

國立交通大學

電子工程學系 電子研究所碩士班

碩士論文

H.264 編碼器及其可調適延伸版  
解碼器之加速和 TI DSP 系統平台之實現



**Acceleration and Implementation of  
H.264 Encoder and Scalable Extension of H.264  
Decoder on TI DSP Platform**

研究生：鄭凱庭

指導教授：杭學鳴 博士

中華民國九十六年六月

# H.264 編碼器及其可調適延伸版

## 解碼器之加速和 TI DSP 系統平台之實現

### Acceleration and Implementation of H.264 Encoder and Scalable Extension of H.264 Decoder on TI DSP Platform

研究生：鄭凱庭

Student: Kai-Ting Cheng

指導教授：杭學鳴

Advisor: Dr. Hsueh-Ming Hang

國立交通大學  
電子工程學系 電子研究所碩士班  
碩士論文



Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of Master

in

Electronics Engineering

June 2007

HsinChu, Taiwan, Republic of China

中華民國九十六年六月

# H.264 編碼器及其可調適延伸版

## 解碼器之加速和 TI DSP 系統平台之實現

研究生：鄭凱庭

指導教授：杭學鳴 博士

國立交通大學

電子工程學系 電子研究所碩士班

### 摘要

隨著數位訊號處理的進步，及時的視訊傳輸已成為生活必需的一部份。本篇論文主要是利用數位訊號處理器去實現基本的 H.264/AVC 解碼器以及 H.264/AVC 可調適延伸版的解碼器，此數位訊號處理器環境為 Sundance 的 SMT395 型號，其上核心為德州儀器公司 TMS320C6416T，是個擁有強大的數學運算功能處理器。

在程式執行方面，針對 H.264/AVC 編碼器，是以公開軟體 x264 為基礎來移植於數位處理器平台，另外 Mode decision 為主要加速的部份，我們使用一些判斷式減少一些不需要的 Mode 的計算量，這樣可以節省 13%的編碼時間，在 DSP 實現方面，我們使用 TI DSP 編譯器所提供的各種最佳化的相關工具來加速，並支援 2 層 Cache 的模式，這樣可以達到 19 倍左右的加速，另外針對 DSP 的架構使用了一些程式技巧，包括定點式資料型態、記憶體規劃、TI DSP 所支援的特殊指令群等等，可以減少 50%的運算量，以 QCIF 的圖形，在某些畫面下，最後可以達到每秒編碼 40 張左右的速度。

針對延伸式 H.264/AVC 解碼器部份，主要是使用參考軟體 JSVM 5.0 做修改，延伸式 H.264/AVC 主要分為 Temporal、Spatial、SNR 等 3 種不同型態的可調適，在 DSP 實現方面，我們先針對 3 種不同可調適環境做分析，在三種型態合併的情況下，針對最耗費計算量部份做加速，主要為 FGS 和 Inter-layer Prediction 這 2 部分，在 FGS 部分，做程式的修正，避免不必要的計算，在 Inter-layer Prediction 部分主要是減少 Intra 和 Residual 的計算量，配合 DSP 所提供的一些最佳化方法，在合併的情況下，可以減少 49%的計算量。

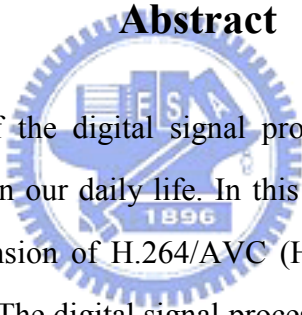
# Acceleration and Implementation of H.264 Encoder and Scalable Extension of H.264 Decoder on TI DSP Platform

Student: Kai-Ting Cheng

Advisor: Dr. Hsueh-Ming Hang

Department of Electronic Engineering &  
Institute of Electronics  
National Chiao Tung University

## Abstract



With the advancement of the digital signal processing, real-time video transmission becomes an essential element in our daily life. In this thesis, we implement the H.264/AVC encoder and the scalable extension of H.264/AVC (H.264/AVC SVC) decoder by using a digital signal processor (DSP). The digital signal processing environment is Sundance module SMT395. The core of the DSP is the Texas Instrument's TMS320C6416T which is a powerful signal processor with strong arithmetic operation capability.

For the H.264/AVC encoder, the open source code x264 is used as the basis to build a DSP-executable program. The mode decision module is the key element being accelerated. We develop an early termination method to reduce the calculation of the dispensable modes. This saves up to 13% of the encoding time. For the DSP implementation, we start with the optimization tools provided by the TI DSP compiler. We also make use of the two-level cache module on the DSP platform. This can speed-up the system by about 19 times. Furthermore, we use several DSP codes acceleration techniques including fixed-point data types, TI DSP intrinsic functions and others. Through the code modifications, we can reduce the computation by 50%. Finally, the overall system can encode up to 40 QCIF frames per second on test video sequences.

For the H.264/AVC SVC decoder, we start with the MPEG reference software JSVM 5.0. The H.264/AVC SVC includes three types of scalability, namely, Temporal, Spatial and SNR scalability. For the DSP implementation, we accelerate the parts which take the most computing time in the combined scalability. These two parts are the FGS and the inter-layer prediction modules. For FGS, we refine the codes to reduce the computation redundancy. For the inter-layer prediction, we reduce the up-sampling operations for the intra texture and the residuals. In addition, we also use the acceleration techniques supported by the TI DSP. The final H.264/AVC SVC decoder can reduce the computation by 49% in the combined scalability.



## 誌謝

在這 2 年的研究所生涯中，最感謝的是我的指導教授 杭學鳴老師，老師不僅在研究上給予專業的指導，讓我可以學問上有所進步，也教育我們研究學問的方法，為往後的工作奠下基礎。老師除了豐富的學識和研究，謙虛、溫和的態度，讓我了解一個優秀的工程師所應有的涵養，是我景仰和學習的目標。

另外也要感謝 彭文孝老師，感謝老師在專業領域上給予我的指導，讓我可以對於視訊壓縮有更深入的了解，老師對於研究方面的專業和研究的態度，讓我學習到很多研究方面的知識，感謝老師在我疑惑時給予解答並提供我正確的研究目標。

這篇論文可以完成，最主要的是要感謝鴻志學長和威年、尚諭 2 位學弟，感謝他們在 Video 領域中可以讓我有可以討論，解惑，並且在實作中給於協助，沒有他們的幫忙，論文也無法順利完成，實驗室除了提供專業的設備，也提供了良好的研究環境，感謝繼大、朝雄、家揚、建志學長給予我研究上的協助，也感謝實驗室的夥伴志岡、育成、浩庭、耀全、介遠、耀均、柏昇、政達、錫祺，在修課上遇到問題時可以有人可以互相討論，解決問題，並在生活中帶給我歡樂和成長。

感謝週遭的朋友和大學時的同學給予我的鼓勵，讓我可以不斷的往前走，還要感謝詠詩的陪伴，在我生活中給予我的支持和鼓勵，讓我在研究生涯中不孤單。

最後，最重要的要感謝我的爸爸、媽媽和 2 位姐姐，不論在生活上或求學生給予我的鼓勵，他們讓我可以心無旁騖的從事研究，陪伴我度過所有難關，讓我可以順利完成學業。

在此僅將論文獻給所有幫助過我，陪伴我走過求學生涯的所有師長，朋友、同學和家人，謝謝!!

鄭凱庭

民國九十六年六月

# Contents

摘要 .....	i
Abstract .....	ii
誌謝 .....	iv
<b>Chapter 1 Introduction</b> .....	1
1.1 Introduction and Motivation .....	1
<b>Chapter 2 H.264 Video Coding</b> .....	3
2.1 Overview of H.264 Encoder .....	3
2.2 Slice and Slice Groups .....	5
2.3 Inter Prediction .....	6
2.3.1 Motion Vector Prediction .....	8
2.4 Intra Prediction .....	9
2.5 Mode Decision .....	10
2.6 Loop Filter .....	13
2.7 Transform and Quantization .....	13
2.8 Entropy Coding .....	16
<b>Chapter 3 Scalable Extension of H.264</b> .....	17
3.1 The Architecture of Scalable Extension of H.264 .....	18
3.2 Temporal Scalability .....	20
3.2.1 MCTF .....	20
3.2.2 Scalability Dimensions .....	22
3.3 Spatial Scalability .....	24
3.4 SNR Scalability .....	25
3.4.1 CGS .....	26
3.4.2 FGS .....	27
3.5 Combined Scalability .....	29
<b>Chapter 4 DSP Implementation Environment</b> .....	30
4.1 The DSP Board .....	30
4.2 The TMS320C6416T DSP Chip .....	31

4.2.1 Central Processing Unit of C64x .....	33
4.2.2 Memory Architecture and Peripherals .....	36
4.3 TI DSP Code Development Environment .....	36
4.3.1 Code Composer Studio .....	36
4.3.2 Simulation Tools .....	38
4.4 Code Development Flow .....	39
4.4.1 Compiler Optimization Options .....	41
<b>Chapter 5 H.264 Encoder Implementation and Optimization on DSP Platform .....</b>	<b>43</b>
5.1 Introduction to x264 .....	43
5.2 Proposed Acceleration Method .....	44
5.2.1 Mode decision in x264 .....	45
5.2.2 Fast Algorithms .....	49
5.2.3 Experimental Results .....	54
5.3 Complexity Analysis on DSP .....	59
5.3.1 Complexity Analysis on Various Simulator .....	59
5.3.2 Memory System .....	61
5.4 DSP Code Acceleration Methods .....	63
5.4.1 Compiler Options .....	63
5.4.2 Fixed-point Coding .....	64
5.4.3 Loop Unrolling .....	64
5.4.4 Linear Assembly .....	65
5.4.5 Other Acceleration Techniques .....	66
5.5 Experimental Results .....	67
5.5.1 Simulation and Acceleration Results .....	67
5.5.2 Encoding Speed on DSP board .....	68
<b>Chapter 6 H.264/AVC SVC Decoder Implementation and Optimization on DSP Platform .....</b>	<b>71</b>
6.1 System Architecture .....	71
6.2 Procedure of the Implementation Work .....	73
6.3 JSVM 5.0 Decoder Complexity Analysis .....	74



6.4 DSP Code Acceleration Methods .....	77
6.4.1 Packet Data Processing.....	78
6.4.2 Intrinsic.....	78
6.4.3 Memory Allocation Optimization.....	79
6.4.4 DSP library .....	80
6.5 Fast Algorithms for the SVC Decoder.....	80
6.5.1 FGS.....	80
6.5.2 Inter-layer Prediction.....	84
6.6 Final Simulation and Acceleration Results.....	88
<b>Chapter 7</b> .....	92
7.1 Conclusion.....	92
7.2 Future Work.....	93
<b>References</b> .....	94
<b>自    傳</b> .....	97



# List of Figures

Figure 2-1 H.264/AVC encoder structure [2] .....	4
Figure 2-2 Subdivision of a picture into slices [2] .....	6
Figure 2-3 Macroblock partitions: 16x16, 16x8, 8x16 and 8x8.....	6
Figure 2-4 Macroblock sub-partitions: 8x8, 8x4, 4x8 and 4x4 .....	7
Figure 2-5 Filter for fractional-sample accurate motion compensation [2] .....	8
Figure 2-6 Choice of neighboring partitions [3] .....	9
Figure 2-7 Intra 4x4 prediction mode [3] .....	10
Figure 2-8 Intra 16x16 prediction mode [3] .....	10
Figure 2-9 H.264 mode decision algorithm .....	12
Figure 2-10 Zig-Zag Scan.....	14
Figure 2-11 Flow of transform and quantization [4] .....	15
Figure 3-1 Example of Scalable Video Coding.....	18
Figure 3-2 Basic structure for the scalable extension of H.264/AVC [8] .....	19
Figure 3-3 Lifting representation of an analysis-synthesis filter bank [8].....	20
Figure 3-4 Illustration of temporal scalability [9] .....	22
Figure 3-5 Hierarchical-B prediction structure.....	23
Figure 3-6 Up-sampling of motion data [8] .....	25
Figure 3-7 Up-sampling of intra texture [8] .....	25
Figure 3-8 CGS SNR scalable coding scheme [10] .....	26
Figure 3-9 Scheme of Cyclical Block Coding .....	27
Figure 3-10 Example of significant and refinement pass [11].....	28
Figure 3-11 Combined scalability.....	29
Figure 4-1 SMT395 module .....	31
Figure 4-2 Block diagram of the TMS320C64x DSPs [13] .....	32
Figure 4-3 Development cycle [16] .....	37
Figure 4-4 Code composer studio development [16] .....	37
Figure 4-5 Code development [17] .....	40
Figure 4-6 C/C++ compiler [18] .....	42

<b>Figure 5-1 Rate-distortion of JM9.8 and x264 .....</b>	<b>44</b>
<b>Figure 5-2 Spatial prediction for skip mode decision.....</b>	<b>47</b>
<b>Figure 5-3 Mode decision algorithm of x264.....</b>	<b>48</b>
<b>Figure 5-4 Block diagram of x264 and Algorithm A .....</b>	<b>50</b>
<b>Figure 5-5 Difference between x264 and Algorithm B1 .....</b>	<b>51</b>
<b>Figure 5-6 Fast Algorithm B2 .....</b>	<b>52</b>
<b>Figure 5-7 Fast Algorithm C .....</b>	<b>53</b>
<b>Figure 5-8 Rate-distortion (PSNR vs. bit-rates) of the x264 and the modified Algorithm .....</b>	<b>58</b>
<b>Figure 5-9 Complexity profiling of the H.264 encoder on the C6416 simulator .....</b>	<b>60</b>
<b>Figure 5-10 Complexity profiling of the H.264 encoder on the C64xx simulator .....</b>	<b>60</b>
<b>Figure 5-11 C6416 memory configuration [14].....</b>	<b>62</b>
<b>Figure 5-12 Profile with L2 cache using C6416 simulator .....</b>	<b>63</b>
<b>Figure 6-1 System architecture of the SVC video decoder .....</b>	<b>72</b>
<b>Figure 6-2 Complexity profiling of the Temporal Scalability of JSVM 5.0 decoder .....</b>	<b>75</b>
<b>Figure 6-3 Complexity profiling of the Spatial Scalability of JSVM 5.0 decoder.....</b>	<b>76</b>
<b>Figure 6-4 Complexity profiling of the SNR Scalability of JSVM 5.0 decoder.....</b>	<b>76</b>
<b>Figure 6-5 Complexity profiling of the Combined Scalability of JSVM 5.0 decoder.....</b>	<b>77</b>
<b>Figure 6-6 SIMD example of using the word instructions for adding short data.....</b>	<b>78</b>
<b>Figure 6-7 Use of intrinsic function in the SVC decoder .....</b>	<b>79</b>
<b>Figure 6-8 Flow chart of FGS in the JSVM 5.0 decoder.....</b>	<b>81</b>
<b>Figure 6-9 Complexity profiling of FGS on C6416 simulator .....</b>	<b>82</b>
<b>Figure 6-10 Block algorithm of the original inter-layer intra prediction.....</b>	<b>85</b>
<b>Figure 6-11 Block algorithm of the modified inter-layer intra prediction .....</b>	<b>86</b>
<b>Figure 6-12 Difference in the inter-layer residual prediction procedures .....</b>	<b>87</b>

# List of Tables

Table 2-1 Quantization step size in H.264 [3].....	16
Table 4-1 Functional units and operations performed [15] .....	34
Table 4-2 Functional units and operations performed [15] .....	35
Table 5-1 Performance of JM9.8 vs. x264.....	44
Table 5-2 The experimental parameters.....	54
Table 5-3 Performance of x264.....	54
Table 5-4 Performance Comparison between x264 and the modified Algorithm .....	55
Table 5-5 Cycles on different simulators .....	59
Table 5-6 Effect of using L2 cache memory .....	62
Table 5-7 Size of different data type.....	64
Table 5-8 Processing time on the C64x for different data types.....	64
Table 5-9 Comparison between C code and linear assembly code.....	65
Table 5-10 Example of linear assembly code.....	66
Table 5-11 Comparison using the C6416 simulator with and without the L2 cache .....	67
Table 5-12 Comparison using C6416 simulator with original and accelerated code .....	68
Table 5-13 Results of the “foreman” sequence on the C6416 emulator.....	69
Table 5-14 Results of the “akiyo” sequence on the C6416 emulator.....	69
Table 5-15 Results of the “mobile” sequence on the C6416 emulator .....	70
Table 5-16 Results of the “stefan” sequence on the C6416 emulator.....	70
Table 6-1 Simulation parameters .....	74
Table 6-2 Cycles on different scalability .....	75
Table 6-3 Performance using intrinsic.....	79
Table 6-4 Reduction Ratio of FGS block.....	83
Table 6-5 Performance of the modified FGS.....	84
Table 6-6 Distribution of mode in spatial scalability.....	85
Table 6-7 Reduction ratio of Inter-layer prediction .....	87
Table 6-8 Performance of the modified inter-layer prediction.....	88
Table 6-9 Comparison of using C6416 simulator with and without the L2 cache.....	89
Table 6-10 Effect of using L2 cache memory .....	89

**Table 6-11 Comparison using C6416 simulator on the original and the modified codes.90**  
**Table 6-12 Performance on the C6416 emulator .....91**



# Chapter 1

## Introduction

### 1.1 Introduction and Motivation

With the growing popularity of mobile communication, video transmission over wireless channel will become an essential element in our daily life. Many international video compression standards such as H.261, H.263, MPEG-2 and MPEG-4 have already been widely used in different situations. In this thesis, we concentrate on the standard H.264/AVC and the newest standard scalable extension of H.264/AVC (H.264/AVC SVC). Our focus is to implement the H.264/AVC encoder and H.264/AVC SVC decoder on the digital signal processors (DSPs).

H.264/AVC is a recent standard defined by the ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. It provides better compression of video images together with a range of features supporting high-quality, low-bitrates streaming video. The basic functional elements (prediction, transform, quantization, entropy encoding) are similar to those in the previous standards but the important fine-tune in H.264 occur in the details of each functional element.

Scalable video coding is currently being developed as an extension of H.264/AVC. The Joint Video Team of the ISO/IEC MPEG and the ITU-T VCEG is now standardizing this new standard. It is intended to encode the signal once, but allow decoding from the partial streams at the specific rate and resolution required by a certain application. Its basic design idea is to extend the hybrid video coding approach of H.264/AVC to efficiently incorporate the spatial, SNR and temporal scalability.

The environment of our DSP implementation involves a host PC, DSP board and DSP

chips on the board. The DSP chips are Texas Instruments (TI)'s TMS320C6416T. The TMS320C6416T is a fixed-point DSP with 1 ns (1 GHz clock) instruction cycle time. It adopts the advanced VelociTI Very Long Instruction Word (VLIW) architecture that enables sustained throughput of up to eight instructions in parallel and thus it allows the processor running faster. In addition, we accelerate the H.264/AVC encoder and H.264/AVC SVC decoder by some DSP coding techniques and several efficient algorithms.

This thesis is organized as follows. Chapter 2 is an overview of the H.264/AVC video standard. Chapter 3 introduces the H.264/AVC SVC. Chapter 4 gives a brief description of the TI DSP chip and its development environment. In chapter 5, we describe the algorithm and code acceleration methods of the H.264/AVC encoder and show the experimental results on DSP. Chapter 6 describes the acceleration of the H.264/AVC SVC decoder for DSP and presents experimental results. Finally, chapter 7 contains the conclusion.



# Chapter 2

## H.264 Video Coding

H.264/MPEG-4 AVC (Advanced Video Coding) is a video coding standard of the ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. The first draft design for H.264/AVC coding standard was adopted in October of 1999. VCEG and the MPEG formed a Joint Video Team (JVT) and drafted new video coding standard as H.264/AVC [1] in March of 2003.

H.264 is a standard developed based on H.26L and it promises to significantly outperform MPEG4 and H.263, providing better compression of video images together with a range of features supporting high-quality, low bit-rate streaming video. It uses the state-of-the-art coding tools and provides enhanced coding efficiency for a wide range of applications, including video telephony, digital video authoring, digital camera, and many others. In this chapter, the H.264/AVC standard will be described.

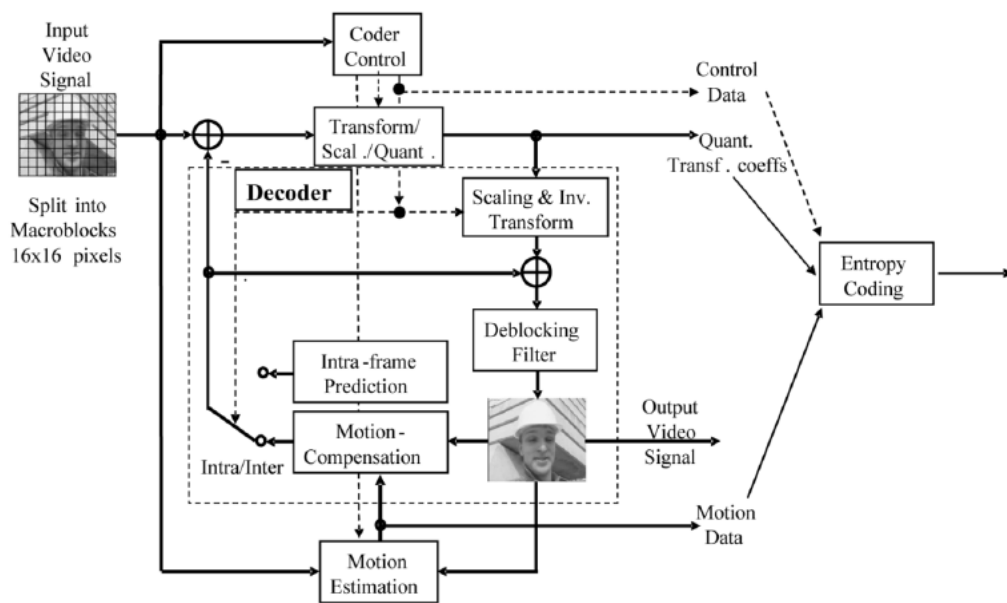
### 2.1 Overview of H.264 Encoder

The H.264/AVC standards include a Video Coding Layer (VCL), designed to efficiently represent video contents, and a Network Abstraction Layer (NAL), designed to format the VCL representation and it provides header information in a manner appropriate to be conveyed by a variety of transport layers or storage media.

A Block diagram of the basic H.264/AVC encoder is shown in Figure 2-1. The H.264/AVC encoder is consistent with a decoder scheme. Its main elements are motion-estimation, motion-compensation, transform and quantization, deblocking filter and entropy coding (CABAC or CAVLC). Motion-compensated is used to remove temporal redundancy. The purpose of transform (Integer-DCT) and quantization is to remove spatial



redundancy. The entropy coding removes syntax redundancy. In the block diagram, the video frames are captured into intra or inter prediction parts. If the frame type is intra, the inter prediction is disabled. Multiple references and variable block size motion estimations are used for the inter prediction. The best mode among these prediction modes is chosen in the mode selection block. The input frame is then subtracted from the prediction and forms the residual block. The residual blocks are transformed by using a 4x4 integer DCT transformer for luminance and a 2x2 transform for the chrominance DC coefficient, scan and quantization procedures are then applied to the coefficients. The entropy coder receives these quantized coefficients and generates codeword. The reconstruction loop includes the dequantization, inverse transform and deblocking filter. Finally, the reconstruction frame is written to the frame buffer for motion estimation.



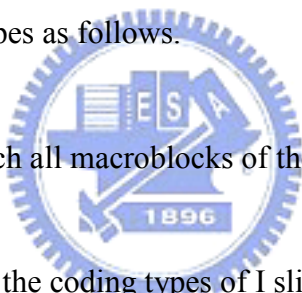
**Figure 2-1** H.264/AVC encoder structure [2]

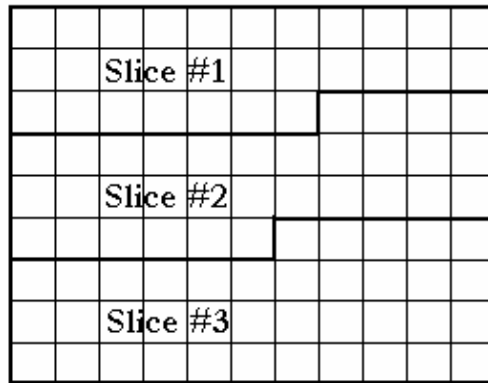
There are three profiles defined in the H.264/AVC standard: baseline, main and extended profile. The baseline profile is for real-time communication, the main profile is for digital storage application, and the extended profile is for network streaming application. In the baseline profile, it supports the intra coding and the inter coding with entropy coding using

CAVLC. In the main profile, B-frame coding is used and entropy coding using CABAC. While the extended profile has all the features of the baseline profile while B-frame coding, SI-frame, and SP-frame coding are included.

## 2.2 Slice and Slice Groups

Slices are a sequence of macroblocks which are processed in the order of a raster scan. A picture is split into one or several slices as shown in Figure 2-2. A picture, therefore, is a collection of one or more slices in the H.264/AVC. Slices are self-contained in the sense that given the active sequence and picture parameter sets, their syntax elements can be parsed from the bit-stream. Furthermore, the values of the samples in the area of the picture that the slices represents can be correctly decoded without the use of data from other slices provided that the utilized reference pictures are identical at encoder and decoder. Each slice can be coded using different coding types as follows.

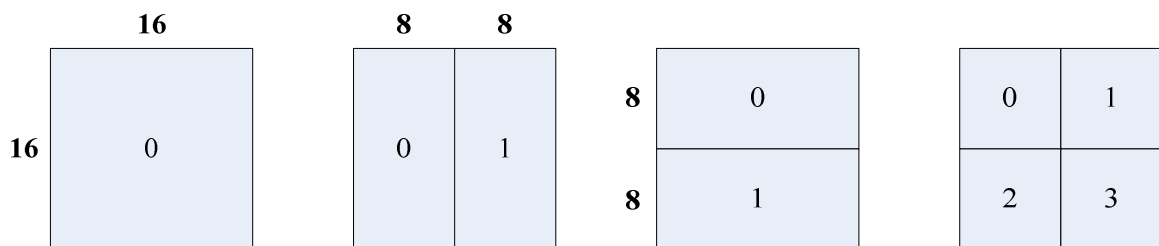
- 
- **I slice:** A slice in which all macroblocks of the slice are coded using intra prediction.
  - **P slice:** In addition to the coding types of I slice, some macroblocks of the P slice can also be coded using inter prediction with at most one motion-compensated prediction signal per prediction block.
  - **B slice:** In addition to the coding types available in a P slice, some macroblocks of the B slice can also be coded using inter prediction with two motion-compensated prediction signals per prediction block.
  - **SP slice:** A so-called switching P slice that is coded such that efficient switching between different pre-coded pictures becomes possible.
  - **SI slice:** A so-called switching I slice that allows an exact match of a macroblock in a SP slice for random access and error recovery purposes.



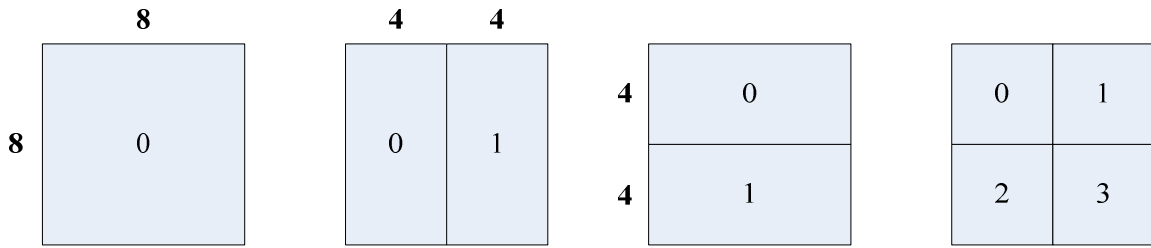
**Figure 2-2** Subdivision of a picture into slices [2]

## 2.3 Inter Prediction

High quality video sequences usually have high frame rates. Therefore, two successive frames in a video sequence are very likely to be similar. The goal of inter prediction is to utilize this temporal correlation to reduce data need to be encoded. Inter prediction creates a prediction model from one or more previously encoded video frames. The model is formed by shifting samples in the reference frame(s). H.264/AVC supports motion compensation block sizes ranging from 16x16 to 4x4 luminance samples with many options between the two sizes. The luminance component of each macroblock (16x16 samples) may be split up in 4 ways as shown in Figure 2-3: 16x16, 16x8, 8x16, or 8x8. If the 8x8 mode is chosen, each of the four 8x8 macroblock partitions within the macroblock may be split in a 4 ways as shown in Figure 2-4: 8x8, 8x4, 4x8, or 4x4.



