國立交通大學

電機資訊學院 電子與光電學程

碩 士 論 文

使用具單指令多資料流程之 Intel 處理器實現 MPEG-4 即時視訊編碼

Real-Time Implementation of MPEG-4 Video Encoder Using SIMD-Enhanced Intel Processor

研 究 生：劉夢遠

指導教授：林大衛 博士

中 華 民 國 九 十 三 年 七 月

使用具單指令多資料流程之 Intel 處理器實現 MPEG-4 即時視訊編碼
Real-Time Implementation of MPEG-4 Video Encoder Using SIMD-Enhanced Intel Processor

研 究 生：劉夢遠　　Student：Meng-Yuan Liu

指導教授：林大衛 博士　　Advisor：Dr. David W. Lin

國 立 交 通 大 學

電機資訊學院 電子與光電學程

碩 士 論 文

A Thesis

Submitted to Degree Program of Electrical Engineering Computer Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Electronics and Electro-Optical Engineering

July 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年七月

使用具單指令多資料流程之 Intel 處理器實現 MPEG-4 即時視訊編碼

研究生: 劉夢遠　　指導教授: 林大衛 教授

國立交通大學電機資訊學院 電子與光電學程（研究所）碩士班

摘要

MPEG-4 提供一些新的架構與工具來達成高壓縮率的視訊編碼。在本篇論文中，我們使用具單指令多資料流程之 Intel 處理器實現即時 MPEG-4 視訊編碼。主要以 Intel MMX 技術為主，包含 SSE 及 SSE2。MMX 是 Intel 公司為了在 Intel Architecture（IA）的微處理器上能夠用來加強多媒體及通訊的處理能力所增加的延伸技術，採用了單指令多資料流程架構（SIMD）來平行處理資料運算。

在程式執行方面，我們使用一公開的程式 Microsoft MPEG-4 Visual Reference Software 加以修改以完成 MPEG-4 即時視訊編碼。由於 MPEG-4 的壓縮方式需要非常大的計算量，因此要達成即時的壓縮和解壓縮必須要有高速的硬體和有效率的軟體互相配合。為了解決龐大運算量的問題，平行處理是一個相當有效的方式。平行處理簡單的說就是把原本需要排隊循序處理的工作變成讓數個工作能同時獨立地運算，以加速工作的進行。本篇論文即是使用 Intel 的 MMX 指令集來改寫 Microsoft MPEG-4 Visual Reference Software 的部分核心程式，增加平行處理的程度，達成加速程式執行的目的。而最後程式在 Intel Pentium 4 CPU 2.66G, 480MB RAM 及 Microsoft Windows XP Professional 作業系統下實際測試的結果，使用 MMX 技術配合其他演算法壓縮有形狀訊息的 CIF foreman 測試檔案可達每秒 30 張約為原始程式的 6 倍左右。

在本篇論文中，我們會先簡單介紹 MPEG-4 的系統架構與 Intel MMX 技術及指令，然後我們會針對 MPEG-4 實作時的改善與加速提供詳細的介紹，我們會將

加速後的程式與原先的程式作比較，並討論其優缺點。論文最後做一結論並提出未來可再繼續發展的主題。

Real-Time Implementation of MPEG-4 Video Encoder Using SIMD-Enhanced Intel Processor

Student: Meng-Yuan Liu Advisor: Dr. David W. Lin

Degree Program of Electrical Engineering Computer Science
National Chiao Tung University

Abstract

The MPEG-4 standard is a very efficient coding standard for multimedia data defined by ISO/IEC MPEG. In this thesis, we use SIMD-enhanced Intel Processor to deal with MPEG-4 video encoding and to achieve the goal of real-time coding. The main technology is Intel's MMX including SSE and SSE2. The Intel MMX technology was introduced into the Intel Architecture (IA) processor. The extension introduced in the MMX technology support a single-instruction, multiple-data (SIMD) execution model that is designed to accelerate the performance of advanced media and communications applications.

In this thesis, we use the public-domain software, Microsoft MPEG-4 Visual Reference Software, to establish an MPEG-4 coding and decoding system. We need high-processing-speed hardware and effective software to achieve real-time MPEG-4 video compression and decompression, and parallel processing is the practical method that can solve huge computation problem in MPEG-4 encoding and decoding. Parallel processing means letting several independent operations or tasks run in parallel simultaneously, and then it can speed up the whole processing by this method. In this

thesis, we modify some kernels of Microsoft MPEG-4 Visual Reference Software using Intel's MMX technology to get more parallel processing ability to speed up the encoding processing. After optimization, we can encoder CIF foreman test sequence with shape information up to 30 frame per second on our test system. The test system is based on Intel Pentium 4 CPU 2.66G, 480MB RAM and Microsoft Windows XP Professional Version 2002. The speed-up is approximately 6 times than the original reference software.

In our thesis, we introduce the MPEG-4 and Intel's MMX technology first. Then we discuss the optimization of the MPEG-4 video encoder by using Intel's MMX technology. We also present experimental results on the speed and the rate-distortion performance of the optimized code. Finally, we give a conclusion and point out some subjects for potential future work.

誌謝

本篇論文的產生，最感謝的是我的指導教授－林大衛老師。身為一位在職研究生要兼顧研究學習及工作本來是件不容易的事，但老師給予我多方面的協助與鼓勵，讓我在研究學習中能夠順利進行，使我在研究學習及工作中得以兼顧而能完成本篇論文。老師除給了我完善的專業知識訓練外，在論文實作部份，老師更培養了我認真踏實的的研究態度，讓我獲益良多。

實驗室完善的資源，讓我可以順利的克服許多實作上所遇的的困難。感謝詹益鎬學長不吝提供其相關的研究經驗以及建議，讓我獲益多。另外，感謝岳賢、沛昀、彥福、建興、宗書、崑健、建統等實驗室所有同伴，他們的勉勵與幫助，讓我在工作中亦保有快樂的研究生生涯。

最後，要特別感謝我的爸媽、弟弟與我摯愛的妻子惠貞。尤其是母親與妻子，他們對於家庭的細心照顧讓我可在無後顧之憂下全力進行研究學習及工作，也給了我精神上的支持與鼓勵，陪伴我度過所有的艱難。在此，我要將我的論文獻給所有關心與幫助我的人。

劉夢遠

民國九十三年七月於新竹

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The MPEG-4 standard was originally intended for very high compression coding of audio-visual information at very low bit-rate. Later the scope of MPEG-4 was extended to address not only compression, but also new audio-video coding techniques for content-based interactivity and universal access. In addition to the conventional "frame" based functionalities of MPEG-1 and MPEG-2 standards, the MPEG-4 video coding will also support access and manipulation of "objects" within video scenes.

Because the computation of MPEG-4 video encode is quiet huge, we need high-processing-speed hardware and effective software to achieve real-time MPEG-4 video compression and decompression, and parallel processing is a practical technique that can solve huge computation problem in MPEG-4 encoding and decoding. Parallel processing means letting several independent operations or tasks run in parallel simultaneously. Then the speed of processing can be increased.

We consider implementation of the MPEG-4 video encoder in software on Intel processor. The implementation is based on the code from Microsoft MPEG-4 Visual Reference Software. It is a public source for MPEG-4 encoding and decoding. In order to achieve real-time performance, we use Intel's MMX instructions to modify some kernels of the reference software. Intel's MMX technology was introduced in to the IA architecture processor [8]. The extension introduced in the MMX technology support a single-instruction, multiple-data (SIMD) execution model that is designed to accelerate the performance of advanced media and communications applications [8].

This thesis is organized as follows. Chapter 2 is an overview of MPEG-4. Chapter 3 describes Intel's MMX technology and some software tools that we use. Chapter 4 discusses the detailed optimization methods for using Intel's MMX technology. The overall experimental results of the MPEG-4 encoder after optimization are described in Chapter 5. Finally, Chapter 6 contains the conclusion.

Chapter 2

Overview of MPEG-4

MPEG-4 is an ISO/IEC standard developed by MPEG (Moving Picture Experts Group), the committee that also developed the well known MPEG-1 and MPEG-2 standards. These standards made interactive video on CD-ROM, DVD and Digital Television possible. MPEG-4 is a newer standard started in 1994, with the mandate to standardize algorithms for audio-visual coding in multimedia applications. MPEG-4, formally designated "ISO/IEC 14496", was finalized in October 1998 and became an International Standard in the first months of 1999. The fully backward compatible extensions under the title of MPEG-4 Version 2 were frozen at the end of 1999, to acquire the formal International Standard Status early in 2000. Several extensions were added since and work on some specific items is still in progress [2]. MPEG-4 builds on the proven success of three fields:

- digital television,
- interactive graphics applications (synthetic content), and
- interactive multimedia (World Wide Web, distribution of and access to content).

In this chapter, we introduce the overall organization of the MPEG-4 standard, its video texture coding scheme, and some special video coding tools.

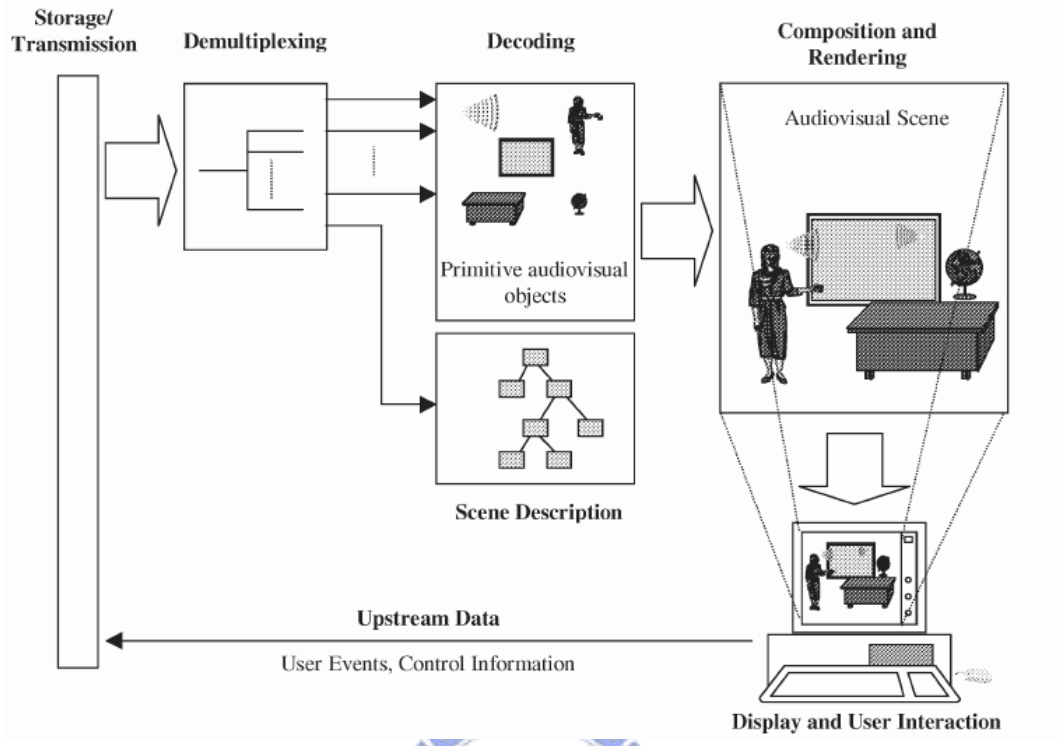


Figure 2.1: A high level view of an MPEG-4 terminal (from[5]).

2.1 Organization of the MPEG-4 Standard

The MPEG-4 standard addresses the generic coding of audio-visual objects, as illustrated in Figure 2.1. It (ISO/IEC 14496) consists of the following basic parts (The following description of the different parts are mainly taken from [1] and [2]).

1. ISO/IEC 14496-1: Systems

   The MPEG-4 Systems specification defines architecture and tools to create audio-visual scenes from individual objects. A major tool for MPEG-4 systems is scene description. The MPEG-4 scene description, a totally new component in the MPEG specifications, is based on VRML (virtual reality modeling language) and specifies the spatial-temporal composition of objects in a scene. The scene description is at the core of the systems specification, and allows easy creation of compelling audio-visual content.

2. ISO/IEC 14496-2: Visual

The MPEG-4 visual specification defines the main video codec. It consists of natural, arbitrary shape and synthetic video coding.

For natural video coding, the main video coding tools are still texture coding, similarly to MPEG-1 and MPEG-2. For intra coding, the MPEG-4 visual specification uses DCT, IDCT, intra prediction, quantization and de-quantization to reduce spatial redundancy. For inter coding, the MPEG-4 visual specification uses motion estimation and motion compensation to reduce temporal redundancy. In visual coding, the major difference from MPEG-1 and MPEG-2 is object coding. In MPEG-4, each picture is considered as consisting of objects, since some MPEG-4 functionalities require access not only to entire pictures but also to objects.

For synthetic video coding, in MPEG-4, mesh-based representation is useful. MPEG-4 includes a tool for triangular mesh-based representation of general objects.

3. ISO/IEC 14496-3: Audio

   ISO/IEC 14496-3 (MPEG-4 Audio) is a new kind of audio standard that integrates many different types of audio coding: natural sound with synthetic sound, low bitrate delivery with high-quality delivery, speech with music, complex sound tracks with simple ones, and traditional content with interactive and virtual-reality content. MPEG-4, unlike previous audio standards created by ISO/IEC and other groups, does not target at a single application such as real-time telephony or high-quality audio compression. MPEG-4 Audio is a rather generic standard that applies to applications requiring the use of advanced sound compression, synthesis, manipulation, or playback. The subparts specify state-of-the-art coding tools in several domains. However, MPEG-4 Audio is more than just the sum of its parts. As the tools described are integrated with the rest of the MPEG-4 standard, new possibilities for object-based audio coding, interactive presentation, dynamic sound tracks, and other sorts of new media, are enabled.

4. ISO/IEC 14496-4: Conformance Testing

   This part of ISO/IEC 14496 specifies how tests can be designed to verify whether bitstreams and decoders meet requirements specified in parts 1, 2, and 3 of ISO/IEC

14496. In this part of ISO/IEC 14496, encoders are not addressed specifically. An encoder may be said to be an ISO/IEC 14496 encoder if it generates bitstreams compliant with the syntactic and semantic bitstreams requirements specified in parts 1, 2 and 3 of ISO/IEC 14496.

5. ISO/IEC 14496-5: Reference Software

   Reference software is normative in the sense that any conforming implementation of the software, taking the same conforming bitstreams, using the same output file format, will output the same file. Complying ISO/IEC 14496 implementations are not expected to follow the algorithms or the programming techniques used by the reference software. Although the decoding software is considered normative, it cannot add anything to the technical description included in parts 1, 2, 3 and 6 of ISO/IEC 14496.

6. ISO/IEC 14496-6: DMIF

   DMIF, or Delivery Multi-media Integration Framework, is an interface between the application and the transport, which enables the MPEG-4 application developer to stop worrying about the transport. A single application can run on different transport layers when supported by the right DMIF instantiation.

   MPEG-4 DMIF supports the following functionalities:

   - A transparent MPEG-4 DMIF-application interface irrespective of whether the peer is a remote interactive peer, broadcast or local storage media.
   - Control of the establishment of FlexMux channels.
   - Use of homogeneous networks between interactive peers: IP, ATM, mobile, PSTN, Narrowband ISDN.
   - Support for mobile networks, developed together with ITU-T.
   - User commands with acknowledgment messages.
   - Management of MPEG-4 Sync Layer information.

2.2 MPEG-4 Video Coding Overview (from [3])

The target of MPEG-4 video is providing standardized core technologies allowing efficient storage, transmission and manipulation of video data in multimedia environments. It provides technologies to view, access and manipulate objects rather than pixels, with great error robustness at a large range of bit-rates. In order to achieve this broad goal, video activities in MPEG-4 aim at providing solutions in the form of tools and algorithms enabling functionalities such as efficient compression, object scalability, spatial and temporal scalability, error resilience, and fine granularity scalability. The standardized MPEG-4 video provides a toolbox containing tools and algorithms bringing solutions to the above mentioned functionalities and more.

2.2.1 Structure of MPEG-4 Video Data

An input video sequence can be defined as a sequence of related snapshots or pictures, separated in time. Many of MPEG-4 functionalities require access not only to entire sequence of pictures, but to an entire object, and further, not only to individual pictures, but also to temporal instances of these objects within a picture.

The concept of Video Objects (VOs) and their temporal instances, Video Object Planes (VOPs) is central to MPEG-4 video. A VOP can be fully described by a set of luminance and chrominance values and shape representation. In Figure 2.2, we show the decomposition of a picture into a number of separate VOPs.

Each VO is encoded separately and multiplexed to form a bitstream that users can access and manipulate. The encoder sends, together with VOs, information about scene composition to indicate where and when VOPs of a VO are to be displayed. Figure 2.3 shows the organization of coded MPEG-4 Video in a top-down hierarchical structure.

- VideoSession (VS): A Video session is the highest syntactic structure of the coded visual bitstream and simply consists of an ordered collection of video objects. The complete MPEG-4 scene which may contain any 2-D or 3-D natural or synthetic objects.

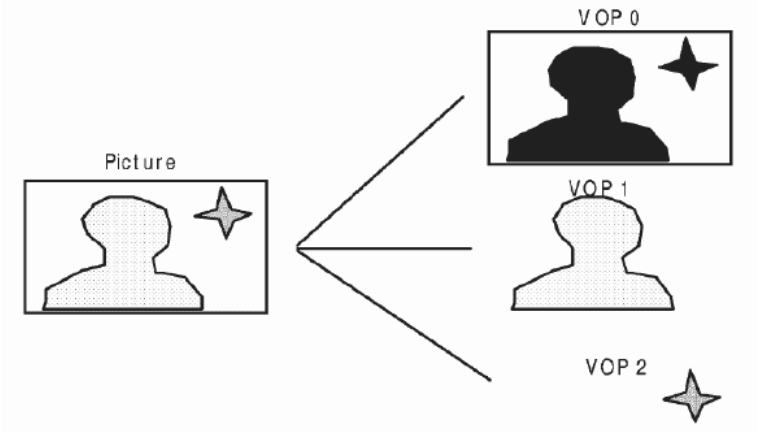


Figure 2.2: Segmentation of a picture in to VOPs (from [5]).

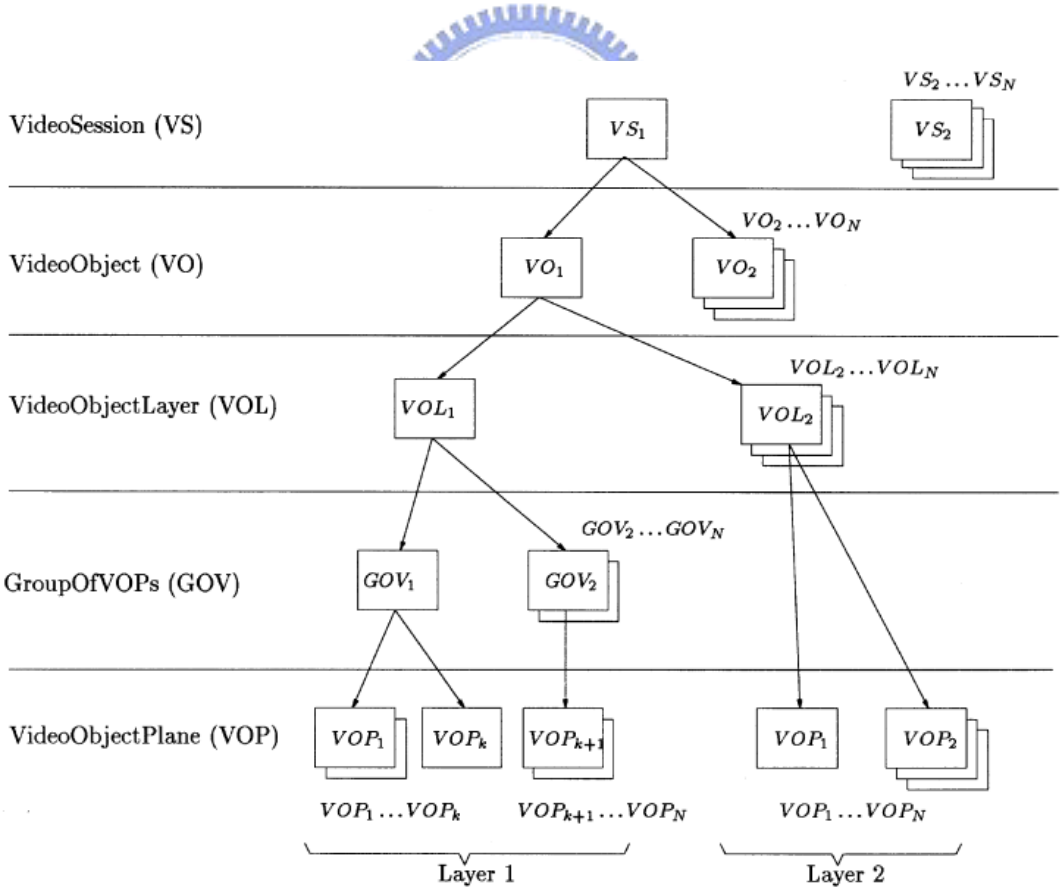


Figure 2.3: Logical structure of coded video data (from [7]).

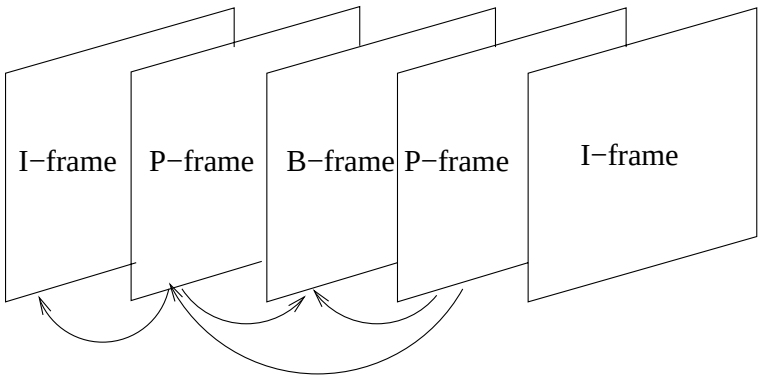


Figure 2.4: Types of VOP.

- VideoObject (VO): A Video object (2D + time) represents a complete scene or a portion of a scene with a semantic. In the simplest case this can be a rectangular frame, or it can be an arbitrarily shaped object corresponding to a physical object or background of the scene.

