國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

採用以 MPEG-4 物件形式視訊編碼之視訊會議傳送端之整合

Integration of Videoconference Transmitter with MPEG-4 Object-based Video Encoding

研 究 生：蔡鎮宇

指導教授：林大衛 博士

中 華 民 國 九 十 四 年 六 月

採用以 MPEG-4 物件形式視訊編碼之視訊會議傳送端之整合

Integration of Videoconference Transmitter with MPEG-4 Object-based Video Encoding

研 究 生：蔡鎮宇　　Student: Chen-Yu Tsai

指導教授：林大衛 博士　　Advisor: Dr. David W. Lin

國 立 交 通 大 學

電子工程學系　電子研究所碩士班

碩 士 論 文

A Thesis

Submitted to Institute of Electronics

College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of Requirements

for the Degree of

Master of Science

in

Electronics Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 四 年 六 月

採用以 MPEG-4 物件形式視訊編碼之視訊會議傳送端之整合

研究生：蔡鎮宇　　指導教授：林大衛教授

國立交通大學　電子工程系 電子研究所碩士班

摘要

在本篇論文中，我們設計並實現一個在個人電腦上的物件形式視訊會議傳送端。並將此系統用來組成一個在個人電腦上的多點視訊會議系統。採用物件形式視訊編碼的最主要理由是為了節省資料量。

此傳送端的架構是由擷取、前置處理、MPEG-4 編碼器、即時傳輸規約(RTP)所組成。在影像擷取的控制上，我們引用了 VfW 的模組來達成；而在聲音擷取的控制上，我們則是引用了 MCI 模組來達成。

我們在前置處理時則能得到影像切割畫面。這一級的基本概念是將畫面與其相對應的背景相減以得到移動的物件。首先，我們估計攝影機的雜訊，並且把此結果拿來當做往後參數調整的參考。為了消除因為物體內部的平坦區域所造成的錯誤背景，首先，我們先取得一個初步的背景。接著，我們利用影像變化加上填補及收縮的技巧來取得一個粗略的物件輪廓，並利用此資訊來修正初步的背景。

接下來，我們使用一公開的程式 Microsoft MPEG-4 Software 加以修改以完成編碼與解碼系統。希望能有高速的硬體及有效率的軟體，我們採用了平行處理方式，而用 Intel 的 MMX 指令集來實現此方法。

希望我們所採用的網路傳輸協定的主要目的是為了實現多點即時系統，但我們仍能控制所傳送的封包，首先，採用了媒體傳送整合框架，雖然它隸屬於 MPEG-4 規格的一部份，但並沒有被完善發展。所以我們改採用即時傳輸規約來滿足我們的需求。

最後，在配備 Intel Centrino Pentium M 1.5 GHz cpu 及 512 MB DDR RAM 之個人電腦及 Microsoft Windows XP Professional 作業系統下的測試結果，我們所傳送的影像平均每秒 10.7 張。

Integration of Videoconference Transmitter with MPEG-4 Object-based Video Encoding

Student: Chen-Yu Tsai Advisor: Dr. David W. Lin

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

Abstract

We consider the design and implementation of an object-based videoconference transmitter on personal computer (PC). The purpose is to support multi-point virtual conferencing. The main reason that we introduce object-based video encoder into video conference is saving data amount.

The structure of the transmitter system consists of capture, pre-processing, MPEG-4 encoder, RTP (Real-time Transport Protocol). The video capture process is aided by the VfW (Video for Windows package), and the audio capture process is aided by MCI (Media Control Interface) library.

We get the segmented image in the pre-processing stage. The basic idea of the stage is a background subtraction technique. First, we estimate the camera noise and the result is used to decide the thresholds. Due to the problem of flat inner regions, we use short-term background to obtain an initial background which usually includes many flat inner regions at first. Second, a temporary foreground mask is obtained to remove the flat inner regions in the short-term background.

Next, we use the public-domain software, Microsoft MPEG-4 software, to establish an MPEG-4 coding and decoding system. Hope to use

high-processing-speed hardware and effective software to achieve real-time MPEG-4 encoder, we introduce parallel processing which we implied with Intel's MMX technology into this software.

Hope to use network protocol which goal is to realize multi-point real-time system, but we could still control the package we delivery. First, we choose DMIF, Delivery Multi-media Integration Framework, which is belonged to MPEG-4 format, as the protocol. However, the algorithm of DMIF isn't developed well and is almost given up in MPEG-4 conference. Hence we choose RTP, Real-time Transport Protocol to satisfy the need.

Finally, the average frame rate we deliver is 10.7 frames per second on our test system. The test system is based on Intel Centrino Pentium M 1.5GHz, 512 MB DDR RAM and Microsoft Windows XP Professional Version 2002.

誌謝

本論文承蒙恩師林大衛教授細心的指導與教誨，方得以順利完成。在兩年的研究所生涯中，林教授不僅在學術研究上予以學生指導，在研究態度上亦給予相當多的建議，讓我學到相當多治學的態度以及心態，在此對林教授獻上最大的感激之意。

此外，感謝通訊電子與訊號處理實驗室所有的成員，包含各位師長、同學、學長姐與學弟妹們。感謝岳賢學長、夢遠學長給予我在研究過程上的指導與建議，還有昱昇、景中、汝芩、家揚學長、崑健學長，以及最重要的伙伴，志凱，與我彼此勉勵、互相討論，讓我在這兩年的研究生涯充滿歡樂與回憶。

最後，感謝我的家人和朋友，在我的求學過程當中總是不斷的鼓勵我，提供我心靈上的支持，陪我走過我的不安、徬徨、憂愁，也與我分享我的驕傲、快樂、心得。

在此，我誠摯的對這些幫助過我的人表達我的謝意，也將我的論文獻給所有關心與幫助我的人。

蔡鎮宇

民國九十四年六月 於新竹

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We consider the design and implementation of an object-based videoconference transmitter on personal computer (PC). The purpose is to support multi-point virtual conferencing. A companion thesis [1] describes the receiver.

To support object-based video composition in the receiver, we first apply video segmentation to obtain the foreground (the conferee image). Then we send the segmented image into the MPEG-4 video encoder, and send the compressed data to the network later. The video segmentation algorithm is a modified version of that in [2], which uses a background subtraction technique. The details are reviewed and discussed in chapter 2.

The MPEG-4 standard addresses the generic coding of audio-visual objects. It consists of six basic parts. They are systems, visual, audio, conformance testing, reference software, and DMIF (Delivery Multi-media Integration Framework). We employ the MPEG-4 video encoder in this study because of its object-based coding functionality, the high compression ability and the availability of an optimized software [3].

Originally, we hoped that we could use DMIF as the network interface. Unfortunately, the support of DMIF is questionable over the last few years and the available software is difficult to use. Hence we turn to the RTP, Real-time Transport Protocol, as the network protocol. RTP is constructed on the basis of UDP, User Datagram

Protocol, which is part of the TCP/IP suite. However, it also provides error detection ability, so we could detect and correct errors, unlike UDP which provides no reliability measures.

In the receiver end of the videoconference system, after receiving the data from network, we use MPEG-4 video decoder to decode the signal, and compose the scene using all the received videos [1].

What we discuss above is the video part of videoconference system. The audio part is processed similarly. We encode it with the MPEG-4 audio encoder, and send the compressed data using RTP. At the receiver, we may decode all received audio signals and form a composite.

The general scheme at the transmitter side considered in this thesis for videoconference contains the following steps: capture, pre-processing, MPEG-4 encoder, and RTP, as shown in Fig. 1.1.

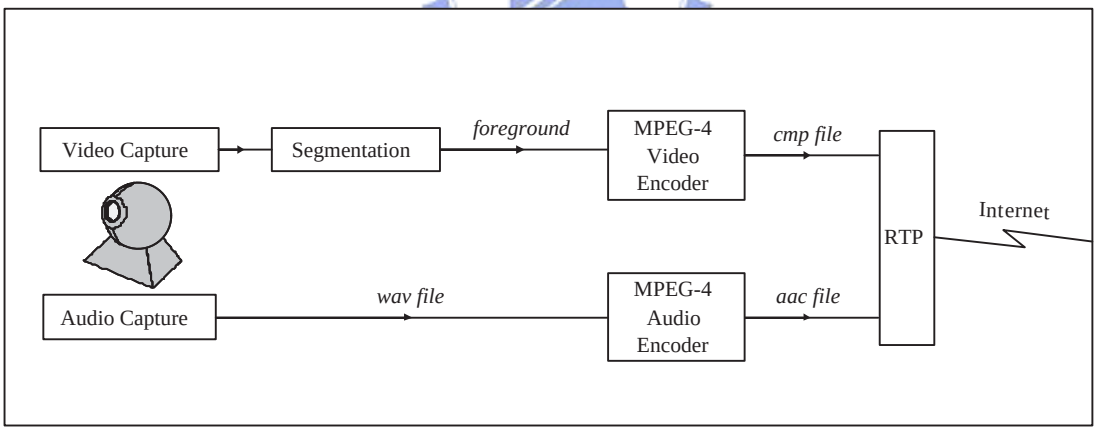


Fig. 1.1: A basic videoconference system.

1. Capture: We use a camera device to capture video, and MCI (Media Control Interface) to get audio input, both in real-time.