**Figure 2-3** Macroblock partitions: 16x16, 16x8, 8x16 and 8x8



**Figure 2-4** Macroblock sub-partitions: 8x8, 8x4, 4x8 and 4x4

Each partition in an inter-coded macroblock is predicted from an area of the same size in a reference picture. The distance between the two areas (the motion vector) has 1/4-pixel resolution (for the luma component). In case the motion vector points to an integer-sample position, the prediction signal consists of the corresponding samples of the reference pictures. Otherwise, the corresponding sample is obtained using interpolation to generate non-integer positions. The prediction values at half-sample positions are obtained by applying a one-dimensional 6-tap FIR (Finite Impulse Response) filter horizontally and vertically. For example in Figure 2-5., half-pixel sample *b* is calculated from the 6 horizontal integer samples *E*, *F*, *G*, *H*, *I* and *J*:

$$b = ((E - 5F + 20G + 20H - 5I + J) + 16) \gg 5$$

Similarly, *h* is interpolated by filtering *A*, *C*, *G*, *M*, *R* and *T*. When all of the samples horizontally and vertically adjacent to integer samples are calculated, the remaining half-pixel positions are calculated by interpolating between six horizontal or vertical half-pixel samples from the first set of operations. For example, the sample at half sample positions labels as *j* are obtained by

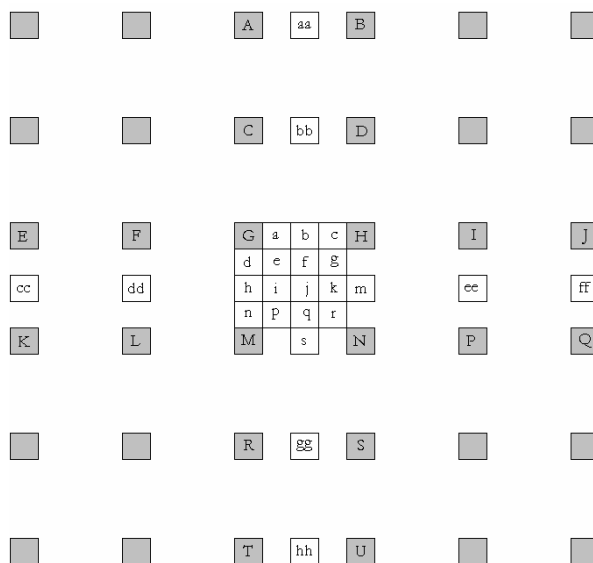
$$j = ((cc - 5dd + 20h + 20m - 5ee + ff) + 512) \gg 10$$

Once all the half-pixel samples are available, the quarter-pixel positions are produced by linear interpolation. Quarter-pixel positions with two horizontally or vertically adjacent half- or integer-pixel samples are linearly interpolated between these adjacent samples. For example:

$$a = (G + b + 1) \gg 1$$

The prediction values for chroma component are always obtained by bilinear interpolation. Since the sampling grid of chroma has lower resolution than the sampling grid

of the luma, the displacements used for chroma have one-eighth sample position accuracy.



**Figure 2-5** Filter for fractional-sample accurate motion compensation [2]

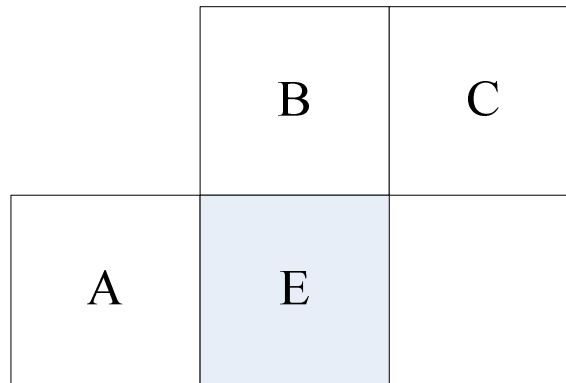


### 2.3.1 Motion Vector Prediction

Encoding a motion vector for each partition can take a significant number of bits, especially if small partition sizes are chosen. Motion vector for neighboring partitions are often highly correlated and so each motion is predicted from vectors of nearby. A predicted vector  $MV_p$  (Motion Vector Prediction) is formed based on previously calculated. The motion vectors  $MVD$  (Motion Vector Difference), the difference between the current vector and the predicted vector, is coded and transmitted. The method of forming the prediction  $MV_p$  depends on the motion compensation partition size on the availability of nearby vectors.

Let  $E$  be the current macroblock, macroblock partition or sub-partition; let  $A$  be the partition or sub-partition immediately to the left of  $E$ ; let  $B$  be the partition or sub-partition immediately above  $E$ ; let  $C$  be the partition or sub-partition above and to the right of  $E$ . If there is more than one partition immediately to the left  $E$ , the topmost of these is chosen as  $A$ . If there is more than one partition immediately above  $E$ , the leftmost of these is chosen as  $B$ . Figure 2-6 illustrates the choice of neighboring partitions when all the partitions have the

same size (16x16 in this case). The MVp of current macroblock E is calculated from the motion vector of macroblock A, B and C. In the decoder, the predicted vector MVp is formed in the same way and added to the decoded vector difference MVD.

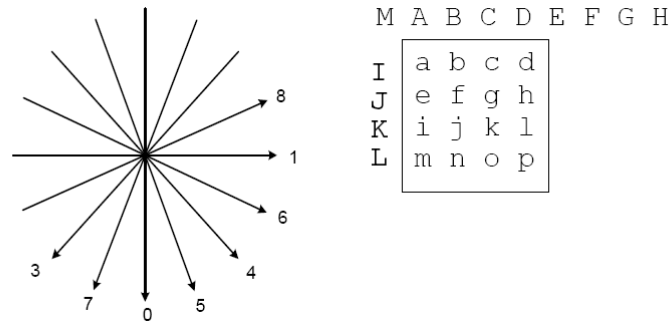


**Figure 2-6** Choice of neighboring partitions [3]

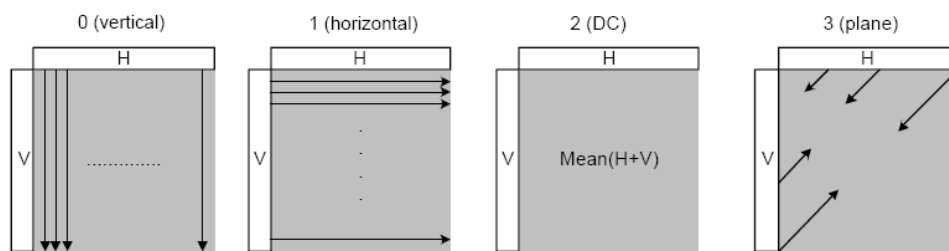
## 2.4 Intra Prediction



In H.264/AVC standard, each 16x16 is a basic unit to be encoded. For intra prediction, utilizing high correlation of neighboring samples in spatial domain, the prediction block is conducted based on previously coded and reconstructed blocks which are to the left /or above the block before deblocking filter. For the luma samples, each prediction block may be formed for each 4x4 block (denoted as I4MB), or for an entire MB (denoted as I16MB). When utilizing I4MB prediction, each 4x4 block is predicted from spatially neighboring samples and will choose one of nine prediction modes as the best one. In addition to DC prediction mode, eight directional prediction modes are supported shown in Figure 2-7. Those modes are suitable to predict directional structures in a picture such as edges at various angles. When utilizing I16MB prediction, which is well suited for smooth image areas, the whole luma component of an MB performs a uniform prediction. There are four prediction modes which are shown in Figure 2-8. The chroma samples of an MB are predicted using a similar prediction technique as for the luma component in I16MB prediction.



**Figure 2-7** Intra 4x4 prediction mode [3]



**Figure 2-8** Intra 16x16 prediction mode [3]

## 2.5 Mode Decision

In H.264/AVC, the high complexity mode of the standard, the macroblock mode decision is done by minimizing the Lagrangian function [1]:

$$J(s, c, MODE | QP, \lambda_{MODE}) = SSD(s, c, MODE | QP) + \lambda_{MODE} \times R(s, c, MODE | QP)$$

Where  $J$  denotes the cost function and depends on  $s$  (the original signal macroblock),  $c$  (the reconstructed signal macroblock) and  $MODE$  (select from a set of modes).  $J$  is found given  $QP$  (the macroblock quantization parameter) and  $\lambda_{MODE}$  (the Lagrange multiplier for mode decision).  $SSD$  is the sum of the squared differences between the original macroblock and its reconstruction with  $QP$  and it also depends on the original and reconstructed macroblock, as well as the mode decision ( $MODE$ ). The Lagrange multiplier,  $\lambda_{MODE}$ , depends on the slice type (B or {SP, Intra, P}) and the quantization parameter per macroblock ( $QP$ ).

For Intra or P slices in particular:

$$\lambda_{MODE} = 0.85 \times 2^{(QP-12)/3}$$

Finally, the rate  $R(s, c, MODE | QP)$  depends on the original and reconstructed macroblock with quantization parameter  $QP$ , as well as chosen  $MODE$ , and reflects the number of bits produced for header(s) (including  $MODE$  indicators), motion vector(s) and coefficients.

In H.264,  $MODE$  is chosen from a set of potential prediction modes as follows:

For Intra slices:

$$MODE \in \{I4MB, I16MB\}$$

For P slices: {single reference forward or backward prediction}

$$MODE \in \{I4MB, I16MB, SKIP, P_{16 \times 16}, P_{16 \times 8}, P_{8 \times 16}, P_{8 \times 8}\}$$

For B slices: {bi-directionally predicted slices}

$$MODE \in \{I4MB, I16MB, DIRECT, P_{16 \times 16}, P_{16 \times 8}, P_{8 \times 16}, P_{8 \times 8}\}$$

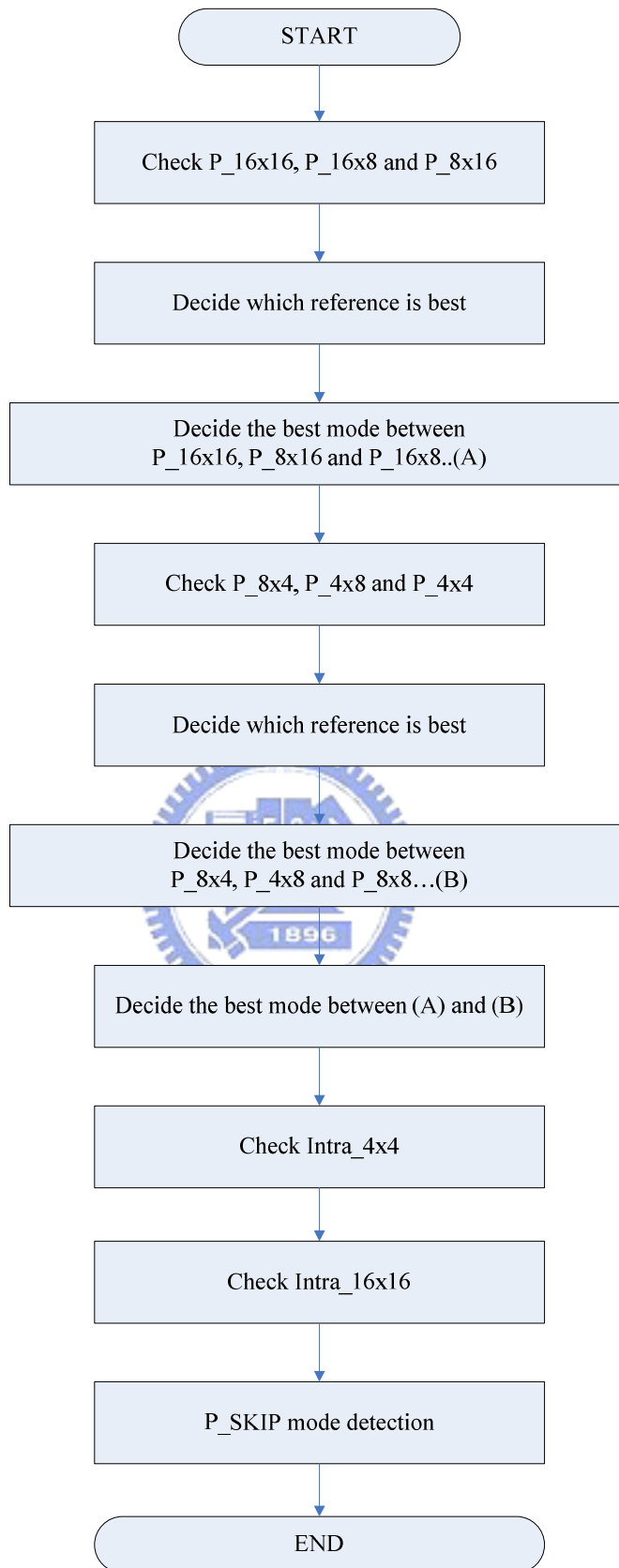
The  $DIRECT$  mode is particular to the B slices, while the  $SKIP$  mode implies that no motion or residual information will be encoded.

In the above mode sets, when the best mode is intra mode (I4MB, I16MB), the mode is chosen through evaluation of the Lagrangian function with mode choices from the mode described in section 2.4

When the best mode is inter mode, the best inter mode is chosen from 7 different block ( $P_{16 \times 8}$ ,  $P_{8 \times 16}$ ,  $P_{16 \times 8}$ ,  $P_{8 \times 8}$ ,  $P_{8 \times 4}$ ,  $P_{4 \times 8}$ , and  $P_{4 \times 4}$ ) shown in Figure 2-3 and Figure 2-4. Figure 2-9 shows the flow chart of H.264/AVC mode decision algorithm.

In order to evaluate the least RD cost for a single mode, we need to calculate the rate and distortion for all modes. For example when we choose the best mode for a  $16 \times 16$  macroblock belonging to a P or B slices (luma component only), we need 144 cost evaluations for the best I4MB mode (9 modes time 16 partitions of  $4 \times 4$  blocks), 4 more evaluations for the I16MB case, 16 more for the best  $P_{8 \times 8}$  inter mode (4 modes times 4 partitions of  $8 \times 8$  blocks) and 4 more for selecting the minimal cost among the rest of the modes results in 168 evaluations. Coupled with similar cost evaluations for each of the chroma components, the complexity analysis clearly shows that the mode decision process is computationally intensive. There are a lot of methods to reduce the complexity from mode decision in the H.264/AVC encoder. And x264 uses some algorithm to reduce the complexity of the mode decision module. In Chapter 5, we will introduce the method and algorithm which is used in x264 algorithm.





**Figure 2-9** H.264 mode decision algorithm

## 2.6 Loop Filter

One particular characteristic of block-based coding is the accidental production of visible block structures. Block edges are typically reconstructed with less accuracy than interior pixels and “blocking” is generally considered to be one of the most visible artifacts with present compression methods. H.264/AVC defines an adaptive in-loop deblocking filter, where the strength of filtering is controlled by the values of several syntax elements. The deblocking filter is applied after the inverse transform. The filter has two benefits: (1) block edges are smoothed, improving the appearance of decoded images and (2) the filtered macroblock is used for motion-compensated prediction of further frames in the encoder, resulting in a smaller residual after prediction. The basic of filter is that if a relatively large absolute difference between samples near a block edge is occurred, we can use a QP threshold to measure. So it is quite likely a blocking artifact and should be reduced. However, if the magnitude of that difference is so large that it cannot be explained by the coarseness of the quantization used in the encoding, the edge is more likely to reflect the actual behavior of the source picture and should not be smoothed over. The deblocking filter is an adaptive filter that adjusts in strength depending upon compression mode of a macroblock, the quantization parameter, motion vector, frame or field coding decision and the pixel values. When the quantization step size is decreased, the effect of the filter is reduced, and when the quantization step size is very small, the filter is shut off. The filter can also be shut off explicitly or adjusted in overall strength by an encoder at the slice level. More details about deblocking filter are described in [3].

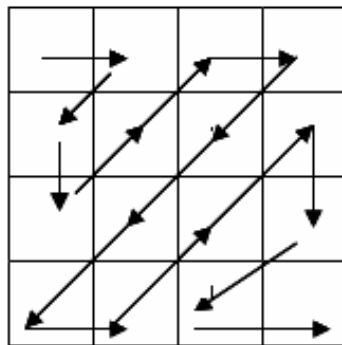
## 2.7 Transform and Quantization

The difference between the actual and predicted data is called residual error data. Discrete Cosine Transform (DCT) is a popular block-based transform for image and video compression. Similar to previous video coding standards, H.264/AVC utilizes transform coding of the prediction residual. H.264/AVC uses three transforms depending on the type of residual data that is to be coded: a transform for the 4x4 array of the luma DC coefficients in

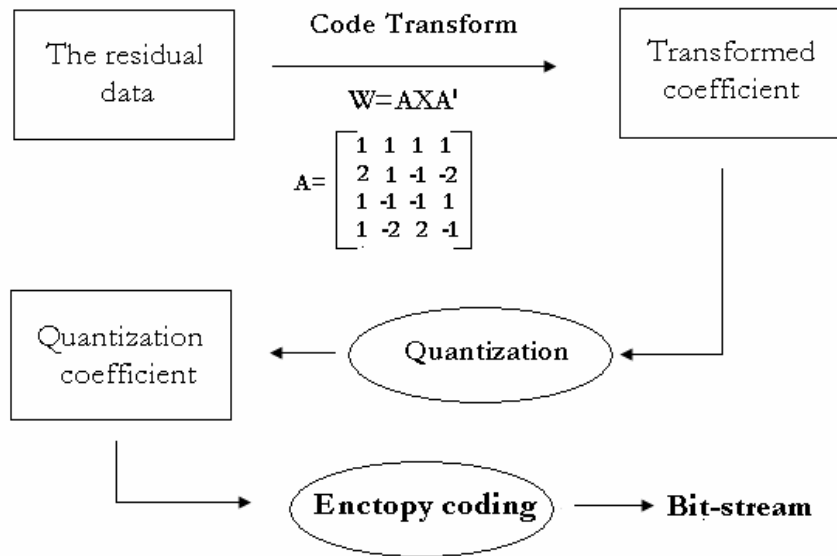
intra macroblocks, a transform for the 2x2 array of the chroma DC coefficients and a transform for all other 4x4 blocks in the residual data. In H.264/AVC, the transform is applied to 4x4 blocks, and instead of 4x4 DCT, a separable integer transform with similar properties as a 4x4 DCT is used. The 4x4 DCT integer transform is approximation of original floating point DCT transform. Since the inverse transform is defined by exact integer transform, inverse transform mismatches is avoided. The 4x4 integer transform is designed to be so simple that it can be implemented using just a few additions, subtractions, and bit shifts. The transform matrix is given as:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$$

The basic transform coding process is very similar to that of previous standards including a forward transform, zig-zag scanning shown in Figure 2-10, scaling, and rounding as the quantization process followed by entropy coding. The flow is shown in Figure 2-11.



**Figure 2-10** Zig-Zag Scan



**Figure 2-11** Flow of transform and quantization [4]

The main functionality of quantization is to scale down the transformed coefficients and to reduce the coding information. Because of human visual system is less sensitive to high frequency image component. Some video and image compression standards may use higher scaling-value for high frequency data. H.264/AVC uses a scalar quantizer. The basic forward quantizer operation is as follows:

$$Z_{ij} = \text{round}(Y_{ij} / Qstep)$$

Where  $Y_{ij}$  is a coefficient of the transformed described above,  $Qstep$  is a quantizer step size and  $Z_{ij}$  is a quantized coefficient. A total of 52 values of  $Qstep$  are supported by the standard and these are indexed by a quantization Parameter,  $QP$ . The values of  $Qstep$  corresponding to each  $QP$  are shown in Table 2-1. Note that  $Qstep$  doubles in size for every increment in  $QP$ . The wide range of quantizer setup sizes makes it possible for an encoder to accurately and flexibly control the trade-off between bit rate and quality.

**Table 2-1** Quantization step size in H.264 [3]

QP	0	1	2	3	4	5	6	7	8	9	10	11	12	
Qstep	0.63	0.69	0.81	0.88	1	1.13	1.25	1.38	1.63	1.75	2	2.25	2.5	
QP		18		24		30		36		42		48		51
QStep		5		10		20		40		80		160		224

## 2.8 Entropy Coding

The entropy encoder is responsible of converting the syntax elements to bit stream and then the entropy decoder can recover syntax elements from bit stream. H.264/AVC standard defines two entropy coding methods: Context Adaptive Variable Length Coding (CAVLC) and Context Based Adaptive Arithmetic Coding (CABAC). For the baseline profile, only CAVLC is employed. For the main profile, both CAVLC and CABAC must be supported.



# Chapter 3

## Scalable Extension of H.264

Motion pictures are to be transmitted over variable bandwidth channels, both in wireless and cable networks. They have to be stored on media of different capacity and displayed on a variety of devices, ranging from small mobile terminals to high-resolution video projection systems. Scalable video coding schemes are intended to encode the signal once at highest resolution, but enable decoding from partial streams at the specific rate and resolution required by a certain application. This scheme provides a simple and flexible solution for transmission over heterogeneous networks, additionally providing adaptability for bandwidth variations and error concealment. An example of applications is shown in Figure 3-1.

The scalable extension of H.264/AVC has been chosen to be the starting point of MPEG Scalable Video Coding (SVC) standardization project in October 2004. In January 2005, MPEG and the Video Coding Experts Group (VCEG) of the ITU-T agreed to jointly finalize the SVC project as an amendment of their H.264/AVC standard. The working draft provides a specification of the bit-stream syntax and the decoding process. The reference encoding process is described in the Joint Scalable Video Model (JSVM). Both can be downloaded from the web site [5]. The new standard is based on the architecture of H.264 [2] and provides types of scalability i.e. temporal, spatial and SNR. More details about the scalable extension of H.264/AVC can be found in [6] [7].

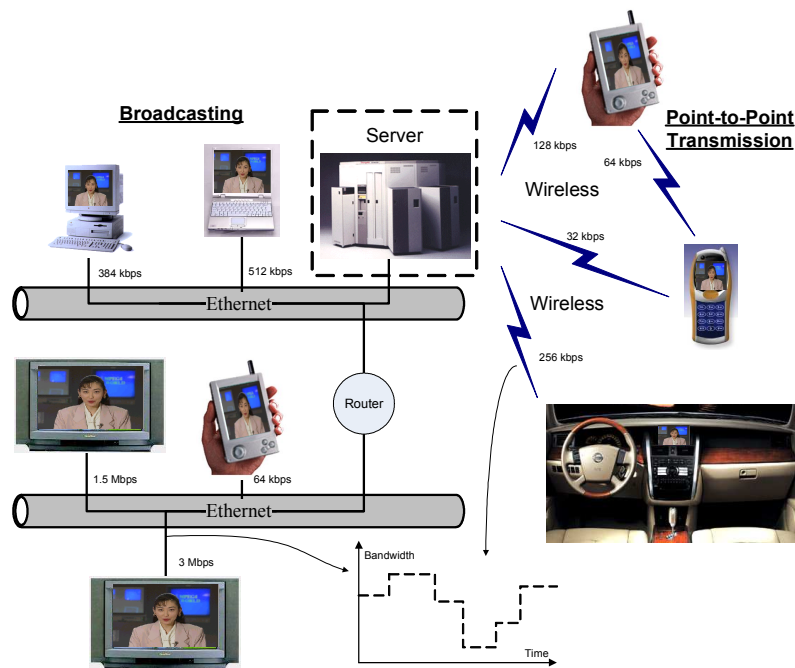
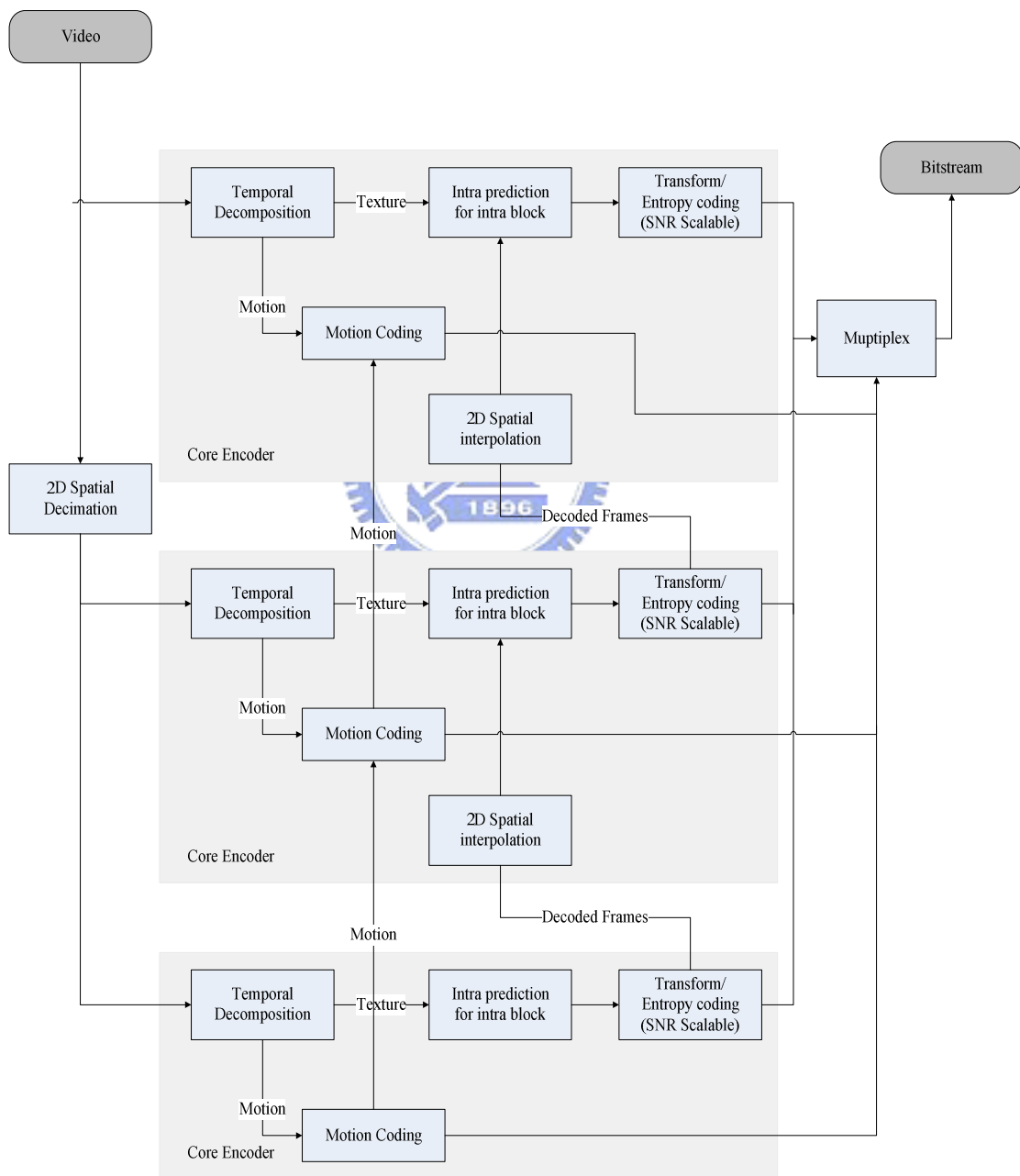


Figure 3-1 Example of Scalable Video Coding

### 3.1 The Architecture of Scalable Extension of H.264

The overall structure of scalable extension of H.264 encoder is shown in Figure 3-2. It encodes the video into multiple spatial, temporal and SNR layers for combined scalability. The spatial scalability can be realized by a layered approach. When we compress a frame, we separate different coding layer for different frame resolution. The base layer contains a lowest spatial resolution version of each coded frame. The enhancement layers have higher resolution and can be predicted from the base layer pictures and previously encoded enhancement layer pictures. The information of enhancement layer can be predicted from the base layer includes the motion vector, intra texture and the residual. The constrained inter-layer prediction is used for reduced decoder complexity. In the same spatial resolution, the temporal scalability means the change of frame rate. The temporal scalability is to extend the hybrid video coding approach of H.264/AVC towards motion compensated temporal filtering (MCTF) by using a lifting framework. By using the highly efficient motion model of H.264/AVC in conjunction with a block-adaptive switching between the Haar and the 5/3 transform, both the prediction and the update step are similar to motion compensation

techniques with hierarchical-B frame of H.264/AVC. We can use the MCTF to achieve the scalability of frame rate. In addition, the SNR scalability can be achieved by residual quantization with very little changes to H.264/AVC. This method is similar as the FGS bit-plane coding of MPEG-4 to achieve the scalability of quality. The SNR scalability includes two aspects: Fine Granularity Scalability (FGS) and Coarse Granularity Scalability (CGS).



**Figure 3-2** Basic structure for the scalable extension of H.264/AVC [8]

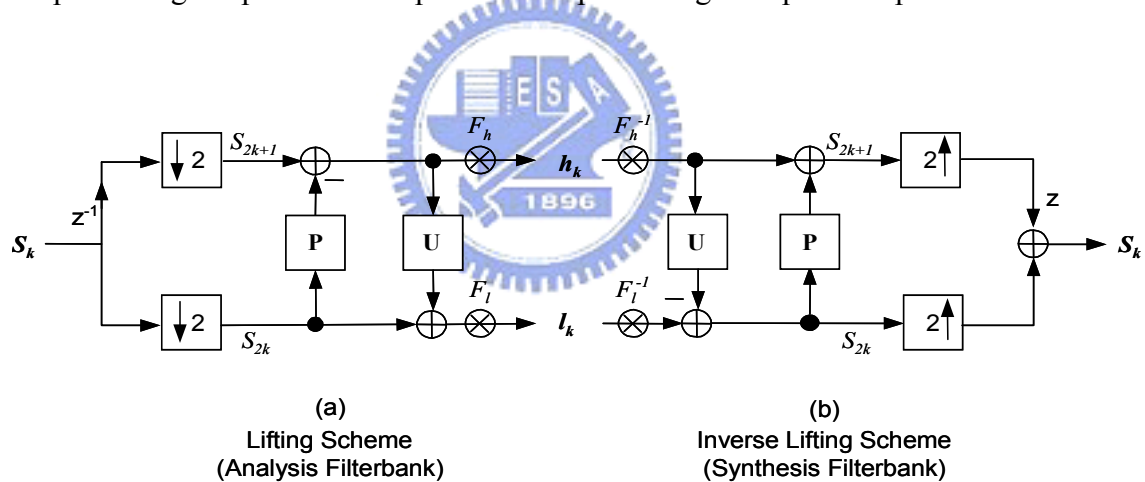


## 3.2 Temporal Scalability

Temporal scalability is often used in practice, as reduction of the video frame rate. It is a common approach in cases where insufficient transmission capacity is available. MCTF is a main feature for spatiotemporal wavelet filtering techniques.