- VideoObjectLayer (VOL): Each video object can be encoded in scalable (multi-layer) or non-scalable form (single layer), depending on the application, represented by VOL. The VOL provides support for scalable coding. A video object can be encoded using spatial or temporal scalability, going from coarse to fine resolution.

- GroupOfVideoObjectPlanes (GOV): Group of video object planes are optional entities. The GOV groups together video object planes. GOVs can provide points in the bitstream where video object planes are encoded independently from each other, and can thus provide random access points into the bitstream.

- VideoObjectPlane (VOP): A VOP is a time sample of a video object. Figure 2.4 shows three of the four types of VOP that use different coding methods:

1. An Intra-coded (I) VOP is coded using information only from itself.

2. A Predictive-coded (P) VOP is a VOP which is coded using motion compensated prediction from a past reference VOP.

3. A Bidirectionally predictive-coded (B) VOP is a VOP which is coded using motion compensated prediction from a past and/or future reference VOP(s).

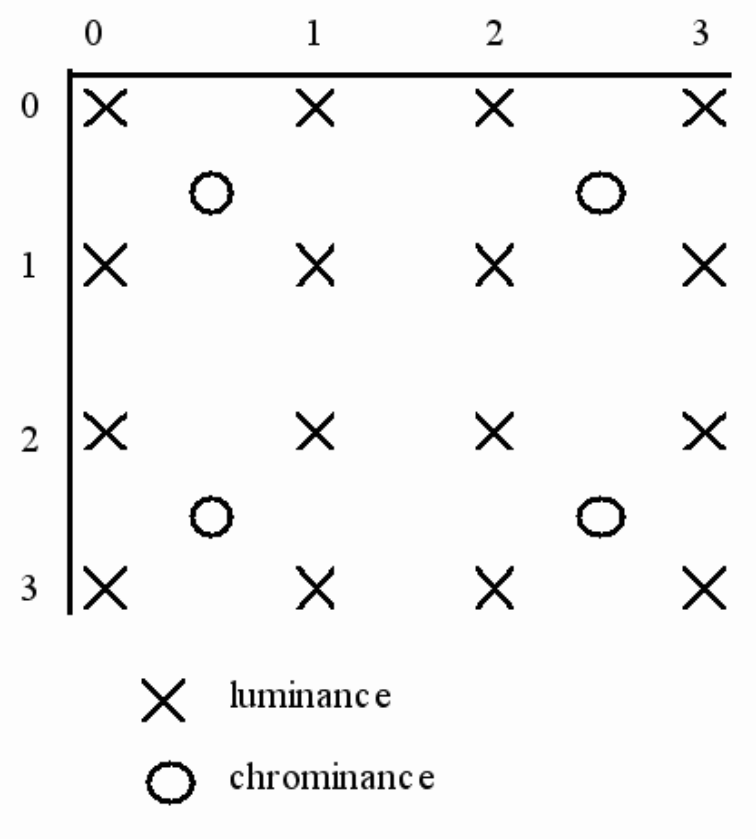


Figure 2.5: Positions of luminance and chrominance samples in 4:2:0 data (from [6])

4. A sprite (S) VOP is a VOP for a sprite object or a VOP which is coded using prediction based on global motion compensation from a past reference VOP.

The macroblock (MB) is a basic coding structure constructing VOP. In the MPEG-4 standard, a macroblock contains a section of the luminance component and the sub-sampled chrominance components in 4:2:0 format. In this format, there are 4 luminance blocks and 2 chrominance blocks in a macroblock. The luminance and chrominance samples are positioned as shown in Figure 2.5.

2.3 MPEG-4 Video Texture Coding (from [5], [6] and [7])

Figure 2.6 shows a high level logical structure of a VO based encoder. The main components are VO segmenter/formatter, VO encoders, system multiplexer/demultiplexer, VO decoders and VO compositor. We will introduce more details of VO encoders in this

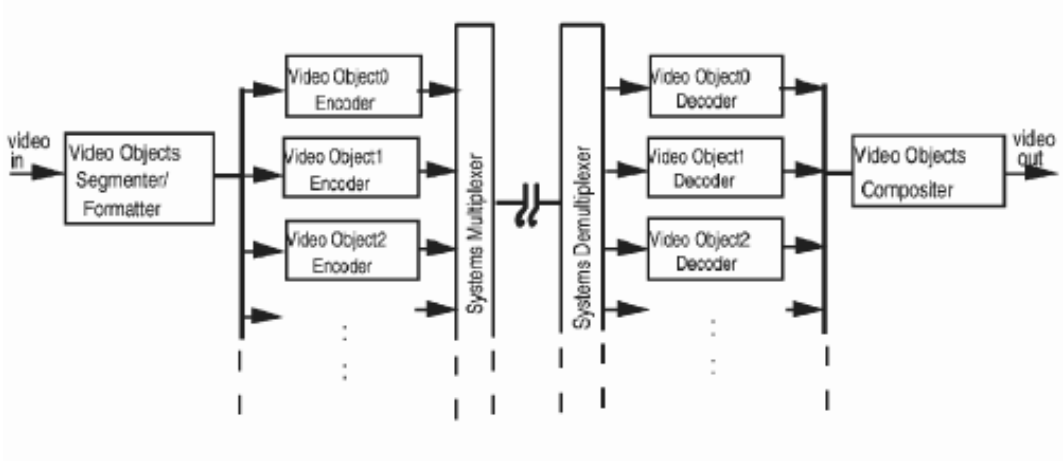


Figure 2.6: High level structure of VO based encoder (from [5]).

section.

Figure 2.7 presents the internal structure of the VO encoder. The same encoding scheme is applied in coding all the VOPs of a given session. The encoder has an entirely new component compared to previous video coding standards: arbitrary shape coding.

2.3.1 VOP Formation

After segmentation, the video object shape information is obtained. The shape information is hereafter referred to as alpha plane. There are two kinds of alpha plane. One is binary alpha plane which contains two kinds of data. The value 255 is assigned to pixels belonging to the objects and 0 is assigned to pixels outside the objects. The other one is grey scale alpha plane which is used for hybrid (of natural and synthetic) scenes generated by blue screen composition and is represented by an 8-bit component.

The alpha plane is used to form a VOP. For the binary alpha plane, a rectangular bounding box enclosing the shape to be coded is formed such that its horizontal and vertical dimensions are extended to multiples of 16 pixels (MB size). For efficient coding, it is important to minimize the number of macroblocks contained in the bounding box.

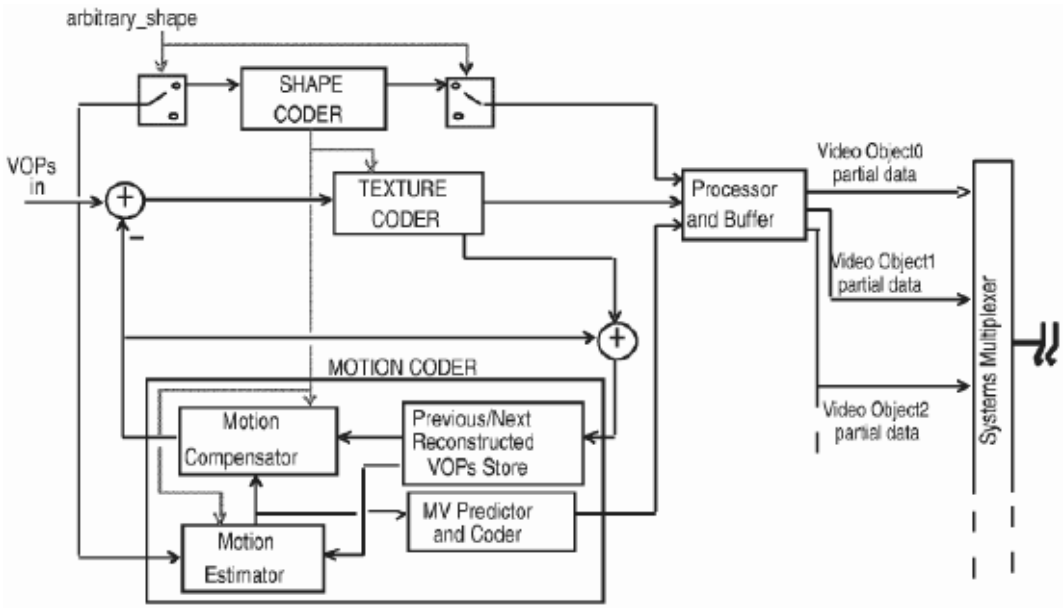


Figure 2.7: Detailed structure of VO encoder (from [5]).

2.3.2 Shape Coding

After VOP formation, the alpha plane of VOP will be coded prior to coding motion vector and texture based on the VOP image bounding box. Binary alpha planes are encoded by modified context-based arithmetic encoding (CAE) while grey scale alpha planes are encoded by motion compensated DCT similar to texture coding. An alpha plane is also bounded by an extended rectangular bounding box. The bounded alpha plane is partitioned into blocks of 16×16 samples (hereafter referred to as alpha blocks) and the encoding/decoding process is done per alpha block.

Binary Shape Coding

The basic tools for encoding binary alpha blocks (BABs) are CAE and motion compensation. InterCAE and IntraCAE are the variants of the CAE algorithm used with and without motion compensation, respectively. Motion vectors can be computed by searching for a best match position. The motion vectors themselves are differentially coded. Every BAB can be coded in one of the following modes:

1. The block is all transparent. In this case no coding is necessary. Texture information

is not coded for such blocks either.

2. The block is all opaque. Again, shape coding is not necessary for such blocks, but texture information needs to be coded (since they belong to the VOP).

3. The block is coded using IntraCAE without use of past information.

4. Motion vector difference (MVD) is zero but the block is not updated.

5. MVD is non-zero, but the block is not updated.

6. MVD is zero and the block is updated. InerCAE is used for coding the block update.

7. MVD is non-zero, and the block is coded by InterCAE.

If the encoder need rate control and rate reduction, the encoder realizes these through size-conversion of binary alpha information. The estimation of conversion ratio (CR) is iterative and consists of using the same factor in both dimensions and determining the acceptability of resulting shape quality. To be specific, a 4:1 downsampled binary alpha block is used first and if the shape errors are higher than acceptable, a 2:1 downsampled binary alpha block is used next, again if it is found unacceptable, an unsubsampled binary alpha block is used. Figure 2.8 shows the block diagram of CR determination. The selection is done based on the conversion error between the original BAB and the BAB which is once down-sampled and then reconstructed by up-sampling. The conversion error is computed for each 4$times$4 sub-block respectively by taking the sum of the absolute difference. If the sum is greater than a designated threshold value, this sub-block is called "Error-PB (Pixel Block)"

CAE encoding is used to code each binary pixel of the BAB. Prior to coding the first pixel, the arithmetic encoder is initialized. Each binary pixel is then encoded in raster order. The process for encoding a given pixel is the following:

1. Compute a context number.

2. Index a probability table using the context number.

3. Use the indexed probability to drive an arithmetic encoder.

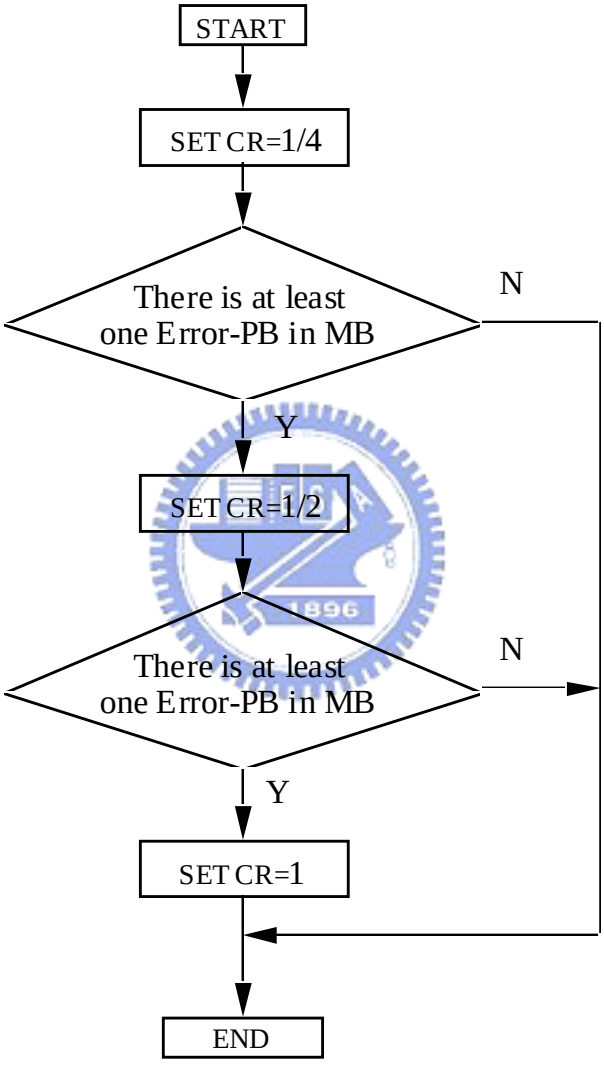


Figure 2.8: CR determination algorithm (from [6]).

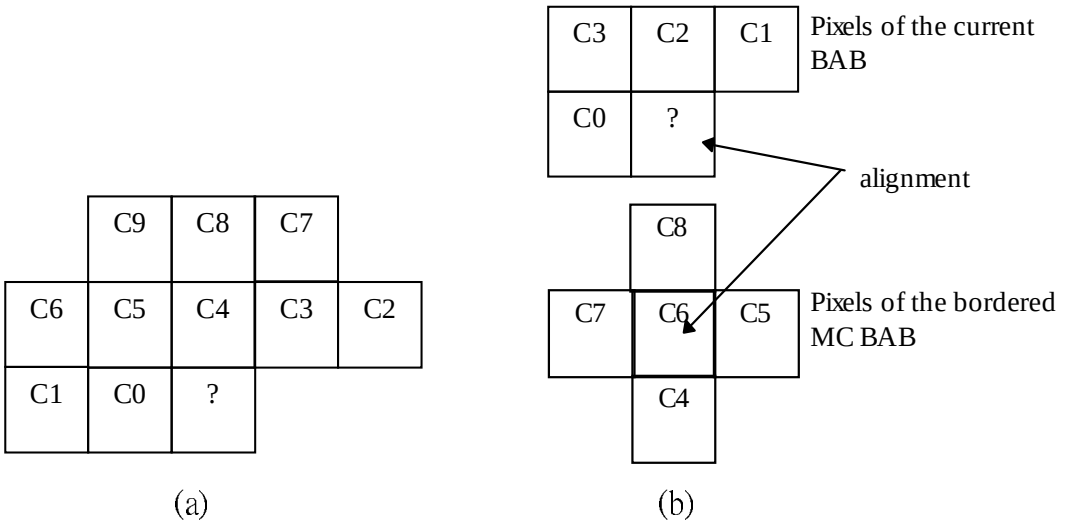


Figure 2.9: Pixel templates used for (a) INTRA and (b) INTER context determination of BAB. The pixel to be coded is marked with "?".

When the final pixel has been processed, the arithmetic code is terminated. Figure 2.9 shows the computation of the contexts for INTRA and INTER modes.

Gray Scale Shape Coding

The gray scale shape information has a structure similar to that of binary shape with the difference that every pixel can take on a range of values (usually 0 to 255) representing the degree of the transparency of that pixel. The gray scale shape corresponds to the notion of alpha plane used in computer graphics, in which 0 corresponds to a completely transparent pixel and 255 to a completely opaque pixel. Intermediate values of the pixel correspond to intermediate degrees of transparencies of that pixel.

Gray level alpha plane is encoded as its support function and the alpha values on the support. The support is obtained by thresholding the gray level alpha plane by 0. The support function is encoded by binary shape coding as described previously and the alpha values are encoded using a block based motion compensated DCT similar to that of texture coding. Figure 2.10 shows the block diagram of gray shape coding.

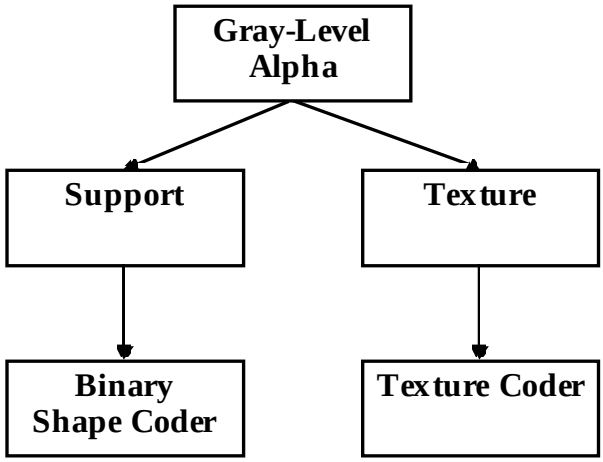


Figure 2.10: Gray shape coding (from [6]).

2.3.3 Motion Coder

There are four types of VOPs (see Figure 2.4 and associated discussion) that use different coding methods. Motion coding is necessary only for P-VOP and B-VOP to reduce temporal redundancy. The motion coder consists of a motion estimator, motion compensator, previous/next VOPs store and motion vector (MV) predictor and coder. In order to perform motion prediction on a per VOP basis, the motion estimation of the blocks on the VOP borders has to be modified from block matching to polygon matching. Furthermore, a special padding technique is required for the reference VOP.

Padding Process

The padding process defines the values of luminance and chrominance samples outside the VOP for prediction of arbitrarily shaped objects. Figure 2.11 shows a simplified diagram of this process.

A decoded MB $d[y][x]$ is padded by referring to the corresponding decoded shape block $s[y][x]$. A MB that lies on the VOP boundary is padded by replicating the boundary samples of the VOP towards the exterior. This process is divided into horizontal repetitive padding and vertical repetitive padding. The remaining MBs that are completely outside the VOP are filled by extended padding.

- Horizontal repetitive padding: Each sample at the boundary of a VOP is replicated

horizontally to the left and/or right direction in order to fill the transparent region outside the VOP of a boundary macroblock. If there are two boundary sample values for filling a sample outside of a VOP, the two boundary samples are averaged.

- Vertical repetitive padding: The remaining unfilled transparent samples from above procedure are padded by a similar process as the horizontal repetitive padding but in the vertical direction. The samples already filled in the horizontal repetitive padding are treated as if they were inside the VOP for the purpose of this vertical pass.

- Extended padding: Exterior MBs immediately next to boundary macroblocks are filled by replicating the samples at the border of the boundary macroblocks. Note that the boundary macroblocks have been completely padded in horizontal and vertical repetitive padding. If an exterior macroblock is next to more than one boundary macroblocks, one of the macroblocks is chosen, according to the priority shown as Figure 2.12. The exterior macroblock is then padded by replicating upwards, downwards, leftwards, or rightwards the row of samples from the horizontal or vertical border of the boundary macroblock having the largest priority number. The remaining exterior macroblocks (not located next to any boundary macroblocks) are filled with 128.

Motion Estimation

Motion estimation (ME) is a method of prediction between adjacent frames/pictures. This technique falls into two categories, pixel-based algorithms and block-based algorithms (BMA). The motion estimation method used in MPEG-4 encoder is block-based.

In general, the ME techniques used in MPEG-4 can be seen as an extension of standard MPEG-1/2 or H.263 block matching techniques with modified block (polygon) matching.

Figure 2.13 illustrates an example for polygon matching. The bounding rectangle of the VOP is first extended on the right-bottom side to multiples of macroblock size. Zero stuffing is used for these extended pixels. The alpha value of the extended pixels is set to zero. The MBs are formed by dividing the extended bounding rectangles into 16×16 blocks. SAD is used as error measure. The original alpha plane for the VOP is used to

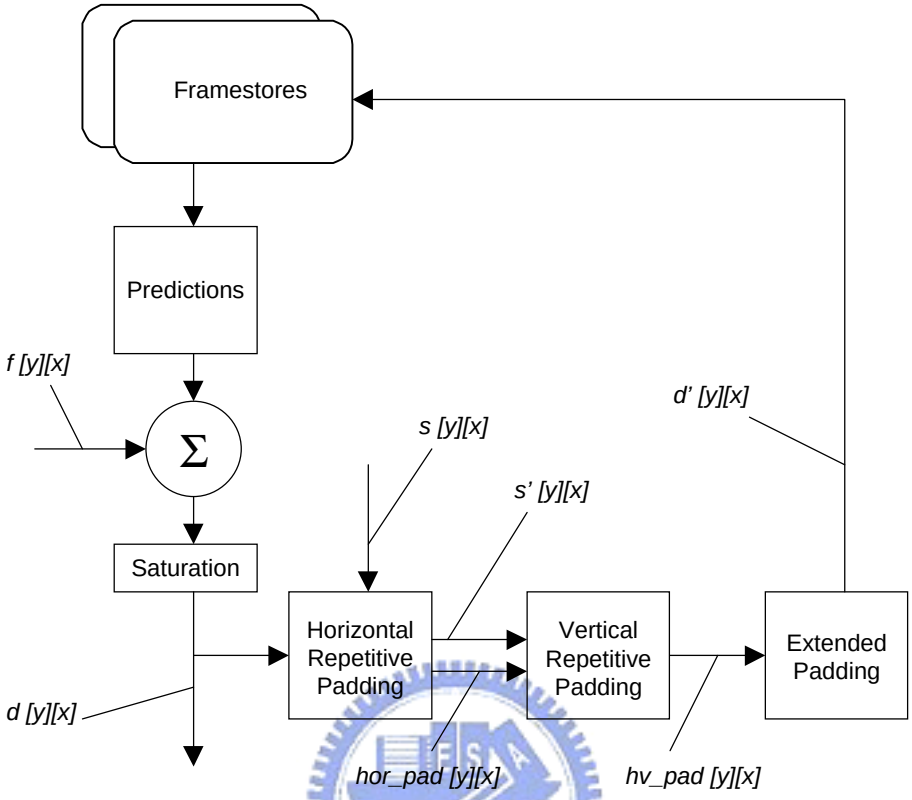


Figure 2.11: Padding process (from [6]).