2. Pre-processing: With the real-time video we get from the first step, we apply segmentation to the video. The basic idea of the system is a background

subtraction technique. Then the moving objects of current frame can be obtained by extracting the region different between the current frame and background [2]. The background is obtained by gathering the stationary regions during the process of segmentation.

3. MPEG-4 encoders: We consider the optimized implementation of the MPEG-4 video encoder in software on Intel MMX processor [3]. The implementation on the code from Microsoft MPEG-4 Visual Reference Software, which is a public source for MPEG-4 encoding and decoding. We use faac software for audio encoding [34].

4. RTP: Real-time Transport Protocol (RTP) provides end-to-end network transport functions suitable for applications transmitting real-time data. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services. The data transport is augmented by a control protocol (RTCP) to allow monitoring of the data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality. RTP and RTCP are designed to be independent of the underlying transport and network layers. The protocol supports the use of RTP-level translators and mixers [5].

This thesis is organized as follows. Chapter 2 is an overview of the video segmentation. Chapter 3 introduces the MPEG-4 encoders. Chapter 4 describes the RTP. Chapter 5 considers the integration of the video conference. Experimental results of the implemented system are described in Chapter 6. Finally, Chapter 7 contains the conclusion.

Chapter 2

The Video Segmentation Algorithm

2.1 Video Segmentation Overview

Our video segmentation method is based on [2]. It is a background subtraction-based scheme. The block diagram is shown in Fig. 2.1. To start, we estimate the camera noise and some thresholds are decided according to the estimated camera noise. We use a "temporary foreground mask" and "short-term background" to generate a stationary background buffer. The object mask of each frame can be obtained by finding the difference between the current frame and the stationary background buffer. If scene change occurs, we may apply global motion estimation to generate a panorama background buffer and recover the stationary background buffer.

2.2 Two-Stage Noise Estimation

2.2.1 Influence of Noise

In this system, the image is captured by camera and then we get the initial image from the output of camera. In the process of capturing, the image may suffer from

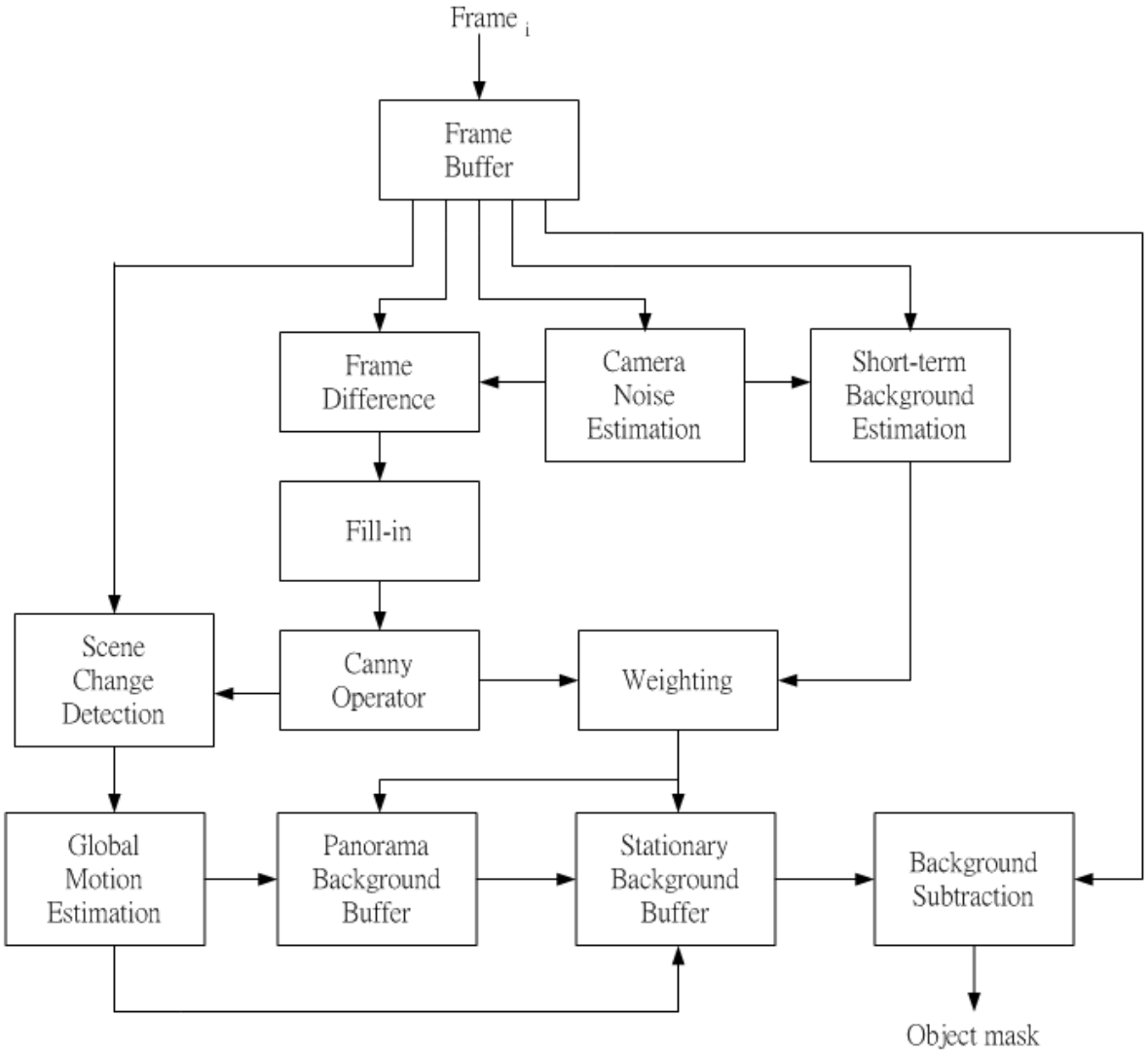


Fig. 2.1: Structure of the video segmentation algorithm (from[2]).

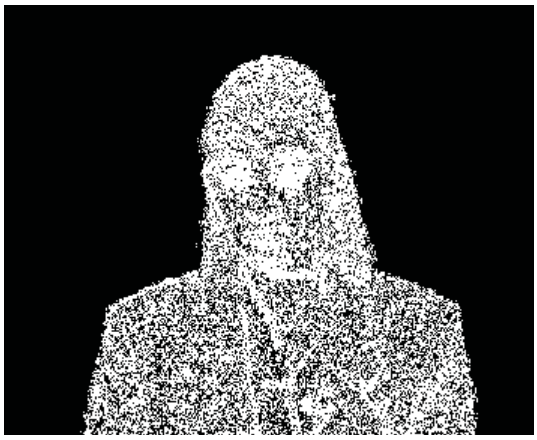


Fig. 2.2: A frame difference map of the Akiyo sequence (from[2]).

Fig. 2.3: A frame difference map of the Mother-and-Daughter sequence (from[2]).

camera noise and therefore the stationary background usually shows some difference in successive frames. In general, larger camera noise makes good segmentation more difficult to achieve. For example, when change detection-based technique [6] is applied, the frame difference map of a sequence with larger noise (e.g., Fig. 2.2) includes more background pixels than one with smaller noise (e.g., Fig. 2.3). It is apparent that the former needs more processing to obtain a more accurate object mask.

2.2.2 Motivation of Noise Estimation

In the various steps of the segmentation algorithm, we use some thresholds or parameters to make decisions and the thresholds are usually adjusted to counter the influence of noise. We can adjust those parameters manually, but this is inconvenient since we have to tune them for different situations and it usually needs some experience. In order to reduce the complexity of threshold decision, these parameters are adjusted based on the estimated camera noise level.

2.2.3 Camera Noise Model

We assume that the difference d_k of stationary pixels between successive frames obeys a zero mean Gaussian distribution $N(0, \sigma)$ with variance σ^2, that is,

$$p(d_k|H_0) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\{-\frac{d_k^2}{2\sigma^2}\}$$

where H_0 denotes the null hypothesis, i.e., the hypothesis that there is no change at pixel k. As in [6], assume that the camera noise is uncorrelated between different frames. Then the variance σ^2 is equal to twice the variance of the assumed Gaussian camera noise distribution.

2.2.4 Procedure for Noise Estimation

In order to estimate the variance σ^2, the sample space should include those pixels belonging to stationary background and exclude pixels belonging to moving objects. Our idea to discriminate the two kinds of pixel is based on the observations illustrated in Fig. 2.4 for a particular scene. The lighter pixels which represent larger differences are usually lumped together or are distributed like a strip when they are introduced by moving objects. On the other hand, the larger frame differences caused by camera noise are usually randomly distributed. Hence we reject the pixels

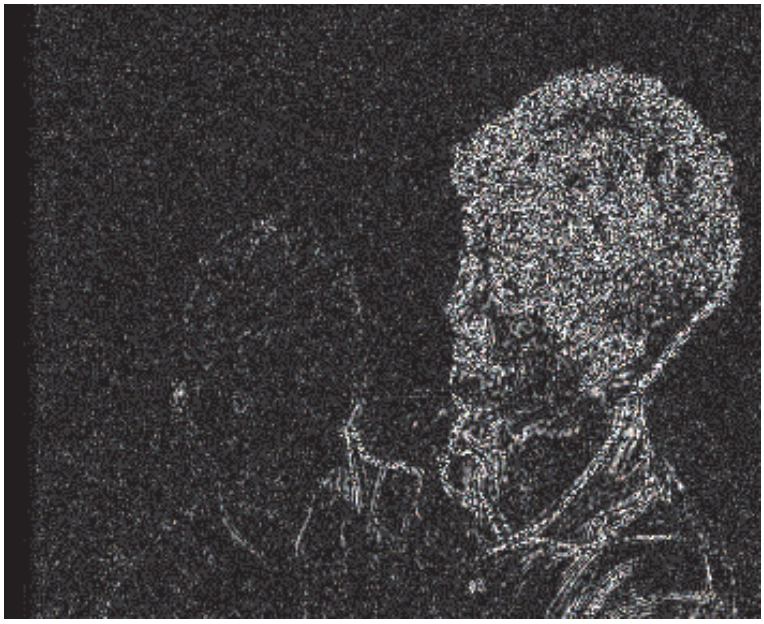


Fig. 2.4: A frame difference map in the Mother-and-Daughter sequence (from[2]).