### 3.2.1 MCTF

The Motion Compensated Temporal Filtering (MCTF) is based on the lifting scheme. The lifting scheme has two main advantages: It provides a way to compute the wavelet transform in an efficient way and it insure perfect reconstruction of the input in the absence of quantization of the wavelet coefficients. The generic lifting scheme consists of three steps: poly-phase operation, prediction, and update. Figure 3-3 shows a two-channel filter bank with “P” representing the prediction step and “U” representing the update step.



**Figure 3-3** Lifting representation of an analysis-synthesis filter bank [8]

At the analysis side (a), the odd samples  $s[2k+1]$  of a given signal  $s$  are predicted by a linear combination of the even samples  $s[2k]$  using a prediction operator  $\mathbf{P}(s[2k])$  and a high pass signal  $h[k]$  is formed by the prediction residuals. A corresponding low-pass signal  $l[k]$  is obtained by adding a linear combination of the prediction residuals  $h[k]$  to the even samples  $s[2k]$  of the input signal  $s$  using the update operator  $\mathbf{U}(h[k])$ :

$$h[k] = s[2k+1] - P(s[2k]) \quad \text{with} \quad P(s[2k]) = \sum_i p_i s[2(k+i)]$$

$$l[k] = s[2k] + U(h[k]) \quad \text{with} \quad U(h[k]) = \sum_i u_i h[k+i]$$

Where the prediction operator “P” and the update operator “U” for the temporal decomposition using the lifting representation can be used by the Haar wavelet or 5/3 transform. For the Haar wavelet are given by

$$P_{\text{Haar}}(s[x, 2k]) = s[x, 2k]$$

$$U_{\text{Haar}}(h[x, k]) = \frac{1}{2} h[x, k]$$

For the 5/3 transform, the prediction and the update operators are given by

$$P_{5/3}(s[x, 2k]) = \frac{1}{2} (s[x, 2k] + s[x, 2k+2])$$

$$U_{5/3}(h[x, k]) = \frac{1}{4} (h[x, k] + h[x, k-1])$$

Where  $s[x, k]$  be a video signal with the spatial coordinate  $\mathbf{x}=(x, y)^T$  and the temporal coordinate  $k$  in scalable video coding.

The extension to motion-compensated temporal filtering is realized by modifying the prediction and the update operators as follows

$$P_{\text{Haar}}(s[x, 2k]) = s[x+m_{p_0}, 2k-2r_{p_0}] \quad U_{\text{Haar}}(h[x, k]) = \frac{1}{2} h[x+m_{u_0}, k+r_{u_0}]$$

$$P_{5/3}(s[x, 2k]) = \frac{1}{2} (s[x+m_{p_0}, 2k-2r_{p_0}] + s[x+m_{p_1}, 2k+2+2r_{p_1}])$$

$$U_{5/3}(h[x, k]) = \frac{1}{4} (h[x+m_{u_0}, k+r_{u_0}] + h[x+m_{u_1}, k-1-r_{u_1}])$$

The reference indices  $r$  allows a general frame-adaptive motion-compensated filtering. The  $\mathbf{m}$  means the motion vectors.

### 3.2.2 Scalability Dimensions

The temporal coding structure of MCTF is changed relative to hybrid video coding in that not only high-pass pictures  $H^k$  are resulting from the prediction step but also low-pass pictures  $L^k$  are resulting from the update step. Figure 3-4 is an example for the temporal decomposition of a group 8 pictures (GOP = 8) using 3 decomposition stages. This structure provides a non-dyadic decomposition in the contrast layer. If only the level 3 pictures are obtained after the third decomposition stage is transmitted, the picture sequence that can be reconstructed at the decoder side has the 1/4 of the temporal resolution of the input sequence. By additionally transmitting the higher level (level 2) pictures, the decoder can reconstruct an approximation of the picture sequence that has 1/2 of the temporal resolution of the input sequence. And finally, if the highest level (level 1) pictures are transmitted, a reconstructed version of the original input sequence with the full temporal resolution is obtained.

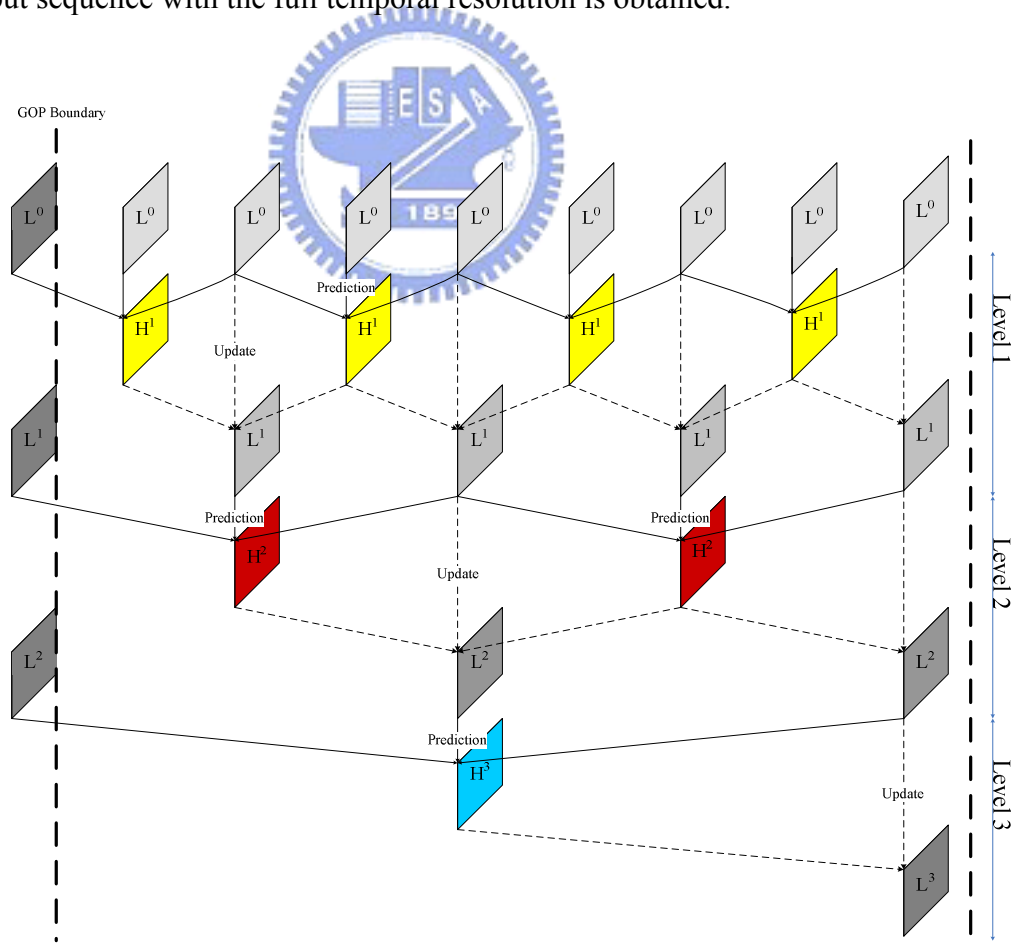
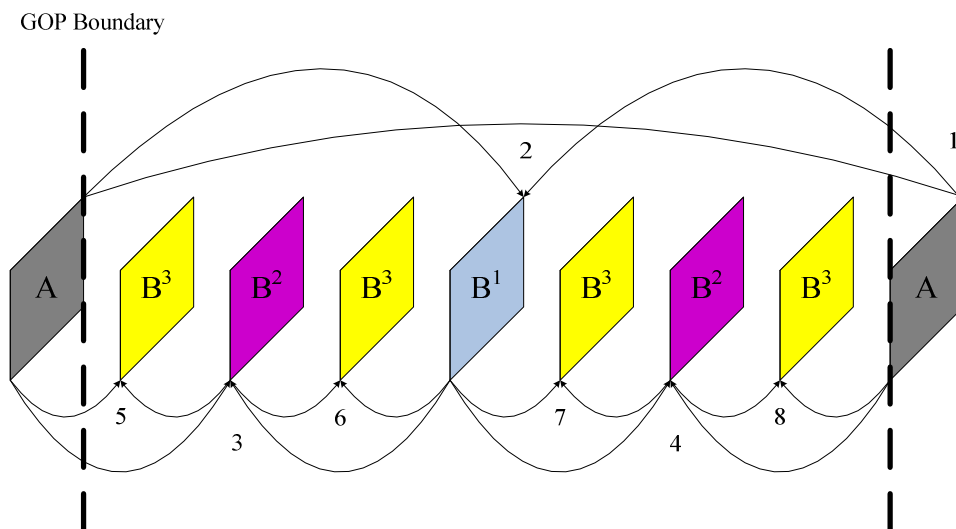


Figure 3-4 Illustration of temporal scalability [9]

The temporal coding structure of MCTF is an open-loop structure. With MCTF, the encoder can provide better prediction. However, it may cause mismatch error between encoder and decoder in the presence of quantization error and the update step increase the complexity and memory requirement. In order to justify the complexity of the update step, temporal scalability uses a closed-loop structure which is known as “hierarchical-B”. The hierarchical architecture of temporal scalability is described in Figure 3-5. This is an example of the prediction structure for a group of eight pictures. The first picture of a video sequence is intra-coded as the instantaneous decoder refresh (IDR) picture that is a kind of the key picture. The key picture of the sequence is independent from any other pictures of the video sequence, and it generally represents the minimal temporal resolution that can be decoded. It is either intra-coded or inter-coded. When the key picture is decoded, the picture  $B^1$  is predicted by using the surrounding key pictures A as references. It depends only on the key pictures, and represents the next higher temporal resolution together the key pictures, the pictures  $B^2$  of the next temporal level are predicted by using only the picture of the lower temporal resolution as references, etc. It is obviously that this hierarchical prediction structure inherently provides temporal scalability. The main idea is similar as the B frames of the H.264/AVC.



**Figure 3-5** Hierarchical-B prediction structure

### 3.3 Spatial Scalability

In the spatial scalability, we use an over-sampled pyramid structure to represent multiple resolutions (ex. QCIF, CIF, and 4CIF) and code the various spatial resolutions independently of each other. The information of a higher spatial layer is affected by the information of the lower spatial layers. We can code the higher spatial layer efficiently by predicting from the lower spatial layers. For that, the following techniques turned out to provide gains and are described below:

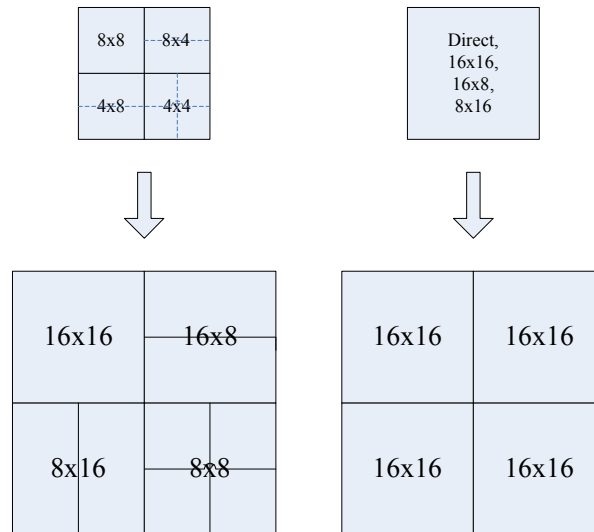
1. Prediction of a macroblock using the up-sampled lower resolution signal
2. Prediction of motion vectors using the up-sampled lower resolution motion vectors
3. Prediction of the residual signal using the up-sampled residual signal of the lower resolution layer

In inter-layer prediction, motion prediction is used to remove the redundancy of motion information, including macroblock partition, reference picture index, and motion vector among layers. The macroblock partitioning is obtained by up-sampling the partitioning of the corresponding 8x8 block of the lower resolution layer. For the obtained macroblock partitions, the corresponding sub-macroblock partition of the base layer block is used as shown in Figure 3-6. The motion vector is scaled by a factor of 2. For the motion information, we introduce two additional modes. While for the first of these modes (Base\_layer\_mode) no additional motion information are coded, for the second one (Qpel\_refinement\_mode), a quarter-sample motion refinement (-1, 0, or +1 for each motion vector component) is transmitted for each motion vector.

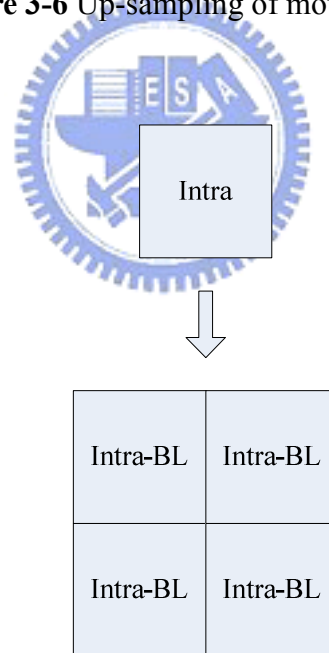
Intra texture prediction uses the reconstructed image of the reference layer to predict an enhancement layer. For intra texture prediction, we use the “Intra\_BL” mode. The “Intra\_BL” mode is only allowed for macroblock, for which the corresponding 8x8 block of the base layer is located inside an intra-coded macroblock. This is described in Figure 3-7. In this mode, the prediction signal is directly obtained by de-blocking and up-sampling the 8x8 luma block inside the corresponding lower layer picture.

In H.264/AVC SVC, the residual prediction is performed in spatial domain. In the inter-layer, consecutive spatial layers may have similar motion information. Thus, the

residuals of consecutive layers may have some correlations. The residual information is coded in the lower resolution layer using a bi-linear filter with constant border extension.



**Figure 3-6** Up-sampling of motion data [8]



**Figure 3-7** Up-sampling of intra texture [8]

### 3.4 SNR Scalability

In the SNR scalability, H.264/AVC-compatible transform coding is used. For the residual macroblocks, the coding as in H.264/AVC including transformation and quantization is

employed. For each macroblock, the coded block pattern (CBP), and the conditioned on CBP the corresponding residual blocks are transmitted together with the macroblocks modes, intra prediction modes, reference picture indices and motion vectors using the B or P slice syntax of H.264/AVC. For that, the quantization error between the SNR base layer and the original sub-band pictures is re-quantized exactly using the same methods as for the base layer but with a finer quantization step size. In the SNR scalability, we can divide into two aspects: Coarse Granularity Scalability (CGS) and Fine Granularity Scalability (FGS).

### 3.4.1 CGS

The mechanism of CGS is similar as spatial scalability. The CGS also can be realized by a layered coding. Each CGS layer has its own motion information and temporal prediction. On top of the SNR base layer, the enhancement layer is coded. For that, the quantization error between the SNR base layer and the original pictures is transformed and quantized exactly using the same method as for the base layer but with a finer quantization step size. The enhancement layer with the base layer can be used the same method again. Figure 3-8 illustrates the idea.

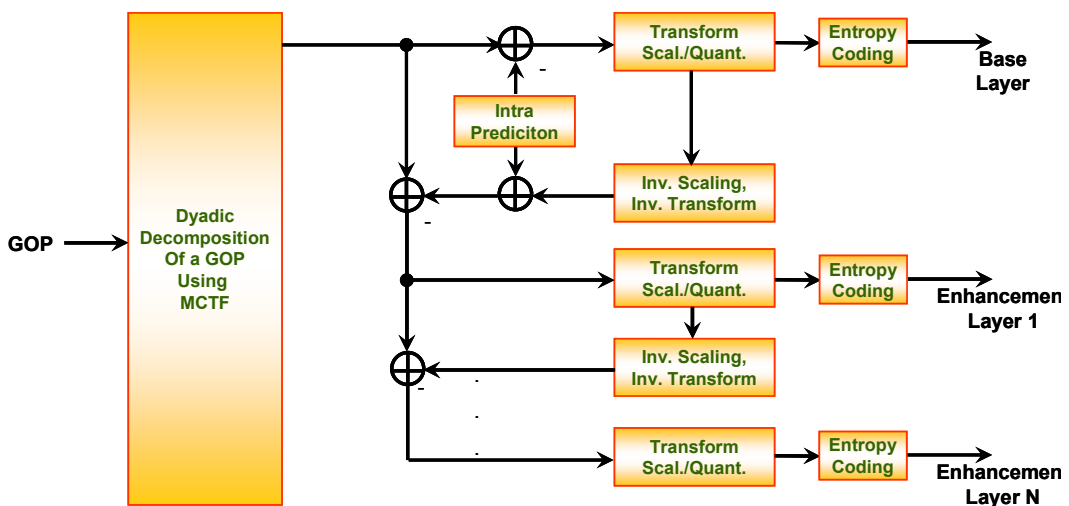
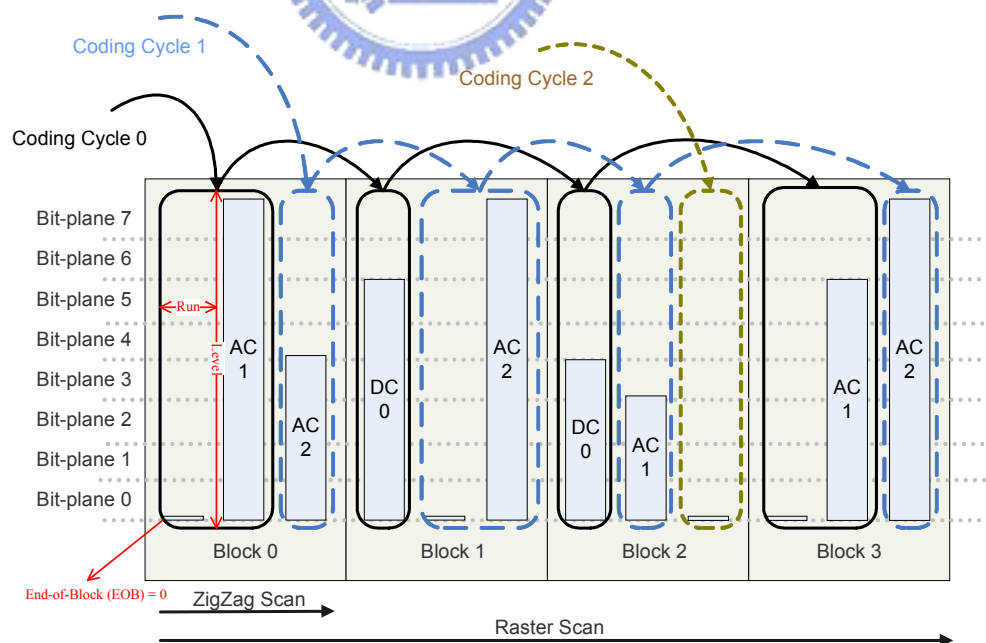


Figure 3-8 CGS SNR scalable coding scheme [10]

### 3.4.2 FGS

In order to support fine granularity scalability (FGS), we have introduced an algorithm so-called progressive refinement slices. The algorithm encodes coefficients in order such that “more significant” coefficients are coded first. By arranging the bit stream in this way, we can truncate the bit stream at any arbitrary point and retain the “more significant” coefficients first, so that the quality of SNR base layer can be improved in a fine granular way. The progressive refinement in FGS uses cyclic block coding. The coefficients are scanned in zig-zag scan as shown in Figure 2-10 and the current scan position in a given coding pass will differ from one block to another. When a coding pass in one block is finished, we need to change the next block to perform a coding pass. In general, the progressive refinement encodes the DC coefficient first in the same cycle for every block. In next cycle encodes the other coefficients. This is shown in Figure 3-9. This progressive refinement slices using cyclical block coding can improve the quality of every block averagely and is in order to support fine granular quality scalability.



**Figure 3-9** Scheme of Cyclical Block Coding



In FGS layer, a block is coded using two passes: significant pass and refinement pass. The significant pass encodes the coefficient that became significant in the enhancement layer. The refinement pass encodes the coefficient for which a nonzero value has already been coded in the previous coding pass. At each cycle, for the significant pass, the coefficients are scanned in zig-zag order for every block, and all zero values are coded up to and including the first nonzero value. Then, the next block is processed. Each coding cycle in a block includes an End-of-Block (EOB) symbol, a Run index, and a non-zero quantization index. In refinement pass, refinement values are coded when all significant values have been coded for all block. Figure 3-10 is an example of a slice consists of four blocks having eight coefficients each.

Block 0	0	0	0	A	0	B	0	1	0	1	C	D	E	0	0	0
Block 1	F	0	1	0	1	0	0	0	0	0	0	0	0	G	0	0
Block 2	1	1	1	1	0	0	0	H	I	0	0	J	0	0	K	0
Block 3	0	L	0	M	0	1	1	0	0	0	0	0	0	0	0	0

Figure 3-10 Example of significant and refinement pass [11]

Initially, in the significant pass, we encode the first nonzero value for each block. So the coefficient for each cycle can be discussed in the follows:

**Cycle 0** = 0 : { 0 0 0 0 0 1 }, 1 : { 0 1 } 2 : { 1 } 3 : { 0 0 0 1 }

**Cycle 1** = 0 : { 0 1 } 1 : { 0 1 } 2 : { 1 } 3 : { 1 }      **Cycle 2** = 1 : { EOB } 2 : { 1 } 3 : { EOB }

**Cycle 3** = 2 : { 1 }      **Cycle 4** = 2 : { EOB }

The symbol of EOB indicates the last significant coefficient flag.

In the refinement pass, each cycle only can encode one refinement value for each block. So the finally coefficient can be presented in the follows:

**Cycle 0** = 0 : { 0 0 0 0 0 1 } 1 : { 0 1 } 2 : { 1 } 3 : { 0 0 0 1 }      **Cycle 1** = 0 : { 0 1 } 1 : { 0 1 } 2 : { 1 } 3 : { 1 }

**Cycle 2** = 0 : { EOB } 1 : { EOB } 2 : { 1 } 3 : { EOB }      **Cycle 3** = 2 : { 1 }      **Cycle 4** = 2 : { EOB }

**Cycle 5** = 0 : { A } 1 : { F } 2 : { H } 3 : { L }      **Cycle 6** = 0 : { B } 1 : { G } 2 : { I } 3 : { M }

**Cycle 7** = 0 : { C } 2 : { J }      **Cycle 8** = 0 : { D } 2 : { K }      **Cycle 9** = 0 : { E }

### 3.5 Combined Scalability

Figure 3-11 is an example of the combination of spatial, temporal and quality scalability. In the same resolution layer, we can use the MCTF to achieve the temporal scalability. In different resolution layer, we can use the inter-layer prediction to code different resolution picture. In addition, in every layer, we can adjust the quantization for quality scalability. This can provide a wide range of temporal, SNR, and spatial scalability.

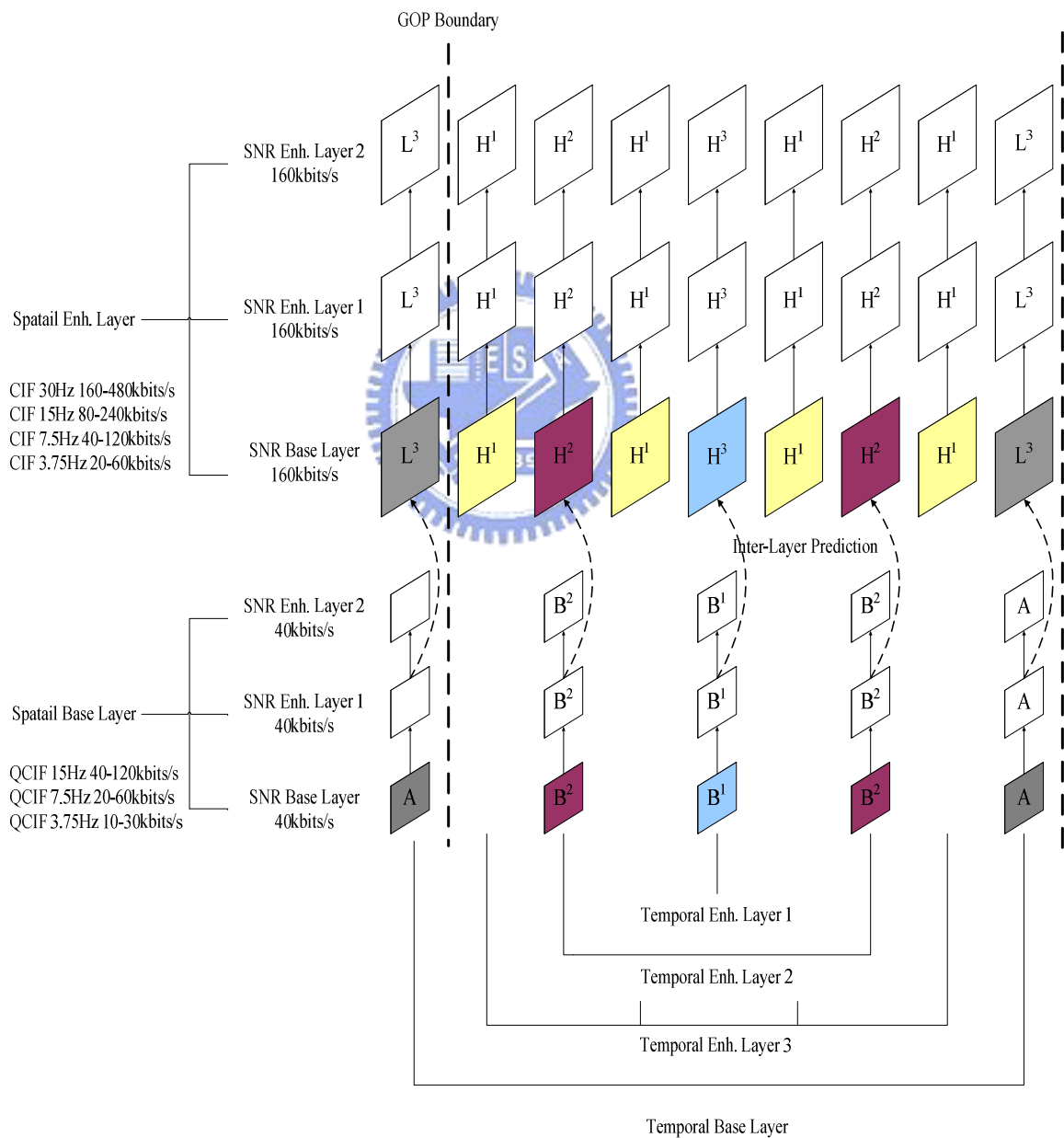
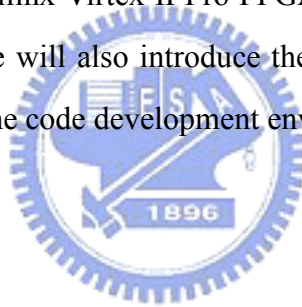


Figure 3-11 Combined scalability

# Chapter 4

## DSP Implementation Environment

As discussed previously, our project involves the implementation on digital signal processors (DSPs). In this chapter, we briefly describe the DSP platform environment. In our DSP board, we use the Sundance module (SMT395). Its main chips are the TMS320C6416T DSP made by Texas Instrument and the Xilinx Virtex II Pro FPGA. We will introduce the DSP chip and the DSP board. In addition, we will also introduce the software development tool, the Code Composer Studio (CCS), and the code development environment for TI DSP.