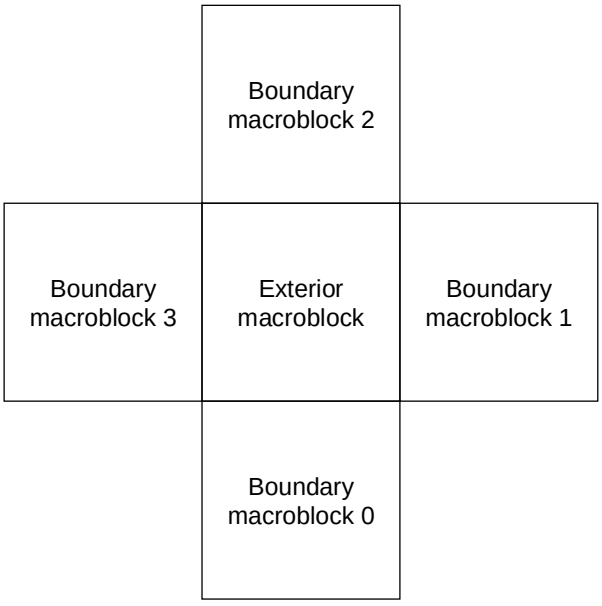


Figure 2.12: Priority of boundary MBs surrounding an exterior MB (from [6]).

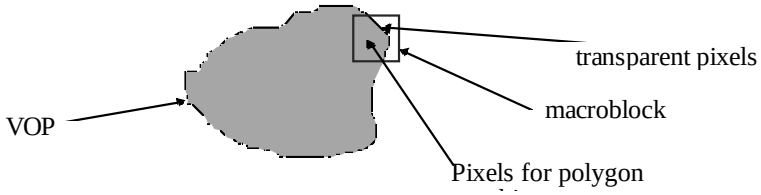


Figure 2.13: Polygon matching for an arbitrary shape VOP (from [6]).

exclude the pixels of the MB that are outside the VOP. SAD is computed only for the pixels with nonzero alpha value. This forms a polygon for the MB that includes the VOP boundary.

The reference VOP is padded based on its own shape information. For example, when the reference VOP is smaller than the current VOP, the reference is not padded up to the size of the current VOP.

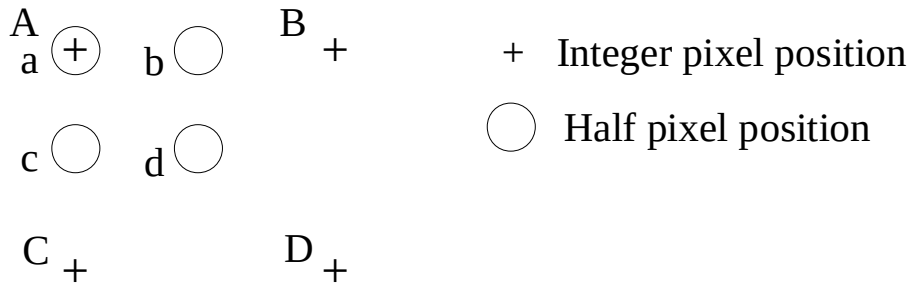
The basic motion estimation is performed on 16×16 luminance MB. The motion vector is specified to half-pixel accuracy. In many coding software implementations, the motion estimation is performed by full search to integer pixel accuracy vector and, using it as the initial estimate, a half pixel search is performed around it.

In the MPEG-4 standard, besides motion vector for 16×16 MB, motion vector can be sent for individual 8×8 blocks to reduce more prediction errors. Both the 8×8 block motion compensation and overlapped motion compensated prediction are referred to as advanced prediction in H.263 and are adapted in MPEG-4 to work with arbitrary shaped VOPs.

Because the motion vector may be non-integer number, sample interpolation is necessary. The process for interpolation of half sample values is carried out only in half sample mode, where the half sample values are calculated by bilinear interpolation as depicted in Figure 2.14. Using interpolation, the half-pixel motion vector can be calculated.

Motion Vector Encoder

When using INTER mode coding, the motion vector must be coded. Horizontal and vertical motion vector are coded differentially by using a spatial neighborhood of three motion vectors already coded (see Figure 2.15). These three motion vectors are candidate pre-



a = A,
b = (A + B + 1 - rounding_control) / 2
c = (A + C + 1 - rounding_control) / 2,
d = (A + B + C + D + 2 - rounding_control) / 4

Figure 2.14: Interpolation scheme for half sample search.

dictors for the differential coding. The differential coding of motion vectors is performed with reference to the reconstructed shape. In the special cases at the borders of the current VOP the following decision rules are applied:

1. If the MB of one and only one candidate predictor is outside the VOP, it is set to zero.

2. If the MBs of two and only two candidate predictors are outside the VOP, they are set to the third candidate predictor.

3. If the MBs of all three candidate predictors are outside the VOP, they are set to zero.

The motion vector coding is performed separately on the horizontal and vertical components. For each component, the median value of the three candidates for the same component is used as predictor, denoted Px and Py, respectively:

$$Px = Median(MV1x, MV2x, MV3x),$$

$$Py = Median(MV1y, MV2y, MV3y).$$

After finding the predictors, the vector differences $MVDx(= MVx - Px)$ and $MVDy(= MVy - Py)$ are coded by variable length coding.

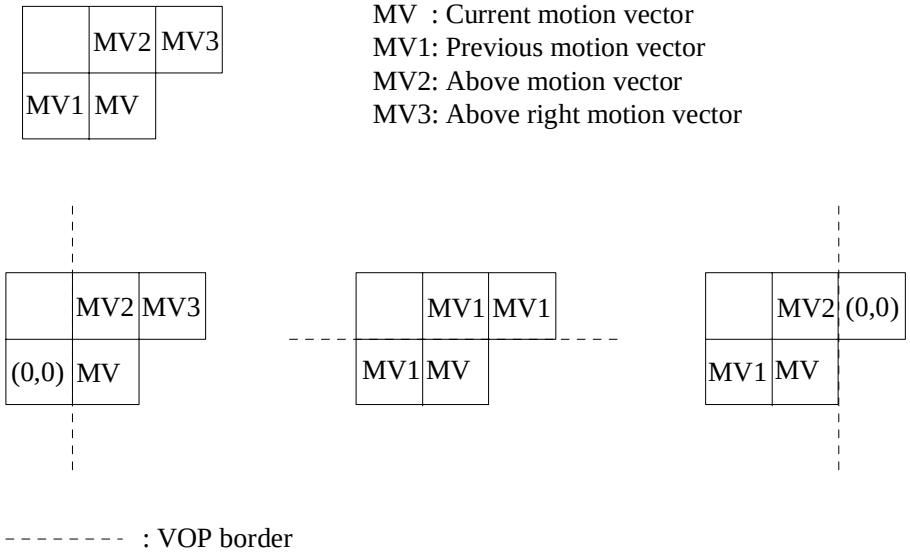


Figure 2.15: Motion vector prediction (from [6]).

Motion Compensation

The motion compensator uses motion vectors to compute motion compensated prediction block, $pred[i][j]$ from the same reference VOP. In addition to basic motion compensation processing, three alternalties are supported, namely, unrestricted motion compensation, four MV motion compensation and overlapped motion compensation.

For unrestricted motion compensation, the motion vectors are allowed to point outside the decoded area of a reference VOP. For an arbitrary shape VOP, the decoded area refers to the area within the bounding box, padded as described above. When a sample referenced by a motion vector is outside the decoded VOP area, an edge sample is used. The $pred[i][j]$ is defined as follows:

$$xref = \min(\max(xcurr + dx, vhmcsr), xdim + vhmcsr - 1),$$

$$yref = \min(\max(ycurr + dy, vvmcsr), ydim + vvmcsr - 1),$$

where $vhmcsr =$ vop_horizontal_mc_spatial_ref, $vvmcsr =$ vop_vertical_mc_spatial_ref, $(ycurr, xcurr)$ are the coordinates of a sample in the current VOP, $(yref, xref)$ are the coordinates of a sample in the reference VOP, (dy, dx) is the motion vector, and $(ydim, xdim)$ are the dimensions of the bounding rectangle of the reference VOP.

One/two/four vectors decision is indicated by the MCBPC codeword and field_prediction flag for each macroblock. If one motion vector is transmitted for a certain macroblock, this is defined as four vectors with the same value as the MV. When two field motion vectors are transmitted, each of the four block prediction motion vectors has the value equal to the average of the field motion vectors (rounded such that all fractional pixel offsets become half pixel offsets). If MCBPC indicates that four motion vectors are transmitted for the current macroblock, the information for the first motion vector is transmitted as the codeword MVD and the information for the three additional motion vectors is transmitted as the codewords MVD2–4. If four vectors are used, each of the motion vectors is used for all pixels in one of the four luminance blocks in the macroblock.

Overlapped motion compensation is performed when the flag obmc_disable = 0. Each pixel in an 8×8 luminance prediction block is a weighted sum of three prediction values, divided by 8. The creation of each pixel $\overline{P}(i,j)$, in an 8×8 luminance prediction block is governed by the following equation:

$$\overline{P}(i,j) = \frac{(p(i+MV_x^0,j+MV_y^0)*H_0(i,j)+p(i+MV_x^1,j+MV_y^1)*H_1(i,j)+p(i+MV_x^2,j+MV_y^2)*H_2(i,j)+4)}{8},$$

where (MV_x^0, MV_y^0) denotes the motion vector for the current block, (MV_x^1, MV_y^1) denotes the motion vector of the block either above or below, (MV_x^2, MV_y^2) denotes the motion vector either to the left or right of the current block, and $H_0(i,j)$, $H_1(i,j)$, and $H_2(i,j)$ denote the weighting of each pixel in the current block and neighbor blocks.

Since the VOP may be coded in P or B mode, there are three types of motion vectors, forward mode, backward mode, and bi-directional mode. The different modes make different predictions $\overline{P}(i,j)$.

1. Forward mode

   Only the forward vector (MVFx,MVFy) is applied in this mode. The prediction blocks $\overline{P}_y(i,j), \overline{P}_u(i,j), \overline{P}_v(i,j)$ are generated from the forward reference VOP.

2. Backward mode

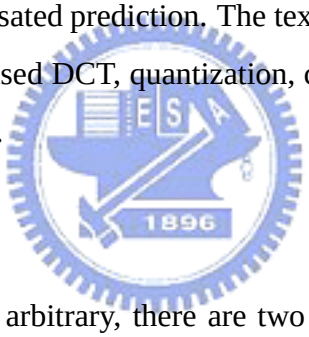
   Only the Backward vector (MVBx,MVBy) is applied in this mode. The prediction blocks $\overline{P}_y(i,j), \overline{P}_u(i,j), \overline{P}_v(i,j)$ are generated from the backward reference VOP.

3. Bi-directional mode

   Both the forward vector (MVFx,MVFy) and the backward vector (MVBx,MVBy) are applied in this mode. The prediction blocks $\overline{P}_y(i,j), \overline{P}_u(i,j), \overline{P}_v(i,j)$ are generated from the forward and backward reference VOPs by doing the forward prediction, the backward prediction and then averaging both predictions pixel by pixel.

2.3.4 Texture Coder

The texture information of a video object plane is present in the luminance Y and two chrominance components Cb and Cr of the video signal. In the case of an I-VOP, the texture information resides directly in the luminance and chrominance components. In the case of motion compensated VOPs the texture information represents the residual error remaining after motion-compensated prediction. The texture coder includes padding process (if needed), 8×8 block based DCT, quantization, coefficient prediction, coefficient scan and variable length coding.



Padding Process

When the shape of the VOP is arbitrary, there are two types of MBs that belong to an arbitrarily shaped VOP:

1. Those that lie completely inside the VOP shape.

2. Those that lie on the boundary of the shape.

The macroblocks that lie completely inside the VOP are coded using a technique identical to the technique used in H.263. The macroblocks that lie on the boundary of the shape need to be padded before texture coding. For residual error blocks after motion compensation, the region outside the VOP within the blocks are padded with zero. For intra blocks, the padding is performed in a three-step procedure called low pass extrapolation (LPE). This procedure is as follows:

1. Compute the arithmetic mean vale m of the pixels $f(i,j)$ in the blocks that belong to the VOP as

$$m = (1/N) \sum_{(i,j) \in VOP} f(i,j),$$

where N is the number of pixels situated with the VOP. Division by N is done by rounding to the nearest integer.

2. Assign m to each block pixel situated outside of the VOP region, that is,

$$f(i,j) = m \text{ for all (i,j)} \notin VOP.$$

3. Apply the following filtering operation to each block pixel $f(i,j)$ outside of the VOP region, in raster-scan order:

$$f(i,j) = [f(i,j-1) + f(i-1,j) + f(i,j+1) + f(i+1,j)]/4.$$

Division is done by rounding to the nearest integer. If one or more of the four pixels used for filtering are outside the block, the corresponding pixels are not included into the filtering operation and the divisor 4 is reduced accordingly. For example, for $i = 0$ and $j = 0$, we have

$$f(i,j) = [f(i,j+1) + f(i+1,j)]/2.$$

After this padding operation the resulting block is ready for DCT coding.

Discrete Cosine Transform Coding

Similar to MPEG-1 and MPEG-2, the 2D (8×8) DCT is used for spatial data compression in MPEG-4 inter and intra coding. The encoder dose forward transform before quantization and inverse transform after inverse quantization in the loop. The reason for inverse quantization and inverse transform is to obtain reconstructed image for the next temporal frame.

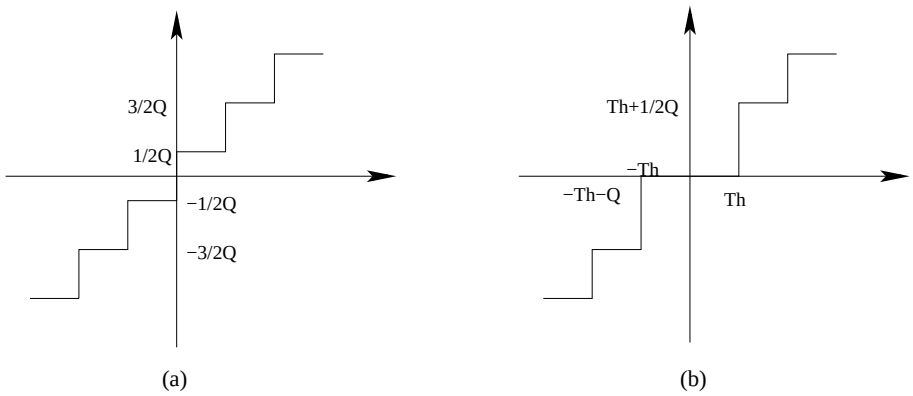


Figure 2.16: Quantizers in H.263. (a) For intra DC coefficient only. (b) For inter DC and all AC coefficients.

Table 2.1: Default Quantization Matrix Q (from [3])

(intra)							
8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

(non intra)							
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16

Table 2.2: Nonlinear Scaler for DC Coefficients of DCT Blocks (from[3])

component	**DC scaler for Quantizer (Q) range**			
	1–4	5–8	9–24	25–31
Luminance	8	2Q	Q+8	2Q+16
Chrominance	8	$\frac{Q+13}{2}$		Q+16

Quantization

MPEG-4 video supports two techniques of quantization (Q), one referred to as the H.263 quantization method and the other, the MPEG quantization method. The H.263 quantization method is with dead zone for intra and inter AC coefficients and with no dead zone for intra DC coefficients. The MPEG quantization method is uniform quantizer with the default matrix.

Figure 2.16 shows the quantizer characteristics in H.263. It has uniform quantization for intra DC coefficients and nearly uniform midtread quantization for the inter DC and all AC coefficients. For AC data, input between $-$Th and $+$Th is quantized to zero. All coefficients in a macroblock go through the same quantizer. The step size Q can be changed in increments of 2 from 2 to 62 depending on rate controller.

In the MPEG quantizer, each coefficient produced by 2D DCT is quantized with a uniform quantizer. The default quantizer matrix is defined as shown in Table 2.1. The default quantizer matrix can be changed by the rate controller if the required channel bandwidth is unavailable.

Typically, the DC coefficients of DCT of blocks belonging to an intra macroblock are scaled by a constant scaling factor of 8. However, in MPEG-4 video, a nonlinear scaler as shown in Table 2.2 is used to provide a higher coding efficiency. The characteristics of nonlinear scaling are different between the luminance and chrominance blocks and further depend on the quantizer used for the block.

Intra Prediction

After quantization, the DC coefficients and many AC coefficients of an intra block are coded by intra prediction. Intra prediction is a new operation used in MPEG-4 standards to reduce the spatial redundancy between 8×8 blocks. There are two types of prediction, DC prediction and AC prediction.

Figure 2.17 shows the prediction of DC coefficients in intra 8×8 blocks. The quantized intra coefficients are predicted with three previous decoded DC coefficients. For example, the DC coefficients of block X is predicted from the DC coefficients of blocks A, B and C. Unlike MPEG-2, the method of prediction in MPEG-4 standards is gradient

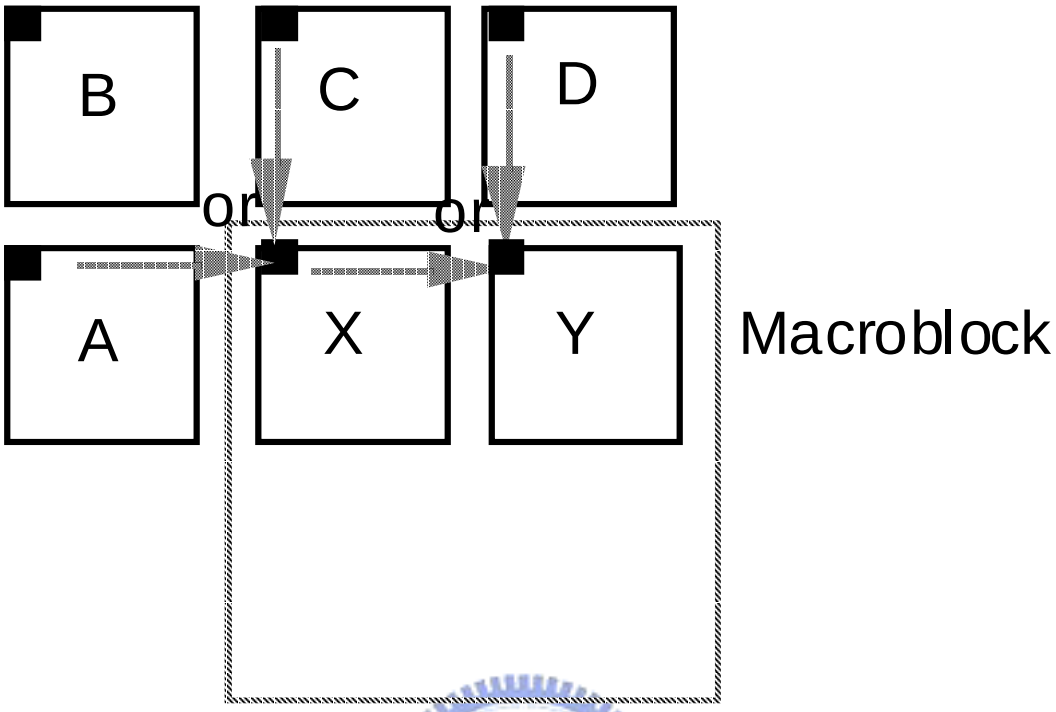


Figure 2.17: Prediction of DC coefficients of blocks in an intra MB (from[5]).

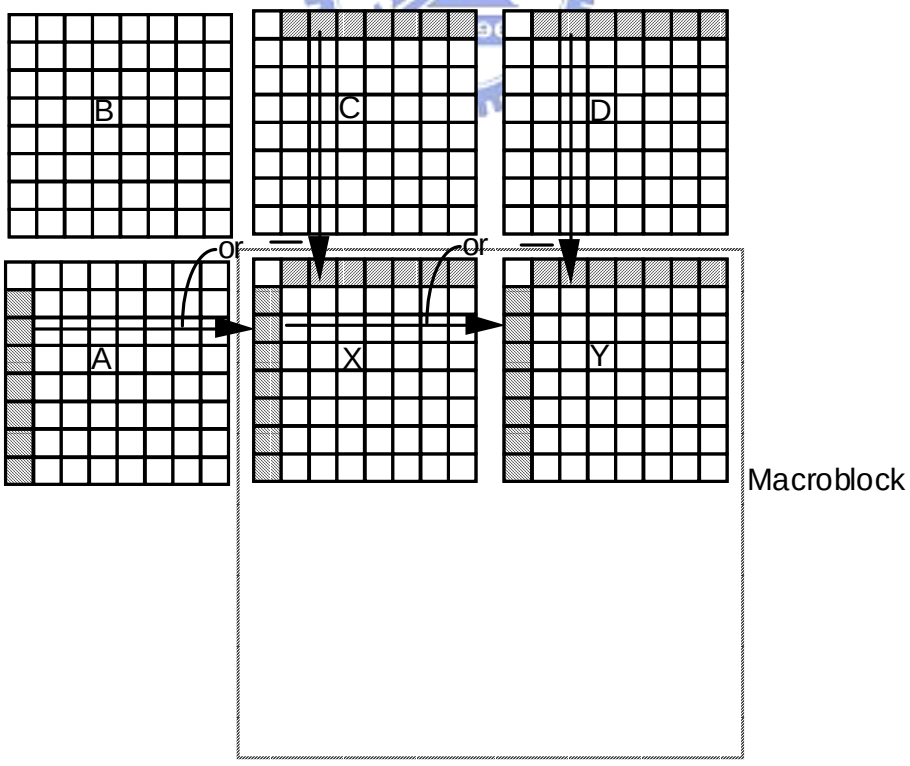


Figure 2.18: Prediction of AC coefficients of blocks in an intra MB (from[5]).

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

zigzag

0	1	2	3	10	11	12	13
4	5	8	9	17	16	15	14
6	7	19	18	26	27	28	29
20	21	24	25	30	31	32	33
22	23	34	35	42	43	44	45
36	37	40	41	46	47	48	49
38	39	50	51	56	57	58	59
52	53	54	55	60	61	62	63

alternate-horizontal

0	4	6	20	22	36	38	52
1	5	7	21	23	37	39	53
2	8	19	24	34	40	50	54
3	9	18	25	35	41	51	55
10	17	26	30	42	46	56	60
11	16	27	31	43	47	57	61
12	15	28	32	44	48	58	62
13	14	29	33	45	49	59	63

alternate-vertical

Figure 2.19: Scans for 8×8 blocks (from[3]).

based. In computing the prediction of block X, if the absolute value of a horizontal gradient is less than the absolute value of a vertical gradient, then the QDC of block C is used as the prediction, else QDC value of block A is used.

The AC prediction depends on DC prediction, as shown in Figure 2.18. The AC coefficients in the first row or in the first column are predicted with three previous decoded AC coefficients. The direction of prediction is the same as DC prediction.