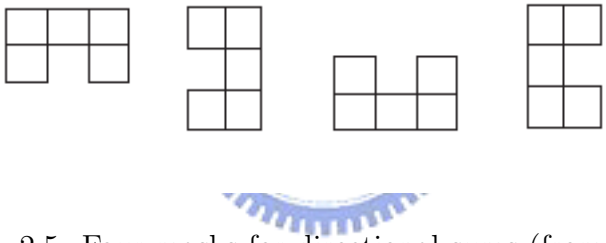


Fig. 2.5: Four masks for directional sums (from[2]).

whose neighbors have larger frame differences from the sample space during noise estimation.

We use similar masks of [7] to find out those pixels that belong to moving objects. For each pixel, we consider the four directional sums in the frame difference map as shown in Fig. 2.5. If one of the four directional sums is larger than certain threshold, we assume that the pixel belongs to a moving object.

The problem now is how we choose the threshold. Up to the present, we can only calculate the variance σ_G^2 of the frame difference of the entire frame, and therefore it is natural that we initially adjust the threshold based on σ_G^2. If one of the four directional sums of a pixel is larger than $\alpha\sigma_G^2$, where α is some suitable constant,

Fig. 2.6: Mother-and-Daughter sequence (from[2]).

Fig. 2.7: Claire sequence (from[2]).

the pixel is classified to the group of pixels influenced by moving objects. After we remove those pixels influenced by moving objects, the remaining pixels are used to estimate σ^2. In order to verify the performance of the method, the author of [2] first manually chooses the pixels belonging to stationary background to estimate the variance σ^2. In Figs. 2.6 and 2.7, the white areas are chosen to estimate σ^2 and the estimation result is regarded as exact. As we can see in Figs. 2.8 and 2.9, this method can effectively remove most pixels influenced by moving objects.

It is obvious that the results in Figs. 2.8 and 2.9 are still influenced by moving objects, because we adjust the threshold based on variance σ_G^2 of entire frame which

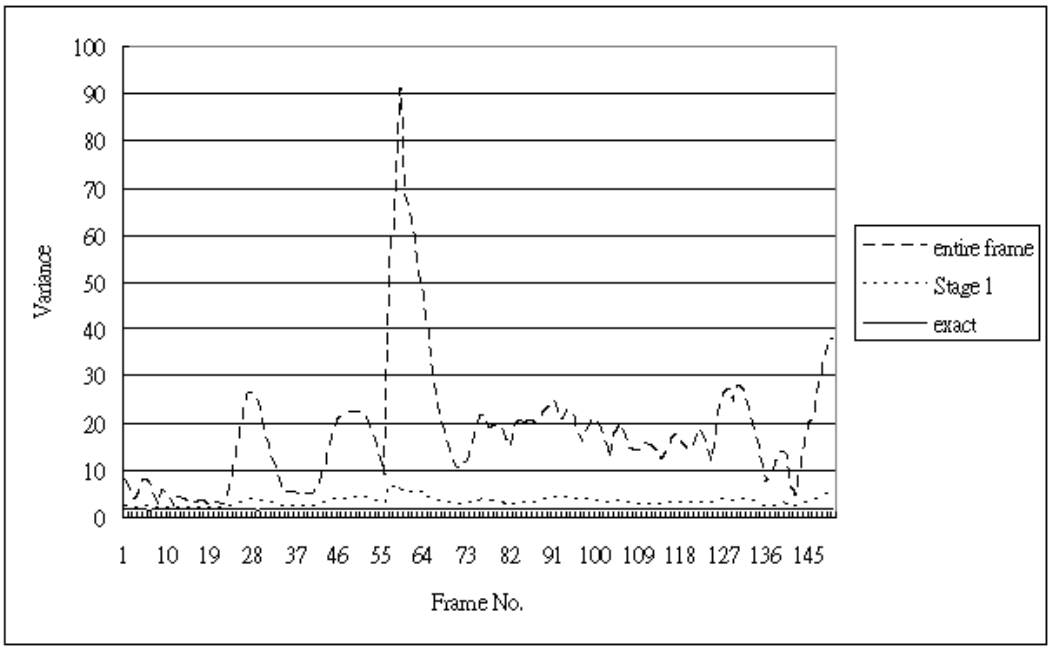


Fig. 2.8: Noise estimation of Mother-and-Daughter sequence (from[2]).

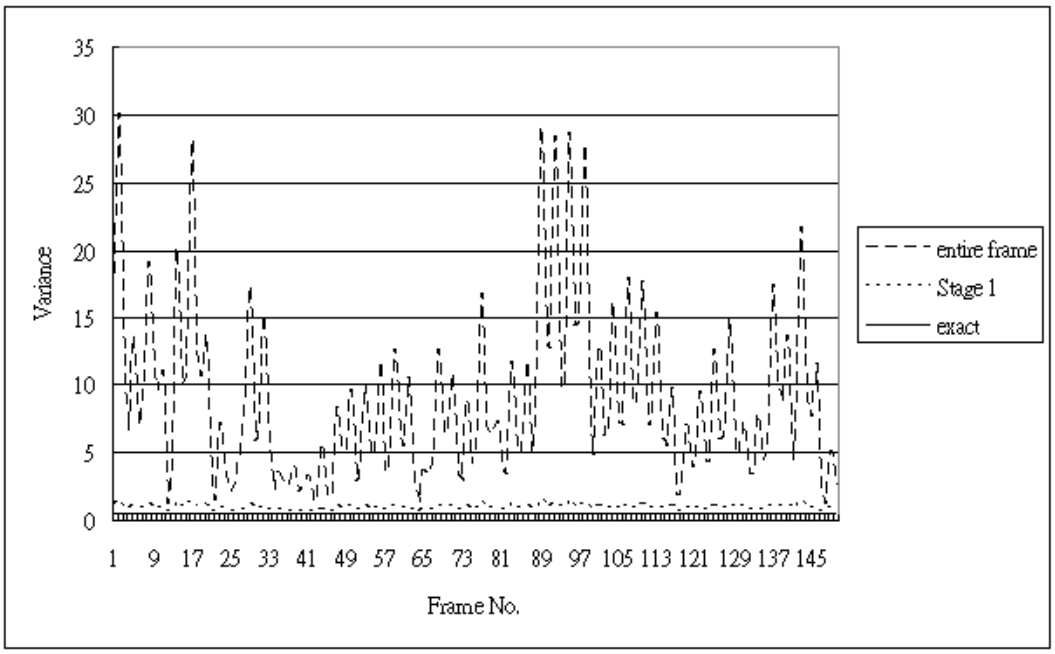


Fig. 2.9: Noise estimation of Claire sequence (from[2]).

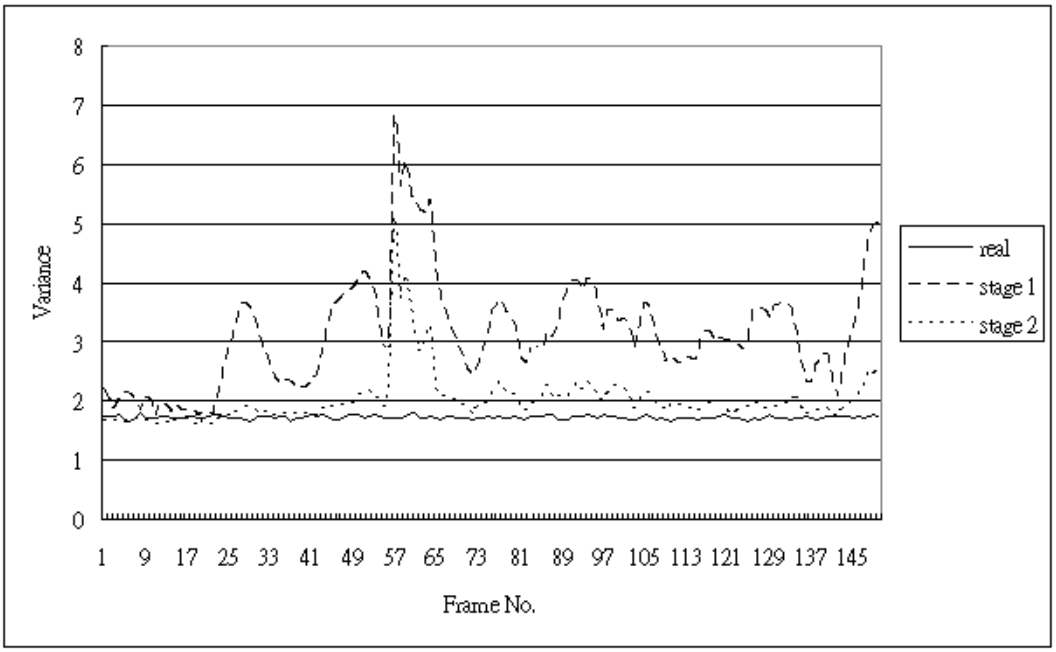


Fig. 2.10: Noise estimation of the Mother-and-Daughter sequence for the two-stage method (from[2]).

has high relationship with moving objects. In order to reduce this problem, the author of [2] considers a two-stage noise estimation method. In the first stage, he uses $\alpha\sigma_G^2$ as the threshold and get the variance σ_1^2 of stage one. In the second stage, he uses $\beta\sigma_1^2$ as the threshold and then he can obtain the final result σ_2^2 of stage two. The final result is shown in Figs. 2.10 and 2.11. It can be seen that the result of the two-stage method is closer to the exact value.