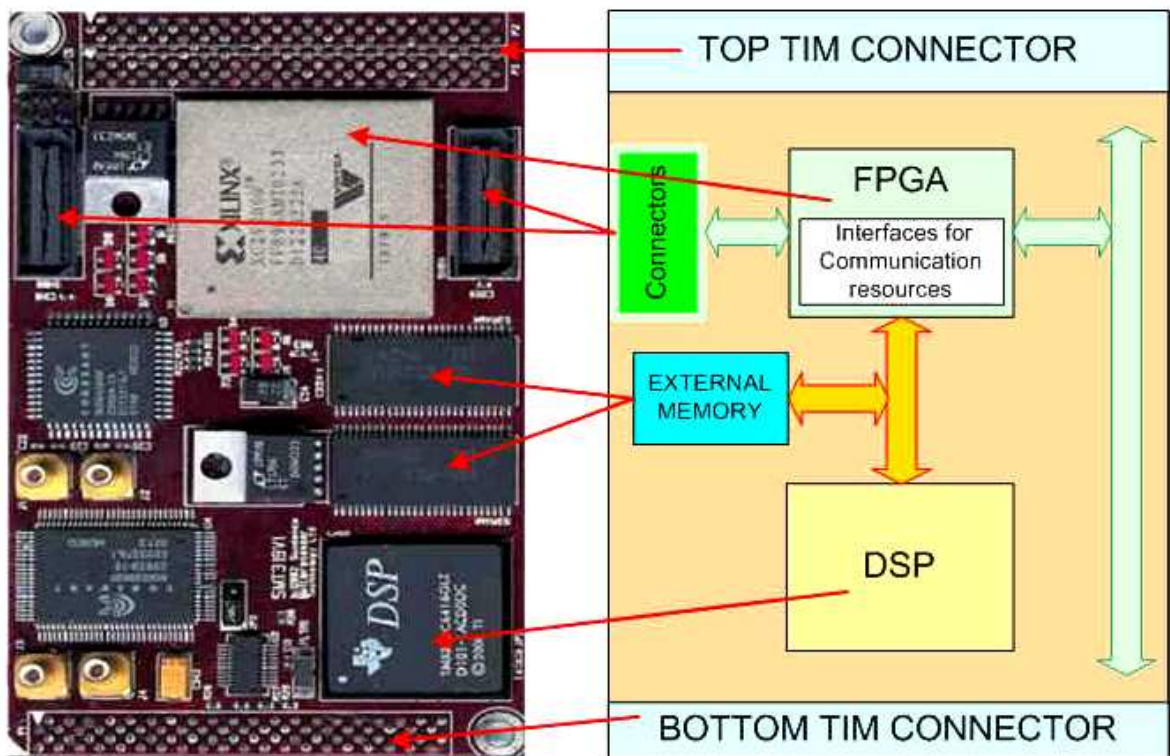


### 4.1 The DSP Board

The DSP board use in our implementation is the Sundance module (SMT395) shown in Figure 4-1. SMT395 is used the 1GHz 64-bit TMS320C6416T DSP, which is manufactured using the latest 90nm wafer technology and it offers high fixed-point processing power. The SMT395 is supported by the TI Code Composer Studio and 3L\_Diamond\_RTOS to enable full MultiDSP systems with minimum efforts by the programmers. It provides a flexible platform for the next generations of telecom systems, image processing applications, medical equipment and industrial solutions. We list some specifications of SMT395 modules as follows [12].

- 1GHz TMS320C6416T Fixed Point DSP
- 8000MIPS peak performance

- Xilinx Virtex II Pro FPGA. XC2VP30-6 in FF896 package.
- 256Mbytes of SDRAM @ 133MHz
- Two Sundance High-speed Bus (50MHz, 100Mhz or 200MHz) ports 32 bits wide
- Eight 2.5Gbit/sec Rocket Serial Links (RSL) for Inter Module communications
- 8Mbytes FLASH ROM for configuration/booting
- JTAG Diagnostics Port

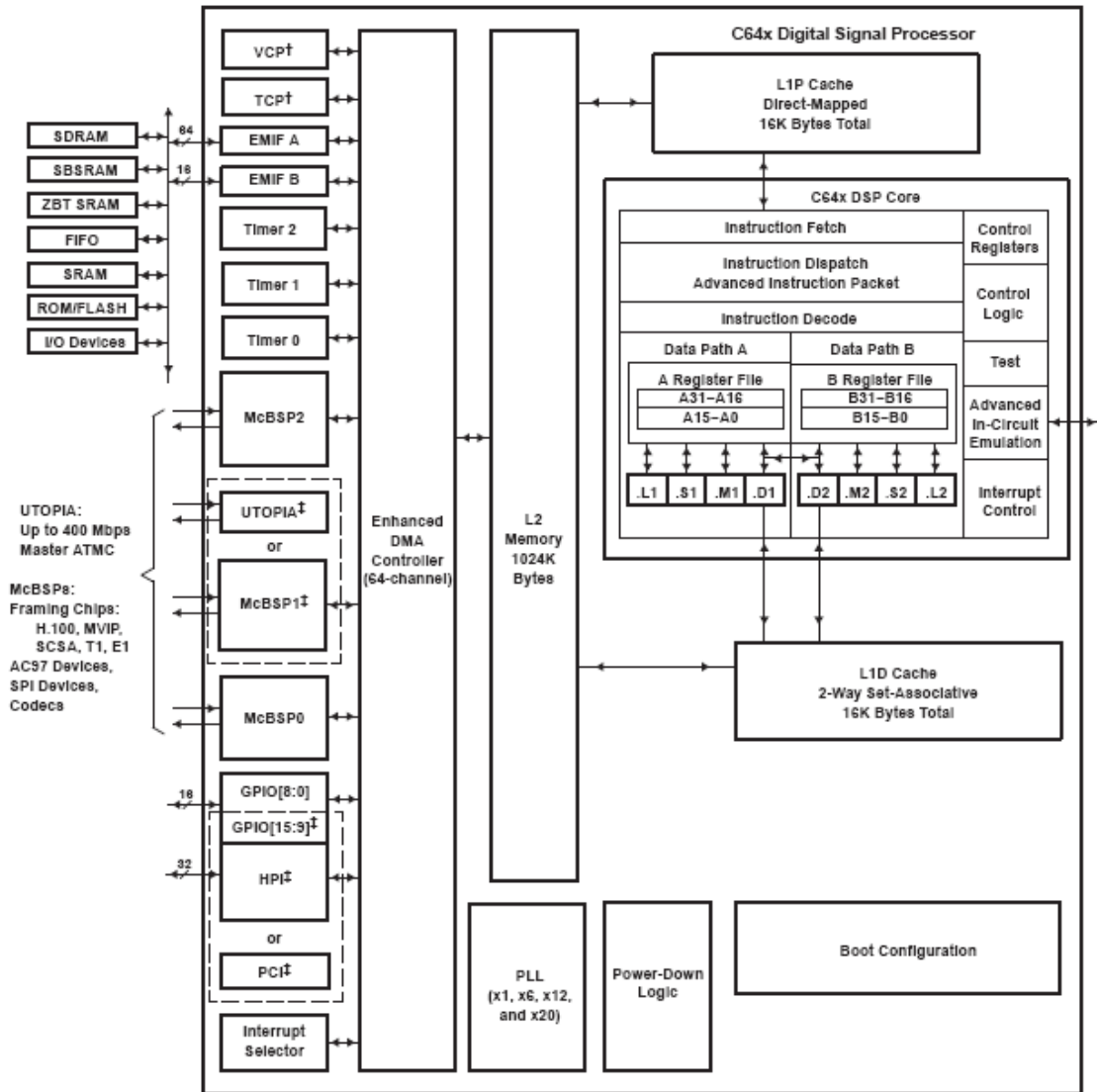


**Figure 4-1** SMT395 module

## 4.2 The TMS320C6416T DSP Chip

The TMS320C6416T is the highest-performance fixed-point DSP generation in the TMS320C64x series of the TMS320C6000 DSP platform family. It is based on the second-generation high-performance, advanced VelociTI Very-Long-Instruction-Word (VLIW) architecture developed by Texas Instruments [13], making this DSP an excellent choice for

wireless infrastructure applications. The functional block and DSP core diagram of TMS320C64x series is shown in Figure 4-2.



**Figure 4-2** Block diagram of the TMS320C64x DSPs [13]

The C6000 core CPU consists of 64 general-purpose 32-bits register and 8 function units. Features of C6000 devices include [14]:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic

- Executes up to eight instructions per cycle.
- Allows designers to develop highly effective RISC-like code for fast development time.
- Instructing packing:
  - Gives code size equivalence for eight instructions executed serially or in parallel.
  - Reduces code size, program fetches, and power consumption.
- Conditional execution of all instructions:
  - Reduces costly branching.
  - Increases parallelism for higher sustained performance.
- Efficient code execution on independent functional units:
  - Efficient C compiler on DSP benchmark suite.
  - Assembly optimizer for fast development and improved parallelization.
- 8/16/32-bit data support, providing efficient memory support for a variety of applications.
- 32 x 32-bit integer multiply with 32- or 64-bit result.

The C64x extensions add enhancements to the C6000 architecture which includes:

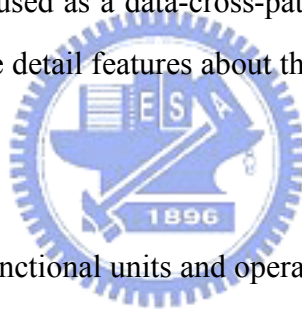
- Register file enhancement.
- Quad 8-bit and dual 16-bit extensions for data flow.
- Additional functional unit hardware.
- Increased orthogonally instruction set.

#### 4.2.1 Central Processing Unit of C64x

The C64x DSP core contains 64 32-bit general purpose register, program fetch unit, instruction decode unit, two data path which each with four function units, control register, control logic, advanced instruction packing, test unit , emulation logic and interrupt logic. The program fetch, instruction fetch, and instruction decode units can arrange eight 32-bit instructions to the eight function units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B) shown in Figure 4-2, each of which contains

four functional units and one register file. The four functional units can divide into four operations. The first unit is for multiplier operations (.M). The second unit is for arithmetic and logic operations (.L). The next one is for branch, byte shifts, arithmetic operations (.S). The last unit is for linear and circular address calculation to load and store with external memory operations (.D). The details of functional units are described in Table 4-1.

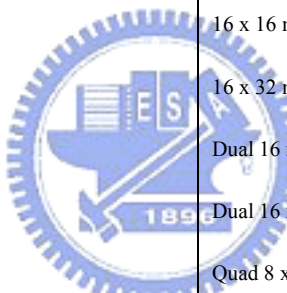
Each register file consists of 32 32-bit registers for each four functional unit reads and writes directly within its own data path. That is, the functional units .L1, .S1, .M1, .D1 can only write to register file A. The same condition occurs in register file B. However, two cross-paths (1X and 2X) allow functional units from one data path to access a 32-operand from the opposite side register file. The cross path 1X allow data path A to read their source from register file B. The cross path 2X allow data path B to read their source from register file A. In the C64x, CPU pipelines data-cross-path accesses over multiple clock cycles. This allows the same register to be used as a data-cross-path operand by multiply functional units in the same execute packet. The detail features about the C64x CPU are introduced in [13].



**Table 4-1** Functional units and operations performed [15]

Function Unit	Operations
.L unit(.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits Byte shifts Data packing/unpacking 5-bit constant generation Dual 16-bit and Quad 8-bit arithmetic operations Dual 16-bit and Quad 8-bit min/max operations

**Table 4-2** Functional units and operations performed [15]

Function Unit	Operations
.S unit (.S1, .S2)	<p>32-bit arithmetic operations</p> <p>32/40-bit shifts and 32-bit bit-field operations</p> <p>32-bit logical operations</p> <p>Branches</p> <p>Constant generation</p> <p>Register transfers to/from control register file (.S2 only)</p> <p>Byte shifts</p> <p>Data packing/unpacking</p> <p>Dual 16-bit and Quad 8-bit compare operations</p> <p>Dual 16-bit and Quad 8-bit saturated arithmetic operations</p>
.M unit (.M1, .M2)	 <p>16 x 16 multiply operations</p> <p>16 x 32 multiply operations</p> <p>Dual 16 x 16 and Quad 8 x 8 multiply operations</p> <p>Dual 16 x 16 multiply with add/subtract operations</p> <p>Quad 8 x 8 multiply with add operations</p> <p>Bit expansion</p> <p>Bit interleaving/de-interleaving</p> <p>Variable shift operations</p> <p>Rotation</p> <p>Galois Field Multiply</p>
.D unit (.D1, .D2)	<p>32-bit add, subtract, linear and circular address calculation</p> <p>Loads and stores with 5-bit constant offset</p> <p>Loads and stores with 15-bit constant offset(.D2 only)</p> <p>Loads and stores doubles words with 5-bit constant</p> <p>Loads and store non-aligned words and double words</p> <p>5-bit constant generation</p> <p>32-bit logical operations</p>



## 4.2.2 Memory Architecture and Peripherals

The C64x DSP is a two level cache-based architecture. The level 1 cache can be separated into program and data space. The level 1 program cache (L1P) is a 16 K-bytes direct mapped cache and the level 1 data cache (L1D) is a 16 K-bytes 2-way set-associative mapped cache. The level 2 (L2) consists 1024 K-bytes memory space for cache (up to 256K-bytes) and unified mapped memory.

The EMIF provides the interfaces for the DSP core and external memory, such as synchronous-burst SRAM (SBSRAM), synchronous DRAM (SRAM), SDRAM, FIFO and asynchronous memories (SRAM and EPROM). The EMIF also provides 64-bit-wide (EMIFA) and 16-bit-wide (EMIFB) memory read capability.

The C64x contains some peripherals such as enhanced direct-memory-access (EDMA), host-port interface (HPI), PCI, three multi-channel buffered serial ports (McBSPs), three 32-bit general-purpose timers and sixteen general-purpose I/O pins. The EDMA controller handles all data transfers between the level 2 (L2) cache/memory and the device peripheral. The C64x has 64 independent channels. The HPI is a 32-/16-bit wide parallel port through which a host processor can directly access the CPUs memory space. The PCI port supports connection of the DSP to a PCI host via the integrated PCI master/slave bus interface.

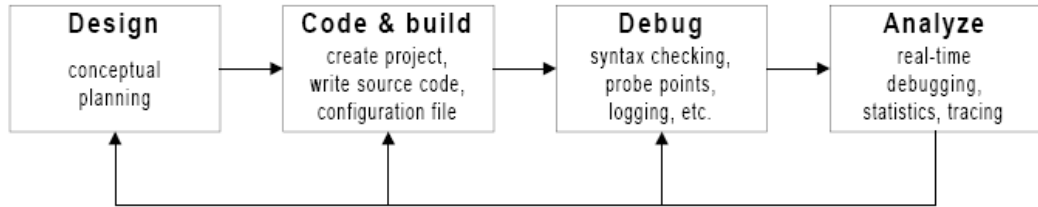
## 4.3 TI DSP Code Development Environment

TI supports a useful GUI development to DSP users for developing and debugging their project: the code composer studio (CCS). In this section, we will give a briefly introduction about this development environment. The tutorial [16] introduces the key features of CCS. A DSP users needs to familiar with the coding development tool for building project on DSP platform efficiently.

### 4.3.1 Code Composer Studio

Code Composer Studio (CCS) is a key element of the DSP software and development tools from Texas Instruments. It extends the basic code generation tools with a set of debugging and

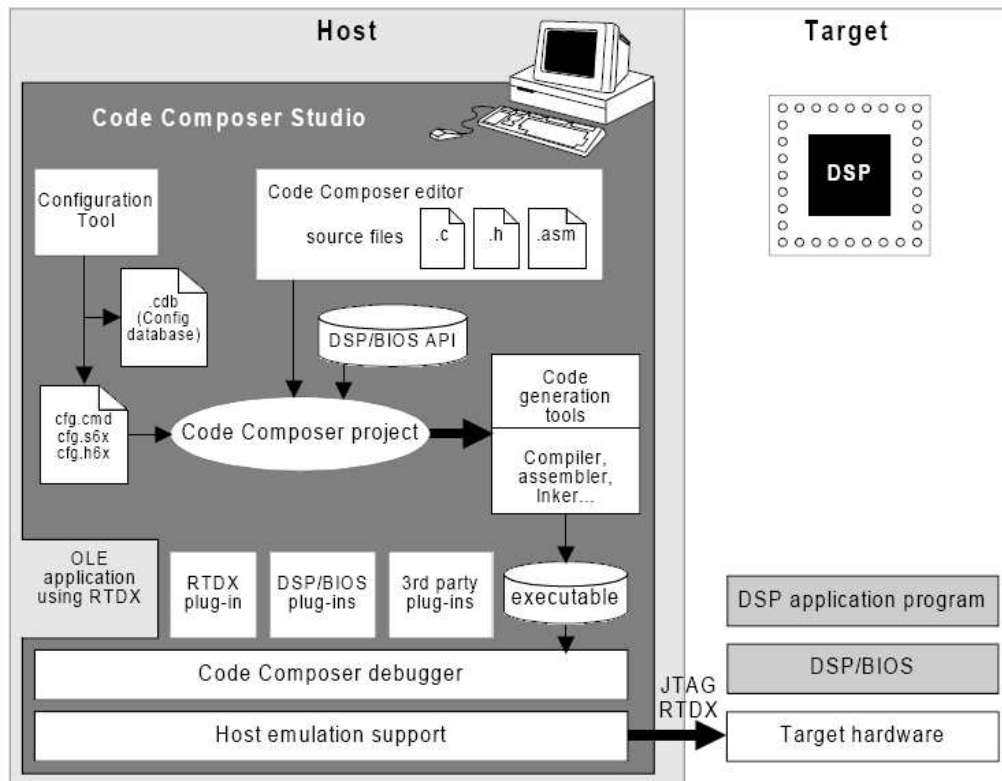
real-time analysis capabilities which supports all phases of the development cycle shown in Figure 4-3.



**Figure 4-3** Development cycle [16]

The CCS has the following components which work together as shown in Figure 4-4:


- TMS320C6000 code generation tools
- Code Composer Studio Integrated Development Environment (IDE)
- DSP/BIOS plug-ins and API
- RTDX plug-in, host interface, and API



**Figure 4-4** Code composer studio development [16]

The code generation tools provide the foundation for the development environment provided by the Code Composer Studio such as C compiler, assembler, assembly optimizer, linker, archiver, run-time-support libraries, cross-reference lister and absolute lister. The code composer studio integrated development environment (IDE) is for editing, building, and debugging DSP target programs. The code composer studio plug-ins provided with DSP/BIOS support such real-time analysis. It can be used by programmers to visually probe, trace, and monitor a DSP application with minimal impact on real-time performance. The DSP/BIOS API provides the real-time analysis capabilities such as: program tracing, performance monitoring and file streaming. In addition, TI DSP provides on-chip emulation supports that enables code composer studio to control program execution and monitor real-time program activity. The RTDX capability is exposed through host and DSP APIs, allowing for bi-directional real-time communications between the host and DSP.

### 4.3.2 Simulation Tools

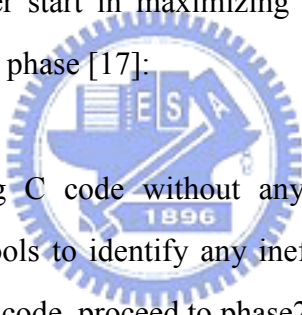


In the code development, we use the DSP board SMT395 to run the project. But the simulator can profile for analyzing coding efficiency. In CCS, the C64x CPU Cycle Accurate Simulator can simulate the cycle of the C64xx processor accurately. This is faster than the real system because it does not simulate the peripherals and the cache system. Instead we use the C6416 Device Cycle Accurate Simulator to simulate the C64x XDS510 emulator. It simulates the C64x processor and supports L1D, L1P, L2 Cache, EDMA, QDMA, Interrupt Selector, McBSP(3), Timer(3), TCP, VCP,EMIF. It also supports the interface with SDRAM and Generic sync RAM Memory models. The C6416 Device Cycle Accurate Simulator is closer to the real system. We use C64xx XDS510 emulator to verify the project. The TMS320C6416T hardware is connected via the XDS510 emulator, which sets the I/O ports on the DSP platform. Usually, the cycle provided by the simulator is less the real system.

## 4.4 Code Development Flow

The recommended code development flow involves utilizing the C6000 code generation tools to aid in optimization rather than coding by hand in assembly. The advantages allow the compiler to do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation. These features simplify the maintenance of the code, as everything resides in a C framework that is simple to maintain, support, and upgrade.

The recommended code development flow for the C6000 involves the phases described in Figure 4-5. It includes phases 1-3. These phases instruct us when to go to the tuning stage of phase 3. What is learned is the importance of giving the compiler enough information to fully maximize its potential. An added advantage is that the compiler provides direct feedback, there are some very simple steps we can take on pass complete and better information to the compiler allowing us a quicker start in maximizing compiler performance. The following items describe the goal for each phase [17]:

- 
- Phase 1: Developing C code without any knowledge of the C6000. Use the simulator profiling tools to identify any inefficient area in C code. Improving the performance of the C code, proceed to phase2.
  - Phase 2: Use techniques that belongs the DSP to improve the C code. Use the simulator profiling tools to check the performance. If the code is not as efficient as we would like it to be, go to phase3.
  - Phase 3: Extract the time-critical areas from the C code and rewrite the code in linear assembly. We can use the assembly optimizer to optimize this code.

TI provides high performance C program optimization tools, and they do not suggest the programmer to code by hand in assembly. Coding the program in phase 1 is easier than phase 3 and can cost less time.

**Phase 1:  
Development C Code**

Write C code

Compile

Profile

Efficient?

Yes → Complete

No

**Phase 2:  
Refine C Code**

Refine C code

Compile

Profile

Efficient?

Yes → Complete

No

Yes

More C optimization?

No

**Phase 3:  
Write Linear  
Assembly**

Write linear assembly

Assembly optimize

Profile

Efficient?

Yes

Complete

No

**Figure 4-5** Code development [17]

## 4.4.1 Compiler Optimization Options

The CCS compiler can accept C/C++ source code and produce C6x assembly language source code. The Figure 4-6 shows the working flow of the compiler. It is able to perform various levels of optimization. This compiler can be used to optimize code size or executing time. In this project, we only concern the executing time. The compiler specifies the `-On` option in the command line which can control the optimization level. The `n` denotes the level of optimization (0, 1, 2, and 3) and their options are given below [18]:

- `-o0`
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Perform loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- `-o1`

Performs all `-o0` optimizations, plus:

  - Performs local copy/constant propagation
  - Remove unused assignments
  - Eliminates local common expression
- `-o2`

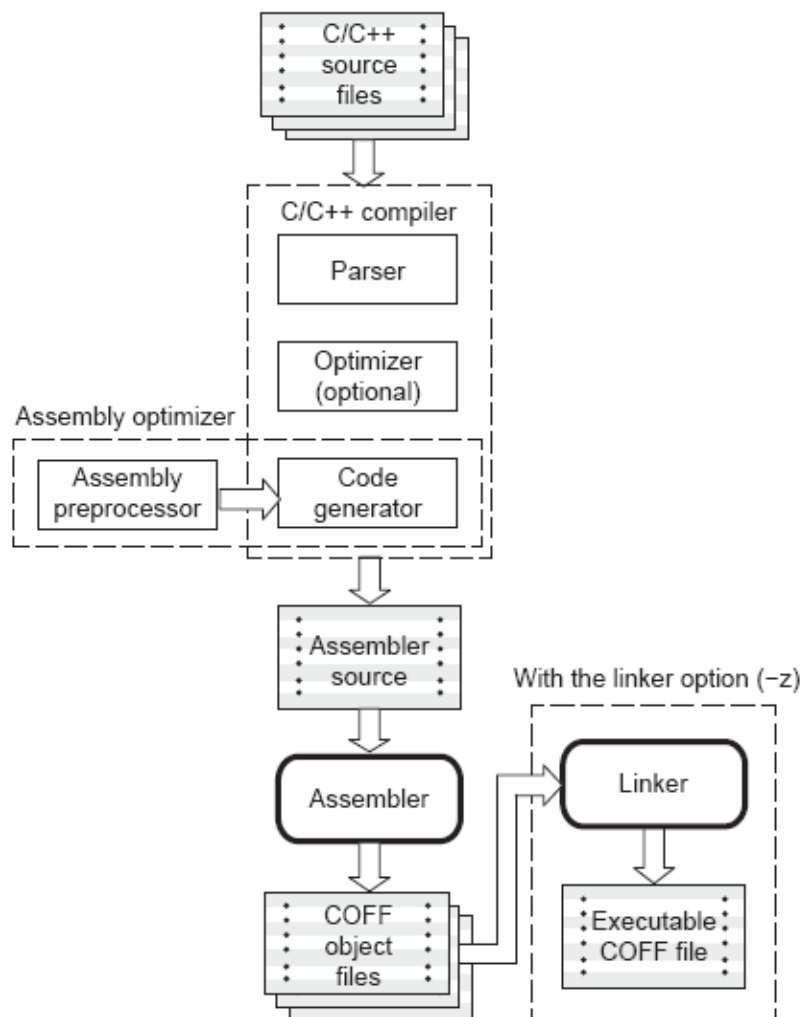
Performs all `-o1` optimizations, plus:

  - Performs software pipelining
  - Performs loop optimizations
  - Eliminates global common sub-expressions
  - Eliminates global unused assignments
  - Converts array references in loops to incremented pointer form
  - Performs loop unrolling
  -

- -o3

Performs all -o2 optimizations, plus:

- Removes all functions that are never called
- Simplifies functions with return values that are never used
- Inline calls to small functions
- Reorders function declarations; the called functions attributes are known when the caller is optimized
- Propagates arguments into functions bodies when all calls pass the same value in the same argument position
- Identifies file-level variable characteristics

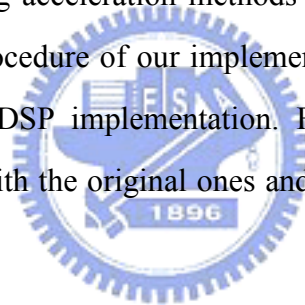


**Figure 4-6** C/C++ compiler [18]

# Chapter 5

## H.264 Encoder Implementation and Optimization on DSP Platform

In the early chapters, we introduce the H.264/AVC and the environment of the DSP implementation. In this chapter, we discuss the implementation of H.264 video encoder on the DSP C6416T board. In our implementation, we only support the H.264 baseline profile. First we describe the source code x264 which is a free library for encoding H.264/AVC streams. Then we expand some existing acceleration methods on the mode decision to speed up the x264. We also describe the procedure of our implementation work and how we optimize the x264 video encoder for the DSP implementation. Finally, we compare the performance between the modified codes with the original ones and give some experimental results of the whole system.



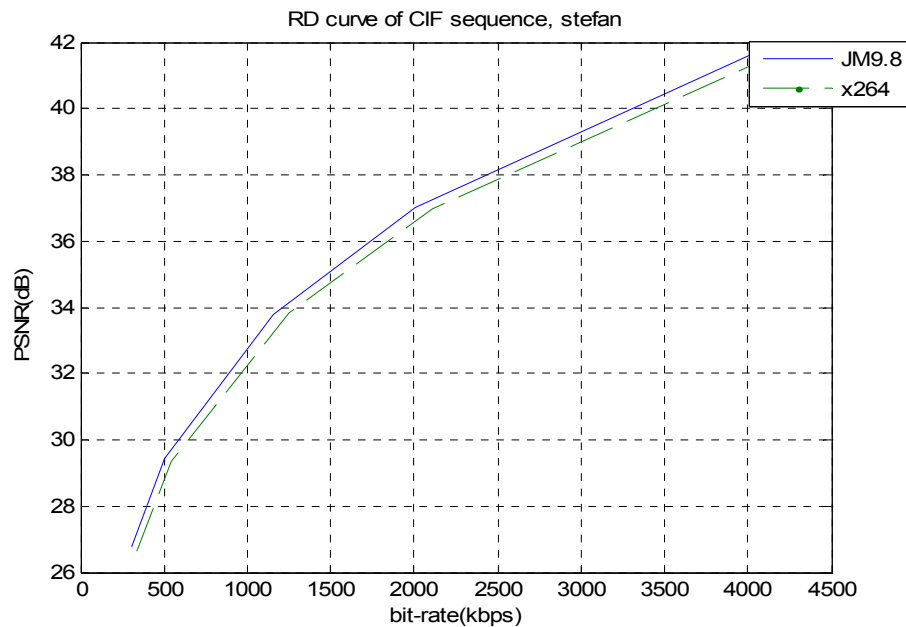
### 5.1 Introduction to x264

Like the previous video standards, H.264/AVC specifies only a decoder, therefore allowing for improvement in compression rate, quality and speed in designing the encoder. The x264 is an open source encoder of H.264 and has been used in many popular applications such as ffdshow [19] and ffmpeg [20]. All the documents and software could be downloaded from [21]. The x264 shows better quality than several commercial H.264/AVC encoders. Table 5-1 shows that when we encode the same pictures, x264 is about 18 times faster than the reference software JM9.8 [22]. Figure 5-1 is the rate-distortion comparison between JM9.8 and x264. At the same PSNR, x264 produces less than 6% more bitrates of JM. Its rate-distortion performance is close to that of JM9.8. So we choose the open source code x264 instead of JM as the starting point for implementation on DSP.



**Table 5-1** Performance of JM9.8 vs. x264

Test Sequence	QP	JM9.8(fps)	x264(fps)
Stefan	28	1.4	16.8
	36	1.45	20.1
Akiyo	28	1.8	38.6
	36	1.92	44.2
Simulation Condition	Frame:300 Intra period:100 IPPPP		
Environment	CPU: 2.2G RAM: 256kB		



**Figure 5-1** Rate-distortion of JM9.8 and x264

## 5.2 Proposed Acceleration Method

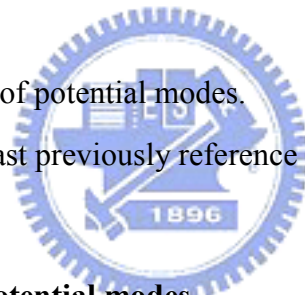
In the early chapter, H.264 has shown significant rate distortion improvements as compared to the other standards for video compression. It can be coded with 7 different block sizes for motion-compensation in the inter mode, and various spatial directional prediction modes in the intra mode. To achieve a higher coding efficiency, H.264 calculates rate distortion cost (RD cost) of all possible modes and chooses the best one having the minimum RD cost. This

increases the complexity and takes a lot of computing time. In order to reduce the high computational complexity, the x264 uses a few techniques to increase the speed in mode decision. In the next section we describe these speed-up techniques in x264. Then we propose a few other methods and show the simulation results which can efficiently reduce the computational cost by 13% with similar visual quality and bits rate of original x264.