Scan and VLC

The predicted DC and AC coefficients (as well as the un-predicted AC coefficients) of DCT blocks are scanned by one of three scans: alternate-horizontal, alternate-vertical and zigzag (normal scan used in H.263 and MPEG-1) to change the 2D image to one dimensional data, see Figure 2.19. The actual scan used depends on the coefficient predictions used. For instance, if the DC prediction refers to the horizontally adjacent block, alternate-vertical scan is selected for the current block. If the DC prediction refer to the vertically adjacent block, alternate-horizontal scan is used for the current block. For all

other blocks, the 8×8 blocks of transform coefficients are zigzag scanned.

The coefficients after scan usually become data with many zeros at the end. This kind of a data stream is good for run-length coding. In the MPEG-4 standard, differential DC coefficients in intra blocks are encoded in variable length codes. However, the AC coefficients are encoded by the variable length codes for EVENTs. An EVENT is a combination of a last non-zero coefficient indication, the number of successive zeros preceding the coded coefficient (RUN), and the non-zero value of the coded coefficient (LEVEL). Some statistically rare events have no variable length codes to represent them. For them an escape coding method is used.

2.4 Other Video Coding Tools (from[5]) and Profiles and Levels (from[3])

2.4.1 Other Video Coding Tools

In addition to texture video coding, there are some special tools defined in MPEG-4. In this section, we shortly introduce robust video coding and scalable coding.

Robust Video Coding

Since the MPEG-4 standard supports the ability to access audio or video data over a diverse range, especially over wireless networks, error resilience is necessary. In the error resilient mode, the MPEG-4 video offers a number of tools as follows:

1. Object priorities

   The object based organization of MPEG-4 video potentially makes it easier to achieve a higher degree of error robustness due to the possibility of prioritizing each semantic object based on its relevance.

   Further, VOP types lend themselves to a form of automatic prioritization since, B-VOPs are noncausal and do not contribute to error propagation and thus can be assigned a lower priority and perhaps even be discarded in case of severe errors.

2. Resynchronization

   It is possible for an encoder to offer increased error resilience by placing resynchronization (resync) markers in the bitstreams with approximately constant spacing, such as beginning of each MB.

3. Data partitioning

   Data partitioning provides a mechanism to increase error resilience by separating the normal motion and texture data of all macroblocks in a video packet and send all of the motion data followed by a motion marker, followed by all of the texture data.

4. Reversible VLCs

   The reversible VLCs offer a mechanism for a decoder to recover additional texture data in the presence of errors since the special design of reversible VLCs enables decoding of codewords in both the forward (normal) and the reverse direction.

5. Intra update and scalable coding

   To prevent error propagation, intra update is a simple method to reduce this problem. However, more intra reduces less coding efficiency. Another method is scalable coding, which can prevent error propagation without more intra coding.

Scalable Coding

The scalability tools in MPEG-4 Video are designed to support applications beyond that supported by single layer video. The applications of scalability include internet video, wireless video, multi-quality video services, video database browsing, etc. In scalable video coding, it is assumed that given a coded bitstream, decoders of various complexities can decode and display appropriate reproductions of coded video. MPEG-4 Video provides several different forms of scalabilities that address non-overlapping applications with corresponding complexities.

The basic scalability tools offered are temporal scalability and spatial scalability. The Fine Granularity Scalability (FGS) which supports continuous scalability of bit rate and

video quality is also defined.

2.4.2 Profiles and Levels

Although there are many tools in the MPEG-4 standard, not every MPEG-4 decoder will have to implement all of them. Similar to MPEG-2, profiles and levels are defined as subsets of the entire bitstreams syntax of all the tools. The purpose of defining conformance points in the form of profiles and levels is to facilitate interchange of bitstreams among different applications. There are eight profiles defined by MPEG-4: simple, core, main, simple scalable, animated & mesh, basic animated texture, still scalable texture profile and simple face. The detailed definitions are given in Table 2.4.

Compared with the previous standards, the simple profile of MPEG-4 is similar to the coding method in H.263. The difference is that the simple profile has error resilience but does not have B-frame coding. The simple scalable profile is the same as simple profile, but with the rectangular scalability added. The core profile is the profile with all tools of the simple profile, temporal scalability, B-VOP coding and binary shape coding. The main profile is the profile with all tools in core profile, gray shape coding, interlace and sprite coding. The other profiles are for particular purposes, such as 2D dynamic mesh coding and facial animation coding.

Table 2.3: Profiles and Tools (from[3])

Visual Tools	Simple	Core	Main	Simple Scalable	Animated 2D Mesh	Basic Animated Texture	Still Scalable Texture	Simple Face Face
Basic *1. I VOP* *2. P VOP* *3. AC/DC Prediction* *4. 4MV Unrestricted MV*	V	V	V	V	V			
Error resilience *1. Slice Resynchronization* *2. Data Partitioning* *3. Reversible VLC*	V	V	V	V	V			
Sort Header	V	V	V		V			
B-VOP		V	V	V	V			
Method 1/Method 2 quantization		V	V		V			
P-VOP based temporal scalability *1. Rectangular* *2. Arbitrary Shape*		V	V		V			
Binary Shape		V	V		V			
Grey Shape			V					
Interlace			V					
Sprite			V					
Temporal Scalability (Rectangular)				V				
Spatial Scalability (Rectangular)				V				
Scalable Still Texture					V	V	V	
2D Dynamic Mesh with uniform topology					V	V		
2D Dynamic Mesh with Delaunay topology					V			
Facial Animation Parameters								V

Chapter 3

Intel's MMX Technology and Tools for Software Optimization

As discussed previously, our goal is to achieve real-time implementation of MPEG-4 video encoder using Intel's MMX technology. In this chapter, we will introduce Intel's MMX technology including features of MMX, instruction set of MMX and extensions of MMX termed SSE and SSE2. We also introduce the software tools we use to help development.

3.1 Intel's MMX Technology (from [8], [9] and [10])

The multimedia extensions (MMX) for the Intel Architecture (IA) were designed to enhance performance of advanced media and communication applications. The MMX technology introduces new general-purpose instructions. These instructions operate in parallel on multiple data elements packed into 64-bit quantities. These instructions accelerate the performance of applications with compute-intensive algorithms that perform localized, recurring operations on small native data. This includes applications such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, video conferencing, 2D graphics, and 3D graphics.

The MMX technology uses the single instruction, multiple data (SIMD) technique. This technique speeds up software performance by processing multiple data elements in

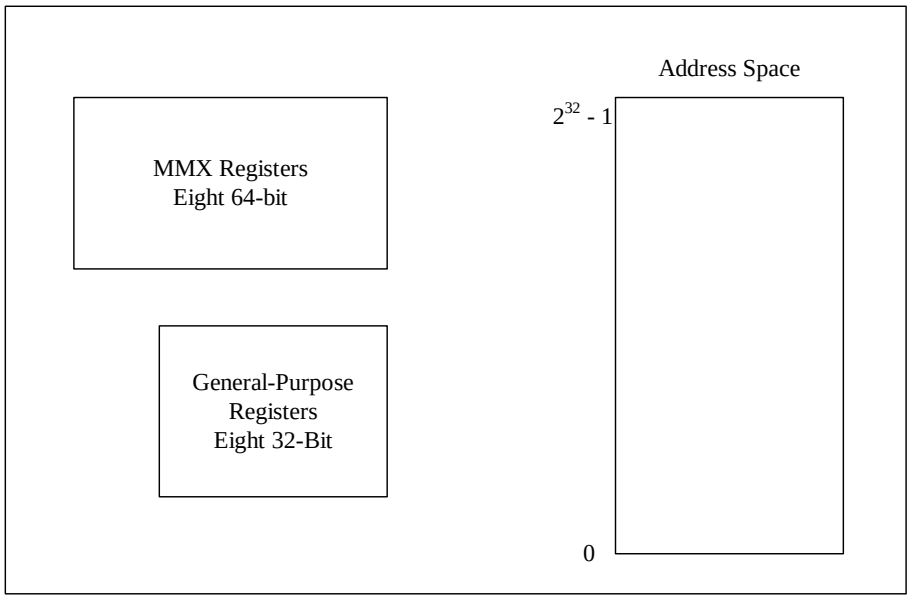


Figure 3.1: MMX execution environment.

parallel, using a single instruction. The MMX technology supports parallel operations on byte, word, and doubleword data elements, and the new quadword (64-bit) integer data type.

3.1.1 MMX Technology Overview

The MMX technology defines a simple and flexible SIMD execution model to handle 64-bit packed integer data. This model adds the following new features to the IA: New data types, MMX registers and enhanced instruction set. All MMX instructions operate on MMX registers, the general-purpose registers, and/or memory as shown in Figure 3.1.

- MMX registers: These MMX registers are used to perform operations on 64-bit packed integer data.
- General-purpose registers: The eight general-purpose registers are used along with the existing IA-32 addressing mode to address operands in memory.

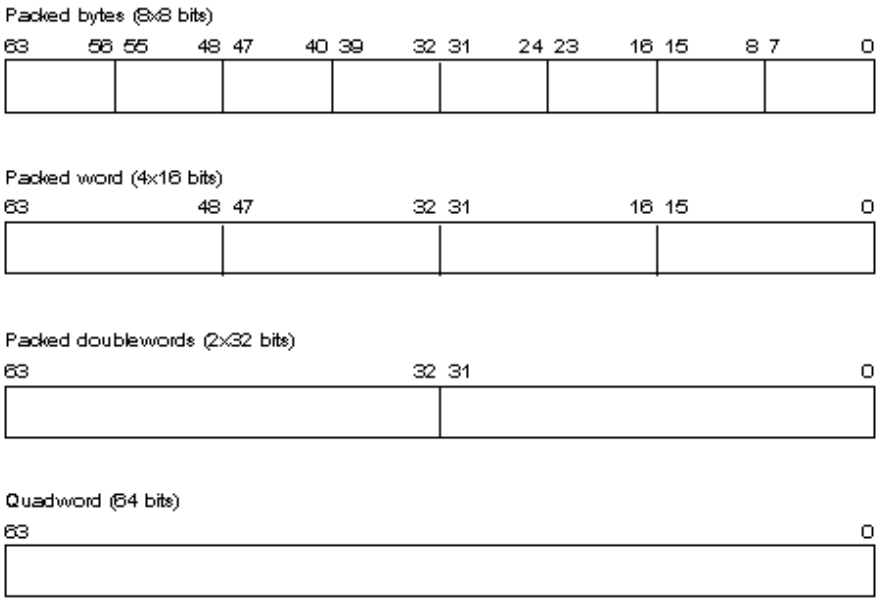


Figure 3.2: MMX packed data types (from [8]).

MMX Data Types

The MMX technology introduced the following four new 64-bit data types as illustrated in Figure 3.2:

- Packed byte: 8 bytes packed into one 64-bits quantity.
- Packed word: 4 words packed into one 64-bits quantity.
- Packed doubleword: 2 doubleword packed into one 64-bits quantity.
- Packed quadword: One 64-bits quantity.

The 64 bits are numbered 0 through 63. Bit 0 is the least significant bit (LSB), and bit 63 is the most significant bit (MSB). The low-order bits are the lower part of the data element and the high-order bits are the upper part of the data element. Bytes in a multi-byte format have consecutive memory addresses. The ordering is little endian. That is, the bytes with lower addresses are less significant than the bytes with higher addresses.

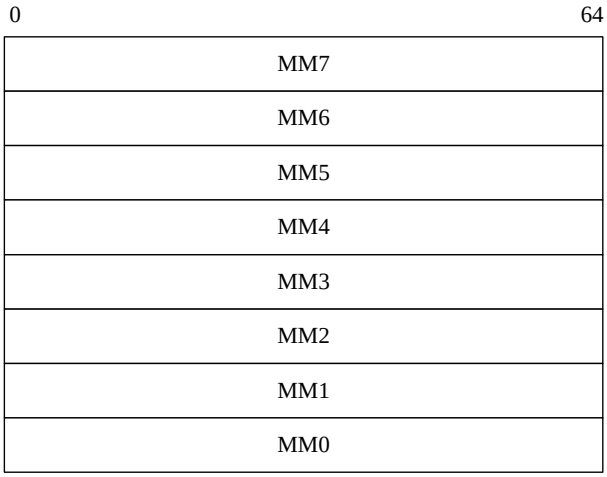


Figure 3.3: MMX register set.

MMX Registers

The MMX register set consists of eight 64-bit registers as shown in Figure 3.3, which are used to perform calculations on the MMX packed data but cannot be used to address memory. Values in MMX registers have the same format as a 64-bit quantity in memory. These registers are aliased to the floating-point registers. The MMX instructions access the MMX registers directly using the register names MM0 to MM7.

Enhanced Instruction Set

The MMX instruction set supplies a set of instructions that operate in parallel on all data elements of a packed data type. The MMX instructions implement two principles: operation on packed data and saturation arithmetic.

- Operations on packed data: The MMX uses the SIMD technique for performing arithmetic and logic operations on bytes, words or doublewords packed into MMX registers as shown in Figure 3.4.
- Saturation arithmetic: When performing integer arithmetic, an operation may result in an out-of-range condition, where the true result cannot be represented in the destination format. The MMX technology provide three ways to handle out-of-range conditions.

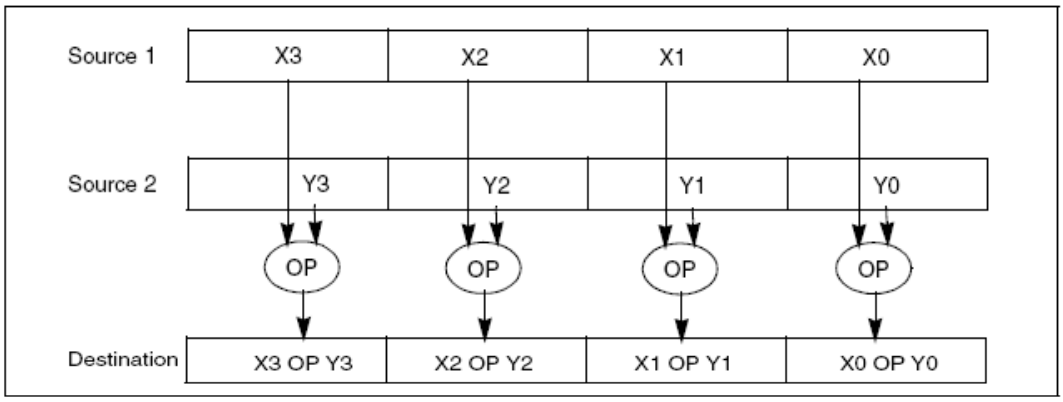


Figure 3.4: SIMD execution model (form [9]).

1. Wraparound arithmetic. With wraparound arithmetic, an out-of-range value is truncated. That is, the carry or overflow bit is ignored and only the least significant bits of the result are return to the destination. Wraparound arithmetic is suitable for applications that control the range of operands to prevent out-of-range results in the end. If the range of operands is not controlled, wraparound arithmetic can lead to large errors.
2. Signed saturation arithmetic. With singed arithmetic, out-of-range values are limited to the representable range of signed integers for the integer size being operated on.
3. Unsigned saturation arithmetic. With unsinged arithmetic, out-of-range values are limited to the representable range of unsigned integers for the integer size being operated on.

3.1.2 MMX Instruction Sets Introduction

This section provides an overview of MMX instruction groups. Detailed information on instructions, can be found in [10]. The MMX instructions are grouped into the following categories:

- Data transfer
- Arithmetic

- Comparison
- Conversion
- Unpacking
- Logical
- Shift
- Empty MMX state instruction (EMMS)

Table 3.1 gives a summary of the instructions in the MMX instruction set.

Data Transfer Instructions

We can transfer 32-bit or 64-bit data from memory to MMX registers and visa versa, or from integer registers to MMX registers and visa versa by a single instruction. We can transfer 32-bit data by MOVD and 64-bit data by MOVQ.

Arithmetic

The arithmetic instructions perform addition, subtraction, multiplication, and multiply-add operation on packed data types. For example, PADDB, PADDSB and PADDUSB instructions add signed or unsigned packed byte integers in wraparound mode, signed packed byte integers in signed saturation mode, unsigned packed byte integers in unsigned saturation mode, respectively.

Comparison Instructions

The comparison instructions compare the packed data in the source and destination operands for equal to or greater than. These instructions generate a mask of ones or zeros which are written to the destination operand.

Table 3.1: MMX Instruction Set Summary

Category	**Wraparound**	**Signed Saturation**	**Usinged Saturation**
	32-bit Transfers		**64-bit Transfers**
Data Transfer			
Register to Register	MOVD		MOVQ
Load from Memory	MOVD		MOVQ
Store to Memory	MOVD		MOVQ
Arithmetic			
Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB PADDUSW
Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
Multiplication	PMULL, PMULH		
Multiply and Add	PMADD		
Comparison			
Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion			
Pack		PACKSSWB, PACKSSDW	PACKUSWB
Unpack			
Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		
	Packed		**Full 64-bit**
Logocal			
And			PAND
And Not			PANDN
Or			POR
Exclusive OR			PXOP
Shift			
Shift Left Logical	PSLLW, PSLLD		PSLLQ
Shift Right Logical	PSRLW, PSRLD		PSRLQ
Shift Right Arithmetic	PSRAW, PSRAD		
Empty MMX State	EMMX		

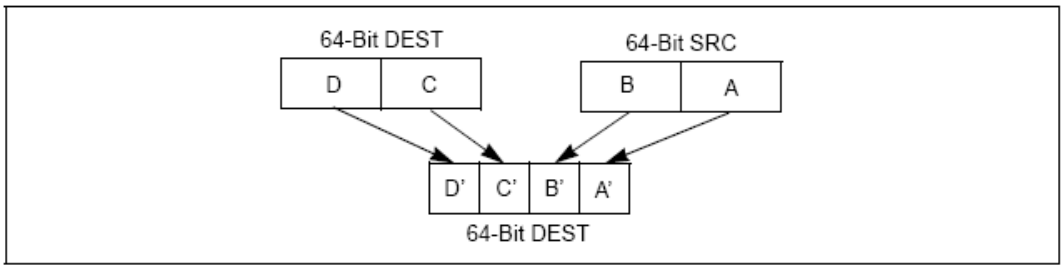


Figure 3.5: PACKSSDW instruction operation using 64-bit operands (form [10]).

Conversion Instructions

The conversion instructions perform conversions between the packed data types. For example, PACKSSDW instruction converts packed signed doubleword integers into packed signed word integers, using saturation to handle overflow conditions as shown in Figure 3.5 for an example of the packing operation.

Unpack Instructions

The unpack instructions unpack bytes, words, or doublewords from the high- or low-order elements of the source and destination operands and interleave them in destination operand. By placing all 0s in the source operand, these instruction can be used to convert byte integers to word integers, word integers to doubleword integers, or doubleword integers to quadword integers. For example, The PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands as shown in Figure 3.6

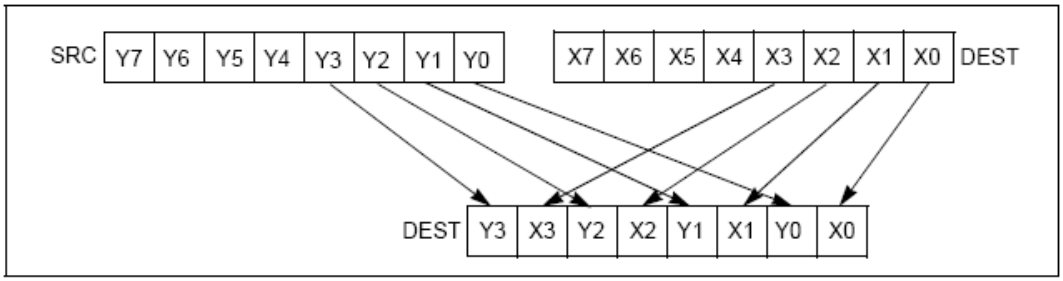


Figure 3.6: PUNPCKLBW instruction operation using 64-bit operands (from [10])

Logical Instructions

The logical instructions perform bitwise logical operations on 64-bit quantities. For example, we can generate a zero register in MM0 by using "PXOR mm0, mm0."

Shift Instructions

The shift instructions have two types: logical shift and arithmetic shift. Logical shift instructions perform a logical left or right shift of the data elements and fill the empty high or low order bit position with zeros. Arithmetic shift instructions perform an arithmetic right shift, copying the sign bit for each data elements into empty bit positions on the upper end of each data elements.

EMMS Instructions

The EMMS instruction empties the MMX state. This instruction must be used to clear the MMX state at the end of an MMX routine before calling other routines that can execute floating-point instructions.

3.1.3 SSE and SSE2, Later Extensions of MMX Technology

The streaming SIMD extensions (SSE) were introduced into IA-32 architecture in the Pentium III processor family and the stream SIMD extensions 2 (SSE2) were introduced into IA-32 architecture in the Pentium 4 and Intel Xeon processor.

Overview of SSE Extensions

The SSE extensions extend the SIMD execution model, by adding facilities for handling packed or scalar single-precision floating-point values contained in 128-bit registers. The SSE extension add the following features to the IA-32 architecture.

- Eight 128-bit data registers, call the XMM registers named by XMM0 to XMM7.
- The 32-bit MXCSR register, which provides control and status bits for operations performed on the XMM registers.

- The 128-bit packed single-precision floating-point data (four IEEE single-precision floating-point values packed into a double quadword).
- Instructions that perform SIMD operation on single-precision floating-point values and that extend the SIMD operations that can be performed on integers:
  - 128-bit packed and scalar single-precision floating-point instructions that operate on operands located in XMM registers.
  - 64-bit SIMD integer instructions that support additional operations on packed integer operands located in the MMX registers.
- Instructions that save and restore the state of MXCSR register.
- Instruction that support explicit prefetching of data, control of the cacheability of data, and control the ordering of store operations.
- Extensions to the CPUID instruction.

SSE Programming Environment

Figure 3.7 shows the execution environment for the SSE extensions. All SSE instructions operate on the XMM registers and/or memory as follows:

- XMM registers: These eight registers are used to operate on packed or scalar single-precision floating-point data. The scalar operations are performed on individual single-precision floating-point values stored in low doubleword of an XMM register.
- MXCSR register: This 32-bit register provides status and control bits used in SIMD floating-point operations.
- MMX registers: This portion is the same as MMX.
- General-purpose registers: This portion is the same as MMX.
- EFLAGS register: This 32-bit register is used to record results of some compare operations.