2.3 Temporary Foreground Mask

Next, we generate a temporary foreground mask and the mask is used in the stationary background buffer, scene change, and global motion estimation. In this stage, we use change detection-based technique to obtain a rough mask. The major advantage of this technique is that the frame difference can be obtained easily and fast. In contrast, many more accurate object boundary identification methods are more complex and time-consuming [8], [6]. In our algorithm, we only need to get

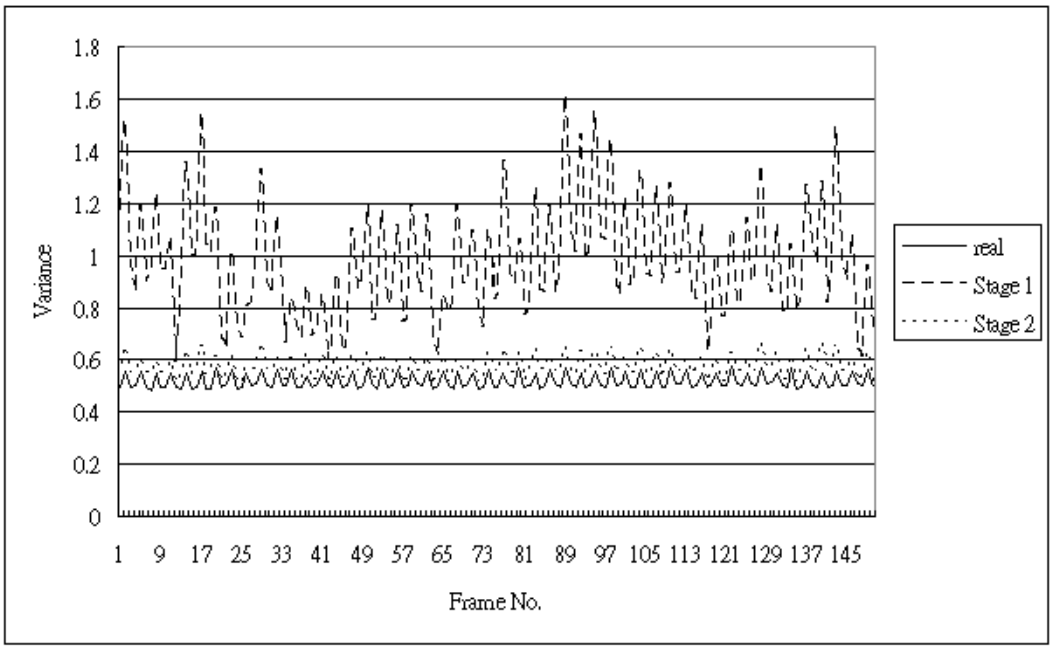


Fig. 2.11: Noise estimation of the Claire sequence for the two-stage method (from[2]).

a rough mask at this stage, and therefore those time-consuming methods are not needed, favoring the speed of the whole system.

2.3.1 Getting the Initial Object Mask

At first, we use a 3×3 window to calculate the mean of squared frame difference at each pixel. If the result is larger than threshold, the pixel is classifed as in a moving object. On the other hand, a pixel is classified as background when the result is smaller than threshold. The threshold here is adjusted based on the camera noise, that is, $\gamma\sigma^2$. An example of thresholded frame difference map is shown in Fig. 2.12. In the second step, we use the fill-in technique proposed in [8] to get a rough mask. At first it assigns the pixels between the first and last white points in each row of Fig. 2.12 to white points. This procedure is then repeated for each column and once more for each row. The step-by-step results are shown in Figs. 2.13, 2.14, and 2.15, respectively.

Fig. 2.12: Thresholded frame difference map of Claire sequence (from [2]).

Fig. 2.13: Fill-in for each row (from [2]).

Fig. 2.14: Fill-in for each column (from [2]).

Fig. 2.15: Second fill-in for each row (from [2]).

Fig. 2.16: Initial object mask of Mother-and-Daughter sequence (from [2]).

2.3.2 Refining the Initial Object Mask

Frequently, a rough mask obtained as in the previous section is enough for the following stages while it may need more improvement in some cases. In Fig. 2.16, for instance, there are two persons sitting side by side. Since the fill-in technique always marks the region between the left and right boundaries, the background between the two persons is always filled in. Although this problem can be mitigated in the following stages, it will be very helpful if the mask here is more accurate.

In this stage, we use the edge information to correct the initial mask and the Canny operator proposed in [9] is adopted to get edge information. The operator performs a gradient operation on the image by convolving it with a gaussian filter

Fig. 2.17: Edge map of Mother-and-Daughter sequence(from [2]).

and then nonmaximum suppression is applied to thin the edge. In the last step, the thresholding operation with hysteresis is used to find and link edges. The thresholding operation employs two thresholds: high-threshold and low-threshold. Pixels whose gradients are larger than the high-threshold are regarded as edges and pixels whose gradients are smaller than the low-threshold are regarded as non-edges. Pixels whose gradient are between the high-threshold and the low-threshold need to check their neighbors. If one of its neighbors is regarded as an edge pixel, then it is classified into edge. The edge map after applying the Canny operator is illustrated in Fig. 2.17 for the Mother-and-Daughter sequence. The related code of Canny operator is obtained from [10].

We refine the initial object mask by shrinking the initial mask to fit the edge map. The initial mask, edge map and shrunk mask are shown in Figs. 2.16, 2.17 and 2.18, respectively, for the Mother-and-Daughter sequence. We can see that the edge map includes many background edges and those background edges usually interfere with the final result. To reduce the influence of the background edges, we use a buffer to store them. When a position of the edge map always has an edge, we assume that there is a background edge at the position. The result after removing the background edges is shown in Fig. 2.19 and the final object mask is shown in

Fig. 2.18: Refined mask of Mother-and-Daughter sequence (from [2]).

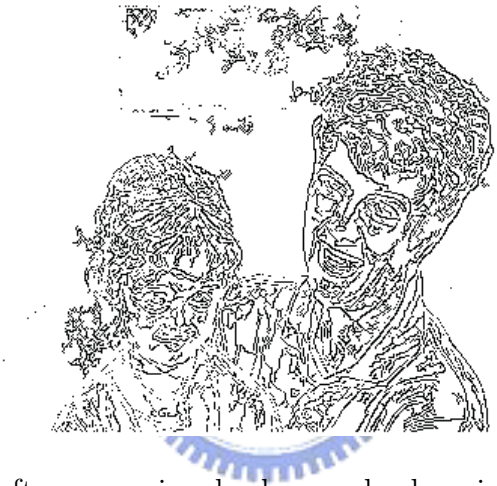


Fig. 2.19: Edge map after removing background edges in Mother-and-Daughter sequence (from [2]).

Fig. 2.20. Comparing Figs. 2.16 and 2.20, we can see that the remaining background due to background edges can be effectively removed.

2.4 Short-Term Background Estimation

The simplest way to judge whether a pixel is background is to check the frame difference at this location. Since the moving objects will cause a larger frame difference, we can assume that a pixel belongs to background when the frame difference at this location is very small from start to finish. For real-time application, we cannot wait until the whole sequence is collected before making a decision. Hence we regard a

Fig. 2.20: Final object mask of Mother-and-Daughter sequence (from [2]).

pixel as background when its frame difference is small for some consecutive frames. The major disadvantage of this method is that it is easier to make a wrong decision when the time of observation is not long enough and therefore the obtained background here is not reliable at some pixels.

We consider using six consecutive frames $f_k(i)$ $(1 \leq k \leq 6)$ as the observation window in time and a 3×3 spatial window is used to calculate the frame difference $d_m(i) = f_6(i) - f_m$ $(1 \leq m \leq 5)$ for each location i in a frame. For every location i, we calculate the mean and the variance of $d_m(i)$ $(1 \leq m \leq 5)$. If the variance is smaller than a threshold, it means the changes over the six frames are small and we can regard the pixel at location i of the sixth frame as background. The threshold here is also based on camera noise, that is, $\lambda\sigma^2$. The result is shown in Fig. 2.21 for the earlier example.

2.5 Construction of Stationary Background Buffer

In this stage, the information from short-term background estimation and temporary foreground is considered to generate the stationary background buffer.