### 5.2.1 Mode decision in x264

In JVT, for P frames, we need to select the best mode from a set of allowable modes: {SKIP, P\_16x16, P\_16x8, P\_8x16, P\_8x8, P\_8x4, P\_4x8, P\_4x4, I4MB, I16MB}. Examining all possible combinations of modes can be a big burden on the encoder. The x264 uses a new algorithm to reduce the complexity on mode decision. The new algorithm includes the following steps:

- A. Reduce the number of potential modes.
- B. Restrict the set of past previously reference pictures.



#### A. Reduced the number of potential modes

The x264 eliminates ME for some block types in the inter mode. The inter modes include the SKIP mode and many different block types (P\_16x16, P\_16x8, P\_8x16, P\_8x8, P\_8x4, P\_4x8, P\_4x4). In x264, calculating RD cost is jointly done with mode decision. For the inter modes, the SKIP mode refers to the 16x16 mode, where no motion and residual information is encoded. So no motion search is required and it has the lowest complexity. In x264, the SKIP has the highest priority. As for the decision on block types, the x264 algorithm checks whether the error surface versus block size is monotonic, that is, whether the current macroblock has the same tendency of using smaller block size (sub-macroblock partition) or larger block size. The error surface is built by initially 3 modes (block sizes): P\_16x16, P\_8x8 and P\_4x4. When RD cost of P\_8x8 mode is less than RD cost of P\_16x16 mode, this means that the current macroblock may have the same tendency of using small block size. Then we check the

P\_4x4 mode. Otherwise, if RD cost of P\_16x16 mode is less than RD cost of P\_8x8 mode, it means the P\_16x16 mode probably has the least RD cost. Then we skip the other inter modes. The next decision of whether to test other modes P\_16x8, P\_8x16, or finer sub-macroblock partition is based on the comparison between RD cost of P\_8x8 mode and RD cost of P\_4x4 mode. If RD cost of P\_8x8 mode is less than RD cost of P\_4x4 mode, it means the best mode is P\_8x8 mode. Then only the P\_16x8, P\_8x16 modes are further tested. Otherwise, if P\_4x4 is the best mode, then the P\_16x8, P\_8x16, P\_4x8, P\_8x4 modes are further tested. Therefore, if P\_4x4 is the best mode among P\_16x16, P\_8x8, P\_4x4, all the inter modes need to be tested. This procedure is restated as follows.

- Step1: Check the SKIP mode. If the condition is satisfied, select the SKIP mode as the best mode, then stop; otherwise go to Step2;
- Step2: Check P\_16x16 and P\_8x8. If ( $RDcost_{16x16} < RDcost_{8x8}$ ), go to Step7; otherwise, go to Step3;
- Step3: Check P\_4x4; if ( $RDcost_{4x4} < RDcost_{8x8}$ ), go to Step4; otherwise, go to Step5;
- Step4: Check P\_16x8, P\_8x16, P\_4x8, P\_8x4; go to Step6;
- Step5: Check P\_16x8, P\_8x16; go to Step6;
- Step6: Select the best inter mode. Then check the intra modes; go to Step7;
- Step7: Choose the best mode among all the tested modes.

In Step1, the SKIP mode condition is a special case. Because in the SKIP mode, where no motion and residual information is encoded. So the SKIP mode can save the encoding time and bit rates. This is why the SKIP mode dominates among all modes at low bit rates. A block is said using the SKIP mode in a P slice when the following set of four conditions is satisfied [23]:

- The best motion compensation block size is P\_16x16.
- The reference frame is the previous frame.
- The best motion vector is the predicated motion vector (regardless of this being the

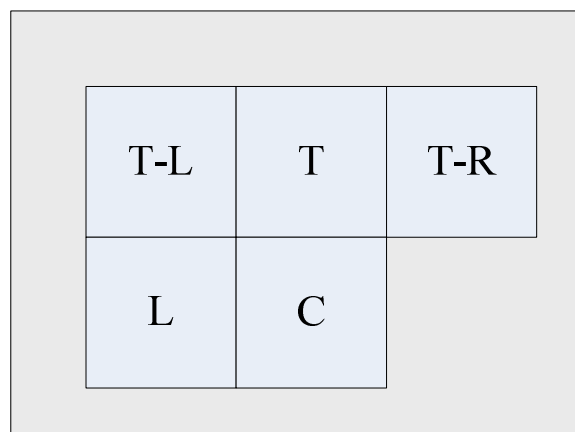
zero motion vectors or a non-zero one).

- The transform coefficients of the 16x16 block size are all quantized to zero.

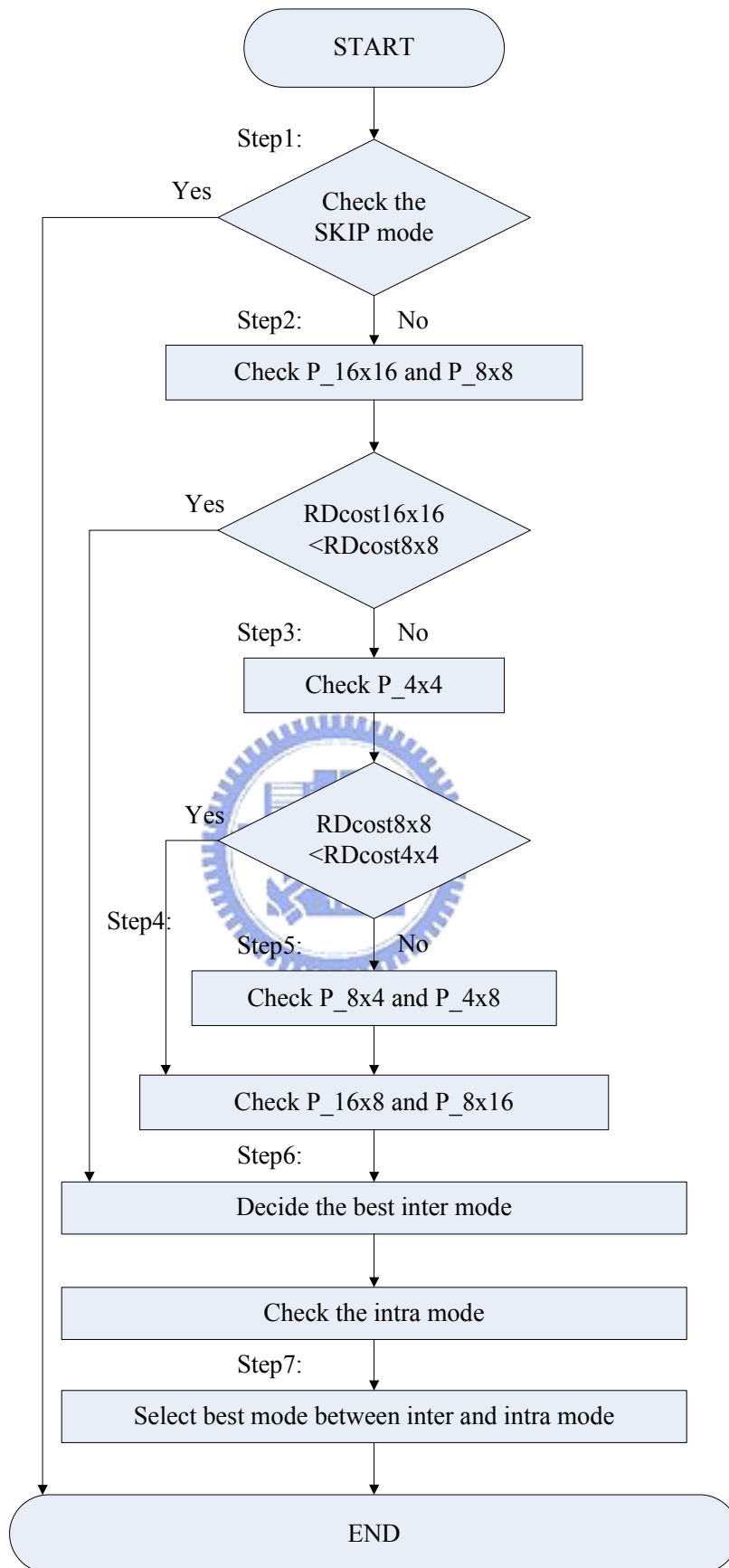
The x264 algorithm, we give the SKIP mode the highest priority. Although the first condition will not sufficient for the SKIP mode prediction, it is “good enough”. In x264, we actually intend to reduce the computational cost by predicting a percentage of macroblocks in the SKIP mode based on the spatial neighborhood information without any overhead computation. The SKIP mode conditions are reduced to the following set of two items:

- The surrounding macroblock is using the SKIP mode.
- The transform coefficients of the 16x16 block size are all quantized to zero.

These two conditions suggest that if one of macroblocks on the top, left, top-left and top-right of the current MB in the current picture is using the SKIP mode, then this mode pattern can be a good indication that the current macroblock can be skipped. This is due to the fact that the likelihood of the current macroblock belonging to a stationary part of the picture is high, and thus it is a candidate for skipping. The proposed spatial predictor is shown in Figure 5-2 using the two conditions. Step1 can save a lot of computation. The flow chart of mode decision algorithm of x264 can be shown in Figure 5-3.



**Figure 5-2** Spatial prediction for skip mode decision



**Figure 5-3** Mode decision algorithm of x264

## **B. Restrict the set of reference pictures in motion estimation**

In JVT, the best reference frame is decided for each macroblock type (P\_16x16, P\_16x8, P\_8x16, and P\_8x8). This takes lots of time and increases the complexity. For example, the standard H.264 has 5 reference frames for P slice to decide, when we evaluate the RD cost for P\_16x16 mode. We need to calculate the RD cost five times for each reference frame to find the minimal RD cost to decide the best reference frame for P\_16x16. The same thing occurs to the other block types. The computational complexity is proportional to the number of reference frames. In order to reduce computations, x264 decides the best reference frame only in P\_16x16 mode. When we evaluate the other macroblock type, we only calculate the RD cost of the reference frame which P\_16x16 decided the best.

### **5.2.2 Fast Algorithms**

As discussed in the previous section, we know x264 can speed up the frame encoding and reduce the complexity of the mode decision of the reference software JM. However, there are other algorithms that we can speed up the mode decision algorithm in x264. The following techniques are included in our proposal.

- A. Set a threshold between the P\_16x16 mode and the P\_8x8 mode [24]
- B. Rearrange the inter mode checking order
- C. Inter mode cost decide whether do intra mode

We will elaborate these techniques below and show the experimental results.

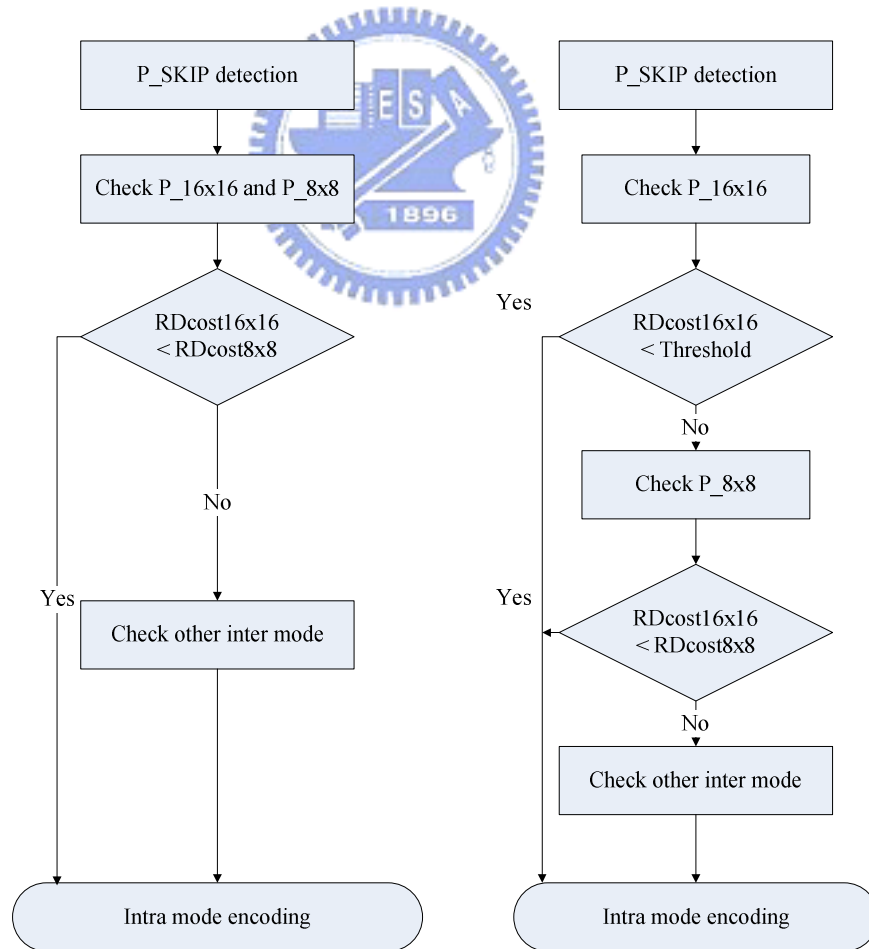
#### **A. Set a threshold between P\_16x16 mode and P\_8x8 mode**

Usually in a picture, large areas of background may be either still or under global motion, which can be predicted well by the motion vectors of the neighboring blocks. In these areas, the SKIP mode or the P\_16x16 mode usually is the best mode since it has no or less MV overheads. Figure 5-3 shows that the x264 mode decision algorithm needs to evaluate the RD

cost of P\_16x16 and P\_8x8 if the macroblock is not predicated as a SKIP mode. Because optimal mode for the background areas is often the P\_16x16 mode, the calculation of the RD cost of P\_8x8 is redundant. Therefore, we check only the P\_16x16 mode at this point. If the cost is smaller than the threshold, the mode decision process stops, the best mode is P\_16x16 (Figure 5-4). To maintain the coding, the threshold is conservatively set by the minimum of the costs of the 20 previous blocks of the same mode (P\_16x16) plus a fixed value. That is,

$$TH_{16 \times 16} = \min_{i \in \{i | 1 \leq i \leq 20, MB \ i \text{ with mode } 16 \times 16\}} (\text{cost } t_i) + \Delta TH_{16 \times 16}$$

In our simulations,  $\Delta TH_{16 \times 16} = 1200$ . In addition, this parameter can also be used to control the tradeoff between the complexity and quality. If  $\Delta TH_{16 \times 16}$  is larger the P\_16x16 mode is used more often and the process faster. This scheme is called Algorithm A (Figure 5-4). More details of it can be found in [24].



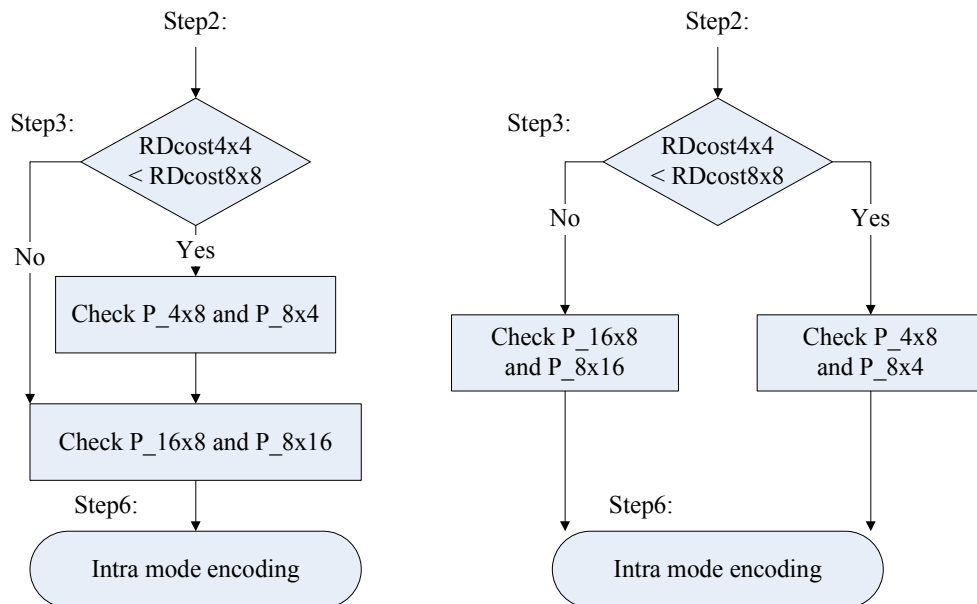
**Figure 5-4** Block diagram of x264 and Algorithm A

## B. Rearrange the inter mode checking order

Another observation is that, if the cost of a larger block-size mode is higher than the cost of the current block-size mode, then the even larger block size modes may be excluded. Similarly, if the cost of a smaller block-size mode is higher than that of the current block-size mode, then the even smaller block-size may be excluded. We can use the idea to accelerate the other inter mode algorithm. Here are the two methods:

1 Skip mode=> 16x16 => 8x8 => if 4x4<8x8 => 4x8, 8x4 (skip 8x16 and 16x8)

In x264, at Step3 (Figure 5-3) if RD cost of P\_4x4 mode is less than RD cost of P\_8x8 mode, we need to check the follows modes P\_16x8, P\_8x16, P\_4x8 and P\_8x4. But since P\_4x4 is the best mode in P\_16x16, P\_8x8, P\_4x4, it often means that the current macroblock has the tendency of using smaller block size. Checking the larger block-size is not necessary. Therefore, we suggest that in Step3, when RD cost of P\_4x4 mode is less than RD cost of P\_8x8, the P\_4x8, P\_8x4, P\_16x8 and P\_8x16 modes are excluded in checking (Figure 5-5). This scheme is called Algorithm B1. This algorithm is similar as the fast mode decision in [25].

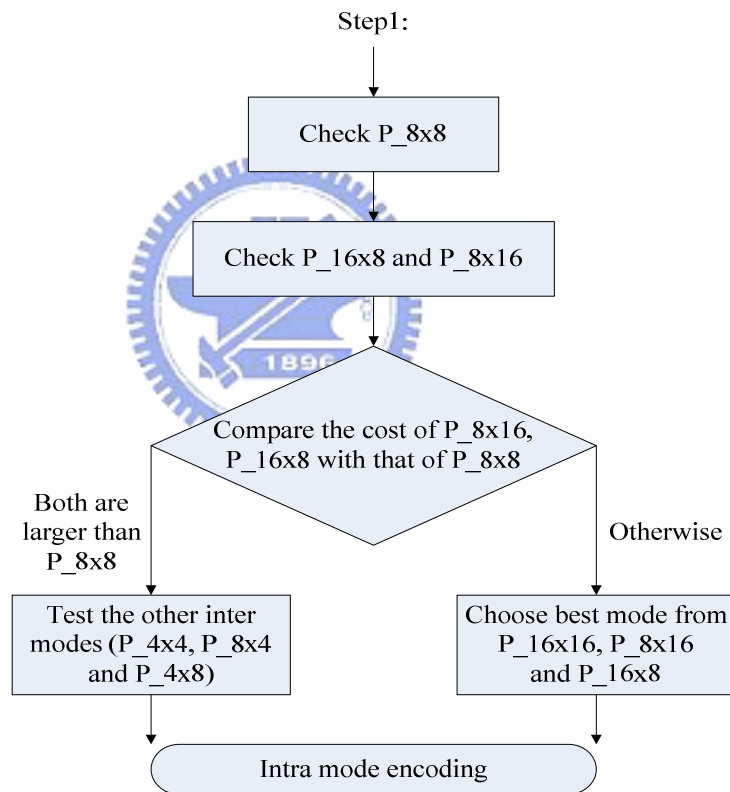


**Figure 5-5** Difference between x264 and Algorithm B1



2 Skip mode=> 16x16 => 8x8 => 16x8, 8x16 => if 16x8 or 8x16 < 8x8 => skip all the other modes

In this method, the P\_16x8 and P\_8x16 modes have higher priority than P\_4x4, P\_8x4, and P\_4x8. If either the cost of P\_8x16 or P\_16x8 is smaller than that of P\_8x8, we assume the larger block-sizes produce smaller costs. So, the best mode is chosen from P\_16x16, P\_8x16 and P\_16x8. Smaller block-size modes (all sub-MB modes of P\_8x4, P\_4x8, and P\_4x4) are excluded. If the cost for both P\_16x8 and P\_8x16 are higher than that of P\_8x8, it means the larger clock modes do not achieve lower costs and the sub\_MB modes (P\_8x4, P\_4x8 and P\_4x4) should be checked (Figure 5-6). This scheme is called Algorithm B2.



**Figure 5-6** Fast Algorithm B2

### C. Inter mode cost decide whether do intra mode

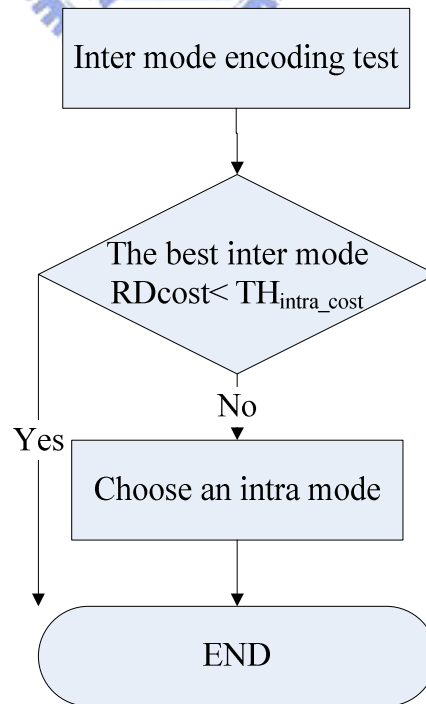
In H.264, after choosing the best inter mode, the cost associated with the spatial prediction mode is calculated and compared with that of the best inter mode. The mode with the minimum cost is determined as the encoding mode. In this process, the spatial prediction

encoding steps are performed for the intra mode. Usually the spatial prediction mode requires more bits than the inter mode. Therefore, the spatial prediction mode is rarely chosen to be mode of a macroblock except the special cases if sense change. The x264 algorithm assumes that the inter mode have the higher priority than the intra modes for P or B images. So if the inter modes have a good performance, no intra modes are to be checked. In order to skip the unnecessary intra mode checking, we can set threshold called  $TH_{intra\_inter}$ . The cost of the best inter mode is compared with  $TH_{intra\_inter}$ . If the cost of the best inter mode is less than  $TH_{intra\_inter}$ , then the inter mode is more efficient and the intra mode encoding is skipped (Figure 5-7). This is called Algorithm C.

We can use a method to set the threshold  $Th_{intra\_inter}$ .

$$TH_{intra\_inter} = \underset{(i|1 \leq i \leq 20, MB_{intra})}{Mean} (cost_i) - \Delta TH_{intra\_inter}$$

Mean value of RD costs of 20 previous macroblock encoded in intra mode of the previous frames and current frame is calculates. The value of  $\Delta TH_{intra\_inter}$  is used to control the tradeoff between complexity and quality. In Algorithm C, we choose the  $\Delta TH_{intra\_inter} = 300$ .



**Figure 5-7** Fast Algorithm C

### 5.2.3 Experimental Results

The experiments are conducted using the parameters in Table 5-2. Table 5-3 is the result of x264 encoding the sequence Foreman, Mobile, Akiyo, Stefan with different QP from 28 to 40.

**Table 5-2** Experimental parameters

Test sequence	Foreman, Mobile, Akiyo, Stefan
Frame number	300
Image size	CIF (352x288)
Intra period	100
Reference frame	1

**Table 5-3** Performance of x264

Sequence	Foreman		
QP	Time(s)	PSNR(dB)	bit-rates(Kbps)
28	16.16	37.4	573.51
32	14.00	34.89	289.13
36	12.24	32.56	159.57
40	10.79	30.33	100.3

Sequence	Mobile		
QP	Time(s)	PSNR(dB)	bit-rates(Kbps)
28	17.85	34.13	2061.21
32	16.88	31.07	1084.72
36	15.19	28.24	507.04
40	13.22	25.75	252.61

Sequence	Akiyo		
QP	Time(s)	PSNR(dB)	bit-rates(Kbps)
28	7.77	41.19	107.28
32	7.24	38.72	57.68
36	6.79	36.37	32.73
40	6.47	34.15	20.45

Sequence	Stefan		
QP	Time(s)	PSNR(dB)	bit-rates(Kbps)
28	17.91	36.03	1551.02
32	16.46	33.17	858.15
36	14.86	30.49	481.96
40	13.20	27.92	290.06

## A. Results of the Modified Algorithm

In the final algorithm, we induce the fast Algorithm A, B and C to speed up the x264 encoding process. For algorithm B we choose the version B2. Table 5-4 shows a comparison between our modified algorithm and x264. In this table, the  $\Delta$ PSNR(dB) and  $\Delta$ bit-rates (%) denote the differences of PSNR and bit rates and they are calculated according to the following equation:

$$\Delta \text{bit-rates} = \frac{\text{Bit number of modified algorithm} - \text{Bit number of x264}}{\text{Bit number of x264}} \times 100(\%)$$

$$\Delta \text{PSNR} = \text{PSNR of x264} - \text{PSNR of modified algorithm}$$

And *Time saving (%)* denotes the amount of encoding time decrease in the total encoding process. The amount of time saving can be calculated according to the following equation:

$$\text{Time saving (Time(\%))} = \frac{\text{Encoding Time of x264} - \text{Encoding Time of modified algorithm}}{\text{Encoding Time of x264}} \times 100(\%)$$

Referring to Table 5-4 it can be seen that if the fast algorithm in use, the total encoding time decreases by 13% on the average. Figure 5-8 are graphs comparing the rate-distortion performed between the modified algorithm and x264. It can be seen that the PSNR of the fast algorithm can achieve almost the same as x264.