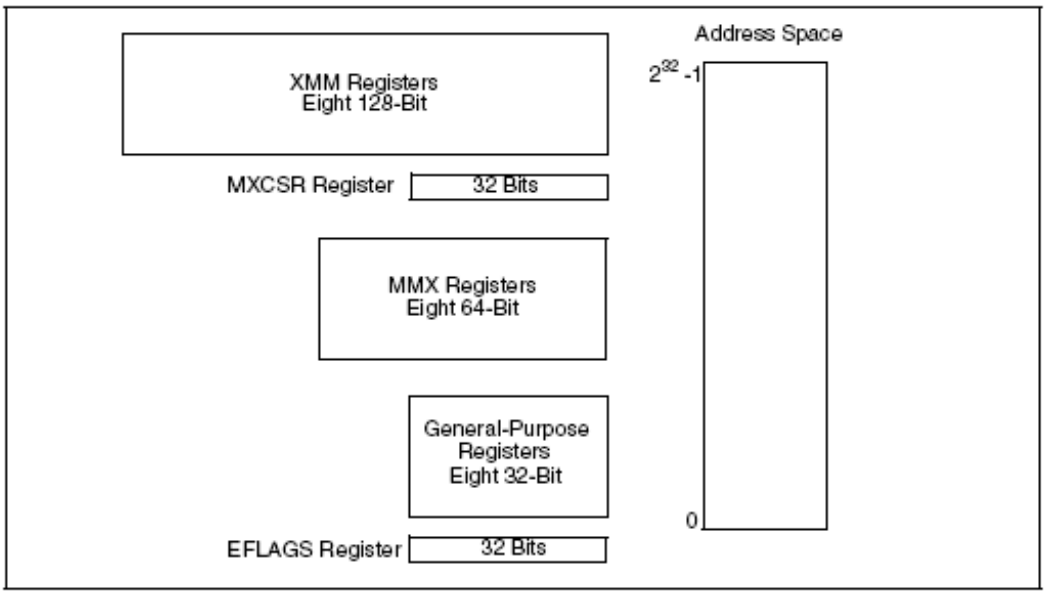


Figure 3.7: SSE execution environment (from [9]).

SSE Instruction Set

The SSE instructions are divided into four functional groups

- Packed and scalar single-precision floating instructions.
- 64-bit SIMD integer instructions
- State management instructions
- Cacheablility control, prefetch, and memory ordering instructions.

The instructions we used are 64-bit SIMD integer instructions for example, PSADBW. Detailed information on SSE instructions can be found in [9]

Overview of SSE2 Extensions

The SSE2 extensions use the same SIMD execution model that is used with the MMX technology and SSE extensions. The SSE2 extensions add the following features to the IA-32 architecture.

- Five data types:

  - 128-bit packed double-precision floating-point (two IEEE Standard 754 double-precision floating-point values packed into a double quadword).
  - 128-bit packed byte integers.
  - 128-bit packed word integers.
  - 128-bit packed doubleword integers.
  - 128-bit packed quadword integers.
  - Instructions that support explicit prefetching of data, control of the cacheability of data, and control the ordering of store operations.

- Instructions to support the additional data type and extend existing SIMD integer operations:
  - Packed and scalar double-precision floating-point instructions.
  - Additional 64-bit and 128-bit SIMD integer instructions.
  - 128-bit versions of SIMD integer instructions introduced with MMX technology and the SSE extensions.
  - Additional cacheability-control and instruction-ordering instructions.

The SSE2 program environment is same as SSE and no new registers are defined with the SSE2 extensions.

SSE2 Instruction Set

The SSE2 instructions are divided into four functional qroups

- Packed and scalar double-precision floating instructions.
- 64-bit SIMD and 128-bit SIMD integer instructions
- 128-bit extensions of SIMD integer instructions introduced with the MMX technology and the SSE extensions
- Cacheablility-control and instruction-ordering instructions.

The instructions we used are 128-bit SIMD integer instructions. All of the 64-bit SIMD integer instructions introduced with the MMX technology and the SSE extensions have been extended with the SSE2 extensions to operate on 128-bit packed integer operands located in the XMM registers. For example, where the 64-bit version of PADDB instruction operates on 8 packed bytes, the 128-bit version has been extended to operate on 16 packed bytes. Detailed information on SSE2 instructions can be found in [9]

3.2 Software Tools for Implementation

In this section, we introduce some tools that help software development. The first is Intel C++ compiler. The compiler is used to compile C and C++ code for Intel IA-32 and Itanium-based systems running Microsoft operating systems. The second is the "VTune Performance Analyzer," which can help to analyze the performance of applications by locating hotspots. Hotspots are areas in code that take a long time to execute.

3.2.1 Intel C++ Compiler (from [11])

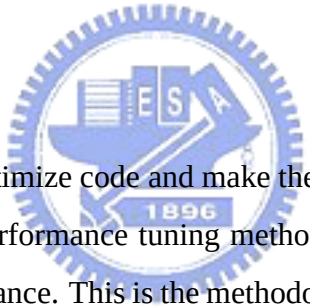
The Intel C++ compiler optimizes performance for applications running on Intel architecture-based computers. The features and benefits of Intel C++ compiler are summarized in Table 3.2. This compiler has minimum system requirements including hardware and software. In order to use this compiler correctly we suggest to read the Release Note first.

3.2.2 Intel VTune (from [12])

Intel VTune Analyzer helps to locate and remove software performance bottlenecks by collecting, analyzing, and displaying performance data from the system-wide level down to the source level. The VTune Analyzer provide multiple profiling technologies that enable optimization across multiple operating system platforms and development environments and support the latest Intel processors.

Table 3.2: Features and Benefits of Intel C++ Compiler (from [11])

Features	Benefits
High Performance	Achieve a significant performance gain by using optimizations
Support for Streaming SIMD Extensions	Advantage of new Intel microarchitecture
Automatic vectorizer	Advantage of parallelism in your code achieved automatically
OpenMP Support	Shared memory parallel programming
Floating-point optimizations	Improved floating-point performance
Data prefetching	Improved performance due to the accelerated data delivery
Interprocedural optimizations	Achieve a significant performance gain by optimizing between modules
Whole program optimization	Improved performance between modules in larger applications
Profile-guided optimization	Improved performance based on profiling frequently-used functions
Processor dispatch	Taking advantage of the latest Intel architecture features while maintaining object code compatibility with previous generations of Intel PentiumR processors



Performance Tuning

Performance tuning helps to optimize code and make the maximum use of the latest Intel architecture. Figure 3.8 is a performance tuning methodology for analyzing and tuning application and system performance. This is the methodology generally recommended by performance analysts at Intel.

Data Collectors

To optimize the performance of our application or system, we can do one or more of the following to find the performance bottlenecks:

- Determine how our system resources, such as memory and processor, are being utilized to identify system-level bottlenecks.
- Measure the execution time for each module and function in our application.
- Determine how the various modules running on our system affect the performance of each other.

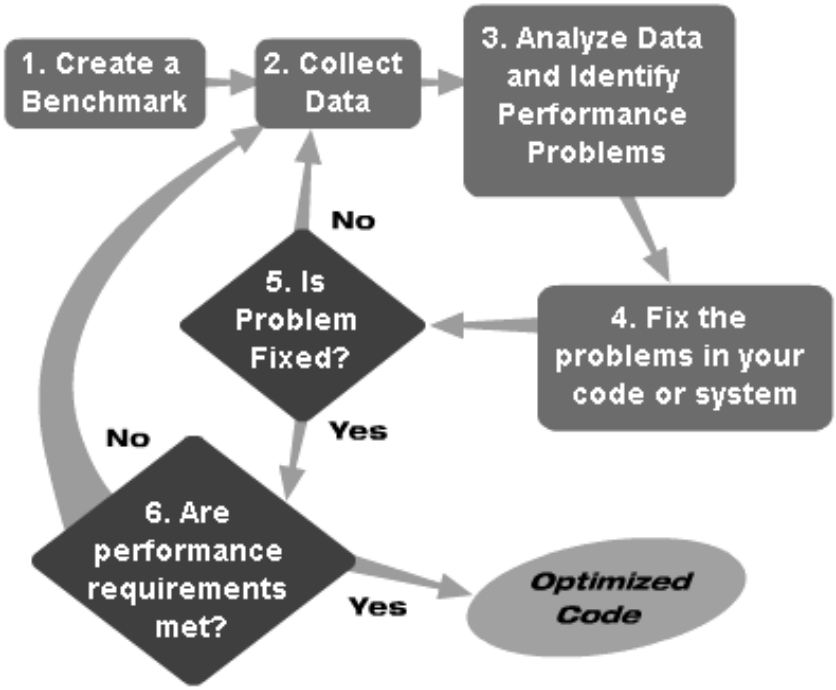


Figure 3.8: Performance tuning methodology (from [12]).

- Identify the most time-consuming function calls and call sequences within our application.
- Determine how our application is executing at the processor level to identify microarchitecture-level performance problems.

The VTune Analyzer can find the above information by automating the process of data collection with three types of data collectors, namely, sampling, call graph, and counter monitor. We need to use the right collector or use a combination of collectors based on their characteristics to collect the type of data that will help address the problem we are trying to isolate. Table 3.3 shows the characteristics of each collector. Knowing the characteristics will help us make the right decision when choosing collectors for performance tuning.

Table 3.3: Functional Units and Operations Performed (from [12])

Characteristics	**Sampling**	**Call Graph**	**Counter Monitor**
Intrusiveness	Non-intrusive.	Intrusive	Non-intrusive
System-wide performance data	Provides system-wide software performance data.	Only application-specific data.	Provides system-wide hardware and software performance counter data.
Parent and child function relationship	Does not determine the relationships.	Determine the relationships and critical calls and call sequences	Does not associate counter data with a specific application or code.
Program flow	Does not determine program flow but provides a statistical analysis of the data collected on an application.	Determines program flow of an application and displays the data function calls and critical call sequences in graphs and charts.	Does not determine program flow since data is not associated with any application.
System and microarchitecture-level performance data	Monitors processor events, such as Cache Misses, to help identify microarchitecture-level performance problems associated with specific sections of your code.	Does not monitor microarchitecture-level performance problems.	Monitors hardware and software performance counters over a specified duration to help identify system and microarchitecture-level performance data.

Chapter 4

MPEG-4 Video Encoder Optimization by Intel MMX Technology

We now discuss our optimization of the the video encoder using Intel MMX technology. Our implementation employs the public source Microsoft MPEG-4 Visual Reference Software (version: Microsoft-FDAM1-2.4-021205) as the basis, and we optimize both frame-based and shape-based video encoding. And our development system is based on Intel Pentium 4 CPU 2.66G, 480MB RAM and Microsoft Windows XP Professional Version 2002, Service Pack 1. In section 4.1, we introduce the Microsoft MPEG-4 Visual Reference Software. In section 4.2, we discuss our approaches to code acceleration. And lastly in section 4.3, we give a conclusion.

4.1 Introduction to Microsoft MPEG-4 Visual Reference Software

The Microsoft MPEG-4 Video Reference Software is a public source for encoding and decoding video sequence using the MPEG-4 compression format. The C++ code of this reference software is provided in three executables. Theses are encoder.exe, decoder.exe and converpar.exe. The convertpar.exe is a utility program for upgrading from old to new parameter files.

Table 4.1: Source Files and Directories Arrangement of MPEG-4 Video Reference Software

Encoder	
\app\encoder\encoder.dsp	Encoder project file
\app\encoder\encoder.cpp	Encoder main()
\sys	Common files
\sys\encoder	Encoder specific
\tools	
\type	Common types
\vtc	Wavelet code
Decoder	
\app\decoder\decoder.dsp	Decoder project file
\app\decoder\decoder.cpp	Decoder main()
\sys	Common files
\sys\decoder	Decoder specific
\tools	
\type	Common types
\vtc	Wavelet code
Paremeter File Conversion Utility	
\app\convertpar\convertpar.dsp	Convertpar project file
\app\convertpar\convertpar.cpp	Convertpar main()

The source files and directories are arranged as Table 4.1. and Table 4.2 indicates which tools are supported in this software. The funtionalities defined by this reference software conforms to main and simple scalable profiles of MPEG-4. Not all the functionalities of MPEG-4 are present, only natural video is covered. System layer functionality and 3D/SNHC parts are not included. We try to optimize this software without affecting the original functionality.

Table 4.2: Funtionalities of Microsoft MPEG-4 Video Reference Software

Tool	Version	Comments
Basic (I-VOP, P-VOP, AC/DC Prediction, 4MV, Unresticted MV)	1	Supported
B-VOP	1	Supported. No MPEG rate control.
P-VOP with OBMC	1	Supported
Method 1, Method 2 Quantisation	1	Supported
Error Resilience	1	Syntax only.No recovery from error supported.
Short Header (H.263 emulation)	1	Decode only.
Binary Shape (progressive)	1	Supported. No automatic VOP generation.
Grayscale Shape	1	Supported
Interlace	1	Supported
N-Bit	1	Supported
Sprite	1	Supported. No warping parameter estimation.
Still Texture	1	Supported
Dynamic Resolution Conversion	2	Supported
NEWPRED	2	Upstream signaling is simulated not implemented.
Global Motion Compensation	2	Supported
Quarter-pel Motion Compensation	2	Supported
SA-DCT	2	Supported
Error Resilience for Still Texture Coding	2	Supported
Wavelet Tiling	2	Supported
Object Based Spatial Scalability (Base)	2	Supported
Object Based Spatial Scalability (Enhancement)	2	Supported
Multiple Auxiliary Components	2	Supported
Complexity Estimation Support	2	Bitstream syntax supported only.

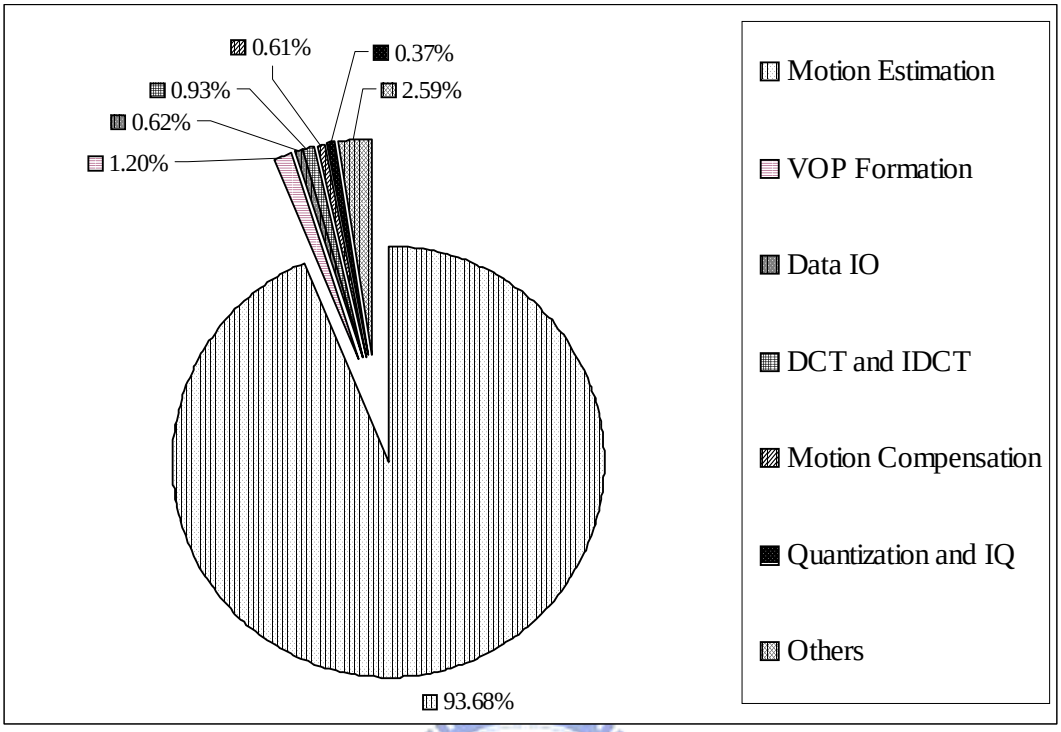


Figure 4.1: Breakdown of execution time in Microsoft MPEG-4 Visual Reference Software.

4.2 Code Acceleration

Figure 4.1 shows breakdown of complexity (execution time) of different functions in the Microsoft MPEG-4 Visual Reference Software. The test sequence is CIF Foreman with the encoding sequence "I B B P I B B P" and we use a binary mask for object based coding. As we can see, the functions motion estimation, VOP formation, DCT, IDCT, motion compensation, quantization and inverse quantization (IQ) occupy most of the execution time. In order to accelerate the encoder, we have to optimize these functions.

We explore methods to accelerate the encoder. These methods fall into two categories. One is to use MMX technology to modify the most computation-intensive kernel operations of the encoder while the other is at algorithm level. The first category attempts to enhance the parallelism by using suitable MMX instructions, while the second modifies the video coding algorithm for decreasing complexity.

Table 4.3: Major Functions of Motion Estimation

Functions	Execution Time Rate w.r.t. whole encoder	Execution Time Rate w.r.t. ME
blkmatch16	64.66%	69.04%
blkmatch16WithShape	23.88%	25.50%
blkmatchForShape	4.07%	4.34%
blockmatch8	0.22%	0.23%
blockmatch8WithShape	0.03%	0.03%
Others	0.79%	0.85%

Hotspots code segment of blkmatch16	Clockticks Events per VOP
for (iy = 0; iy < MB_SIZE; iy++){	17,665,616
for (ix = 0; ix < (MB_SIZE; ix++)	354,874,222
mbDiff += abs (ppxlcTmpC [ix] - ppxlcRefMB [ix]);	624,059,434
if (mbDiff >= iMinSAD)	63,822,424
goto NEXT_POSITION; // skip the current position	
ppxlcRefMB += m_iFrameWidthY;	14,245,596
ppxlcTmpC += MB_SIZE;	11,929,677
}	

Figure 4.2: Code segment of hotspots of blkmatch16.

4.2.1 Motion Estimation Optimization

Figure 4.1 shows that the most computation is spent on functions relating to motion estimation. Hence our first target is to reduce the complexity of these functions. The major functions of motion estimation are summarized in Table 4.3 and we also show the percentage complexity of each function with respect to the encoder and to the motion estimation.

Optimization of blkmatch16

The blkmatch16 function finds the best matched MB in the previous reconstructed VOP and is applied to MBs which are totally in VOP. The search method in the original reference software is spiral full search. To reduce the complexity we need to find the hotspots. The hotspots of blkmatch16 are shown in Figure 4.2.

As we can see, the most complexity is to calculate SAD (sum of absolute differences) at integer pixel displacements. In order to reduce the complexity we use MMX instructions to modify the original loop. Firstly, we do not change the search algorithm (using full search as the original code). We just modify the SAD kernel by using MMX instructions. The modified code is shown in Figure 4.3.

The major instruction we use is "psadbw." The psadbw instruction computes the absolute differences of 8 unsigned byte integers using 64-bit operands. These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4.4 shows the operation of the psadbw instruction. The original C++ code contains a premature breakout mechanism that saves iterations loop by comparing the SAD value accumulated after each row with the current minimum SAD value. According to [13], this premature breakout mechanism will decrease the efficiency. But after experiment we find that if we comment out this mechanism the efficiency will be a little lower than we keep it and unroll the loop four times so that each loop iteration calculates the SAD for 4 rows of the macroblock.

We also modify the SAD kernel of half sample search by MMX instruction. The original C++ code is shown in Figure 4.5.

As we can see, the SAD kernel of half sample motion search is a little different from the SAD kernel of integer pixel search. The difference is that the half sample positions in the reference VOP for calculating SAD are not continuous. We need to modify the MMX SAD kernel of integer pixel search to suit this condition for efficiently. The modified code is shown in Figure 4.6. The major differences with MMX SAD kernel of integer pixel search are that we need to shift pixel values in MMX register left and right to reserve pixel values we need. Then we pack the pixel values in two MMX registers to one for psadbw instruction.

After modifying the original code by the above methods, the number of clockticks events of blkmatch16 is reduced to around 91M clockticks events per VOP, achieving a 1936% speedup comparing with the original code. We show the comparison in Table 4.4.

```
for (iy = 0; iy < 4; iy++){
        __asm
        {
                pxor mm6, mm6;
                pxor mm7, mm7;
                mov edx, ppxlcRefMB;
                mov ebx, ppxlcTmpC;
                movq mm1,[edx];    // read 1st 8 pixels of reference block
                movq mm2,[edx+8];  // read next 8 pixels of reference block
                psadbw  mm1, [ebx]; // calculate SAD of pairs of 1st 8 pixels
                psadbw  mm2, [ebx+8]; // calculate SAD of pairs of next 8 pixels
                paddw   mm6, mm1; // add to buffer for final SAD
                paddw   mm7, mm2; // add to buffer for final SAD
                mov eax, dword ptr [iFrameWidthY] // Calculate SAD of next row
                movq mm1, [edx][eax];
                movq mm2, 8[edx][eax];
                psadbw mm1, [ebx+16];
                psadbw mm2, [ebx+24];
                paddw   mm6, mm1;
                paddw   mm7, mm2;
                mov eax, dword ptr [iFrameWidthYx2]; // Calculate SAD of 3rd row
                movq mm1, [edx][eax];
                movq mm2, 8[edx][eax];
                psadbw mm1, [ebx+32];
                psadbw mm2, [ebx+40];
                paddw   mm6, mm1;
                paddw   mm7, mm2;
                mov eax, dword ptr [iFrameWidthYx3]; // Calculate SAD of 4th row
                movq mm1, [edx][eax];
                movq mm2, 8[edx][eax];
                psadbw mm1, [ebx+48];
                psadbw mm2, [ebx+56];
                paddw   mm6, mm1;
                paddw   mm7, mm2;
                paddd   mm7, mm6; // Calculate the SAD of 4 rows
                movd    eax, mm7;
                add     eax, dword ptr [mbDiff]
                mov     dword ptr [mbDiff], eax
                emms
        }
                if (mbDiff >= iMinSAD)
                        goto NEXT_POSITION; // skip the current position
                ppxlcRefMB += iFrameWidthYx4;
                ppxlcTmpC += iMB_SIZEx4;
}
```

Figure 4.3: Revised code segment of SAD kernel of integer pixel motion search.