Most of wrong decisions in the short-term are due to flat inner regions as shown in Fig. 2.22. If an object has a large flat inner region, the overlap between successive

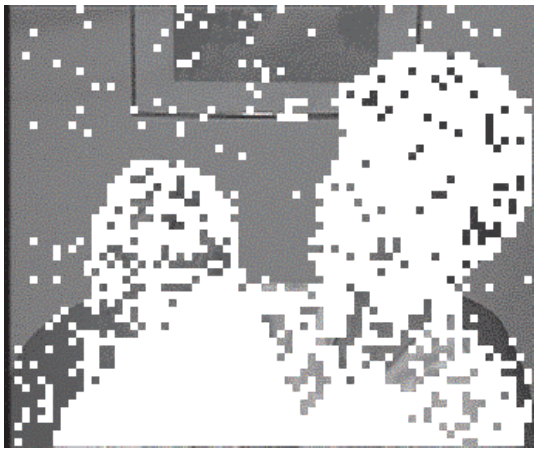


Fig. 2.21: Result of short-term background estimation (from [2]).

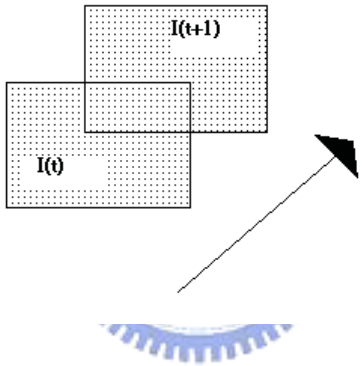


Fig. 2.22: The influence of flat inner region (from [2]).

moving objects is still stationary and is easily regarded as background. In order to reduce the influence of flat inner regions, we use the temporary foreground mask to weight every pixel before we put the short-term background into the final background buffer.

A weighting mask is shown in Fig. 2.23, where the black region represents reliable background and is given higher weight while the white region represents objects and is given zero weight. If a pixel is inside a gray region, it means the pixel is regarded as background in the short-term background and its location is inside the temporary foreground mask, it is easier to suffer from the flat inner region problem and we give it

Fig. 2.23: The weighting mask of Mother-and-Daughter sequence (from [2]).

Fig. 2.24: Final background buffer after observing 280 frames (from [2]).

lower weighting. We accumulate the weighting for every pixel position and the short-term background is put into the stationary background buffer when the accumulated weighting meets a threshold. The lower-weighted points can still become background when these points are always regarded as short-term background for a longer time, to reduce wrong decisions due to flat inner region. The final background buffer after 280 frames are observed is shown in Fig. 2.24.

2.6 Background Subtraction

The final object mask is obtained by finding difference between the current frame and the stationary background buffer. For a better result, both difference in luminance and difference in chromanace between the two frames are considered. In general, the background of the current frame may suffer from light change and shadow and the stationary background may contain some wrongly identified background pixels. Therefore, using subtraction between the current frame and the background may still leave some background pixels in the segmented foreground and there may be small holes in the segmented foreground. For this reason, we have to remove small regions after subtraction. There are two steps to remove the small regions. First, remove the small regions outside the object mask. Second, remove the small regions inside the object mask. In the first step, we check the connected length of object mask for each row and remove pixels whose connected length is less than a threshold. Then, the processing in used for each column. After we remove the regions outside object mask, we have to fill in the wrongly identified object pixels which usually looks like a hole inside object mask and the method is similar to the first step.

An example of the current frame and background is shown in Figs. 2.25 and 2.26. The result after subtraction and thresholding is shown in Fig. 2.27. The final result with small regions removed is shown in Fig. 2.28.

2.7 Experimental Performance and Analysis

We conduct experiments using a laptop computer equipped with Intel Centrino Pentium M 1.5 GHz processor and 512 MB DDR RAM. The relative computing time of every module is shown in Fig. 2.29. We use VTune software, which is developed by Intel to analyze the performance of software, to get the result. The higher time-consuming modules are stationary background buffer and small region

Fig. 2.25: Frame 255 of mother and daughter sequence (from[2]).

Fig. 2.26: Stationary background buffer (from[2]).

Fig. 2.27: Mask after subtraction and thresholding (from[2]).

Fig. 2.28: Final object mask (from[2]).

removal.

However, under the algorithm stated above, we find that for some yet unidentified reasons the algorithm fails easily in certain particular conditions, such as when there is a lattice background. Fig. 2.30 gives such an example. There is no foreground in this scene. However, from Fig. 2.30, we see that the algorithm would view the lattice part of the bag as foreground. No matter we look at the temp foreground or the temp background, the result is the same.

Besides, we find that sometimes the algorithm works well under some conditions, but sometimes it does not work well. Sometimes the algorithm even views whole scene as the foreground. After examining the algorithm carefully, we realize the key issue lies in the threshold. Two-stage noise estimation can yield more accurate noise variance, but sometimes it also causes problem. We may need to adjust the coefficients every time. If the coefficients are set improperly, then the segmentation would not perform well.

We turn the two stage noise estimation to one noise stage estimation to rectify the problem. The noise variance estimation is more rough, but on the average, the performance is better if we do not adjust the parameter every time. Hence we leave the option for the user to choose one stage noise estimation or two stage estimation.

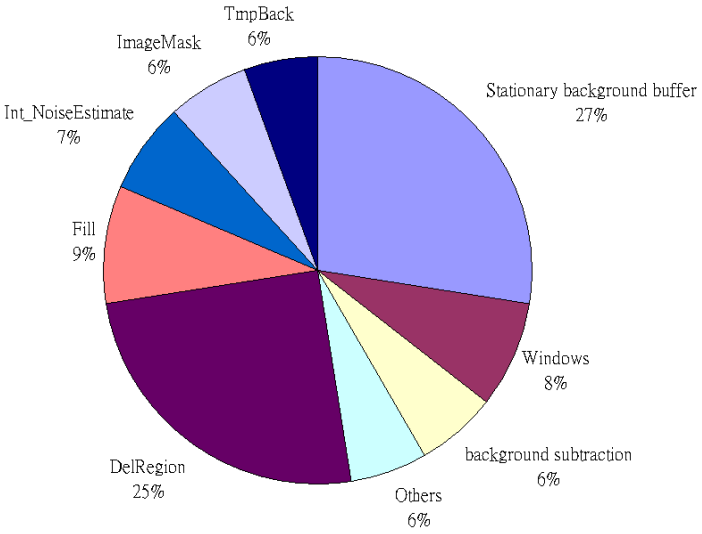


Fig. 2.29: Performance analysis of segmentation system.

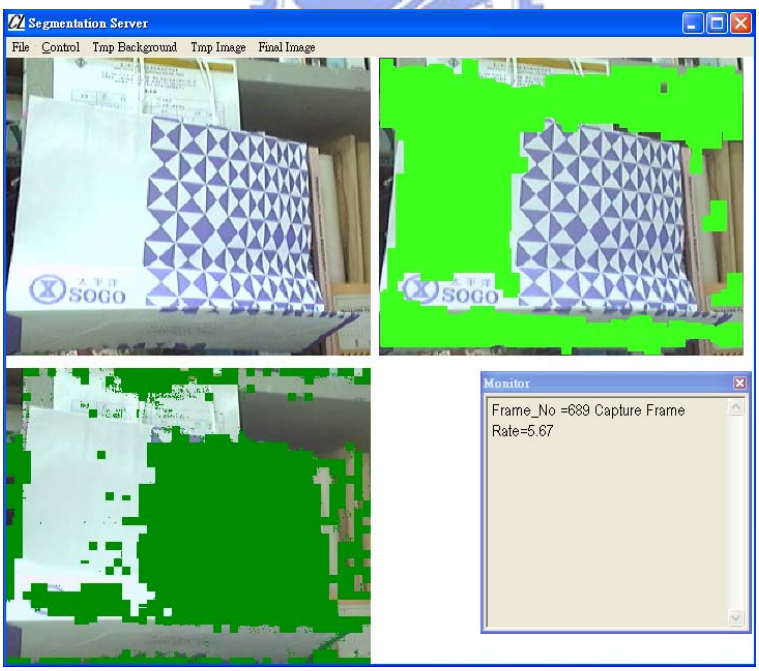


Fig. 2.30: Performance of segmentation algorithm under lattice background. Top left: input; top right: segmented foreground; bottom left: segmented background; bottom right: processing statistics.

2.8 The Modified the Video Segmentation System

2.8.1 Introduction

Fig. 2.31 shows a simplified version of the segmentation algorithm shown in Fig. 2.1. We have made three major modifications as follows.

The first is the noise estimation stage. We modify it to have only one stage rather than the original two stages as described above. Second, we just use the Y pixels component to do video segmentation system, eliminating the use of U and V components. The third is about the user interface. We only show the original video in the system and eliminate the other windows. The goal of the first part is to make the system more robust. The goal of the other two is mainly to optimize the speed.

2.8.2 Modified Noise Estimation Stage

Recall that the performance of two-stage noise estimation is better than one-stage, but we leave the choice open to the users for reason of robustness. Note that we should raise the threshold when there is more noise in the video and lower the threshold otherwise.

In Fig. 2.32, we catch the scene of our lab as the input to collect the data. From Fig. 2.32, we find an interesting fact that, in this particular case, most of the time the sectional rate of processing using two noise stages is higher than using one noise stage. (The sectional processing rate in the average processing speed of video frames over 10-frame sections.) This is because the video environment is not suitable for two noise stages. Therefore, the two stages method can not get a more accurate background, and the segmentation system may not work.