**Table 5-4** Performance Comparison between x264 and the modified Algorithm

Test sequence : foreman_cif.264			
QP	$\Delta$ PSNR (dB)	$\Delta$ bit-rates (%)	Time saving (%)
28	-0.06	1.37	15.89
32	-0.07	1.51	14.47
36	-0.06	1.65	12.69
40	-0.04	1.16	10.74

(a) Foreman

Test sequence : mobile_cif.264			
QP	$\Delta$ PSNR (dB)	$\Delta$ bit-rates (%)	Time saving (%)
28	-0.01	-0.37	12.67
32	-0.03	-0.29	14.24
36	-0.04	0.13	13.62
40	-0.04	0.06	11.99

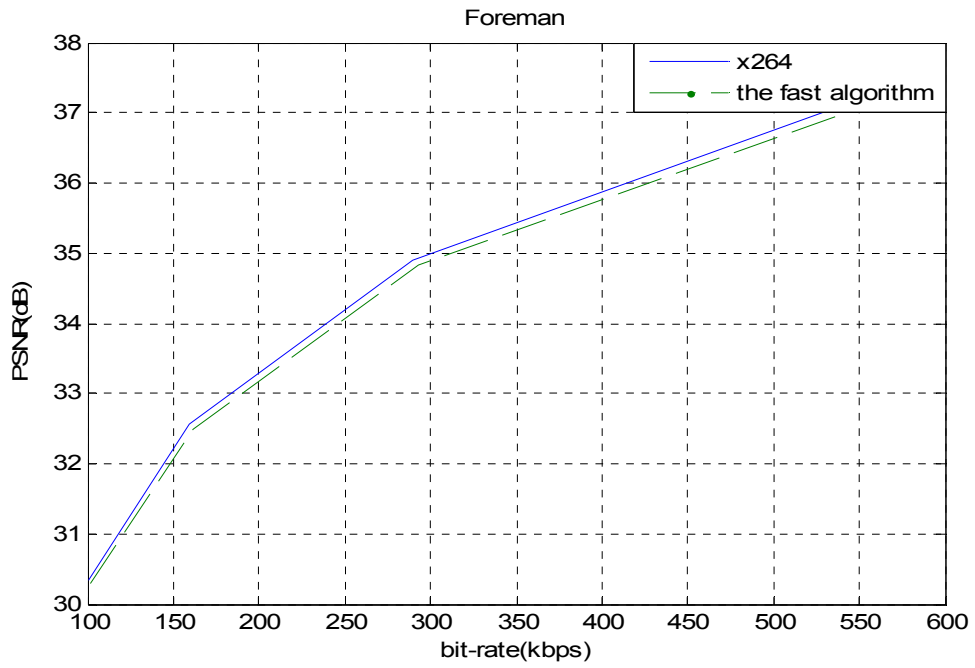
(b) Mobile

Test sequence : Akiyo_cif.264			
QP	$\Delta$ PSNR (dB)	$\Delta$ bit-rates (%)	Time saving (%)
28	-0.03	0.5	6.99
32	-0.04	0.49	5.38
36	-0.03	0.73	3.08
40	-0.02	0.1	2.65

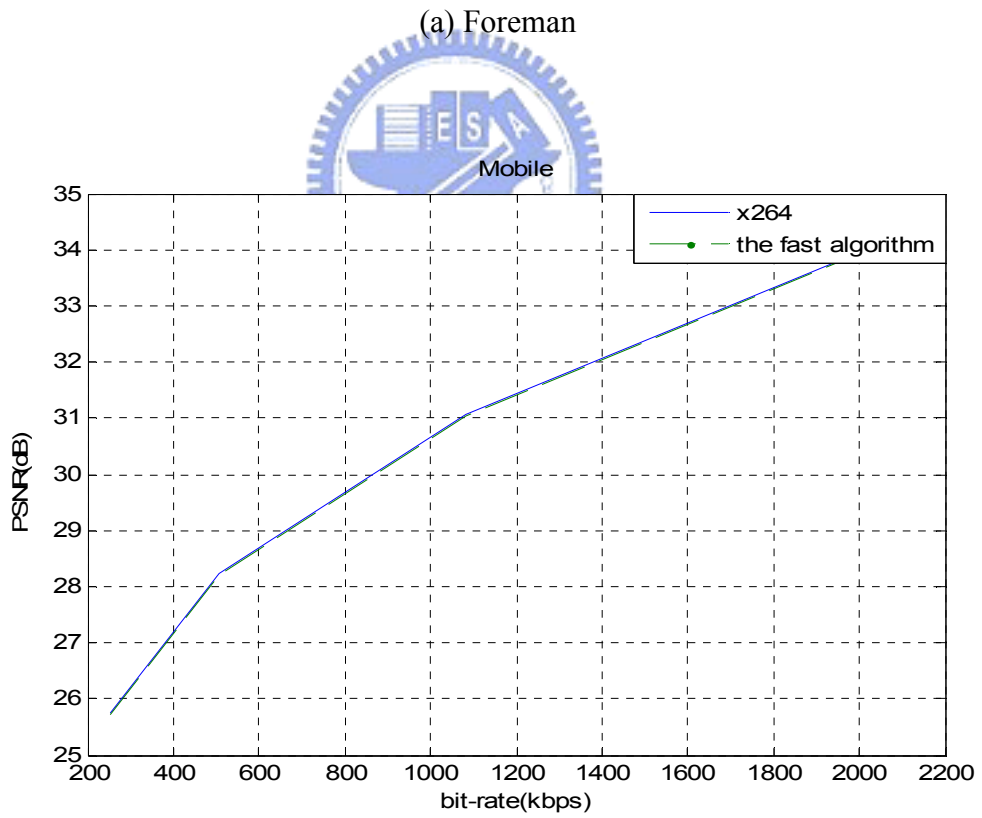
(c) Akiyo

Test sequence : Stefan_cif.264			
QP	$\Delta$ PSNR (dB)	$\Delta$ bit-rates (%)	Time saving (%)
28	-0.03	0.5	16.12
32	-0.04	0.49	16.24
36	-0.06	0.73	14.91
40	-0.05	0.1	13.63

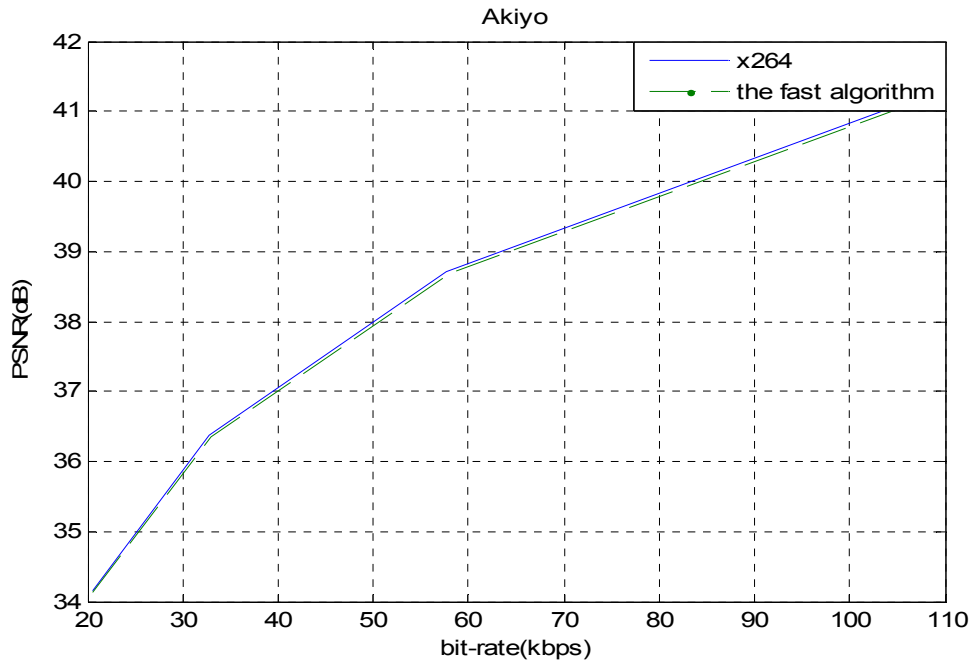
(d) Stefan



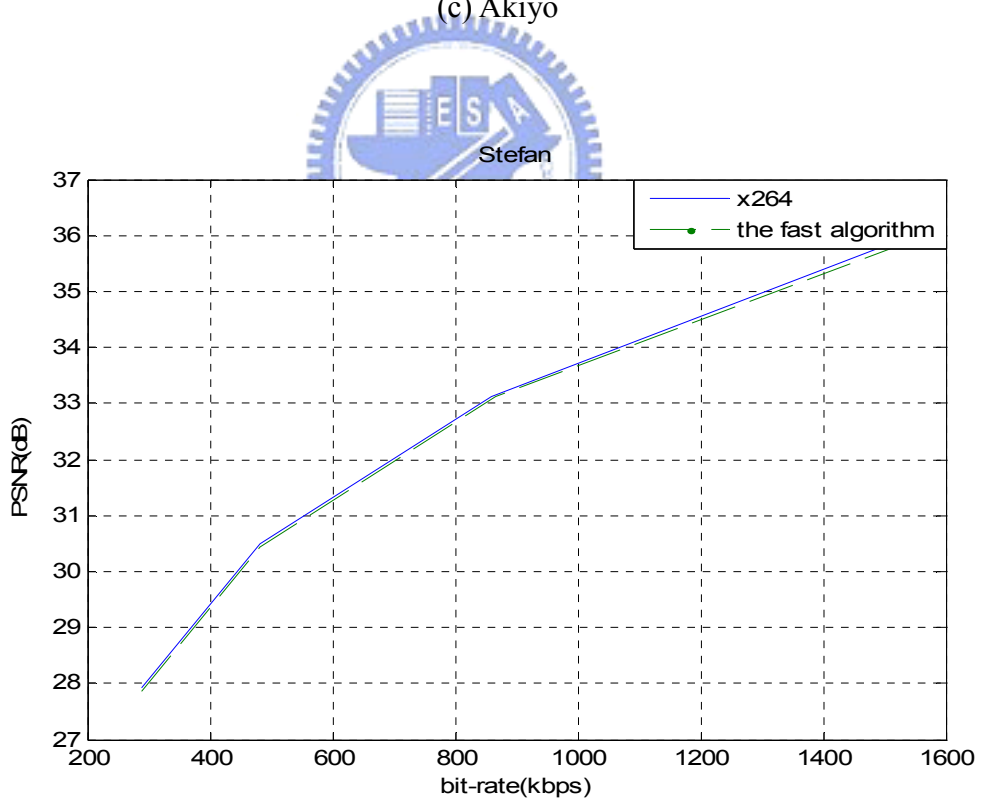
(a) Foreman



(b) Mobile



(c) Akiyo



(d) Stefan

Figure 5-8 Rate-distortion (PSNR vs. bit-rates) of the x264 and the modified Algorithm

## 5.3 Complexity Analysis on DSP

Our goal is to implement x264 on the DSP board. In this section, we will describe the profile of x264 on different DSP simulator. Then we will show the influence of memory system.

### 5.3.1 Complexity Analysis on Various Simulator

In section 4.3.2 we know that different simulator tools show different clock cycle results. In this section, we profile the x264 encoder on different simulators to observe the ratio of each function. We use the C6416 simulator and the C64xx simulator. The profiling results using the two methods are shown in Figure 5-9 and Figure 5-10. The test sequence is 10 frames of foreman and the image size is 176x144 (QCIF) and QP is 28. The profiling results of each function in percentage are almost the same in C64xx and C6416. However, in Table 5-5 we can see that the total cycles of the C6416 simulator are approximately 44 times of that of C64xx. This is because the C64xx simulator only counts the cycles of the DSP core processor, but the C6416 simulator also count the memory access time. In the real system, we use the DSP C64xx XDS510 emulator with 1GHz clock. Its results are almost the same as that of the C6416 simulator. This means that we need 5 seconds to encode the 10 frames of “foreman” sequence. We first need to solve the problem of the memory access time.

**Table 5-5** Cycles on different simulators

Simulator	C6416	C64xx	Ratio (%)
Total cycles	5,396,046,677	122,147,133	44.1



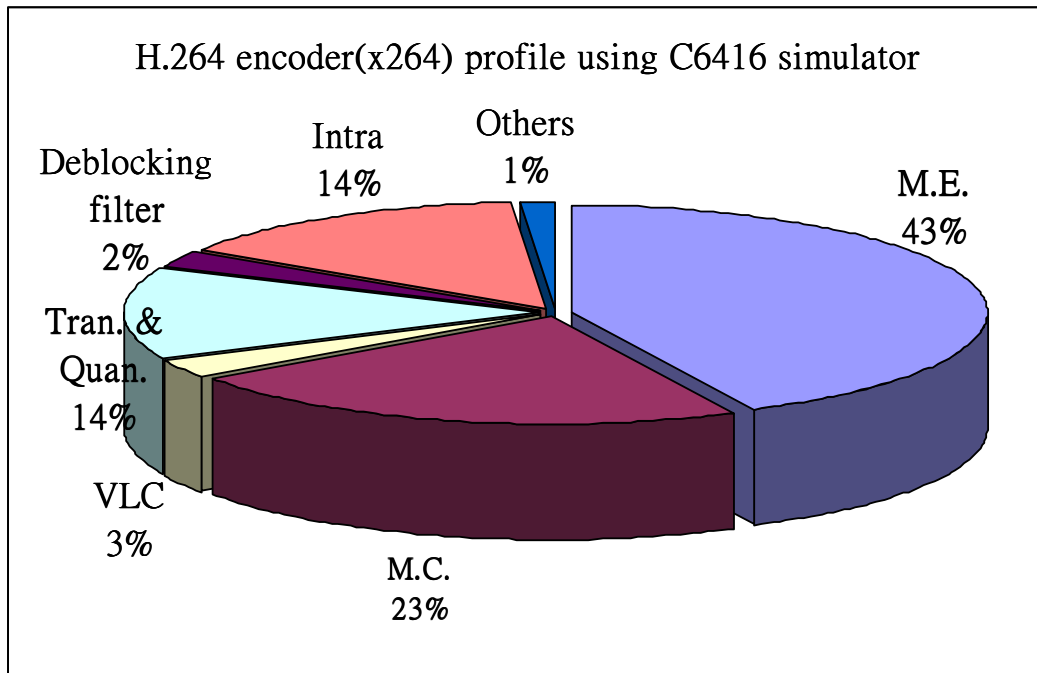


Figure 5-9 Complexity profiling of the H.264 encoder on the C6416 simulator

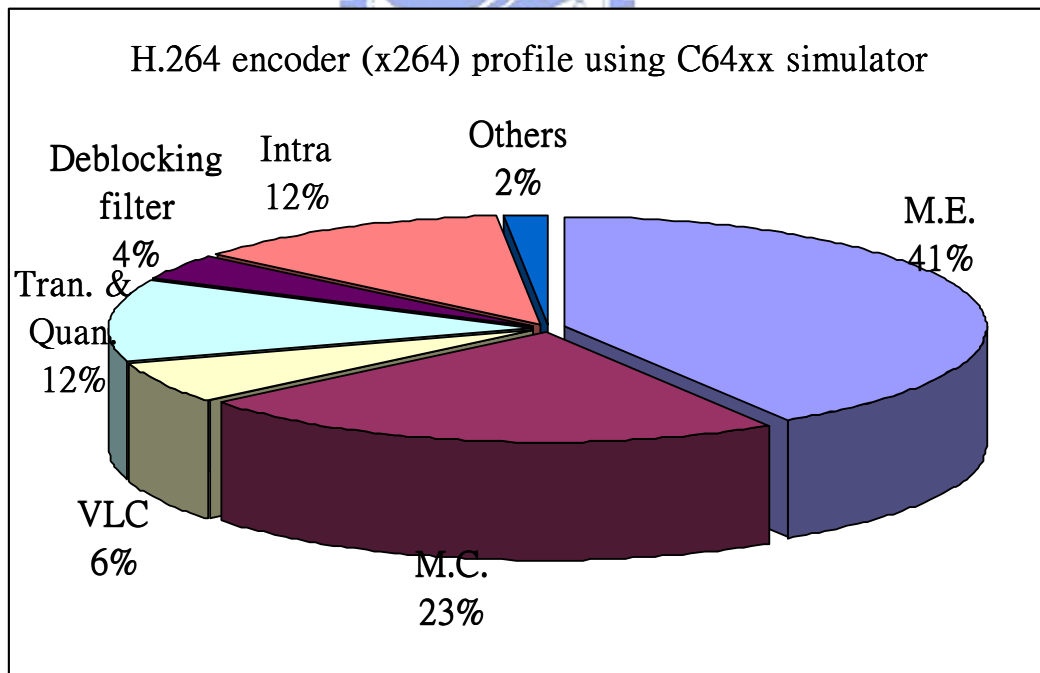


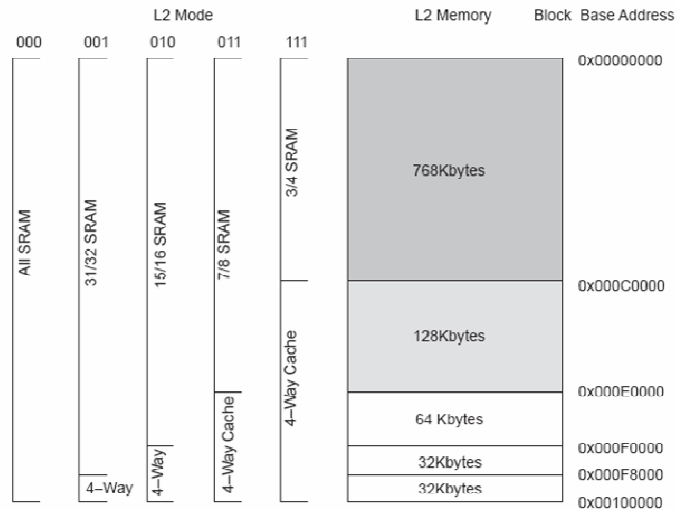
Figure 5-10 Complexity profiling of the H.264 encoder on the C64xx simulator

### 5.3.2 Memory System

In the previous section, we identify the major bottlenecks in running the x264 encoder on a DSP system. The actual cycles on the DSP platform are more than the CPU cycles. The C64xx CPU cycle accurate simulator ignores the time to access the instructions and data. But on the real DSP platform need to consider the memory access time. Table 5-6 shows the cycle distribution generated by the C6416 simulator. The core processing cycles are only 2.25% of the total cycles. The stall cycles are the most critical part in the total cycles. In the memory hierarchy of the DSP platform, the L1D is too small so that the data miss frequently occurs. In Table 5-6 , the data cache miss rate is 96%. If the data miss occurs, the CPU needs longer time to access data from the external memory. The numerous stall cycles means that the system wastes a lot of time in transferring data. If the cache becomes larger, the data miss frequency will decrease.

In section 4.2.2, we know that the DSP platform has 16K-bytes L1D cache and 16K-bytes L1P cache. The L2 is a 1024K-bytes SRAM. Because the L1 cache is too small, we can use the L2 SRAM as a second level cache. In the DSP platform, we can use the flag L2 mode to control the size of SRAM and cache in the L2 internal memory as shown in Figure 5-11. If we use the L2 cache, it brings in a great improvement as shown in Table 5-6. The data cache miss rate has reduced from 96% to 5.3% so that the stall cycles decreases a lot. The percentage of the core cycles arises to 43%. In Table 5-6, we set the L2 cache size to its maximum (256Kbytes located in the SRAM). The two-level cache produces the most benefit.

Because of the benefit of two-level cache, the speed can increase 19 times than the original one. We can encode 10 frames of the sequence “foreman\_qcif” within 0.3 second. Figure 5-12 shows the profile of using L2 cache on the C6416 simulator. In the next section, we use the profile of using the L2 cache as the starting program and accelerate its speed with the two-level cache using various techniques.



**Figure 5-11** C6416 memory configuration [14]

**Table 5-6** Effect of using L2 cache memory

C6416 simulator	Original		L2 cache	
	Count (cycles)	Percentage (%)	Count (cycles)	Percentage (%)
<b>Total Cycles</b>	<b>5,396,104,222</b>		<b>282,805,109</b>	
Core cycles(excl. stalls)	121,411,824	2.25	121,411,880	42.93
NOP cycles	26,737,686	21.89	26,737,704	21.89
Stall Cycles	5,274,885,028	97.75	161,411,584	57.08
L1P Stall Cycles	4,165,028	0.08	83,460,201	29.51
L1D Stall Cycles	5,270,294,459	97.67	75,328,488	26.64
Instruction cache hits	44,494,904	95.94	44,494,990	95.94
Instruction cache misses	1,882,076	4.06	1,881,996	4.06
Data cache hits	2,408,873	3.98	57,282,572	94.71
Data cache read hits	2,377,679	5.75	40,988,195	99.05
Data cache write hits	31,194	0.16	16,294,377	85.3
<b>Data cache misses</b>	<b>58,075,133</b>	<b>96.02</b>	<b>3,201,441</b>	<b>5.29</b>
Data cache read misses	39,004,261	94.25	393,752	0.95
Data cache write misses	19,070,872	99.84	2,807,689	14.7

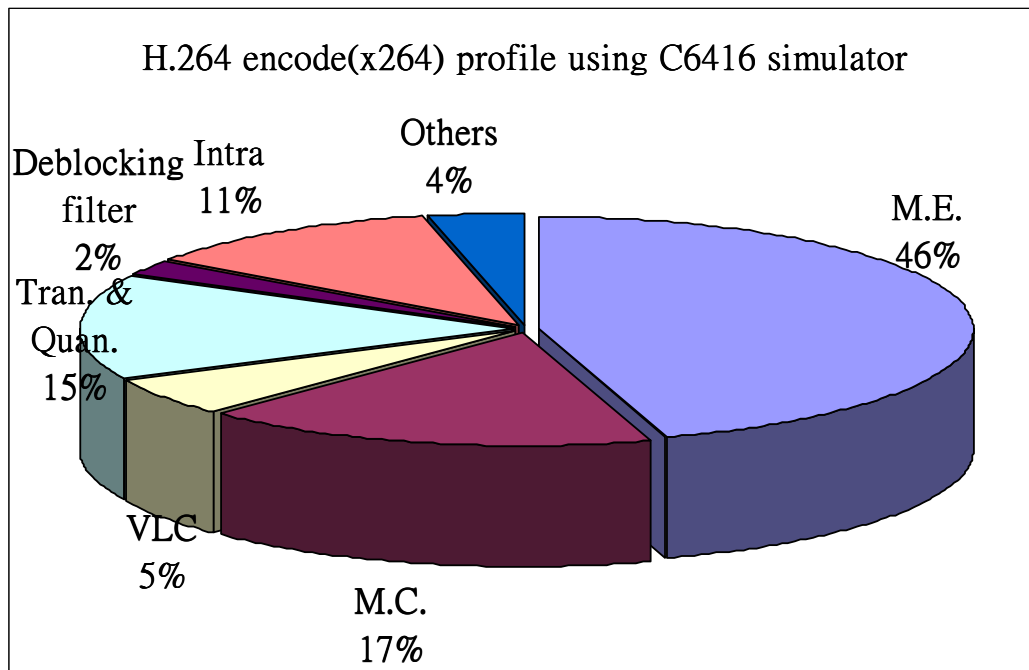


Figure 5-12 Profile with L2 cache using C6416 simulator

## 5.4 DSP Code Acceleration Methods

Improving the clock cycles of the x264 algorithm is the main task of our system implementation on DSP. In this section, we will describe several methods that can accelerate the execution time on the C64x DSP. Some of these methods are supported by the features of the C64x.

### 5.4.1 Compiler Options

As discussed in section 4.4, the CCS compiler can transform C code into efficient assembly code. The compiler options have four optimization levels: register (-o0), local (-o1), function (-o2), file level (-o3). File level is the highest available optimization level. We choose the file level optimization when implementing the x264 on DSP.

## 5.4.2 Fixed-point Coding

The C6000 compiler defines a size for each data type (Table 5-7):

**Table 5-7** Size of different data type

Data Type	Char	Short	Int	Long	Float	Double
Size (bits)	8	16	32	40	32	64

The C64x DSP is a fixed-point processor, so it can only perform fixed-point operations. Although the C64x DSP can simulate floating-point processing, it takes a lot of extra clock cycles to do this job. The “char”, “short”, “int” and “long” are the fixed-point data types, and the “float” and “double” are the floating-point data types. We test C64x DSP processing time of the assembly instructions “add” and “mul” for different data types. Table 5-8 shows the results. We can clearly see that the floating-point data type need more computation time than the fixed-point data types. Hence, we can accelerate our DSP codes in computation time by converting the data types from floating-point to fixed-point.

**Table 5-8** Processing time on the C64x for different data types

Assembly Instruction	Char 8-bit	Short 16-bit	Int 32-bit	Long 40-bit	Float 32-bit	Double 64-bit
add	1	1	1	2	77	146
mul	2	2	6	8	54	69

## 5.4.3 Loop Unrolling

Loop unrolling expands small loops so that all iterations of the loop appear. It can increase the number of instructions available in execution in parallel. It is also suitable for using software pipeline. When the codes have conditional instructions, the compiler may not be known in

advance that the branch will occur or not. It needs more waiting time for the decision of branch operation. If we do loop unrolling, some of the overhead for branching instruction are reduced. Use loop unrolling can decrease the clock cycles but it often increases the code size. So when we use loop unrolling, the code size may be a consideration.

#### 5.4.4 Linear Assembly

Assembly code is generated by the CCS compiler or the assembly optimizer. Sometimes the generated assembly codes are not efficient due to stalls or hazards. Converting parts of the C codes into linear assembly codes is a good way to solve the problem. Rearranging the assembly codes can avoid the stalls and hazards by hand. But this process generally is too detail and very time consumption in practice. Hence, we do this process only on some important functions. In x264, we rewrite the C code to assembly code only on the function of dct and idct. When we use the linear assembly, we do not need to specify the parallel instructions, pipeline latency and register usage. These will be specified by the assembly optimizer. We only need to modify the codes. Table 5-9 is the comparison between original code and modified linear assembly code in dct and idct. And Table 5-10 is an example of the dct2x2 we write in linear assembly code. The details of linear assembly can be found in [17].

**Table 5-9** Comparison between C code and linear assembly code

	Original C code (cycles)	Modified Linear Assembly Code (cycles)	Reduction Ratio (%)
dct2x2dc	16	14	12.5%
dct4x4dc	87	49	44%
idct4x4dc	80	53	34%
sub4x4_dct	97	48	51%

**Table 5-10** Example of linear assembly code

Original C Code	Modified Linear Assembly Code
<pre>static void dct2x2dc( int16_t d[2][2] ) {     int tmp[2][2];      tmp[0][0] = d[0][0] + d[0][1];     tmp[1][0] = d[0][0] - d[0][1];     tmp[0][1] = d[1][0] + d[1][1];     tmp[1][1] = d[1][0] - d[1][1];      d[0][0] = tmp[0][0] + tmp[0][1];     d[0][1] = tmp[1][0] + tmp[1][1];     d[1][0] = tmp[0][0] - tmp[0][1];     d[1][1] = tmp[1][0] - tmp[1][1]; }</pre>	<pre>.global _dct2x2dc _dct2x2dc: .cproc d             .reg d00,d01,d10,d11,             t00,t01,t10,t11              LDH    *+d[0],d00             LDH    *+d[1],d01             LDH    *+d[2],d10             LDH    *+d[3],d11              ADD    d00,d01,t00             SUB    d00,d01,t10             ADD    d10,d11,t01             SUB    d10,d11,t11             ADD    t00,t01,d00             ADD    t10,t11,d01             SUB    t00,t01,d10             SUB    t10,t11,d11             STH    d00,*+d[0]             STH    d01,*+d[1]             STH    d10,*+d[2]             STH    d11,*+d[3]              .endproc</pre>

### 5.4.5 Other Acceleration Techniques

Other techniques for code speed up are: reduce memory access, use bit shift for multiplication or division, declare variable or memory as constant, eliminate the unimportant function, etc.

## 5.5 Experimental Results

After algorithm and code acceleration, we present the experimental results on the speed and the coding performance by encoding different kinds of video sequences.

### 5.5.1 Simulation and Acceleration Results

We first encode different video sequences using the released compilation mode with file level optimization on the C6416 simulator. In Table 5-11, we can clearly see that if we use the two-level cache, the reduction ratio is almost 94.6%. And after modification, the final implementation is almost speedup 50.05% on the C6416 simulator as shown in Table 5-12.

**Table 5-11** Comparison using the C6416 simulator with and without the L2 cache

Sequence	QP	Original	With L2 cache	Reduction Ratio (%)
Foreman	24	5,857,141,388	299,997,333	94.88
	28	5,396,047,003	272,850,099	94.94
	32	4,791,033,572	235,541,515	95.08
	36	4,270,633,918	220,994,332	94.83
Akiyo	24	2,477,711,618	143,512,051	94.21
	28	2,407,798,347	139,733,601	94.20
	32	2,349,115,427	136,120,778	94.21
	36	2,337,257,965	136,816,145	94.15
Mobile	24	5,920,104,560	312,644,701	94.72
	28	5,728,222,344	301,657,203	94.73
	32	5,478,265,825	288,294,260	94.74
	36	4,919,007,494	259,681,978	94.72
Stefan	24	6,282,198,760	323,474,273	94.85
	28	5,891,887,789	300,134,320	94.91
	32	5,523,460,593	279,967,886	94.93
	36	5,064,125,114	259,003,037	94.89



**Table 5-12** Comparison using C6416 simulator with original and accelerated code

Sequence	QP	Original	Accelerated	Reduction Ratio (%)
Foreman	24	299,997,333	141,762,294	52.75
	28	272,850,099	131,384,560	51.85
	32	235,541,515	119,525,198	49.26
	36	220,994,332	110,162,459	50.15
Akiyo	24	143,512,051	78,220,964	45.50
	28	139,733,601	76,657,427	45.14
	32	136,120,778	75,355,147	44.64
	36	136,816,145	74,970,281	45.20
Mobile	24	312,644,701	152,720,909	51.15
	28	301,657,203	143,165,216	52.54
	32	288,294,260	134,215,780	53.44
	36	259,681,978	122,615,959	52.78
Stefan	24	323,474,273	155,155,172	52.03
	28	300,134,320	145,362,395	51.57
	32	279,967,886	135,718,672	51.52
	36	259,003,037	126,202,732	51.27

### 5.5.2 Encoding Speed on DSP board

On the DSP board, we can use an internal timer to count the executing clock cycles on the C6416T emulator. The 32-bit general-purpose timers are included in the DSP core processor. Its clock is closer to the real system than that of the C6416 simulator. So we compute the encoding time on the DSP board by using this internal timer instead of the C6416 simulator. More information about this timer is described in [26]. Table 5-13 to Table 5-16 show the results of encoding different sequences, whose frame number is 50 and picture size is 176x144 (QCIF) show that in encoding the sequence “foreman”, the fps (frames per second)

is 40.4, the fps of “akiyo” is 76, the fps of “mobile” is 24.2 and the fps of “stefan” is 26. We can achieve namely real-time encoding, which is 30 fps. As discussed in the early sections, the PSNR performance of the accelerated version is almost identical to the original x264 version.

