads cannot be scheduled to the other core. This fact leads that deep fine-grain parallelization to two cores is not a good idea.
- Jobs' partition to two cores should always be considered after the data locality optimization using the L1 data SRAM.

- Inline assembly can quickly benefit from the parallelization of memory accesses and mathematical calculations.

To summarize, our proposed optimization methods for JPEG2000 compression on Blackfin BF561 have a speed-up of 1.92x–2.04x compared to conventional single-core execution with -O3 optimization.

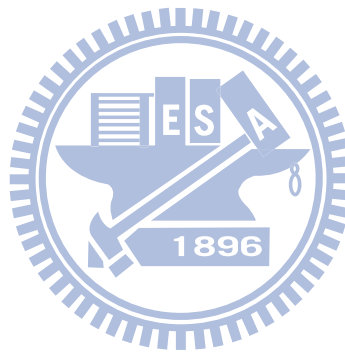
7.2 Future Work

Our future work can be considered from three parts:

- ***The algorithm optimizations to EBCOT Tier-1***—EBCOT Tier-1 occupies the most part of time consumption due to the complicated bit-plane coding. We find that it is potential to optimize the coding algorithm to reduce the time consumption. Some data structures could be reused to save the memory space and access time. This optimization needs to deeply understand the Tier-1 coding algorithms. Furthermore, we can consider to integrate a dedicated hardware architecture to conduct the coding of Tier-1. There are many researches in designing hardware architectures to speed up the coding of Tier-1.
- ***The choices of coding parameters***—The parameters to control JPEG2000 codec are very complicated. We will research these parameters to understand how they affect the coding time of the JPEG2000. In addition to the coding time, the parameters may also affect image size, image quality and so on. To finely utilize these parameters needs to really understand how they control the coding procedure of JPEG2000.
- ***Overcoming the problems about compiler optimizations***—Although this is not a easy work, however, if the compiler problems we encountered in the experiments, which are the disturbance of inter-procedural optimizations and the inefficient generation of parallel instructions, can be improved, the workload of high-level applica-

tion developers can be reduced and the hardware components of BF561 can be fully exploited with few changes in high-level source codes.

Our optimizations presented in this paper mainly focused on the hardware-dependent optimizations. On the other hand, future work will focus on “algorithm-dependent” optimizations. If these two aspects of optimizations could be integrated, more speed-ups can be achieved.



Bibliography

- [1] M. J. Absar and F. Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2, DATE '05*, pages 1162–1167, Washington, DC, USA, 2005. IEEE Computer Society.
- [2] Tinku Acharya and Ping-Sing Tsai. *JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures*. Wiley-Interscience, 2004.
- [3] ADSP-BF561 — Blackfin symmetric multi-processor for consumer multimedia. <http://www.analog.com/en/processors-dsp/blackfin/adsp-bf561/processors/product.html>.
- [4] Analog Devices, Inc. *ADSP-BF561 Blackfin Processor Hardware Reference, Revision 1.2*.
- [5] Blackfin SMP like. <http://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:smp-like>.
- [6] Arnaldo Azevedo and Ben Juurlink. An efficient software cache for h.264 motion compensation. In *Proceedings of the 11th international conference on System-on-chip, SOC'09*, pages 147–150, Piscataway, NJ, USA, 2009. IEEE Press.

- [7] Mikel Azkarate-askasua. Jpeg2000 image compression in multi-processor system-on-chip. In *JPEG2000 Image Compression in Multi-Processor System-on-Chip*, Master's Thesis, Delft University of Technology, The Netherlands, 2008.
- [8] Michael G. Benjamin and David Kaeli. Stream image processing on a dual-core embedded system. In *Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation, SAMOS'07*, pages 149–158, Berlin, Heidelberg, 2007. Springer-Verlag.
- [9] Berkeley Design Technology, Inc. *Choosing a DSP Processor*. 2000.
- [10] Francky Catthoor, Eddy de Greef, and Sven Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
- [11] CH Chen. Implementation and optimization of jpeg2000 wavelet transform on adsp-bf533 blackfin processor. In *Master's Thesis, 2005. National Taiwan University of Science and Technology Repository*, 2005.
- [12] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl*, 4:247–269, 1998.
- [13] Jun-Wei Gao and Ke-Bin Jia. Embedded video surveillance system based on h.264. In *Proceedings of the 2009 International Conference on Multimedia Information Networking and Security - Volume 01*, MINES '09, pages 282–286, Washington, DC, USA, 2009. IEEE Computer Society.
- [14] David J.Katz and Rick Gentile. *Embedded Media Processing*. Kluwer Academic Publishers, 2005.
- [15] The JPEG committee home page: JPEG2000. <http://www.jpeg.org/jpeg2000/>.

- [16] Chung-Jr Lian, Kuan-Fu Chen, Hong-Hui Chen, and Liang-Gee Chen. Analysis and architecture design of block-coding engine for ebcot in jpeg 2000. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(3):219 – 230, mar 2003.
- [17] Peter Meerwald, Roland Norcen, and Andreas Uhl. Parallel jpeg2000 image coding on multiprocessors. In *Journal of Object-Oriented Programming*, page 2. Society Press, 2002.
- [18] The jasper project home page. <http://www.ece.uvic.ca/mdadams/jasper/>.
- [19] Hidemasa Muta, Munehiro Doi, Hiroki Nakano, and Yumi Mori. Multilevel parallelization on the cell/b.e. for a motion jpeg 2000 encoding server. In *Proceedings of the 15th international conference on Multimedia, MULTIMEDIA '07*, pages 942–951, New York, NY, USA, 2007. ACM.
- [20] OpenJPEG library : an open source JPEG2000 codec. www.openjpeg.org.
- [21] Rafael Palomar, José M. Palomares, José M. Castillo, Joaquín Olivares, and Juan Gómez-Luna. Parallelizing and optimizing lip-canny using nvidia cuda. In *Proceedings of the 23rd international conference on Industrial engineering and other applications of applied intelligent systems - Volume Part III, IEA/AIE' 10*, pages 389–398, Berlin, Heidelberg, 2010. Springer-Verlag.
- [22] Majid Rabbani and Rajan Joshi. An overview of the jpeg2000 still image compression standard. In *Signal Processing: Image Communication*, pages 3–48, 2002.
- [23] Hee Seo and Seon Wook Kim. Openmp directive extension for blackfin 561 dual core processor. In *Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference on*, page 49, sept. 2006.
- [24] Sipi image database. <http://sipi.usc.edu/database/>.

- [25] David S. Taubman and Michael W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
- [26] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26:30–44, May 1991.