From section 130 to 160, we see that the section rate of processing of two stage

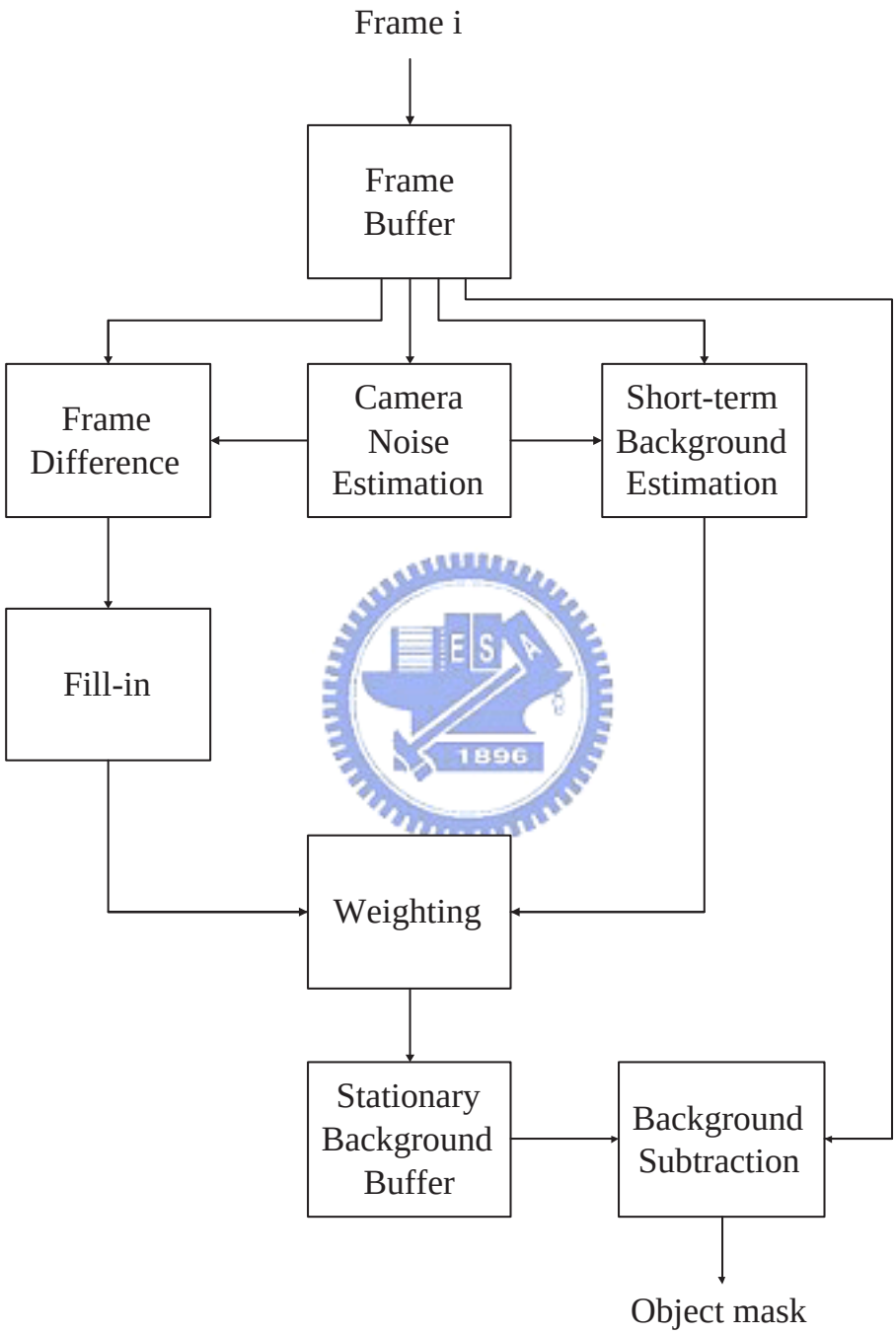


Fig. 2.31: The block diagram of the segmentation system.

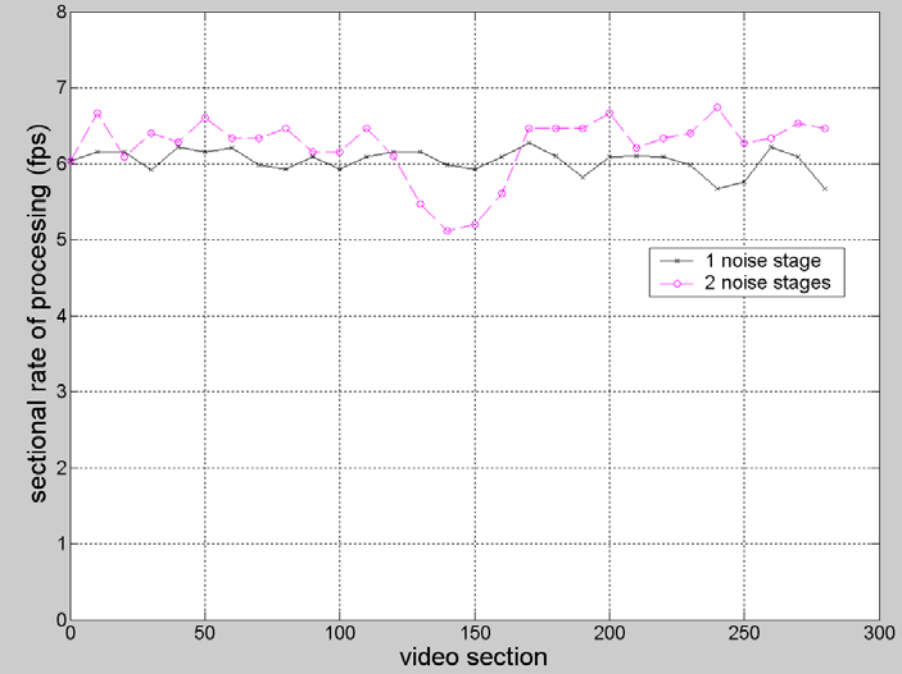


Fig. 2.32: The sectional processing rate of noise estimation

noise is lower than one stage. We note that this is because the video is stationary in this interface. Hence the segmentation system using two noise stages starts to work, but the total complexity is higher than one noise stage. Overall, the sectional rates of processing of these two alternatives are quite close in this case.

2.8.3 Using Only Y Component in Segmentation

These seems to be a bug in the original video segmentation program. After padding from YUV420 to YUV444, the U and V components are not used, even in noise estimation, the computation of temporal foreground, the computation of temporal background, the fill-in function, and the obtainment of the weighting index from the above two temporal masks.

However, the U and V components are used again to get the final image mask which is the final step in the video segmentation program. We consider it a bug because the program computes the noise variance and views it as the base for the

threshold without including U and V components. In other words, we do not use the information of U and V components in the most important factor — the noise variance.

Now we focus on the final image mask function. Fig. 2.33 shows the related code. In it, msey, mseu and msev are the absolute differences between the image pixel value and the final background mask pixel value. IMAGE_RATIO_Y and IMAGE_RATIO_UV are the thresholds we could compute as we get the noise variance. If the identification number of some pixel in the background mask is 255 (the statement: if(backFINALid[index] == 255)), that means we consider the pixel as the background in the final background mask. The program re-confirms that this pixel is in the background if the error is small (by the statement: if(msey < IMAGE_RATIO_Y)&&(mseu < IMAGE_RATIO_UV)&&(msev < IMAGE_RATIO_UV))). Thus this algorithm keeps a background pixel in the background buffer if the value does not change sharply when a new frame comes in.

Note that the final background mask OM_FinalImage is determined by the Y, U, and V components (through msey, mseu, and msev). U and V components are both determined by IMAGE_RATIO_UV, which is obtained through noise variance analysis, though we get the noise variance only through the Y component. Hence we remove "(mseu < IMAGE _RATIO_UV)&&(msev < IMAGE_RATIO_UV)" in Fig. 2.33. The side effect is that we can also eliminate the padding method and speed up the video segmentation program.

2.8.4 Modified User Interface

Fig. 2.34 shows the original interface of the video segmentation system. Besides the preview of input video, it also shows the object mask and the stationary background buffer. The display of the object mask and the stationary background buffer much uses the SetPixel function. It consumes much computing resource. Hence we allow

```
void ImageMask( unsigned char *OM_FinalImage, unsigned char *backFINALid,
                unsigned char *ref6, unsigned char *Utmp, unsigned char *Vtmp,
                unsigned char *backFINAL, unsigned char *backFINALu, unsigned char
               *backFINALv,    int IMAGE_RATIO_Y, int IMAGE_RATIO_UV)
{
   int uv1,uv2;
   int width = 352 ;
   int msey,mseu,msev ;
   int index ;
   //using 3x3 block as basic block
       for(uv1=1;uv1<287;uv1++)
      {
          for(uv2=1 ;uv2<351;uv2++)
          {
            msey = 0 ;
            mseu = 0 ;
            msev = 0 ;

            index = uv1*352+uv2;

            if(backFINALid[index] == 255)
            {
                msey = abs(ref6[index]-backFINAL[index] );
                mseu = abs(Utmp[index]-backFINALu[index]) ;
                msev = abs(Vtmp[index]-backFINALv[index]) ;


              if((msey < IMAGE_RATIO_Y) &&(mseu < IMAGE_RATIO_UV)&&(msev < IMAGE_RATIO_UV))
                                    {   //background
                                            OM_FinalImage[index]  = 255 ;
                                    }
                            }
                    }
          }
}
```

Fig. 2.33: Related code of final image mask function.