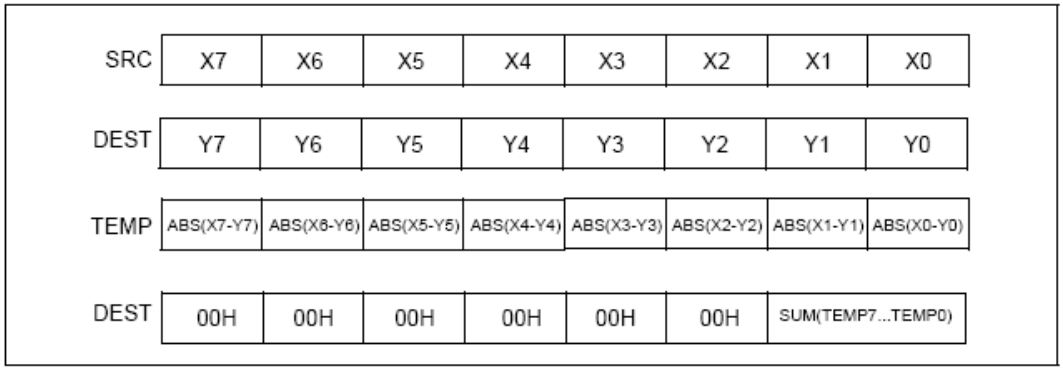


Figure 4.4: PSADBW instruction operation using 64-bit operands(from [10]).

```
for (iy = 0; iy < MB_SIZE; iy++){
        for (ix = 0; ix < MB_SIZE; ix++)
                mbDiff += abs (ppxlcTmpC [ix] - ppxlcRefZoomMB [2 * ix]);
        if (mbDiff > iMinSAD)
                goto NEXT_HALF_POSITION;
        ppxlcRefZoomMB += m_iFrameWidthZoomY * 2;
        ppxlcTmpC += MB_SIZE;
}
```

Figure 4.5: Original code segment of the SAD kernel of half pixel motion search.

Table 4.4: Execution Result of Optimized blkmatch16 Function Using MMX

Function	Clockticks/VOP of Original Code	Clockticks/VOP of Modified Code	Speedup
blkmatch16	1,763,013,601	91,043,523	1936.45 %

```
for (iy = 0; iy < MB_SIZE ; iy++) {
        __asm {
                mov eax, ppxlcTmpC;
                mov ecx, ppxlcRefZoomMB;
                movq mm1, [eax];
                movq mm2, [eax + 8];
                movq mm3, [ecx]; // read 1st 8 pixels of reference zoom block
                movq mm4, [ecx + 8]; // read next 8 pixels of reference zoom block
                movq mm5, [ecx + 16];
                movq mm6, [ecx + 24];
                psllw mm3, 8; // shift left to reserve pixels we need
                psllw mm4, 8;
                psllw mm5, 8;
                psllw mm6, 8;
                psrlw mm3, 8; // then shift right for packuswb instruction
                psrlw mm4, 8;
                psrlw mm5, 8;
                psrlw mm6, 8;
                packuswb mm3, mm4; // pack the two 4 pixels to one MMX register
                packuswb mm5, mm6;
                psadbw mm1, mm3;
                psadbw mm2, mm5;
                paddd mm1, mm2;
                movd temp, mm1;
        }
        mbDiff += temp;
        if (mbDiff > iMinSAD)
                goto NEXT_HALF_POSITION;
        ppxlcRefZoomMB += m_iFrameWidthZoomY * 2;
        ppxlcTmpC += MB_SIZE ;
}
```

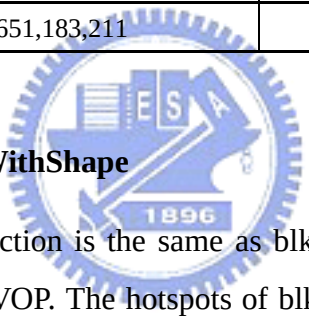
Figure 4.6: Revised code segment of the SAD kernel of half pixel motion search.

```
for (iy = 0; iy < MB_SIZE; iy++) {
        for (ix = 0; ix < MB_SIZE; ix++) {
                if (ppxlcTmpCBY [ix] != transpValue)
                        mbDiff += abs (ppxlcTmpC [ix] - ppxlcRefMB [ix]);
        }
        if (mbDiff > iMinSAD)
                goto NEXT_HALF_POSITION; // skip the current position
        ppxlcRefMB += m_iFrameWidthY;
        ppxlcTmpC += MB_SIZE;
        ppxlcTmpCBY += MB_SIZE;
}
```

Figure 4.7: Code segment of hotspots of blkmatch16WithShape function.

Table 4.5: Execution Result of Optimized blkmatch16WithShape Function Using MMX

Function	Clockticks/VOP of Original Code	Clockticks/VOP of Modified Code	Speedup
blkmatch16WithShape	651,183,211	95,240,399	683.73%



Optimization of blkmatch16WithShape

The blkmatch16WithShape function is the same as blkmatch16 but is applied to MBs which are on the boundary of VOP. The hotspots of blkmatch16WithShape is the same as blkmatch16, that is, integer pixel SAD computation for the MB. The original code is shown in Figure 4.7.

The SAD kernel adds a conditional statement “if (ppxlcTmpCBY [ix] != transp-Value)” because the pixels out of VOP will not be calculated in SAD computation. We modify our optimized integer pixel SAD kernel of blkmatch16 to include this conditional statement. The modified code is in Figure 4.8. The major differences with MMX integer SAD kernel of blkmatch16 are that we use 128-bit SIMD integer instructions of SSE2 to handle 16 pixels in a single instruction and we use pand instruction to substitute the added conditional statement. After modifying the original code, the execution result is shown in Table 4.5.

```
for (iy = 0; iy < MB_SIZE; iy++){
        __asm {
                mov eax, ppxlcTmpCBY; // move the memory address of current BAB MB to eax
                mov ecx, ppxlcTmpC;   // move the memory address of current MB to ecx
                mov edx, ppxlcRefMB;  // move the memory address of reference MB to eax
                movdqu xmm0, [eax];   // read 16 pixels of current BAB MB
                movdqu xmm1, [ecx];   // read 16 pixels of current MB
                movdqu xmm2, [edx];   // read 16 pixels of reference MB
                pand xmm1, xmm0;      // Using pand to substitute the original "if statement"
                pand xmm2, xmm0;
                psadbw xmm1, xmm2;
                movdqa xmm3, xmm1;
                psrldq xmm3,8;
                ddd xmm1, xmm3;
                vd temp, xmm1;
                emms;
        }
        mbDiff += temp;
        if (mbDiff > iMinSAD)
                goto NEXT_POSITION; // skip the current position
        ppxlcRefMB += m_iFrameWidthY;
        ppxlcTmpC += MB_SIZE;
        ppxlcTmpCBY += MB_SIZE;
}
```

Figure 4.8: Revised code segment of integer pixel SAD kernel of blkmatch16WithShape function.

Table 4.6: Execution Result of Optimization of Motion Estimation Using MMX

Encoder Block	Clockticks/VOP of Original Code	Clockticks/VOP of Modified Code	Speedup
Motion Estimation	2,553,489,414	203,983,469	1251.81%

Optimization of Other Functions

The hotspots of the other functions for motion estimation are mostly the integer pixel SAD computation of 16×16 or 8×8 block. The optimization method is almost the same. We only use MMX or SSE2 instructions to modify the SAD kernel with full search. We do not go through the detail methods for optimizing each functions. We introduce each function and summarize the experimental result after we optimize all major functions for motion estimation.

The blkmatchForShape function finds best matched binary alpha plane MB. After searching for 16×16 motion vectors, additional search is made for 8×8 vectors. Again, the search is made with integer pixel displacements and for the Y component. The same as 16×16 motion estimation, the 8×8 integer motion estimation also employs block-matching. Using the 16×16 motion vector as the search center, the search range of 8×8 motion estimation is ± 2 pixels. The blockmatch8 and blockmatch8WithShape are functions to implement the 8×8 integer motion estimation with respect to MB totally in VOP and MB on VOP boundary. Table 4.6 shows the summary result of optimization for motion estimation using MMX. The clockticks per VOP is reduced from 2,553,489,414 to 203,983,469, which is 92.01% reduction.

4.2.2 Motion Estimation Optimization Using Fast Motion Search

Besides using MMX technology with full-search (FS) motion estimation to speed up the functions for motion estimation, we also consider some fast search methods: two dimensional logarithm search (2DLS) [13], diamond search (DS) [16] and new diamond search (NDS) [15] to optimize blkmatch16 and blkmatch16WithShape functions to reduce the execution time furthermore.

The 2DLS is similar to a binary search. First, the algorithm computes SAD(i,j) at 9 locations within the search region: (0,0) and 8 points located along the rectangle $[-SR/2, SR/2]$ where SR is the search range. Using the best matched location as the new starting point (0,0), the algorithm computes SAD(i,j) at 9 more locations: the new starting point and 8 points located along the rectangle $[-SR/4, SR/4]$. This process continues until the search region cannot be divided further. Figure 4.9 show an example of 2DLS.

The method of diamond search is a suitable way for slow-moving video. Figure 4.10 illustrates the method. With the center point corresponding to zero MV as initial point, the first step involves ME based on 1-pel/1-line resolution at the four locations denoted 1. If the best solution is the center point, the motion estimation is finished. Otherwise, repeat the same method of the first step using the best solution of the last search step as the center point.

The new diamond search method is shown in Figure 4.11. The search procedure is summarized as follows.

- Step 1: The initial large diamond search pattern (LDSP) is centered at the origin of the search window, and the 9 check points of LDSP are tested. If the best matched point is located at the center position, go to Step 3; otherwise, go to Step2.
- Step 2: The best matched point found in the previous search step is re-positioned as the center point to form a new LDSP. If the new best matched point obtained is located at the center position, go to Step 3; otherwise, recursively repeat this step.
- Step 3: Switch the search pattern form LDSP to small diamond search pattern (SDSP). The best matched point in this step is the final solution.

The SAD kernel of these methods is the same as described last section and Table 4.7 gives a comparison of these fast motion search methods.

Summary

Table 4.8 gives the summary results of optimization for motion estimation using MMX and fast motion search methods. The clockticks per VOP for full search by using MMX/SSE2

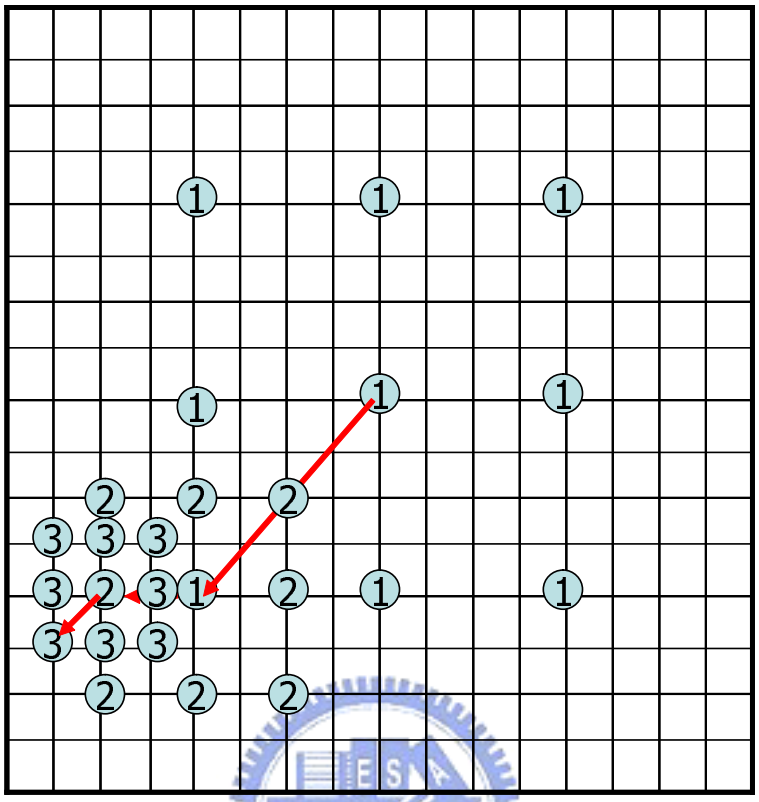


Figure 4.9: The method of 2D logarithmic search (from [13]).

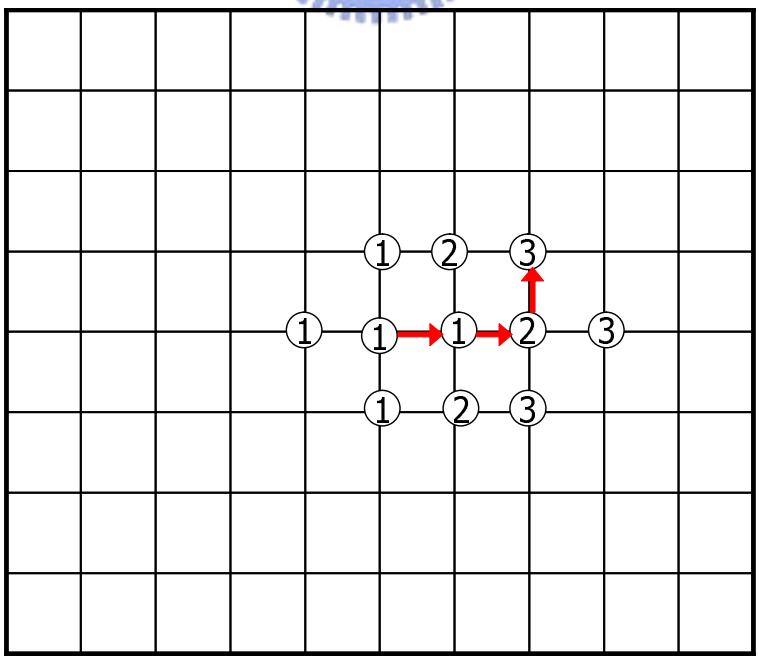


Figure 4.10: The method of diamond search (from [16]).

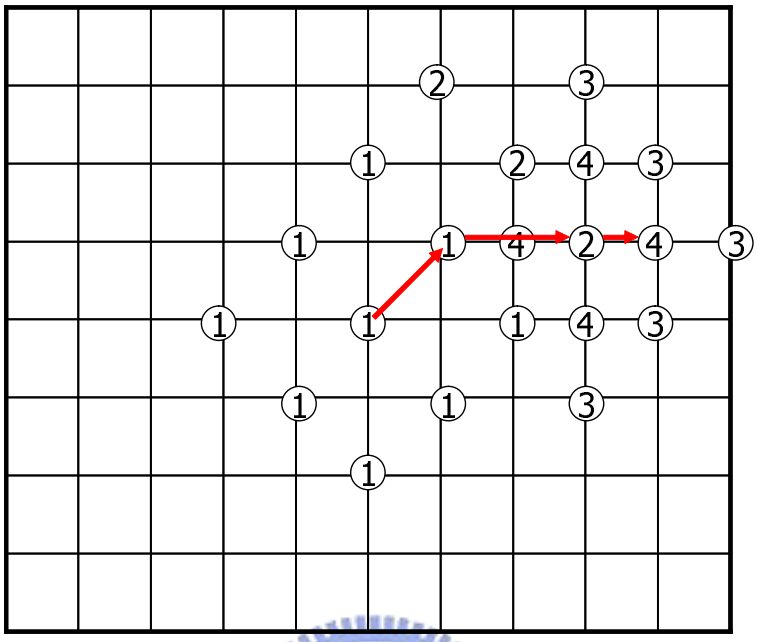


Figure 4.11: The method of new diamond search (from [15]).

Table 4.7: Execution Results of Optimization of blkmatch16 and blkmatch16WithShape Using Fast Motion Search Method

Function	Search Approach	Clockticks/VOP	Speedup Respect to Original Code
blkmatch16	Original Code	1,763,013,601	None
	2D Logarithmic Search	6,381,165	27628.40%
	Diamond Search	3,666,585	48083.26%
	New Diamond Search	4,275,402	41236.20%
blkmatch16WithShape	Original code	651,183,211	None
	2D logarithmic search	5,089,205	12795.38%
	Diamond search	3,224,729	20193.42%
	New diamond search	3,981,227	16356.35%

Table 4.8: Execution Result of Optimization of Motion Estimation

Encoder Block	Search Approach	Clockticks/VOP	Speedup respect to original code
Motion Estimation	Original code	2,553,489,414	None
	Full search with MMX	203,983,469	1251.81%
	2D logarithmic search	29,700,425	8597.48%
	Diamond search	24,448,053	10444.55%
	New diamond search	27,836,810	9173.07%

only is reduced from 2,553,489,414 to 203,983,469, which is 92.01% reduction. The reductions in clockticks per VOP for fast motion estimation method are 98.84%, 99.04% and 98.91% for 2DLS, DS and NDS search methods, respectively.

4.2.3 VOP Formation Optimization

MPEG-4 video coding can handle objects within video scenes. In order to get a higher coding efficiency and to handle VOPs with arbitrary shape, the encoder needs to create a rectangle that completely contains the object but with the minimum number of MBs in it.

The VOP formation has the following procedure (from [6]).

1. Generate the tightest rectangle with even numbered top left position.

2. If the top left position of this rectangle is the same as the origin of the image frame, skip the formation procedure. Form a control MB at the top left corner of the tightest rectangle as shown in Figure 4.12.

3. Count the number of MBs that completely contain the object, starting at each even numbered point of the control MB. The details are as follows:

   (a) Generate a bounding rectangle from the control point to the right bottom side of the object which consists of multiples of 16x16 blocks.

   (b) Count the number of MBs in this bounding rectangle, which contains at least one object pixel. It is sufficient to take into account only the boundary pixels of an MB.

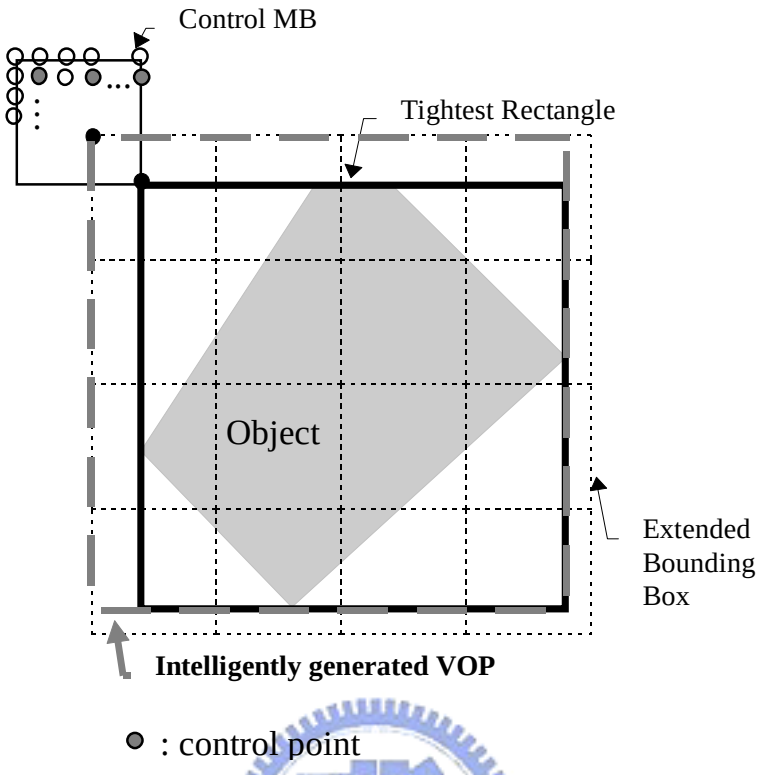


Figure 4.12: VOP formation (from [6]).

4. Select that control point that results in the smallest number of MBs for the given object.

5. Extend the top left coordinate of the tightest rectangle generated in Figure 4.12 to the selected control coordinate. This will create a rectangle that completely contains the object but with the minimum number of MBs in it.

The Microsoft MPEG-4 Visual Reference Software has two functions to implement the above procedure for VOP formation: findTightBoundingBox and findBestBounding-Box. The clockticks of findBestBoundingBox are 20 times than findTightBoundingBox. We target at reducing the complexity of findBestBoundingBox.

After we examine the code, we find that the hotspots is the for-loop shown in Figure 4.13. The loop is used for counting the number of MBs in this bounding rectangle which contains at least one object pixel, as described in Step 3(b) of the above procedure.

We use MMX instructions to sum one row of binary alpha plane to substitute for the "for (UInt iX = 0; iX < MB_SIZE; iX++)" loop. The revised code is shown in Figure

```
UInt nOpaquePixel = 0;
for (UInt iY = 0; iY < MB_SIZE; iY++){
        for (UInt iX = 0; iX < MB_SIZE; iX++)
                nOpaquePixel += *ppxlcBY++;
        ppxlcBY += m_iFrameWidthY - MB_SIZE;
}
if (nOpaquePixel != 0)
        nNonTransparentMB++;
```

Figure 4.13: Code segment of hotspots of findBestBoundingBox function.

```
for (UInt iY = 0; iY < MB_SIZE; iY++) {
        __asm {
                mov edx, ppxlcBY;
                pxor mm0, mm0;   // let mm0 to be all zero
                movq mm1, [edx]; // move 8 pixels of binary block to mm1
                movq mm3, [edx+8]; // move next 8 pixels of binary block to mm3
                psadbw  mm1, mm0;  // sum of 1st 8 pixels
                psadbw  mm3, mm0;  // sum of next 8 pixels
                paddd   mm1, mm3;  // sum of one row
                movd    eax, mm1;
                add     eax, dword ptr[nOpaquePixel];
                mov     dword ptr [nOpaquePixel], eax;
                emms
        }
        ppxlcBY += m_iFrameWidthY;
}
```

Figure 4.14: Revised code segment of hotspots of findBestBoundingBox function.

4.14 and the execution result is shown in Table 4.9.

4.2.4 DCT and IDCT Optimization

DCT and IDCT are used for the 8×8 blocks in the intra and inter pictures. The DCT/IDCT in Microsoft MPEG-4 Visual Reference Software is Fast DCT using floating-point computation. We implement DCT and IDCT by using matrix multiplication as in the following equation:

$$[Y_{mn}]_{8\times8} = [C_{mn}]^T_{8\times8}\,[X_{mn}]_{8\times8}\,[C_{mn}]_{8\times8}$$

where

$$C_{mn} = K_n \cos\left[\frac{(2m+1)n\pi}{16}\right],\ K_n = \begin{cases} 1/2, & \text{n = 0,} \\ 1/2\sqrt{2}, & \text{otherwise.} \end{cases}$$

Table 4.9: Execution Result of Optimization of findBestBoundingBox Function

Function	Clockticks/VOP of Original Code	Clockticks/VOP of Modified Code	Speedup
findBestBoundingBox	27,907,394	5,270,853	529.47%

Table 4.10: Execution Result of Optimization of DCT and IDCT

Function	Clockticks/VOP	PSNR_Y	PSNR_U	PSNR_V
Floating-Point Fast DCT/IDCT	22,966,734	35.4783	40.2630	40.1919
Integer DCT/IDCT with MMX	16,705,303	35.4703	40.287	40.1541

In order to speed up computation, we use scaled 16-bit integer to substitute the floating-point coefficients of matrix $[C]$. We use the method $\lfloor C(m,n) \times 2^{15} + 0.5 \rfloor$ or $\lfloor -C(m,n) \times 2^{15} + 0.5 \rfloor$ [14] to get scaled constants stored in 16-bit integer format in an order suitable for applying the PMADDWD instruction.