**Table 5-13** Results of the “foreman” sequence on the C6416 emulator

Sequence	foreman_qcif			
QP	Clock cycles 50 frames	Average clock cycles per frame	Conversion (sec)	fps
16	1,331,768,824	26,635,376	0.0266	37.5
20	1,269,304,288	25,386,086	0.0254	39.4
24	1,236,622,048	24,732,441	0.0247	40.4
28	1,200,254,168	24,005,083	0.0240	41.7
32	1,198,344,096	23,966,882	0.0240	41.7
36	1,194,814,936	23,896,299	0.0239	41.8

**Table 5-14** Results of the “akiyo” sequence on the C6416 emulator

Sequence	akiyo_qcif			
QP	Clock cycles 50 frames	Average clock cycles per frame	Conversion (sec)	fps
16	666,665,336	13,333,307	0.0133	75.0
20	665,718,728	13,314,375	0.0133	75.1
24	659,000,184	13,180,004	0.0132	75.9
28	654,887,064	13,097,741	0.0131	76.3
32	648,843,728	12,976,875	0.0130	77.1
36	647,099,824	12,941,996	0.0129	77.3

**Table 5-15** Results of the “mobile” sequence on the C6416 emulator

Sequence	mobile_qcif			
QP	Clock cycles	Average clock cycles per frame	Conversion (sec)	fps
16	2,171,652,688	43,433,054	0.0434	23.0
20	2,144,657,712	42,893,154	0.0429	23.3
24	2,143,347,584	42,866,952	0.0429	23.3
28	2,011,821,640	40,236,433	0.0402	24.9
32	1,986,374,176	39,727,484	0.0397	25.2
36	1,943,237,440	38,864,749	0.0389	25.7

**Table 5-16** Results of the “stefan” sequence on the C6416 emulator

Sequence	stefan_qcif			
QP	Clock cycles 50 frames	Average clock cycles per frame	Conversion (sec)	fps
16	1,944,197,344	38,883,947	0.0389	25.7
20	1,920,824,192	38,416,484	0.0384	26.0
24	1,924,583,792	38,491,676	0.0385	26.0
28	1,915,755,304	38,315,106	0.0383	26.1
32	1,918,608,224	38,372,164	0.0384	26.1
36	1,907,222,528	38,144,451	0.0381	26.2

# Chapter 6

## H.264/AVC SVC Decoder

### Implementation and Optimization on DSP Platform

Chapter 3 gives an overview of scalable extension of H.264. We now discuss our implementation of the SVC decoder on DSP. We describe the system architecture and the procedure of our implementation work. Then we analyze the JSVM decoder which is the reference software of H.264/AVC SVC and identify the most complicated elements in the decoder. We also present a few techniques that accelerate code execution and the acceleration methods that take advantages of the features of C64x. Finally, we implement and show some experimental results on the speed and the coding performance of our system.

#### 6.1 System Architecture

Figure 6-1 shows the overall scalable extension of H.264 decoder architecture. When the bit-stream enters the JSVM decoder, it can be split into the base-layer part and the enhancement-layer part. The base-layer of the JSVM decoder is almost the same as the H.264 decoder. It includes entropy decoding, inverse quantization, inverse transform, motion compensation and deblocking filter. The enhancement layer is similar to the H.264 decoder but with a few modifications. The structure of decoder includes three types of scalability. In the spatial scalable coding, the motion-compensated prediction and intra coding tools are employed for both the base and enhancement layers. Each layer supports a different spatial resolution. In order to increase coding efficiency, additional inter-layer prediction mechanisms are incorporated. The inter-layer prediction includes techniques for motion, residual and intra

texture prediction. The information of inter-layer prediction needs to up-sample for the enhancement layer. For improving the coding efficiency, the enhancement layer can take the information from either the reference frame or the inter-layer prediction from the base layer. Signals are controlled by SW3, SW4 and SW5 which is shown in Figure 6-1. For SNR scalability, both CGS and FGS are supported. CGS uses the same inter-prediction mechanism as the spatial scalable coding, but without the up-sampling operation. The other case is FGS. FGS coding is based on so-called progressive refinement slices. The H.264/AVC CABAC is extended to support the FGS. In the temporal scalability coding, the procedure is already discussed in section 3.2.2 In the JSVM decoder, it uses the hierarchical-B prediction structure (shown in Figure 3-5) for the temporal scalable coding. A decoded picture buffer (DPB) method is to implement the temporal scalability.

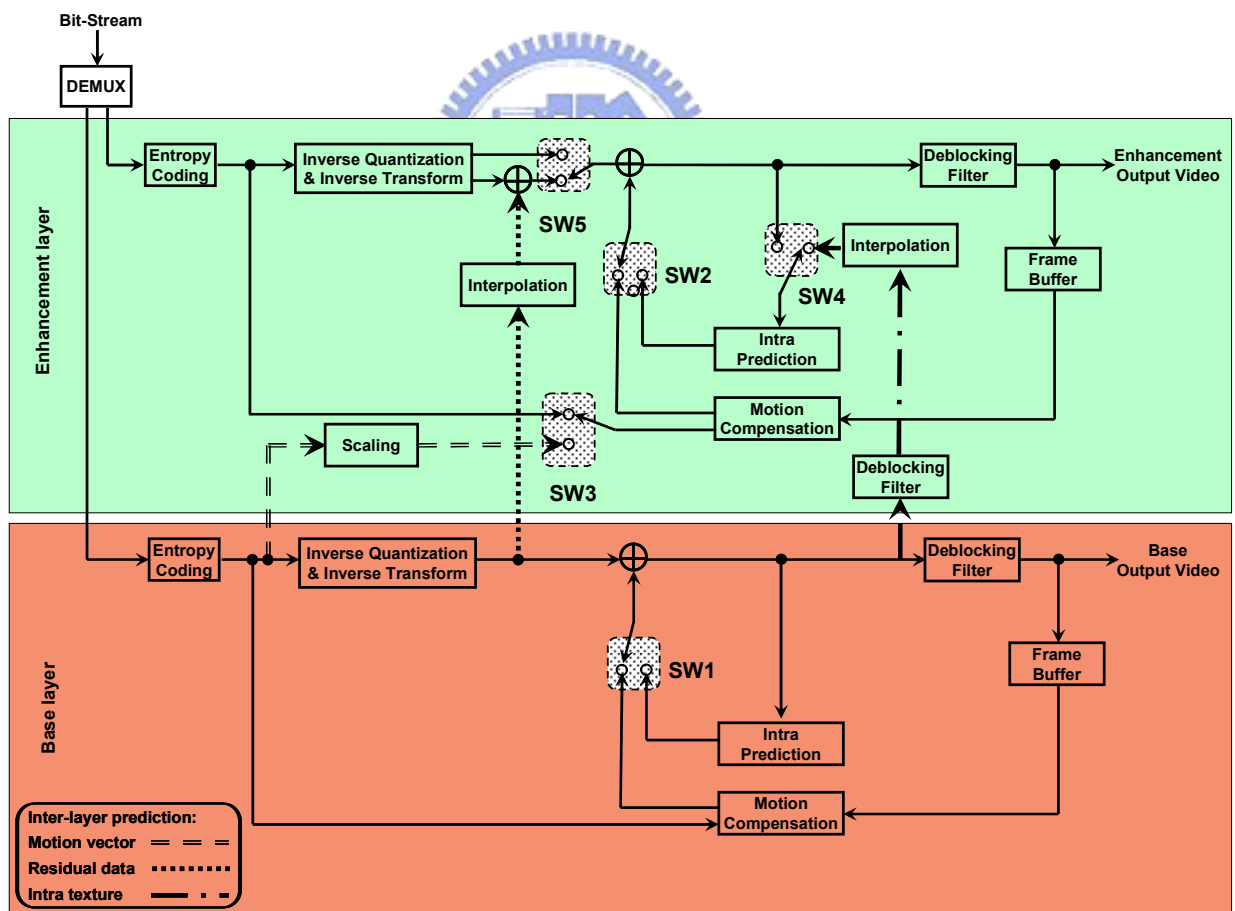



Figure 6-1 System architecture of the SVC video decoder

## 6.2 Procedure of the Implementation Work

As discussed in section 6.1 we port the scalable extension of H.264 decoder on the DSP board. Because JSVM is the only reference software that is available for SVC, it is our starting point for porting. JSVM includes a H.264/AVC SVC encoder, a decoder and other some useful tools, and these different programs share some common functions. It is developed on Visual Studio platform. Hence, the first step is to extract the decoder from JSVM to make it a stand alone program. In our implementation, we extract the H.264/AVC SVC decoder from the reference software JSVM 5.0 [27].

After making the decoder an independent program, the next step is to port the code from the Visual Studio (on PC) to the Code Composer Studio (CCS, the integrated development environment for TI's DSP). Because CCS does not support all Visual Studio C++ programming functionality, in this step there are some problems as described below.

- 
- (1) CCS itself does not provide the Standard Temporal Library (STL), but JSVM uses STL a lot. Therefore, we found a STL called STLport [28] which can be ported to many platforms, and after a proper set-up of configuration, STLport can work correctly on CCS.
  - (2) Exception handing is not supported by CCS. That is, the keywords “try”, “throw” and “catch” of C++ can not be used in CCS, but those keywords are found in JSVM. So all of these keywords must be removed and modified properly to ensure the correctness of the whole code.
  - (3) CCS does not implement some useful headers in C++ such as iostream.h, io.h and so on. Therefore, we replace these codes by the equivalent and supported functions.

Other than the problem of lacking some functionality support in CCS, when we port the JSVM 5.0 decoder on the DSP, we need to check the decoding results on DSP against those on PC. So we need to solve some problems for sending function parameters or the memory

index pointer of the decoder on CCS. After the finishing the porting, we can ensure the result of decoding is correct.

### 6.3 JSVM 5.0 Decoder Complexity Analysis

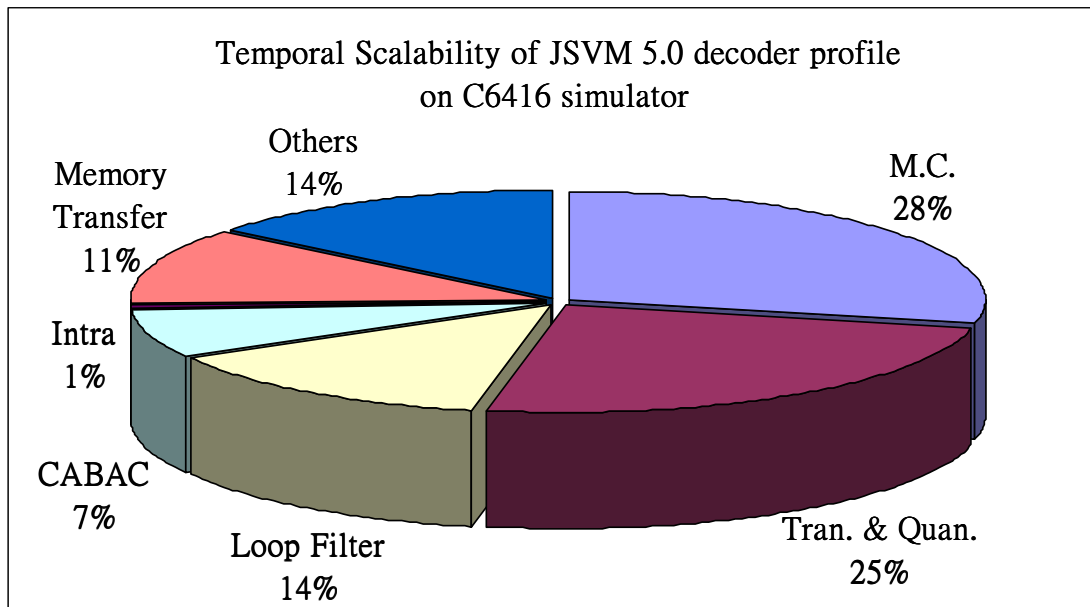
We profile the JSVM 5.0 decoder to find which part takes the most computation time on DSP. In order to profile the decoder, we use the profile of the stand-alone C6416T DSP simulator. We concentrate on the most critical areas and try to accelerate these modules. In chapter 3, we know that the scalable extension of H.264 have three types of scalability. In this section, we profile each scalability separately. Finally, we profile the combined scalability, which contains the spatial, temporal and SNR scalability. The profiling results using different scalability are shown in Figure 6-2 (Temporal), Figure 6-3 (Spatial), Figure 6-4 (SNR) and Figure 6-5 (Combined). The test video sequence is “city.264”, the simulation condition is shown in Table 6-1. And the compiler optimization level configuration of C6416 simulator is the “File” level (-o3) and we already use the L2 cache, which has been described in section 5.3.2 Table 6-2 shows the cycles of different scalability. We can see that the Spatial and SNR scalability need more cycles than temporal scalability. In following section, we will focus on these two types of scalability.

**Table 6-1** Simulation parameters

Test sequence: city.264 IPPPPP 9 Frames				
	GOP size	Frame size	FGS layers	QP
Temporal	4	CIF	0	30
Spatial	1	QCIF,CIF	0	30
SNR (FGS)	1	QCIF	1	30, 24
Combined	4	QCIF, CIF	1	30, 24

**Table 6-2** Cycles on different scalability

Scalability	Total cycles
Temporal	540,286,942
Spatial	7,427,104,703
SNR (FGS)	2,027,805,801
Combined	30,067,421,887



**Figure 6-2** Complexity profiling of the Temporal Scalability of JSVM 5.0 decoder



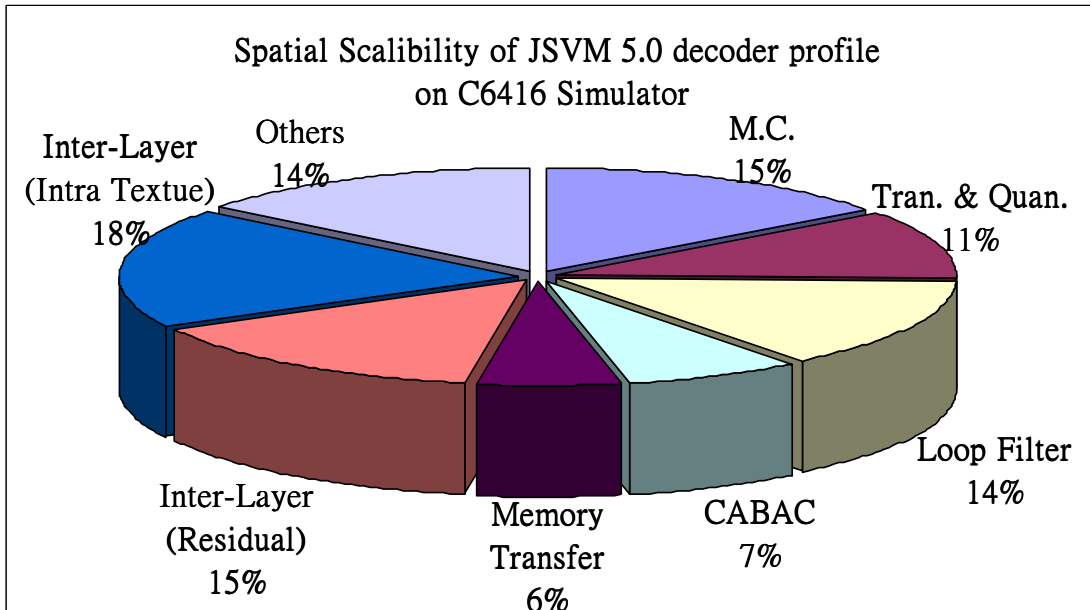


Figure 6-3 Complexity profiling of the Spatial Scalability of JSVM 5.0 decoder

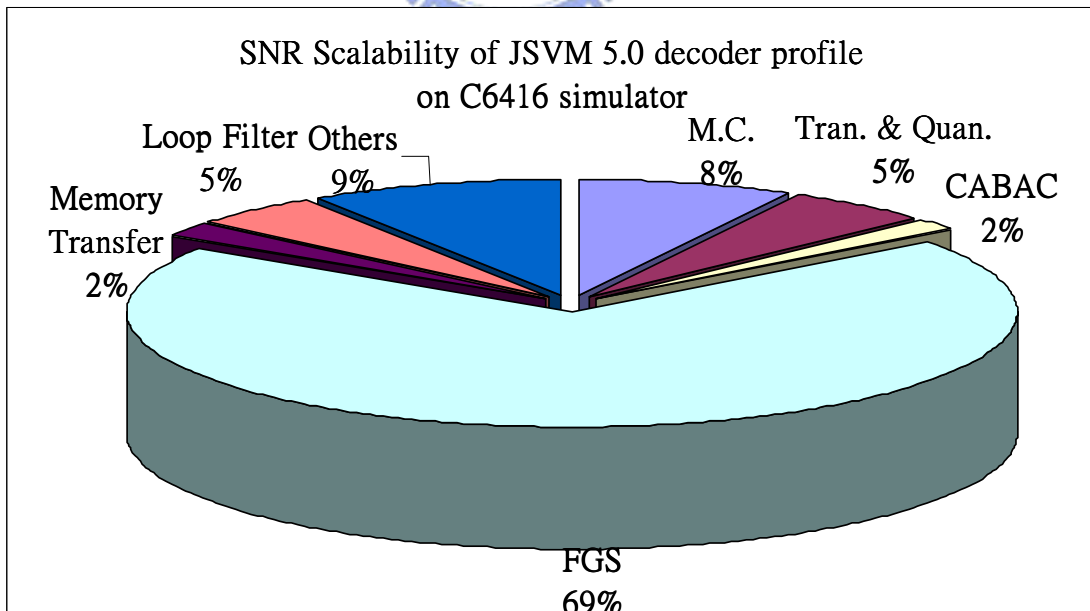
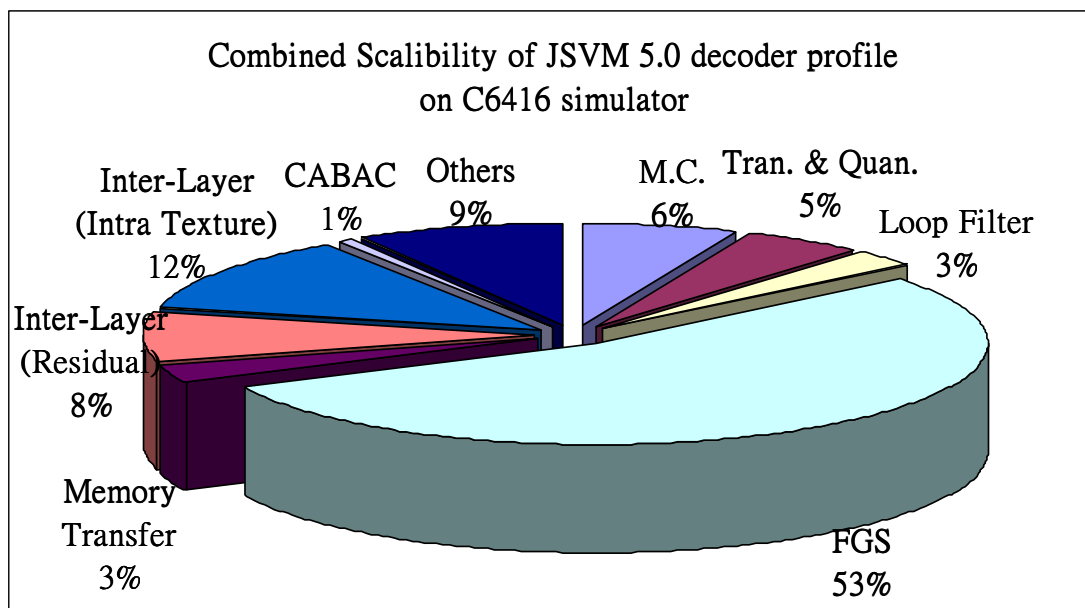


Figure 6-4 Complexity profiling of the SNR Scalability of JSVM 5.0 decoder



**Figure 6-5** Complexity profiling of the Combined Scalability of JSVM 5.0 decoder

In Figure 6-2, the profile of the temporal scalability is almost as same as H.264 decoder. The major complex parts are motion compensation, loop filter, entropy coding (CABAC), transform and quantization. In Figure 6-3, the profile of spatial scalability, the most computation part is inter-layer prediction for residual and intra texture. Its computation percentage is almost 33%. In Figure 6-4, when we profile the SNR scalability the most complex part is FGS. Finally, in Figure 6-5, the major computation parts of combined scalability are inter-prediction which takes about 20% and FGS which takes 53%. In the following sections, we develop several techniques to reduce the complexity of the major computation parts.

## 6.4 DSP Code Acceleration Methods

In this section, we will describe several schemes that we can optimize our C/C++ codes and reduce the DSP execution time on the C6416T DSP platform. These techniques take the advantages of the features of C64x.

### 6.4.1 Packet Data Processing

It is often desirable to use a single load or store instruction to access multiple data values consecutively located in memory. It is called the Single Instruction Multiple Data (SIMD) method. For example, when operating on a bit-stream, we can use word (32-bit) accesses to process read two 16-bit (short) or four 8-bit data (char) values at a time. This method can improve the code efficiency substantially. Figure 6-6 shows an example of using the SIMD method. Some intrinsic functions enhance the efficiency in a similar way.

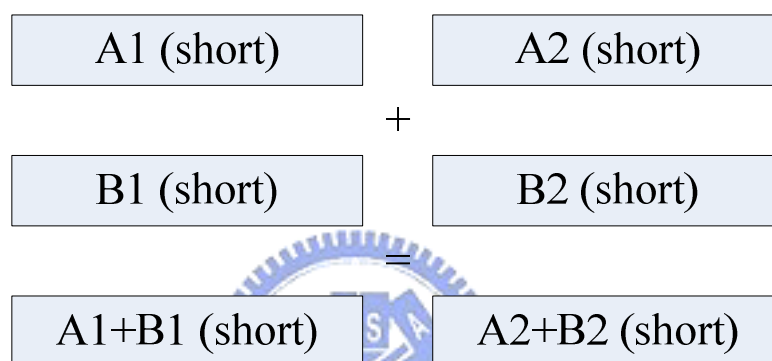
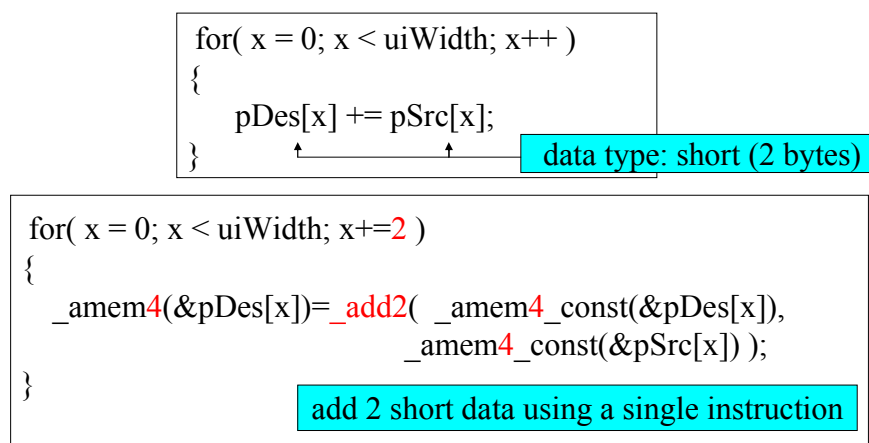


Figure 6-6 SIMD example of using the word instructions for adding short data

### 6.4.2 Intrinsic

The TI C6000 compiler provides many special functions that map directly to the inlined C64x instructions. It speeds up the C codes. These special functions are called intrinsics. If an instruction has equivalent intrinsic functions, we can replace it by intrinsic functions. The execution time will be decreased because of the use of intrinsics. Intrinsics are specified with a leading underscore (`_`) and are accessed by calling them as ordinary functions. There are quite a few intrinsic functions defined for the C6000 series DSP. More details about the TI DSP intrinsic functions are described in [17]. In the JSVM 5.0 decoder, we use the intrinsic function to deal with the calculation of pixels. Such as pixel add, subtract or copy. For example in Figure 6-7, we can use the intrinsic function “`_add2`” to replace the original function. The calculation is reduced by 50% since the “`_add2`” intrinsic can perform two

additions in one instruction. The performance of adopting intrinsic is shown in Table 6-3.



**Figure 6-7** Use of intrinsic function in the SVC decoder

**Table 6-3** Performance using intrinsic

Function	Original cycles	Revised cycles	Reduction Ratio (%)
add	243,320,438	130,990,721	46.1%
copy	488,524,940	159,347,857	67.3%
subtract	58,412,795	14,885,097	74.5%
up-sampleResidual	1,818,090,978	1,711,585,722	5.9%

### 6.4.3 Memory Allocation Optimization

In section 4.2.2, we know that the sizes of the internal program memory and the internal data memory are both 16 K-bytes for C6416T. The code segment should be put into the internal program memory. However, our codes may require a larger memory size than the internal memory. For instance, when dealing with a large image, it can not load the whole image into the internal data memory. For this reason, the data would be put into the external memory. If the accessed data are located in the external memory, it needs more clock cycles to transfer data to CPU. We can use registers to store data to reduce transfer time. In the DSP code, we can rearrange the link.cmd file, which is the memory allocation file. We put different type of

data in different memory sections for acceleration consideration. It also provides the “CODE\_SECTION”, “DATA\_SECTION” key words, which can allocate parts of C code or data in the internal memory. In order to improve the JSVM decoder execution cycles on DSP, we put some frequently used functions into the internal memory. This method can decrease the memory access time.

#### **6.4.4 DSP library**

The TI C64x DSP library is an optimized DSP function fir C programmers using C64x devices. It includes many C-callable, assembly optimized, and general-purpose signal-processing routines. By using these routines, we can achieve execution speed up considerably faster than the equivalent code written in the standard C language. We can use the DSP library (includes convolution, fft, iddt...etc) to replace the original functions in the decoder.

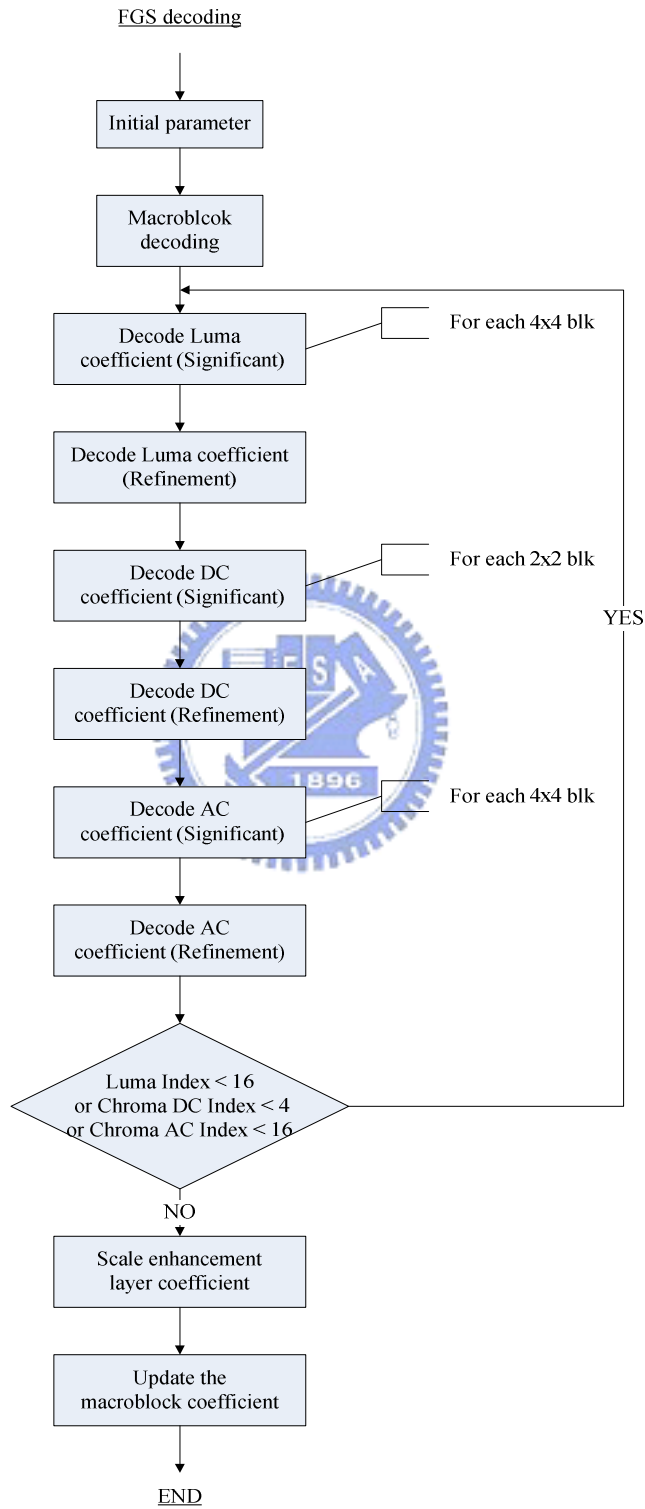
### **6.5 Fast Algorithms for the SVC Decoder**

In this section, we describe the implementation of the inter-layer prediction and the FGS operation in the JSVM decoder on DSP. We also modify some methods wherever possible to reduce the computations.