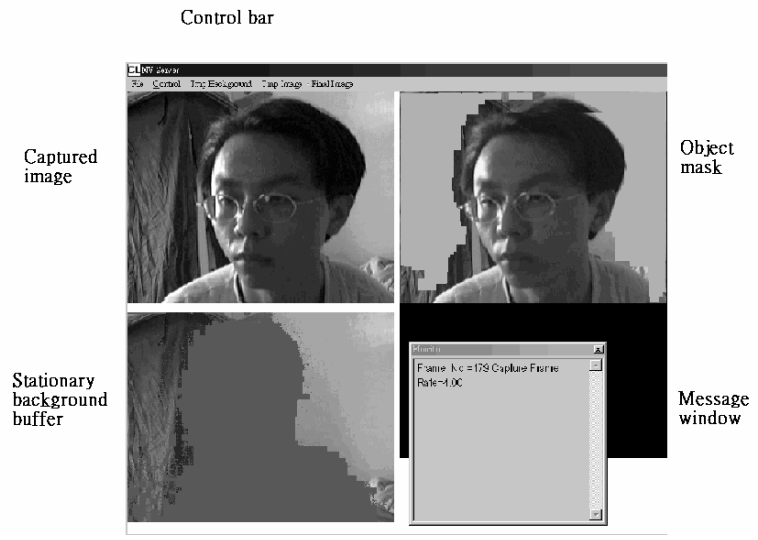


Fig. 2.34: The entire application interface of video segmentation system (from [2]).

Table 2.1: Simulation Result about Windows

Function	Final background window	Final foreground window	Total software
Clockticks	885	1063	6351

the user to remove the displaying of the object mask and the stationary background buffer.

2.8.5 Performance of Modified Interface

Fig. 2.35 shows the performance with the modified user interface in debug mode. The mean sectional rate of processing of the original model is 5.088fps, and the variance is 0.041072; the mean sectional rate of processing with elimination of one window (final background window) is 6.183, and the variance is 0.03503; the mean sectional rate of processing with elimination of two windows is 7.568, and the variance is 0.02526. Compared with the original model, the efficiency of eliminating one window is enhanced by $(6.183 - 5.088)/5.088 = 21.5\%$, and that of eliminating two windows is $(7.568 - 5.088)/5.088 = 38.9\%$. Table 2.1 shows the separate window clockticks and the total segmentation system clockticks, where 1 clocktick represents 1ms.

Fig. 2.36 shows the result under the release mode. The mean sectional rate of processing of the original model is 5.6133fps, and the variance is 0.011315; the mean sectional rate of processing with eliminating one windows (final background window) is 7.4793, and the variance is 0.04000; the mean sectional rate of processing with eliminating two windows is 9.9567, and the variance is 0.041706. Therefore the efficiency of eliminating one window is enhanced by $(7.4793 - 5.6133)/5.6133 = 33.2\%$; the efficiency of eliminating two windows by $(9.9567 - 5.6133)/5.6133 = 77.4\%$.

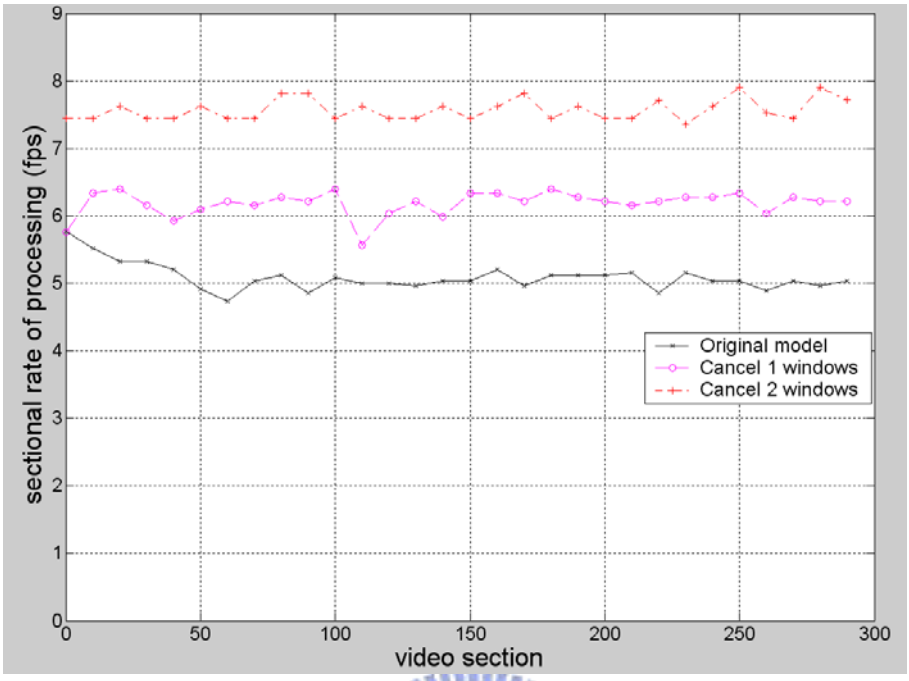


Fig. 2.35: Performance with modified user interface in debug mode.

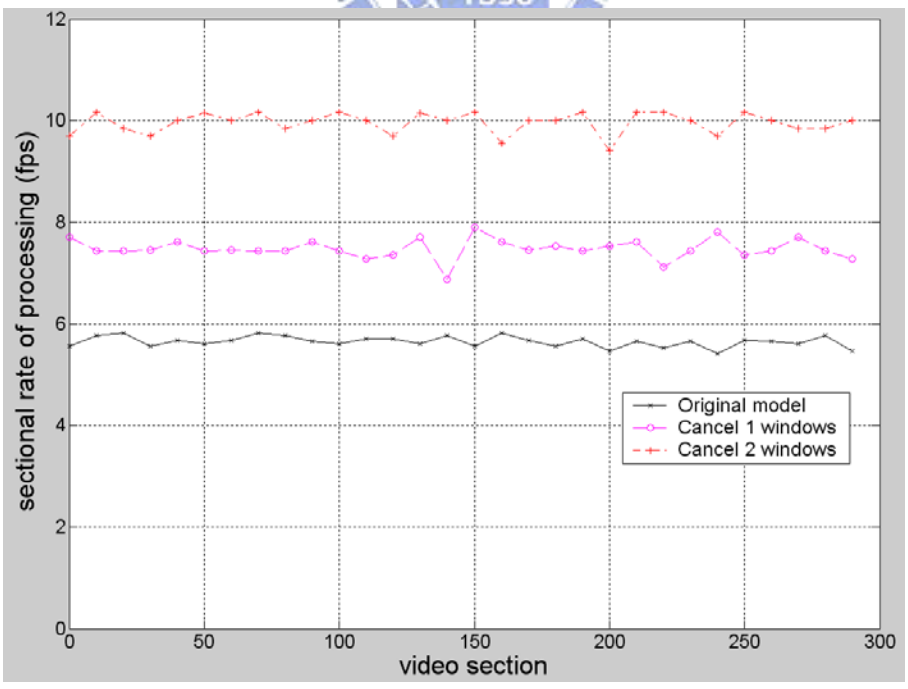


Fig. 2.36: Performance with modified user interface in release mode.

Chapter 3

Overview of MPEG-4

3.1 Introduction

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. 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 [11]. 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).

3.2 Organization of the MPEG-4 Standard

The MPEG-4 standard addresses the generic coding of audio-visual objects, as illustrated in Fig. 3.1. It (ISO/IEC 14496) consists of the following basic parts. The following text is mainly taken from [12], [11].