The revised DCT code is shown in Figure 4.15 and the implementation of IDCT is almost the same as DCT except for the transform matrix.

We compare the performance of the integer DCT/IDCT and the floating-point fast DCT in Table 4.10. As the data show, the PSNR with integer DCT/IDCT is just a little worse than the floating-point with a 137.48% speedup.

4.2.5 Motion Compensation Optimization

The hotspots of Motion Compensation is the function motionCompEncY. This function copies data from reference image to the prediction block. And the bottleneck of motionCompEncY is as shown in Figure 4.16

We use MMX instructions to substitute the memcpy function and the “for (ix = 0; ix $<$ iSize; ix++)” loop in the else-clause. The revised code is shown in Figure 4.17 and Table 4.11 shows the experimental result.

```
short sDCT[64] = { 23170, 23170, 23170, 23170, 23170, 23170, 23170, 23170, // scaled 16-bit integer transform matrix C
            32138, 27246, 18205, 6393 , -6393,-18205,-27246,-32138,
            30274, 12540,-12540,-30274,-30274,-12540, 12540, 30274,
            27246, -6393,-32138,-18205, 18205, 32138,  6393,-27246,
            23170,-23170,-23170, 23170, 23170,-23170,-23170, 23170,
            18205,-32138,  6393, 27246,-27246, -6393, 32138,-18205,
            12540,-30274, 30274,-12540,-12540, 30274,-30274, 12540,
              6393,-18205, 27246,-32138, 32138,-27246, 18205, -6393 };
short y[8][8], *psDCT, *pSrcTemp;
int tempy, tempz, i, j *z = rgiDst;
const PixelC* psrc = rgchSrc;
// 1D row transform
for (i = 0; i < BLOCK_SIZE ; i++) {
            psDCT = sDCT;
            for (j=0 ; j<8 ; j++) {
                        __asm {
                                    mov edx, psrc;
                                    mov ebx, psDCT;
                                    pxor xmm0, xmm0;
                                    movq xmm1, [edx]; // move 8 pixel from src to xmm1
                                    movdqu xmm2, [ebx]; // move DCT coeff. to xmm2
                                    punpcklbw xmm1, xmm0; // unpack 8-bits pixel data to 16-bits
                                    pmaddwd xmm1, xmm2; // multiply and add packed integers
                                    movdqa xmm3, xmm1;
                                    movdqa xmm4, xmm1;
                                    movdqa xmm5, xmm1;
                                    psrldq xmm3, 4;
                                    psrldq xmm4, 8;
                                    psrldq xmm5, 12;
                                    paddd xmm1, xmm3;
                                    paddd xmm4, xmm5;
                                    paddd xmm1, xmm4;
                                    movd tempy, xmm1;
                        }
                        y[j][i] = ((tempy + 32768) >> 16); // Compensation before shift right
                        psDCT += 8;
            }
            psrc += nColSrc;
}
// 1D column transform
pSrcTemp = &y[0][0];
for (i = 0; i < BLOCK_SIZE ; i++) {
            psDCT = sDCT;
            for (j=0 ; j<8 ; j++) {
                        __asm {
                                    mov edx, pSrcTemp;
                                    mov ebx, psDCT;
                                    movdqu xmm1, [edx]; // move 8 value from src after 1D to xmm1
                                    movdqu xmm2, [ebx]; // move DCT coeff. to xmm2
                                    pmaddwd xmm1, xmm2; // multiply and add packed integers
                                    movdqa xmm3, xmm1;
                                    movdqa xmm4, xmm1;
                                    movdqa xmm5, xmm1;
                                    psrldq xmm3, 4;
                                    psrldq xmm4, 8;
                                    psrldq xmm5, 12;
                                    paddd xmm1, xmm3;
                                    paddd xmm4, xmm5;
                                    paddd xmm1, xmm4;
                                    movd tempz, xmm1;
                        }
                        z[j*nColDst + i] = ((tempz + 32768) >> 16); // Compensation before shift right
                        psDCT += 8;
            }
            pSrcTemp += 8;
}
```

Figure 4.15: Revised code segment of DCT.

```
if (!bYSubPxl && !bXSubPxl) {
        const PixelC* ppxlcRefMB = ppxlcRef
        + m_rctRefFrameY.offset (xHalf>>1, yHalf>>1);
        for (iy = 0; iy < iSize; iy++) {
                memcpy (ppxlcPred, ppxlcRefMB, iSize*iUnit);
                ppxlcRefMB += m_iFrameWidthY;
                ppxlcPred += MB_SIZE;
        }
}
else {
        const PixelC* ppxlcPrevZoomY = ppxlcRefZoom
        + m_puciRefQZoom0->where ().offset (xHalf, yHalf);
        for (iy = 0; iy < iSize; iy++) {
                for (ix = 0; ix < iSize; ix++)
                        ppxlcPred [ix] = ppxlcPrevZoomY [2 * ix];
                ppxlcPrevZoomY += m_iFrameWidthZoomY * 2;
                ppxlcPred += MB_SIZE;
        }
}
```

Figure 4.16: Code segment of hotspots of motionCompEncY function.

```
// memcpy (ppxlcPred, ppxlcRefMB, iSize*iUnit);
 __asm {
        mov eax, ppxlcPred;
       mov ecx, ppxlcRefMB;
        movdqu xmm0, [ecx];
        movdqu [eax], xmm0;
        emms
 }
```

```
// for (ix = 0; ix < iSize; ix++)
//       ppxlcPred [ix] = ppxlcPrevZoomY [2 * ix];
__asm {
        mov eax, ppxlcPrevZoomY;
        mov ecx, ppxlcPred;
        movdqu xmm0, [eax];
        movdqu xmm1, [eax+16];
        psllw xmm0, 8;
        psllw xmm1, 8;
        psrlw xmm0, 8;
        psrlw xmm1, 8;
        packuswb xmm0, xmm1;
        movdqu [ecx], xmm0;
        emms
}
```

Figure 4.17: Revised code segment of hotspots of motionCompEncY function.

Table 4.11: Execution Result of Optimization of Motion Compensation

Function	Clockticks/VOP of Original Code	Clockticks/VOP of Modified Code	Speedup
motionCompEncY	15,307,952	8,410,287	182.01%

```
// Input: xmm1
// Output: abs(xmm1)
pxor xmm0, xmm0; // generate a zero register in xmm0
psubw xmm0, xmm1;
pmaxsw xmm1, xmm0; // generate abs(xmm1)
```

Figure 4.18: Code segment of abs using SSE2.

4.2.6 Quantization Optimization

We use H.263 quantization method to quantize the transform coefficients. The quantization parameter QP may take integer values from 1 to 31. The quantization stepsize is $2 \times$ QP. For intra quantization, we have LEVEL = $|\text{COF}| \ / \ (2 \times \text{QP})$, and for inter quantization, we have LEVEL = $(\ |\text{COF}| - \text{QP} \ / \ 2) \ / \ (2 \times \text{QP})$, where COF is a transform coefficient to be quantized and LEVEL is the absolute value of quantized version of transform coefficient. Clipping to $[-127{:}127]$ is performed for all coefficients except intra DC.

The bottleneck of quantization is to calculate abs of COF. We use SSE2 instructions to calculate abs as shown in Figure 4.18 and Table 4.12 shows the experimental result.

Table 4.12: Execution Result of Quantization Optimization

Function	Clockticks/VOP of Original Code	Clockticks/VOP of Modified Code	Speedup
quantizeInterDCTcoefH263	7,123,176	6,016,844	118.39%

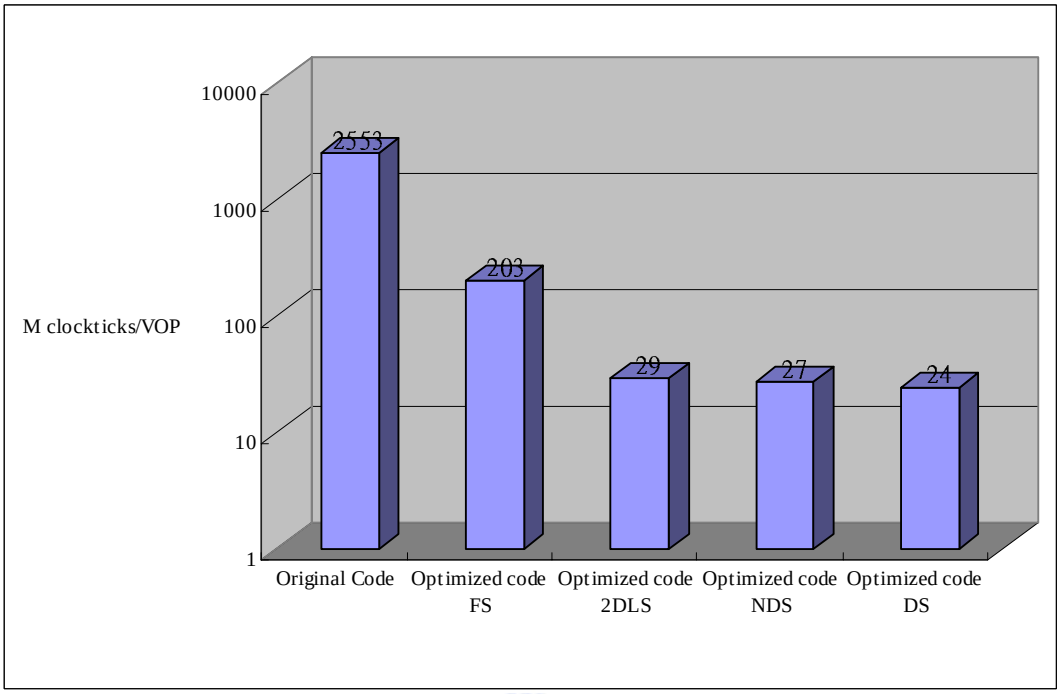


Figure 4.19: Comparison between original reference software and optimized code in execution time for motion estimation.

4.3 Conclusion in Optimization

After optimization, the results are shown as Figures 4.19 and 4.20 for motion estimation and other encoder blocks respectively.

The clockticks per VOP for motion estimation is reduced from 2,553M to 203M, 29M, 27M and 24M for full search, 2D logarithmic search, new diamond search and diamond search, respectively.

The clockticks per VOP for VOP formation is reduced from 29.8M clockticks to 7.6M clockticks which is 74.5% reduction. The clockticks per VOP for DCT/IDCT is reduced from 22.9M clockticks to 16.7M clockticks which is 27.07% reduction. The clockticks per VOP for motion compensation is reduced from 16.6M clockticks to 9.8M clockticks which is 40.9% reduction. The clockticks for quantization and inverse quantisation is reduced from 9.3M clockticks to 8.6M clockticks which is 7.5% reduction.

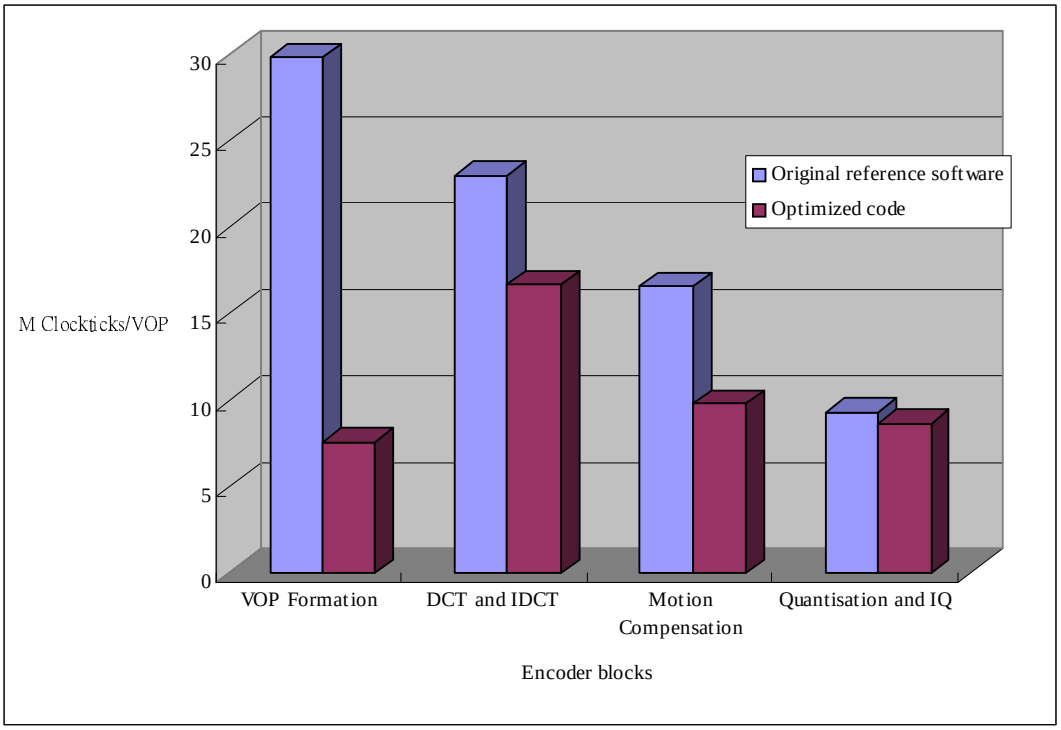


Figure 4.20: Comparison between original reference software and optimized code in execution time for other encoder blocks

Chapter 5

Experimental Results

In the last chapter, we discussed how we optimized the major functions of the original reference software. In this chapter, we present some experimental results on the speed and the coding performance by encoding different kinds of video sequence.

5.1 Encoding Speed Performance

We first consider the overall encoding speed performance of the optimized MPEG-4 encoder in comparison to the original reference software. We will consider two different coding methods, one is coding without shape and the other is with shape information. We also compare the result of different motion search algorithms and different compilation mode: debug mode and release mode. We will discuss rate-distortion performance in the next section.

5.1.1 Frame Based Coding

We first encode video sequence without shape information. And we compare the original reference software and the revised code. Table 5.1 shows the result for different kinds of test sequence using debug compilation mode. The speedup is from 900% to 1400% by using full search method and from 2400% to 4600% by using fast motion search methods.

We also compile codes using release mode and Table 5.2 shows the result for different kinds of test sequence. The speedup is from 160% to 260% by using full search method

Table 5.1: Overall Coding Speed Without Shape in Average CIF Frame per Second Using Debug Compilation Mode

	QP							
Test Sequence / Encoder	4	6	8	12	16	20	24	28
akiyo_cif / Original Code	1.31	1.32	1.31	1.28	1.23	1.19	1.16	1.11
akiyo_cif / Optimized Code (FS)	10.08	10.42	10.64	10.82	10.81	10.73	10.66	10.45
akiyo_cif / Optimized Code (2DLS)	22.86	24.39	25.10	25.88	26.12	26.22	26.21	25.79
akiyo_cif / Optimized Code (NDS)	23.53	25.03	25.88	26.67	26.99	27.10	27.11	26.68
akiyo_cif / Optimized Code (DS)	24.02	25.27	26.06	26.90	27.19	27.39	27.28	26.86
foreman_cif / Original Code	0.47	0.46	0.45	0.43	0.42	0.41	0.40	0.40
foreman_cif / Optimized Code (FS)	5.75	5.75	5.74	5.79	5.81	5.76	5.72	5.68
foreman_cif / Optimized Code (2DLS)	13.99	14.34	14.43	14.60	14.68	14.70	14.73	14.78
foreman_cif / Optimized Code (NDS)	14.42	14.73	15.09	15.42	15.59	15.70	15.74	15.78
foreman_cif / Optimized Code (DS)	14.44	14.83	15.12	15.58	15.84	16.00	16.08	16.11
stefan_cif / Original Code	0.38	0.38	0.38	0.37	0.37	0.36	0.36	0.35
stefan_cif / Optimized Code (FS)	5.41	5.49	5.55	5.54	5.57	5.59	5.56	5.53
stefan_cif / Optimized Code (2DLS)	13.31	13.65	13.94	14.21	14.35	14.44	14.51	14.55
stefan_cif / Optimized Code (NDS)	13.61	14.38	14.83	15.27	15.52	15.68	15.77	15.83
stefan_cif / Optimized Code (DS)	13.72	14.56	15.00	15.53	15.82	15.98	16.05	16.22

Table 5.2: Overall Coding Speed Without Shape in Average CIF Frame per Second Using Release Compilation Mode

	QP							
Test Sequence / Encoder	4	6	8	12	16	20	24	28
akiyo_cif / Original Code	7.80	7.85	7.85	7.70	7.52	7.34	7.13	6.90
akiyo_cif / Optimized Code (FS)	12.55	12.99	13.26	13.43	13.38	13.30	13.16	12.88
akiyo_cif / Optimized Code (2DLS)	32.03	33.91	34.71	35.86	36.17	36.44	36.39	35.85
akiyo_cif / Optimized Code (NDS)	33.29	34.93	36.08	37.49	37.89	38.02	38.10	37.65
akiyo_cif / Optimized Code (DS)	33.65	35.20	36.35	37.65	38.31	38.42	38.59	37.98
foreman_cif / Original Code	3.15	3.02	3.08	2.99	2.91	2.86	2.81	2.75
foreman_cif / Optimized Code (FS)	7.12	7.12	7.09	7.01	6.94	6.87	6.82	6.76
foreman_cif / Optimized Code (2DLS)	19.44	19.76	19.95	20.03	20.18	20.23	20.29	20.29
foreman_cif / Optimized Code (NDS)	20.07	20.49	20.79	21.26	21.48	21.69	21.73	21.80
foreman_cif / Optimized Code (DS)	20.11	20.62	21.00	21.50	21.79	21.98	22.14	22.09
stefan_cif / Original Code	2.58	2.61	2.69	2.65	2.62	2.57	2.54	2.50
stefan_cif / Optimized Code (FS)	6.54	6.68	6.72	6.71	6.67	6.62	6.58	6.53
stefan_cif / Optimized Code (2DLS)	18.71	19.14	19.37	19.62	19.79	19.91	19.96	19.87
stefan_cif / Optimized Code (NDS)	19.29	20.22	20.78	21.27	21.59	21.70	21.94	21.92
stefan_cif / Optimized Code (DS)	19.49	20.40	20.95	21.58	21.90	22.08	22.27	22.30

and from 400% to 900% by using fast motion search methods.

5.1.2 Shape Based Coding

We next encode video sequence with shape information. And we compare the encoding performance of the original reference code and that of the revised code. We only encode the first 200 frames of each test sequence because the VO of foreman_cif.yuv is out of scene after 206 frames.

Tables 5.3 and 5.4 show the result for different kinds of test sequence for debug compilation mode and release compilation mode respectively. As we can see, the encoding speed is variable depending on test sequence. Two major factors affect the execution

Table 5.3: Overall Coding Speed With Shape in Average CIF Frame per Second Using Debug Compilation Mode

	QP							
Test Sequence / Encoder	4	6	8	12	16	20	24	28
akiyo_cif / Original Code	2.03	2.00	2.04	1.99	1.93	1.89	1.84	1.78
akiyo_cif / Optimized Code (FS)	12.12	12.69	13.01	13.27	13.24	13.17	13.07	12.99
akiyo_cif / Optimized Code (2DLS)	23.72	24.81	25.46	26.14	26.51	26.62	26.67	26.37
akiyo_cif / Optimized Code (NDS)	23.65	25.03	25.67	26.70	27.11	27.27	27.23	27.09
akiyo_cif / Optimized Code (DS)	24.63	25.95	26.56	27.36	27.74	27.92	27.97	28.01
foreman_cif / Original Code	0.99	0.98	0.97	0.94	0.92	0.90	0.89	0.87
foreman_cif / Optimized Code (FS)	8.80	8.84	8.82	8.75	8.68	8.62	8.49	8.43
foreman_cif / Optimized Code (2DLS)	19.42	19.60	19.56	19.46	19.46	19.52	19.50	19.39
foreman_cif / Optimized Code (NDS)	19.86	20.27	20.27	20.50	20.60	20.58	20.62	20.61
foreman_cif / Optimized Code (DS)	20.22	20.62	20.75	20.97	21.07	21.14	21.19	21.23
stefan_cif / Original Code	3.74	3.73	3.70	3.66	3.61	3.58	3.53	3.48
stefan_cif / Optimized Code (FS)	26.64	26.87	26.86	26.85	26.71	26.57	26.46	26.29
stefan_cif / Optimized Code (2DLS)	54.17	55.02	55.47	55.89	55.94	55.89	55.94	56.02
stefan_cif / Optimized Code (NDS)	54.96	55.68	56.10	56.31	56.37	56.52	56.52	56.11
stefan_cif / Optimized Code (DS)	55.59	56.95	57.33	58.11	58.34	58.34	58.51	58.62

time: motion and size of VO. The average percentages of VO with respect to whole frame are 37.52%, 37.88% and 5.24% for akiyo_cif.yuv, foreman_cif.yuv and stefan_cif.yuv, respectively. In debug compilation mode, the speedup is from 600% to 900% by using full search method and from 1200% to 2400% by using fast motion search methods. In release compilation mode, the speedup is from 150% to 200% by using full search method and from 350% to 600% by using fast motion search methods.