#### **6.5.1 FGS**

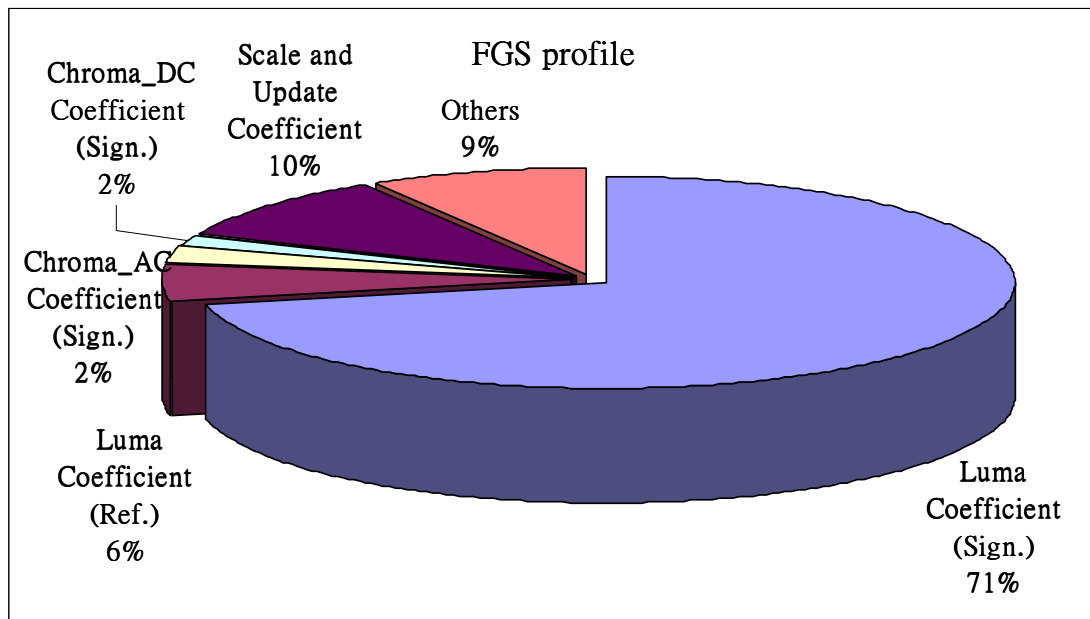
FGS (Fine Granularity Scalability) is one tool used by the SNR scalability. The details of FGS have been described in section 3.4.2 Figure 6-8 shows the block diagram of FGS in the JSVM 5.0 decoder. In the JSVM decoder, the FGS can be divided into three parts: luma, chroma DC and chroma AC. Each part includes the significant path and the refinement path. The refinement path is turned on only when the significant path is completed. When entering the decode luma significant coefficient box, it only deals with the 4x4 blocks. So decoding the luma refinement coefficient operation will begins to execute only when all the significant coefficients in every 4x4 block in a macroblock are completely checked. The same case

occurs in chroma DC and chroma AC. When all these significant and refinement path in a macroblock are completed, the enhancement coefficients are properly scaled and the update the macroblock coefficients. The flow chart of FGS is shown in Figure 6-8.



**Figure 6-8** Flow chart of FGS in the JSVM 5.0 decoder

Figure 6-5 tells us that the FGS takes a large percentage of computing time in the combined scalability case. First, we profile the FGS operation which is shown in Figure 6-9. The complex parts are decoding the luma significant coefficients, luma refinement coefficients, chroma AC significant coefficient, the chroma DC significant coefficient and scaling and updating the macroblock coefficients. In the following sections, we adopt some methods to speed the FGS operation.



**Figure 6-9** Complexity profiling of FGS on C6416 simulator

**(A) Early termination for luma significant coefficient**

In Figure 6-9, decoding luma significant coefficient is the part that takes the most computing time. So, if we would reduce the cycles of FGS. The decoding luma significant coefficient needs modification. In the decoding luma significant coefficient passes, each loop checks a 4x4 block for whether the block have significant coefficients. To speed up this process, we set an early termination point. If all of the significant coefficients in this block are done, in the next loop we skip checking this block again. This can save the time of checking block and setting up parameters.

### **(B) Check skipped blocks for null coefficient blocks**

In the significant and the refinement paths, some coefficients are not significant or refinement. These coefficients are zeros. If all coefficients in a block are all zero, this block is called a null. In the null block, scaling and transforming the coefficients is redundancy. So the block of scaling the enhancement layer coefficients can be skipped. We can detect whether the block is null or not. This method saves the redundant time in calculating the scaling operation.

### **(C) Code refinement**

In the FGS block, some functions are shared with the other components in the JSVM 5.0 decoder. For example, the function “initMB” is a tool which initials all parameters in a macroblock. But some parameters are not need in the FGS process. For example, motion vectors are needed only in motion estimation or motion composition but FGS does not use the parameters of motion vectors. So we can rewrite the function “initMB” that only applies to FGS.

Table 6-4 shows the reduction ratio of each function in all FGS block and Table 6-5 shows the results of accelerating the FGS block on the overall system which condition as same as Table 6-1. We notice that the reduction ratio of the operational cycles is about 61%.

**Table 6-4** Reduction Ratio of FGS block

Function	Original cycles	Revised Cycles	Reduction Ratio (%)
Luma significant coefficients	923,838,292	315,713,695	65.8%
Luma refinement coefficients	70,901,624	30,425,812	57%
Chroma DC Significant coefficients	19,501,221	8,939,283	54%
Chroma AC Significant coefficients	30,558,382	21,322,965	30.2%
Scaling and Updating coefficient	130,753,983	35,332,570	72.9%
Total	1,297,419,071	501,257,788	61.3%



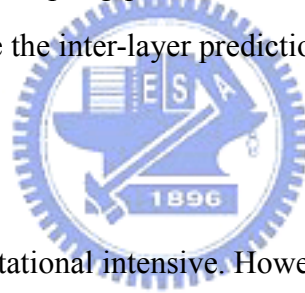
**Table 6-5** Performance of the modified FGS

Test sequence: city_qcif.264			
Type	Original Cycles	Proposed Cycles	Reduction Ratio (%)
SNR	2,017,898,772	870,697,861	56.8%
Combined	28,156,792,014	16,860,453,637	40.1%

### 6.5.2 Inter-layer Prediction

For spatial scalability, the most important component is inter-layer prediction. But from Figure 6-3, we find that the inter-layer prediction decoding takes a large amount of computations. This is because that in decoding the spatial enhancement layer, the motion vectors, residuals, and intra texture data of the base layer should be up-sampled for their use at enhancement layer. The up-sampling process is complex and takes a lot of computations. We design algorithms to reduce the inter-layer prediction computation.

#### (A) Intra texture prediction

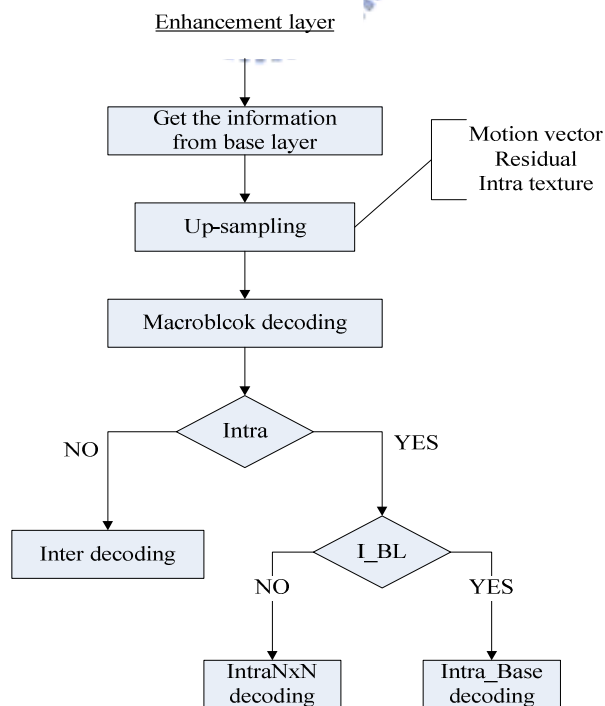


Inter-layer prediction is computational intensive. However, not all of the up-sampling data are needed in the enhancement layer. For example, the inter-layer intra prediction up-samples the reconstructed intra signal of the base layer. In up-sampling the luma component, one-dimensional 6-tap filter FIR filters are applied horizontally and vertically. The chroma components are up-sampled using a simple bi-linear filter. In the JSVM decoder, when the inter-layer prediction is in use, all the reconstructed signals of the base layer are up-sampled. Figure 6-10 shows that the inter-layer prediction is performed before the macroblock decoding. But only a few blocks request for the intra prediction operation typically. In the JSVM 5.0 decoder, only the “Intra\_BL” mode needs to use the information from the base layer. Table 6-6 shows that in the spatial scalability decoding, only 2% of block are using the “Intra\_BL” mode. In this case, up-sampling all the intra texture data is unnecessary. Hence, we first decide whether the intra texture data should be up-sampled or not. The procedure is shown in Figure 6-11. If the current enhancement layer mode is the “Intra\_BL” mode, than

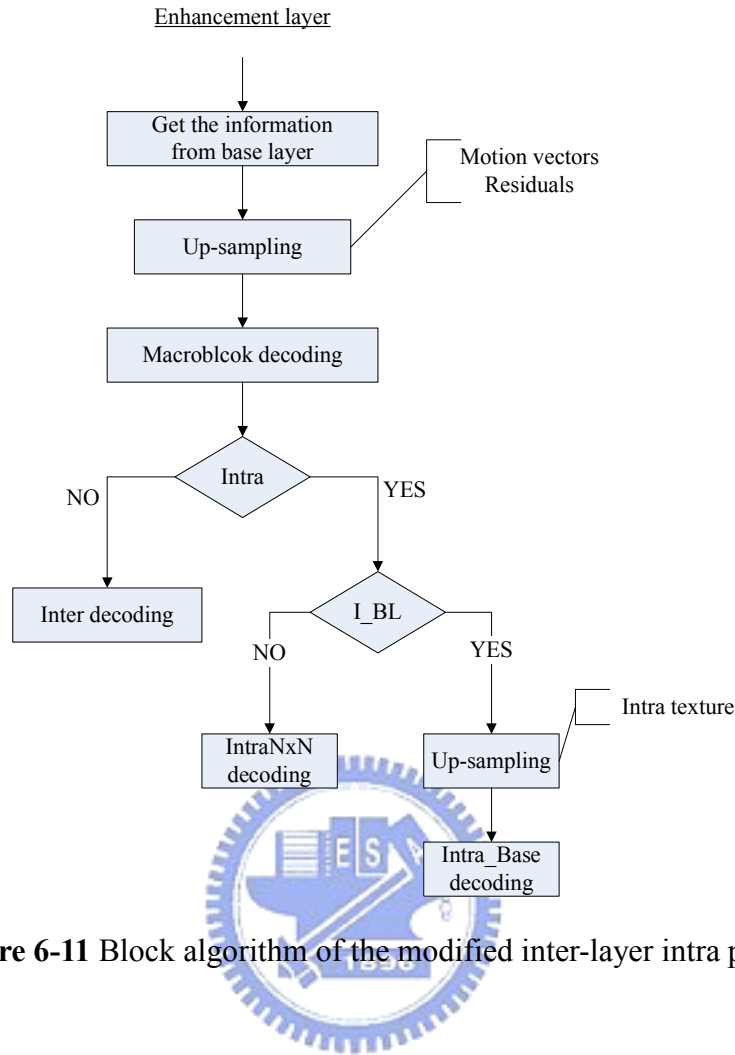
the intra texture data would be up-sampled. Otherwise, it would not be up-sampled.

**Table 6-6** Distribution of mode in spatial scalability

Base layer			Enhancement layer		
Mode	Number	Percentage (%)	Mode	Number	Percentage (%)
Intra_4x4	80	0.54%	Intra_4x4	56	0.31%
Intra_8x8	26	0.18%	Intra_8x8	25	0.14%
Intra_16x16	1	0.01%	Intra_16x16	9	0.05%
Intra_BL	0	0%	<b>Intra_BL</b>	<b>419</b>	<b>2.29%</b>
Inter_8x8	2026	13.64%	Inter_8x8	8048	44.00%
Inter_8x16	2041	13.74%	Inter_8x16	1708	9.34%
Inter_16x8	906	6.10%	Inter_16x8	1104	6.04%
Inter_16x16	3701	24.92%	Inter_16x16	5837	31.91%
Skip	6069	40.87%	Skip	1084	5.93%
<b>Total</b>	<b>14850</b>	<b>100%</b>	<b>Total</b>	<b>18290</b>	<b>100%</b>



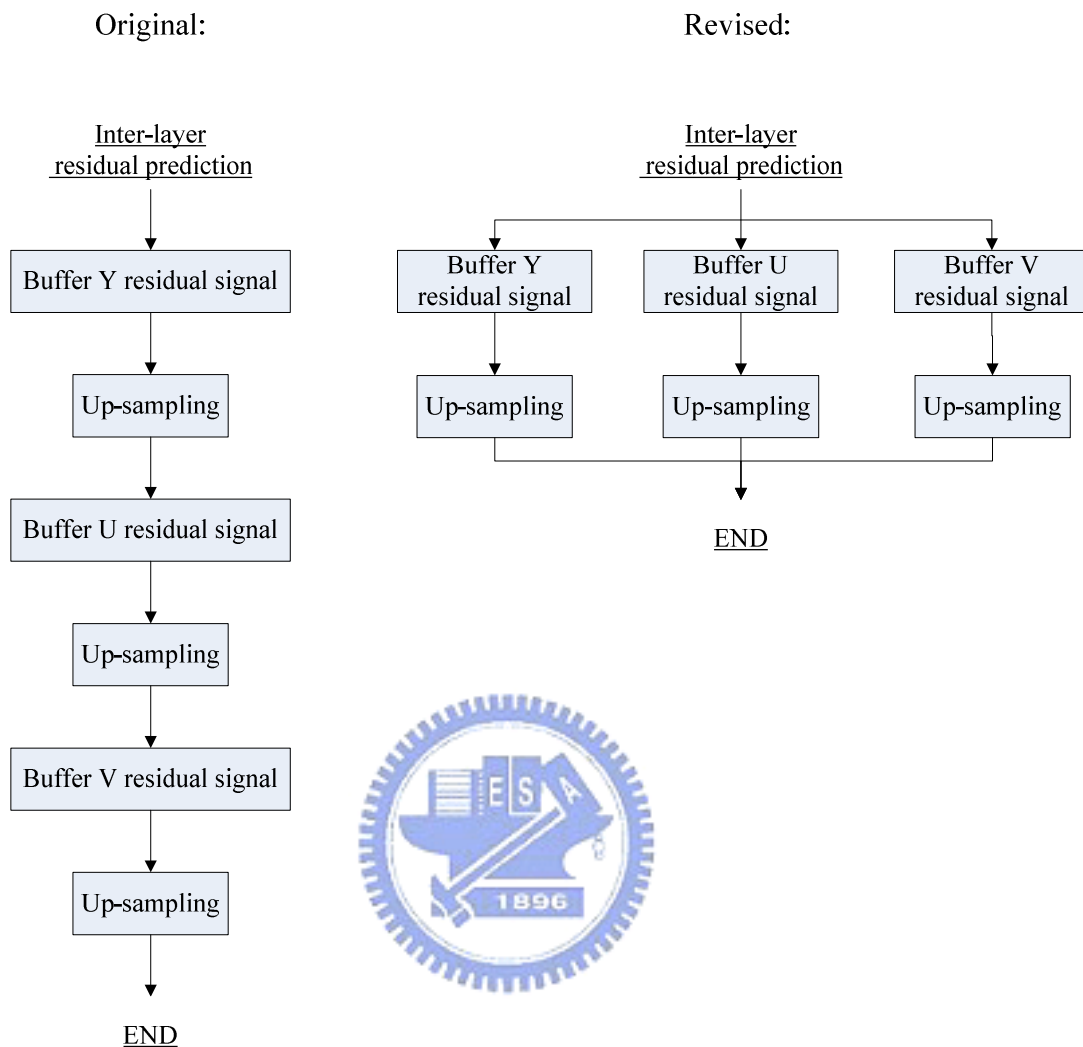
**Figure 6-10** Block algorithm of the original inter-layer intra prediction



**Figure 6-11** Block algorithm of the modified inter-layer intra prediction

### (B) Residual prediction

The inter-layer residual prediction can be employed for all inter-coded macroblocks. The residual signal is up-sampled using a bi-linear filter and used as the prediction value for the residual signal of the enhancement layer. Thus, only the associated difference signal is coded in the enhancement layer. In the JSVM decoder, the inter-layer residual prediction requires a lot of memory transfers. On the DSP platform, the memory transfer is costly in time. So we modify the codes to reduce the number of memory transfer. For the inter-layer residual prediction coding, the residuals are split into Y, U and V residual signal. But in the JSVM decoder, the Y, U and V residual signals are up-sampled together. So we take the Y, U and V residual signals apart. We only up-samples the residual signals that are needed. The modified block diagram is shown in Figure 6-12.



**Figure 6-12** Difference in the inter-layer residual prediction procedures

Table 6-7 shows the reduction ratio of the inter-layer prediction. We rewrite the program in order to accelerate the inter-layer part. Table 6-8 shows that the operational cycles of different scalability are reduced.

**Table 6-7** Reduction ratio of Inter-layer prediction

Function	Original cycles	Revised cycles	Reduction ratio (%)
Residual	889,289,854	757,070,091	14.87%
Intra texture	532,432,371	35,668,577	93.30%

**Table 6-8** Performance of the modified inter-layer prediction

Test sequence : city_qcif.264			
Type	Original cycles	Proposed cycles	Reduction Ratio (%)
Spatial	5,966,597,605	5,198,506,424	12.8%
Combined	16,860,453,637	15,472,723,853	8.2%

## 6.6 Final Simulation and Acceleration Results

After accelerating the codes and modifying the algorithms, we have efficiently reduced the computations of the JSVM 5.0 decoder on DSP. Table 6-9 shows the comparison of the processes with and without the L2 cache. We can clearly see that the reduction ratio achieves 50%. The improvement is not as much as the reduction ratio in the x264 case in Table 5-11, because the code size and data size of the JSVM 5.0 decoder is larger than the x264. Using two-level cache can not reduce the data cache miss a lot. As shown in Table 6-10, the data cache miss rate decreases from 56.63% to 27.12%. The data cache miss still is large. Table 6-11 shows the improvement of the optimized codes compared with the original version. The simulation condition is shown in Table 6-1. After optimizing the codes, the improvement achieves 20% in temporal scalability, 30% in spatial scalability, 55% in SNR scalability and 49% in combined scalability. We can decrease the half execution time on overall system. Table 6-12 shows the real execution time on the C6416 emulator. The testing condition is the same as that in Table 6-1. The resulting system can decode approximately 15 frames per second in the baselayer and the temporal scalability, 1.32 frames in the spatial scalability, 2.84 frames in the SNR scalability and 0.4 frames in the combined scalability finally.

**Table 6-9** Comparison of using C6416 simulator with and without the L2 cache

Sequence	Without L2 cache	With L2 Cache	Reduction Ratio (%)
City	64,576,729,391	30,067,421,887	53.44%
Akiyo	53,685,160,996	24,403,283,420	54.54%
Foreman	65,599,674,888	30,902,096,589	52.89%

**Table 6-10** Effect of using L2 cache memory

C6416 simulator	Without L2 cache		With L2 cache	
	Count	Percentage	Count	Percentage
Total Cycles	64,576,760,635		30,067,433,857	
Core cycles(excl. stalls)	755,799,273	7.82%	755,799,785	16.8%
NOP cycles	2,354,743,973	46.5%	2,354,744,076	46.5%
Stall Cycles	59,526,048,163	92.18%	25,016,840,078	83.2%
Instruction cache hits	953,423,952	97.36%	953,507,967	97.37%
Instruction cache misses	25,882,890	2.64%	25,799,022	2.63%
Data cache references	915,440,375		915,440,489	
Data cache reads	621,323,760	67.87%	621,323,824	67.87%
Data cache writes	294,116,624	32.13%	294,116,672	32.13%
Data cache hits	397,007,357	43.37%	667,191,138	72.88%
Data cache read hits	221,802,890	35.7%	446,661,075	71.89%
Data cache write hits	175,204,467	59.57%	220,530,063	74.98%
<b>Data cache misses</b>	<b>518,433,018</b>	<b>56.63%</b>	<b>248,249,351</b>	<b>27.12%</b>

**Table 6-11** Comparison using C6416 simulator on the original and the modified codes

Sequence	Type	Original cycles	Optimized Cycles	Reduction Ration (%)
City	Baselayer	521,049,251	415,868,509	20.19%
	Temporal	540,286,942	428,750,233	20.64%
	Spatial	7,427,104,703	5,198,506,424	30.01%
	SNR	2,027,805,801	857,551,343	57.71%
	Combined	30,067,421,887	15,472,723,853	48.54%
Akiyo	Baselayer	427,770,004	337728563	21.05%
	Temporal	442,938,814	348379134	21.35%
	Spatial	6,353,717,397	4,077,516,481	35.82%
	SNR	1,743,465,294	790,636,440	54.65%
	Combined	24,403,283,420	12,474,314,013	48.88%
Foreman	Baselayer	491,066,392	383,309,737	21.94%
	Temporal	518,112,745	405,372,731	21.76%
	Spatial	7,327,335,062	5,069,933,810	30.81%
	SNR	1,968,622,159	853,436,646	56.65%
	Combined	30,902,096,589	15,781,271,800	48.93%

**Table 6-12** Performance on the C6416 emulator

Sequence	Type	Execution Time 9 frames (sec.)	Conversion 1 frames (sec)	fps (frames/sec.)
City	Baselayer	0.61	0.068	14.75
	Temporal	0.62	0.069	14.52
	Spatial	7.8	0.87	1.15
	SNR	3.8	0.42	2.37
	Combined	25.7	2.86	0.35
Akiyo	Baselayer	0.41	0.046	21.95
	Temporal	0.41	0.046	21.95
	Spatial	5.5	0.61	1.64
	SNR	2.5	0.28	3.60
	Combined	17.3	1.92	0.52
Foreman	Baselayer	0.55	0.061	16.36
	Temporal	0.55	0.061	16.36
	Spatial	7.7	0.86	1.17
	SNR	3.5	0.39	2.57
	Combined	25.6	2.84	0.35



# Chapter 7

## Conclusions and Future Work

### 7.1 Conclusion

The main goal of this work is to accelerate the H.264/AVC baseline profile encoder and the H.264/AVC SVC decoder implemented on the TI TMS320C6416T DSP processor. The H.264/AVC encoder implementation is developed based on the open source code x264 and the H.264/AVC SVC decoder, which is extracted from the MPEG reference software JSVM 5.0.

For H.264/AVC encoder, we first modify the mode decision module to accelerate the H.264/AVC encoder operations. We use the early termination to reduce the calculation of dispensable mode. This acceleration method can reduce the computational cost by 13% without visual quality and bits rate degradation. For the DSP implementation, one initial problem is the frequent occurring of data cache miss. We use the two-level cache to solve this problem. We also use various code speed-up techniques supported by the C64x to match its hardware architecture. The details are described in chapter 5. The total execution time is reduced by 50%. Finally, the overall system can on the average encode about 40 QCIF frames per second on the C6416 emulator.

Furthermore, we also have successfully implemented the decoder of H.264/AVC SVC on the DSP platform. The H.264/AVC SVC includes three types of scalabilities including temporal, spatial and SNR scalability. Based on the profiling data, the FGS module and inter-layer prediction module are two high computational complexity parts in the JSVM decoder of combined scalability. For the FGS module, we use the early termination to reduce the unnecessary checking of luma significant coefficients and updating the null block. We also refine the FGS codes to reduce the computation. For the inter-layer prediction, we reduce the

unnecessary up-sampling operations for the intra texture between the base and the enhancement layer. Another code acceleration method is the inter-layer prediction which has Y, U and V three components. We only up-sample the residual signals that are needed. We also adopt the coding acceleration techniques on C64x to accelerate the decoder on the DSP platform. The improvement can achieve 49% in combined scalability. More details are shown in chapter 6.

## 7.2 Future Work

The time to read or write file from host is a major problem in the DSP environment. If we can reduce the data transmission time between the host and the target, the system will run faster. Hence, the transmission time reduction techniques should be studied.

For the H.264/AVC encoder, we only support the baseline profile. The H.264/AVC main profile includes more complex element than baseline profile. If we need compress the video more efficiently, the main profile is necessary. Furthermore, we can integrate the FPGA hardware implementation together with DSP to accelerate the overall system.

The code of JSVM decoder is written in C++ language. But the DSP platform can not perform the codes in C++ efficiently. Thus, we should rewrite the entire JSVM decoder using C language. In addition, we only focus on the FGS and inter-layer prediction modules in this thesis. The JSVM decoder can be further accelerated by modifying the other modules.

# References

- [1] Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, “Draft ITU-T Recommendation and Final Draft international Standard of Joint Video Specification (ITU-T Rec. H.264 |ISO/IEC 14496-10 AVC),” JVT-G050, March 2003.
- [2] T. Wiegand, G. J. Sullivan, G. Biontegaard, and A. Luthra, “Overview of the H.264/AVC Video Coding Standard,” IEEE Transactions on Circuit and System for Video Technology, VOL. 13, Issue 7, pp. 560-576, Jul. 2003.
- [3] H.264/MPEG-4 Part 10 tutorials, <http://www.vcodex.com/h264.html>.
- [4] J. W. Chen, C. Y. Kao, and Y. L. Lin “Introduction to H.264 Advanced Video Coding,” Department of Computer Science National Tsing Hua University, Jan. 2006.
- [5] SVC home page: <http://ftp3.itu.int/av-arch/jvt-site/>.
- [6] H. Schwarz, D. Marpe, and T. Wiegand, “Overview of the Scalable Extension of the H.264/MPEG-4 AVC Video Coding Standard,” Oct. 2006.
- [7] J. Reichel, H. Schwarz, and M. Wien, “Joint Scalable Video model JSVM-5,” ISO/IEC JTC1/SC29/WG11/JVT-R202, Bangkok, Thailand, Jan. 2006.
- [8] J. Reichel, M. Wien, and H. Schwarz, “Scalable Video Model 3.0,” ISO/IEC JTC 1/SC 29/WG 11 N6716, October 2004.
- [9] W. H. Peng, C.Y. Tsai, T. Chiang, and H. M. Hang, “Advances of MPEG Scalable Video Coding Standard,” National Chiao-Tung University, International Workshop on Intelligent Information Hiding and Multimedia Signal Processing, Melbourne, Australia, 14 -- 16, Sept 2005.
- [10] H. Schwarz, D. Marpe, and T. Wiegand, “MCTF and Scalability Extension of

- H.264/AVC,” in Proc. of PCS, San Francisco, CA, USA, Dec. 2004.
- [11] J. Ridge. X. Wang, “Simplification and Unification of FGS,” ISO/IEC JTC1/SC29/WG11/JVT-S077, Apr. 2006.
- [12] Sundance home page:<http://www.hitechglobal.com/SunDance/SMT395VP30.htm>.
- [13] Texas Instruments, “TMS320C6416T, TMS320C6415T, TMS320C6416T fixed-point Digital Signal Processors,” Literature number SPRS226J, JULY. 2006.
- [14] Texas Instruments, “TMS320C6000 Peripherals Reference Guide,” Literature number SPRU190D, Feb. 2001.
- [15] Texas Instruments, “TMS320C64x Technical Overview,” Literature number SPRU395B, Jan. 2001.
- [16] Texas Instruments, “TMS320C6000 Code Composer Studio Tutorial,” Literature number SPRU301C, Feb. 2000.
- [17] Texas Instruments, “TMS320C6000 Programmer’s Guide,” Literature number SPRU198I, Mar. 2006.
- [18] Texas Instruments, “TMS320C6000 Optimizing Compiler v 6.0 Beta,” Literature number SPRU187N, July. 2005.
- [19] ffdshow reference software,  
[http://www.afterdawn.com/software/video\\_software/codecs\\_and\\_filters/ffdshow.cfm](http://www.afterdawn.com/software/video_software/codecs_and_filters/ffdshow.cfm).
- [20] ffmpeg Multimedia System, <http://ffmpeg.sourceforge.net/index.php>.
- [21] x264 reference software, <http://developers.videolan.org/x264.html>.
- [22] H.264 JVT reference software, <http://iphome.hhi.de/suehring/tml/>.
- [23] C. Grecos, and M. Y. Yang, “Fast Inter Mode Prediction for P Slices in the H264 Video Coding Standard,” IEEE Transactions on Broadcasting, Vol. 51, No. 2, pp. 256-263, June 2005.

- [24] Z. Zhou, and M. T. Sun, “Fast Macroblock Inter Mode Decision and Motion Estimation for H.264/MPEG-4 AVC,” International Conference on Image Processing, Vol. 2, pp. 789-792, 2004.
- [25] P. Yin, H.Y.C. Tourapis, A.M. Tourapis, and J. Boyce, “Fast Mode Decision and Motion Estimation for JVT/H.264,” Image Processing, 2003. Vol. 3, pp. 853-856, ICIP 2003.
- [26] Texas Instruments, “TMS320C6000 DSP 32-Bit Timer,” Literature number SPRU582B, Jan. 2005.
- [27] JSVM 5.0 Reference Software:  
[http://ftp3.itu.int/av-arch/jvt-site/2006\\_01\\_Bangkok/JVT-R203.zip](http://ftp3.itu.int/av-arch/jvt-site/2006_01_Bangkok/JVT-R203.zip).
- [28] STLport home page: <http://www.stlport.org/>.



## 自 傳

鄭凱庭，民國七十一年生於新竹市。民國九十四年畢業於台灣新竹國立交通大學電機與控制工程學系，之後進入交通大學電子研究所攻讀碩士學位，從事多媒體訊號處理系統之相關研究，研究興趣為數位訊號處理、多媒體系統、視訊壓縮等方面。