5.2 Rate-Distortion (R-D) Performance

In addition to encoding speed, the rate-distortion behavior is also an important characterization for an encoder. In this section we will show the performance of the optimized code

Table 5.4: Overall Coding Speed With Shape in Average CIF Frame per Second Using Release Compilation Mode

	QP							
Test Sequence / Encoder	4	6	8	12	16	20	24	28
akiyo_cif / Original Code	10.33	10.46	10.57	10.45	10.30	10.06	9.83	9.62
akiyo_cif / Optimized Code (FS)	15.65	16.24	16.63	16.90	16.90	16.78	16.51	16.34
akiyo_cif / Optimized Code (2DLS)	34.90	35.58	36.85	38.68	39.29	39.21	39.47	39.47
akiyo_cif / Optimized Code (NDS)	36.05	37.73	39.03	40.02	40.59	40.90	40.90	41.07
akiyo_cif / Optimized Code (DS)	36.65	38.26	39.68	40.68	40.98	41.32	41.32	41.46
foreman_cif / Original Code	5.64	5.62	5.53	5.44	5.33	5.23	5.17	5.09
foreman_cif / Optimized Code (FS)	10.89	10.89	10.83	10.71	10.64	10.52	10.47	10.40
foreman_cif / Optimized Code (2DLS)	29.16	29.43	29.53	29.73	29.67	29.68	29.71	29.74
foreman_cif / Optimized Code (NDS)	30.17	30.62	30.81	31.11	31.11	31.19	31.24	31.30
foreman_cif / Optimized Code (DS)	30.71	31.19	31.45	31.63	31.73	31.80	31.87	31.84
stefan_cif / Original Code	17.21	17.17	17.53	17.40	17.13	17.02	16.88	16.43
stefan_cif / Optimized Code (FS)	34.27	34.14	34.14	33.87	33.68	33.36	32.78	32.94
stefan_cif / Optimized Code (2DLS)	82.54	83.44	84.38	85.47	85.73	86.08	85.84	85.96
stefan_cif / Optimized Code (NDS)	83.79	84.27	85.00	85.84	86.33	86.71	86.83	87.09
stefan_cif / Optimized Code (DS)	85.72	86.44	87.08	87.58	87.60	87.73	87.99	87.09

in comparison to the original software.

5.2.1 Frame Based Coding

Figures 5.1, 5.2 and 5.3 show the R-D performance for the test sequences akiyo_cif.yuv, foreman_cif.yuv, and stefan_cif.yuv without shape information. The original frame rate is 30 frame/sec in each case and we encode 300 frames for each sequence. As shown, the R-D performance under optimized full search motion estimation is almost the same as the original reference software at the same bits/frame, and the performance of fast search motion estimation is about 0.5 to 2.5 dB lower depending on sequence and search methods.

5.2.2 Shape Based Coding

Figures 5.4, 5.5 and 5.6 show the R-D performance for the test sequences akiyo_cif.yuv, foreman_cif.yuv, and stefan_cif.yuv with shape information. The original frame rate is 30 frame/sec in each case and we encode 200 frames for each sequence. Same as frame based coding, the performance under optimized full search motion estimation is almost the same as the original reference software at the same bits/frame, and the performance of fast search motion estimation is about 0.5 to 1 dB lower depending on test sequences and search methods.

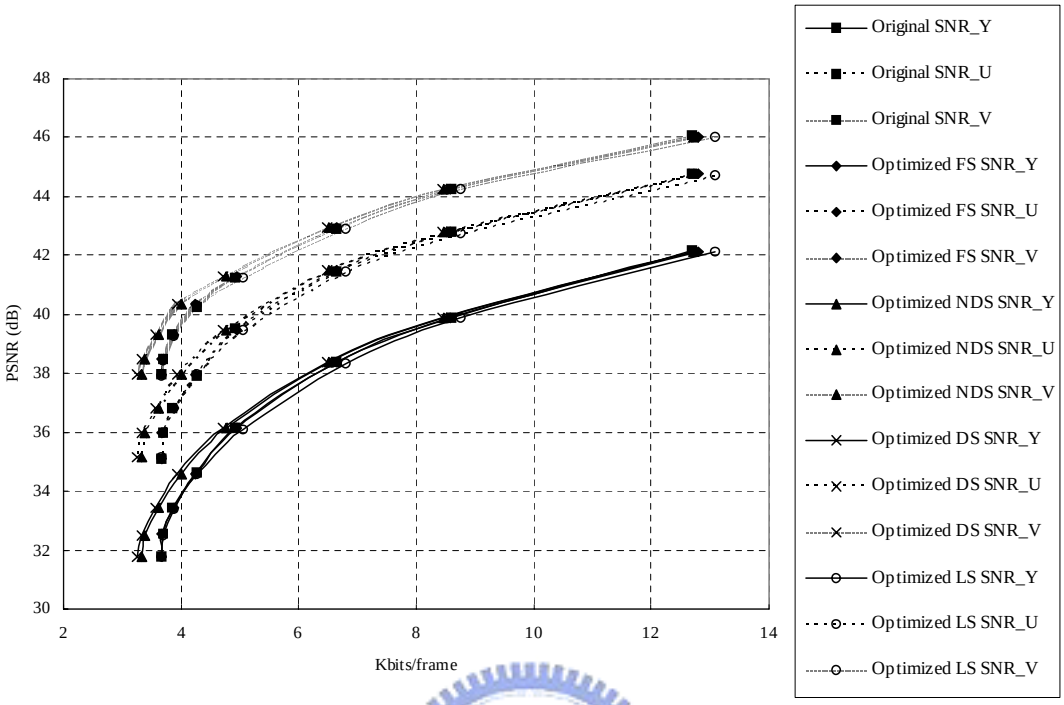


Figure 5.1: R-D performance in coding akiyo_cif without shape.

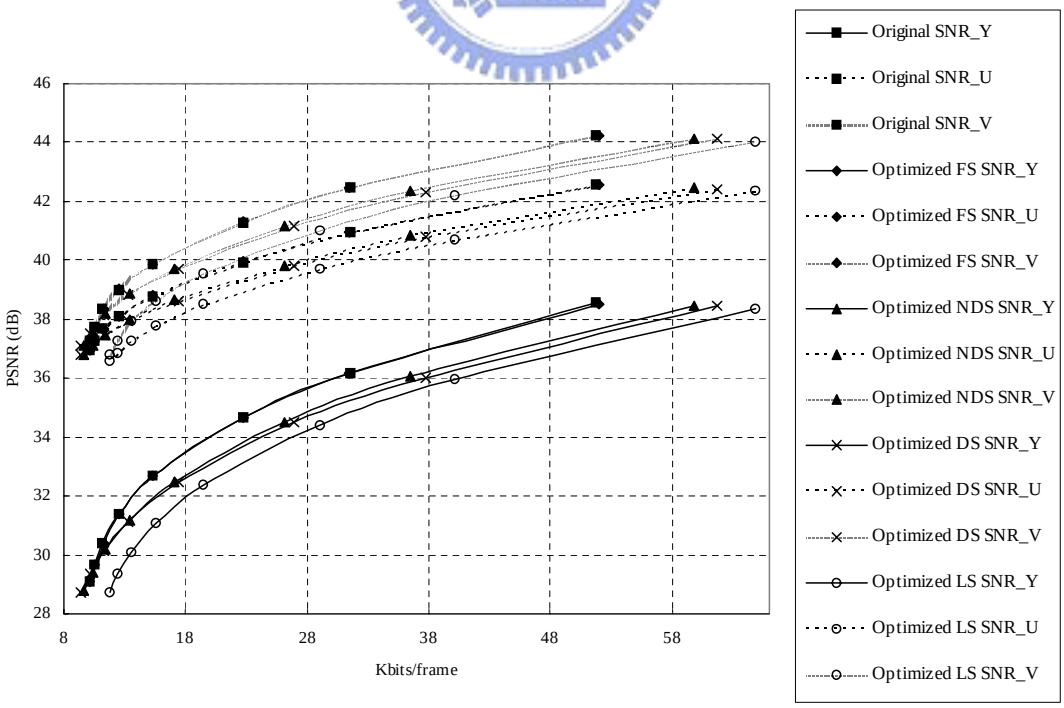


Figure 5.2: R-D performance in coding foreman_cif without shape.

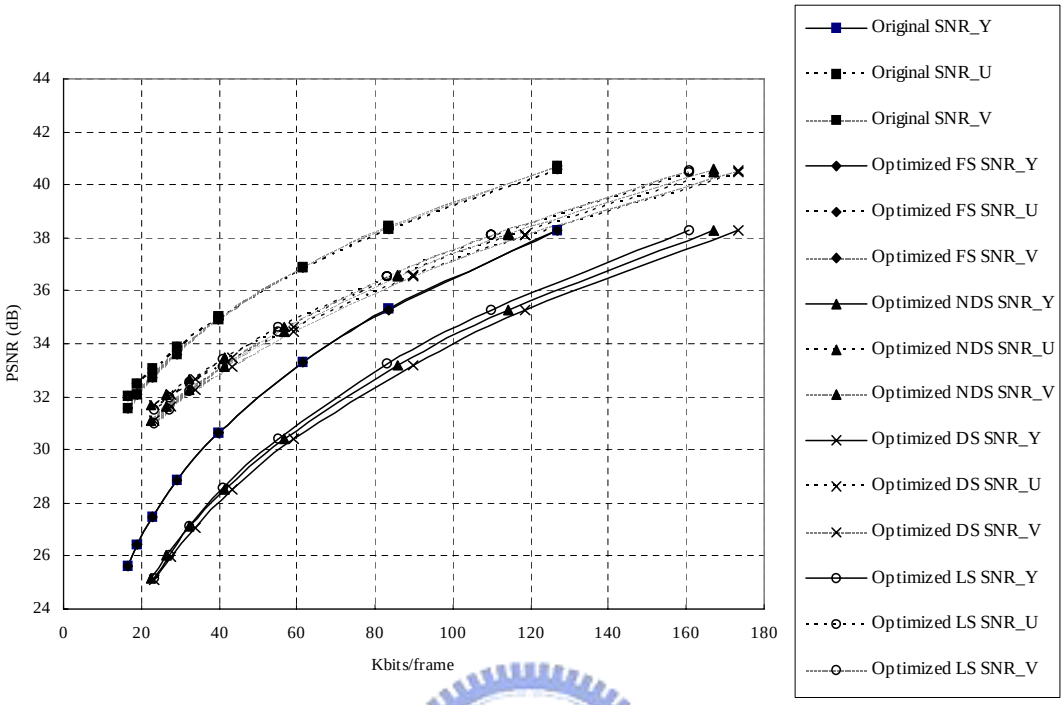


Figure 5.3: R-D performance in coding stefan_cif without shape.

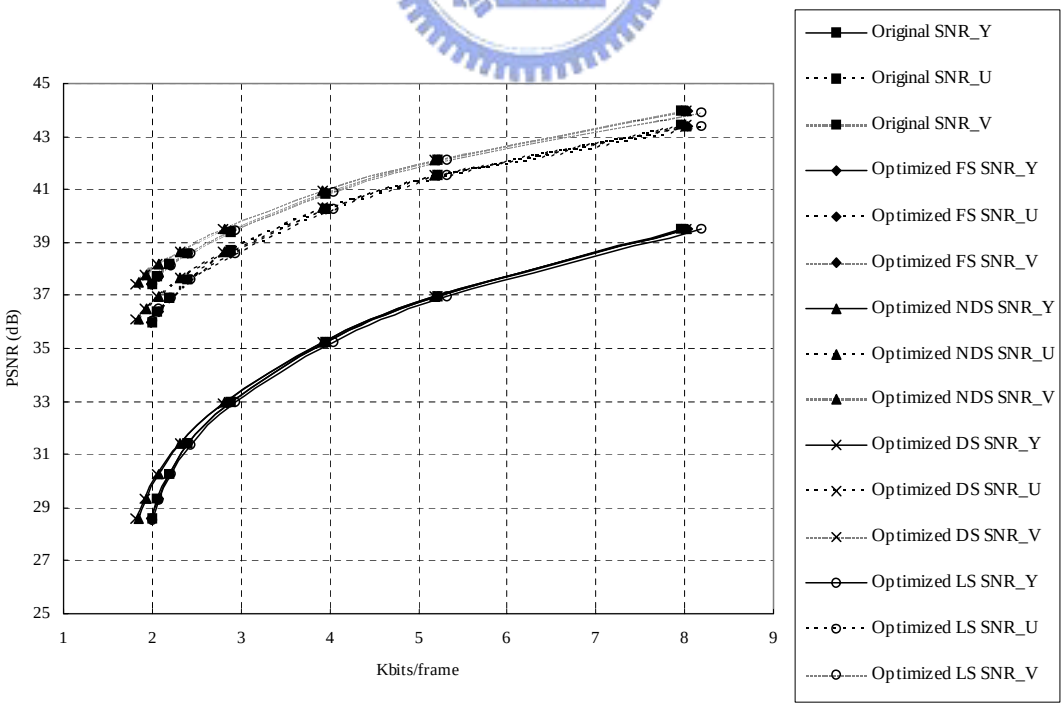


Figure 5.4: R-D performance in coding akiyo_cif with shape.

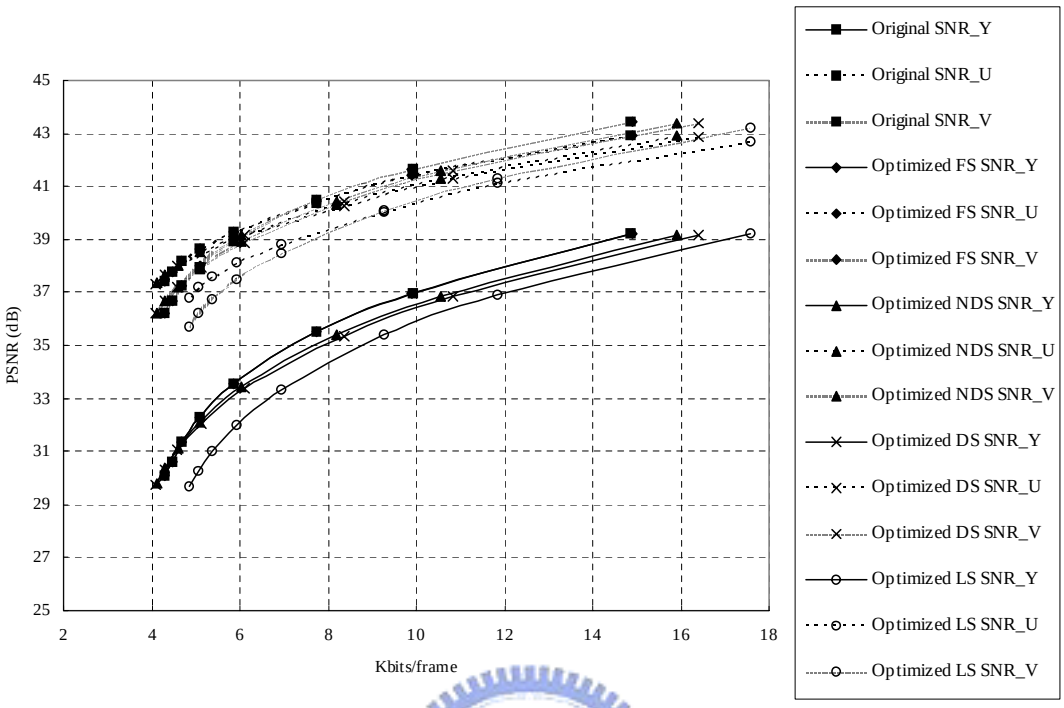


Figure 5.5: R-D performance in coding foreman_cif with shape.

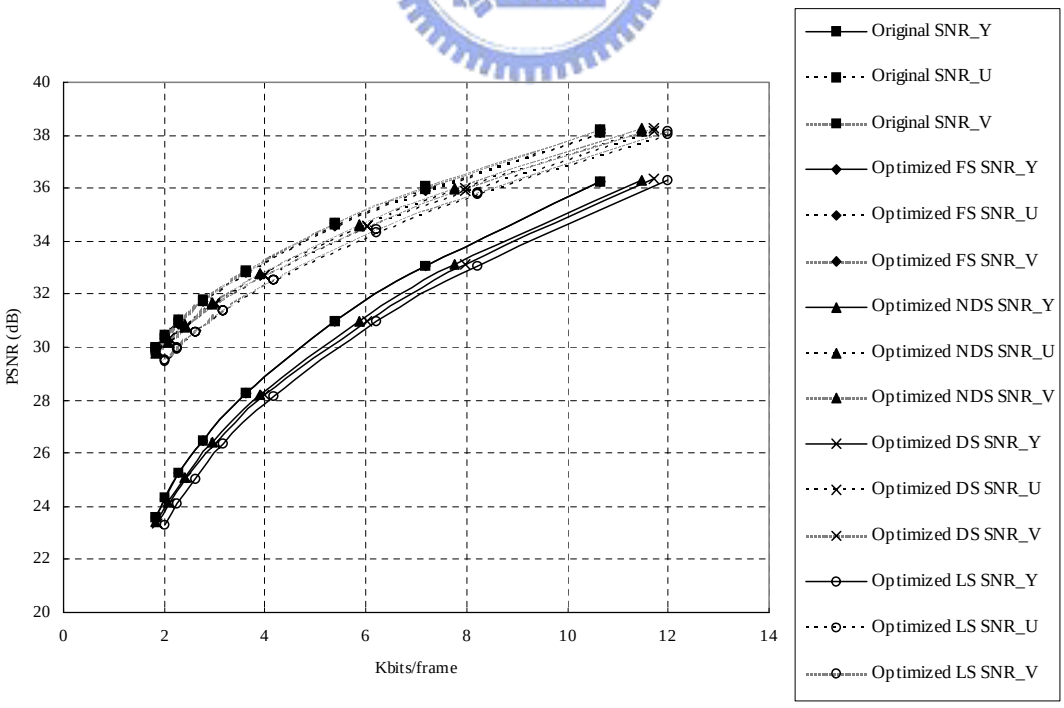


Figure 5.6: R-D performance in coding stefan_cif with shape.

Chapter 6

Conclusion and Future Work

We considered implementation of real-time MPEG-4 video encoder on personal computer. The intended application is PC-based multipoint videoconferencing system.

We introduced the parallelism processing application on video encoding in this thesis and we implemented the optimization of the MPEG-4 encoder using Intel MMX technology and its extensions, SSE and SSE2, based on the public-domain software, Microsoft MPEG-4 Visual Reference Software. In our work, we tried to find out the major complexity portion by using Vtune software tool and reducing the computation. For coding speed, we optimized the major complexity functions which are motion estimation, VOP formation, DCT/IDCT, motion compensation and quantization.

Several experiments are performed to analyze the performance of the process in chapter 5. The frame rate of encoding CIF foreman sequence with shape is approximately 10 frames per second using full search motion estimation and 30 frames per second using fast search motion estimation under release compilation mode. It is almost two and six times speed-up respectively.

For quality improvement we can do some improvements for the main projects, in the future.

1. Other Optimization

   Except full search algorithm, we also using fast motion search methods to speed up the encoder. Although the speed up performance of fast motion search methods

is good, the rate-distortion performance is a little worse than full search algorithm especially for test sequence with fast moving video objects. In order to get the more balance between speed and distortion, using other search algorithm is considerable. DCT/IDCT method we used is matrix multiplication directly, the fast DCT/IDCT method introduced from [14] can be adopted in the future.

2. Object-Based Encoding with Gray Scale Alpha Plane

   The gray scale shape information is used for hybrid scenes generated by blue screen. For synthetic scene application, we also need to optimize the object-based encoder with gray scale alpha plane.

3. Combination MPEG-4 Encoder and Segmenter

   The real-time segmenter is completed by another member in our laboratory. In the future, we will combine the segmenter and MPEG-4 encoder to achieve a whole video conference system.

4. Real-Time Decoder

   Real-time encoder is completed in this project. In order to complete point-to-point communication, real-time decoder is necessary.

Bibliography

[1] International Committee for Information Technology Standards, http://www.ncits.org/.

[2] MPEG-4 Video Group, "MPEG-4 overview — (V.21 Jeju Version)," doc. no. ISO/IEC JTC1/SC29/WG11 N4668, Mar. 2002.

[3] ISO/IEC 14496-2:2001, *Information Technology — Coding of Audio-Visual Objects — Part 2: Visual*. July 2001.

[4] T. Sikora, "The MPEG-4 video standard verification model," *IEEE Trans. Circuits Systems Video Tech.*, vol. 7, no. 1, pp. 19–31, Feb. 1997.

[5] A. Puri and A. Eleftheriadis, "MPEG-4: an object-based multimedia coding standard supporting mobile applications mobile networks and applications," *Mobile Networks Applic.* vol. 3, pp. 5–32, 1998.

[6] MPEG-4 Video Group, "MPEG-4 video verification model version 18.0," doc. no. ISO/IEC JTC1/SC29/WG11 N3908, Pisa, Jan. 2001.

[7] A. Ebrahimi and C. Horne, "MPEG-4 natural video coding – an overview," *Signal Processing Image Commun.* vol. 15., pp. 365–385, 2000.

[8] Intel, *MMX Technology — Programmers Reference Manual*. 2000.

[9] Intel, *IA-32 Intel Architecture Software Developer's Manual, vol. 1*. 2003.

[10] Intel, *IA-32 Intel Architecture Software Developer's Manual, vol. 2*. 2003.

[11] Intel, *Intel C++ Compiler for Windows User's Guide*, http://www.intel.com/support/performancetools/c/windows/sb/cs-007747.htm.

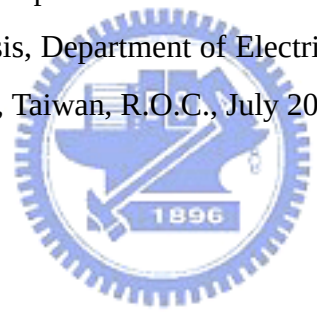
[12] Intel, *Getting Started With the VTune(TM) Performance Analyzer.* 2003.

[13] Intel "Using streaming SIMD extensions in a motion estimation algorithm for MPEG encoding," doc. AP-818, Jan. 1999.

[14] Intel "Streaming SIMD exetensions—a fast precise 8×8 DCT," doc. AP-922, Apr. 1999.

[15] S. Zhu and K.K. Ma, "A new diamond search for fast block-matching motion estimation," *IEEE Trans. Image Processing*, vol. 9, no. 2, pp. 287–290, Feb. 2000.

[16] Pei-Yun Kuo, "Real-time implementation of MPEG-4 video encoder on digital signal processors," M.S. thesis, Department of Electrical Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., July 2003.



作者簡歷

劉夢遠，民國五十八年十一月出生，民國八十二年畢業於交通大學控制工程系，民國八十九年考上交大電資學院專班電子與光電組。民國九十三年取得碩士學位。家中成員還有父母、妻子、弟弟及一女一子。

碩士研究方面，我主要研究 MPEG4 即時編碼，將其在有 Intel 處理器的 PC 平台上實現。