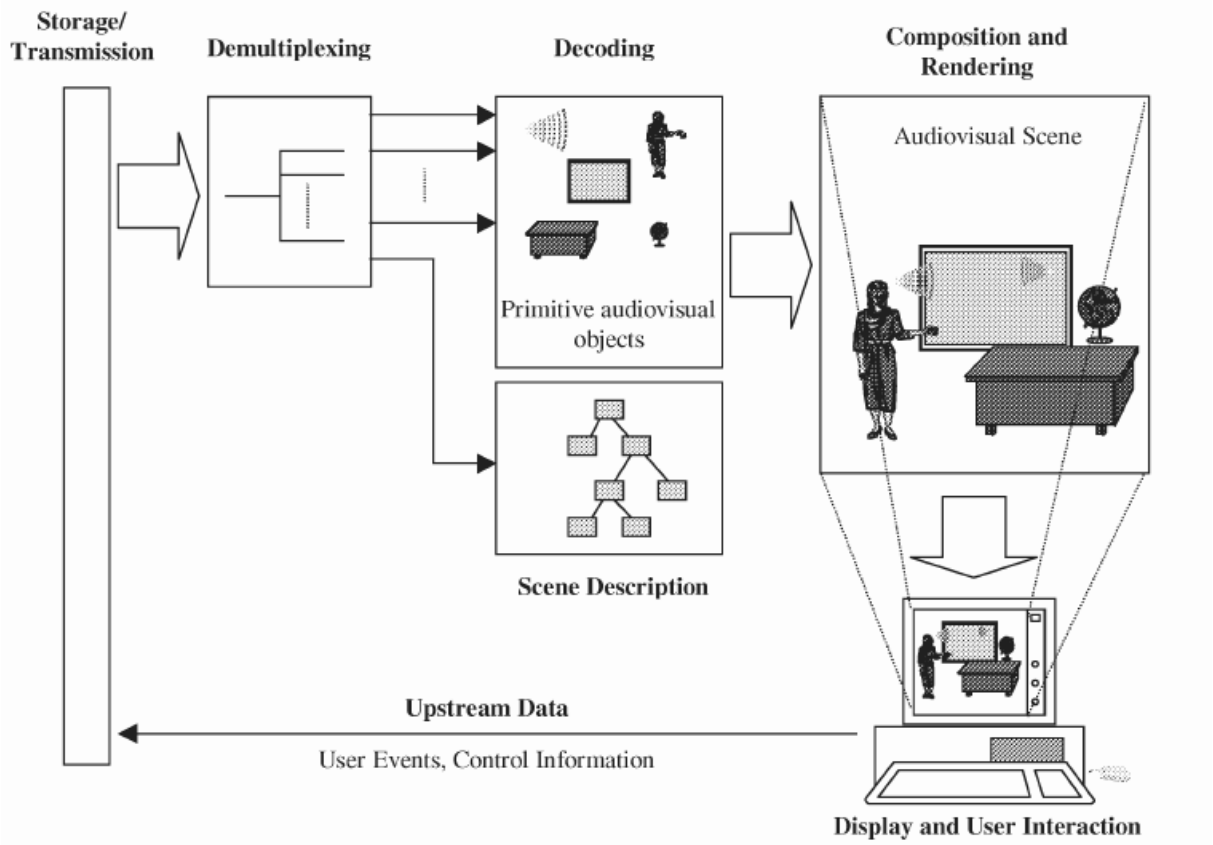


Fig. 3.1: A high level view of an MPEG-4 terminal (from[13]).

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 bit-rate 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.

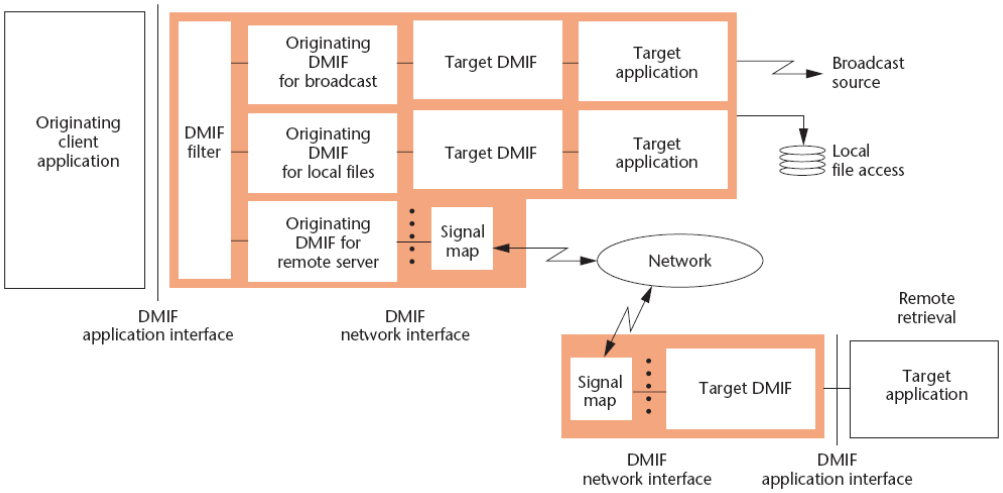


Fig. 3.2: DMIF communication architecture (from [30]).

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.

Fig. 3.2 introduces the architecture of DMIF communication, and Fig. 3.3 introduces the overall MPEG-4 streaming system with DMIF inside. However, the available software has not been developed well. For example, the publicly available IM1 software is not easy to use. Further, the activities in DMIF have waned in recent years.

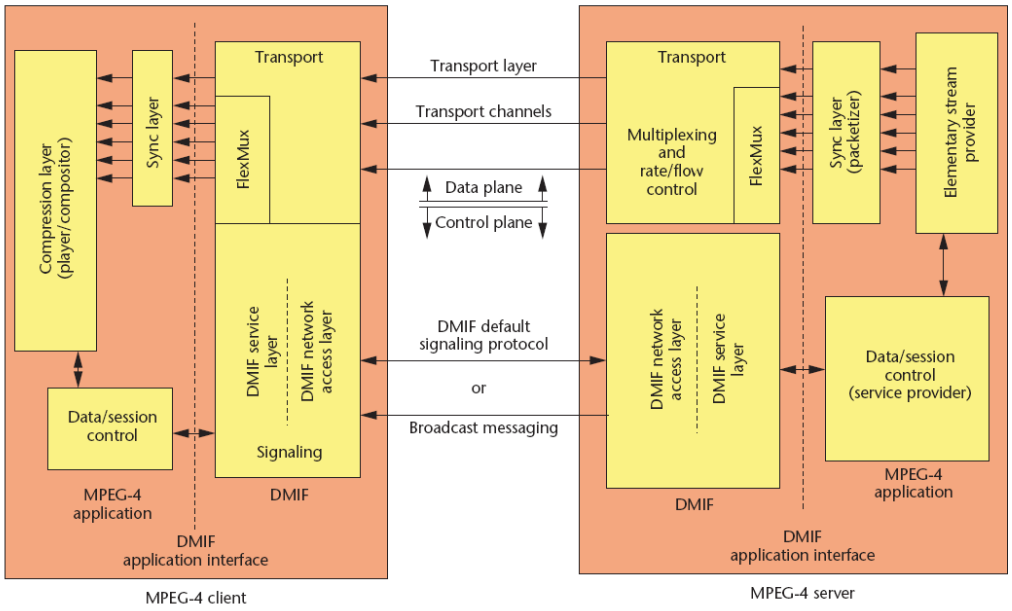


Fig. 3.3: MPEG-4 streaming system architecture (from [30]).

Originally we would like to use DMIF as the network interface because the goal of DMIF is for object-based transmission. However, because of the above reasons we decide to turn to RTP for the network interface.

3.3 MPEG-4 Audio Coding Overview (from [29])

MPEG-4 is based on the notion that the audio part of the audiovisual scene presented at the receiver is composed of one or more so-called audio objects. Different audio compression tools (codecs) are available to enable an efficiently coded representation of the audio objects in a scene. Natural audio objects, such as recorded speech and music, can be coded at bitrates typically ranging from 2 kbit/s (for narrow-band speech) to 64 kbit/s/ch (for CD quality music) using parametric speech coding (HVXC), CELPbased speech coding, parametric audio coding (HILN) or transform-based general audio coding (AAC, TwinVQ). The natural audio and speech coding

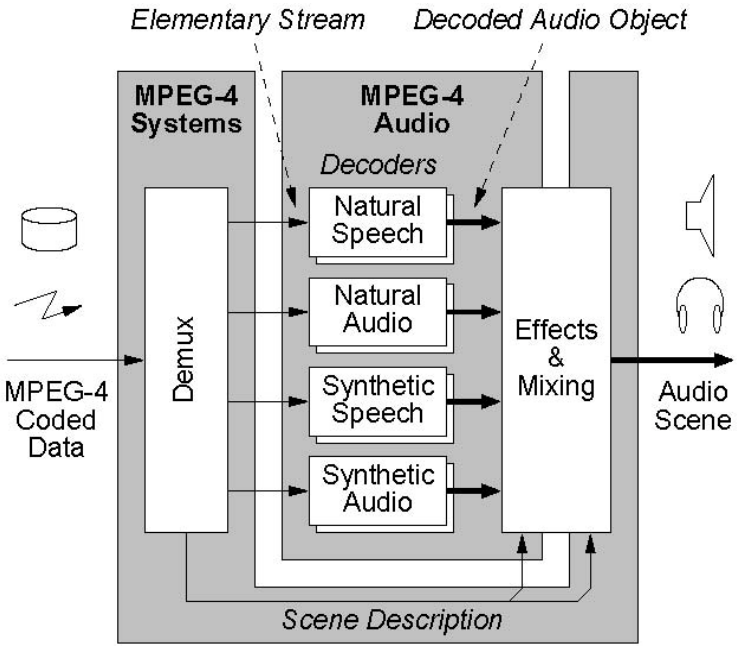


Fig. 3.4: Block diagram of a complete MPEG4 Audio decoder (from [29]).

tools support bitrate scalability, also known as embedded coding. In addition, the parametric coding tools also provide speed and pitch change functionality in the decoder. Synthetic audio objects can be represented using a TextToSpeech Interface (TTSI) or the Structured Audio (SA) synthesis tools. Other uses of the SA tools are adding effects, like reverberation, and mixing different audio objects to compose the final "audio scene" that is presented to the listener.

Fig. 3.4 shows the block diagram of a complete MPEG-4 Audio decoder. It includes the decoding tools for the audio objects defined in Part 3 (Audio) of the MPEG-4 Standard as well as bitstream demultiplexing and scene composition defined in the Systems part.

The information needed to decode an audio object at the decoder is conveyed by means of a so-called elementary stream (ES), as shown in Fig. 3.5 In case of bitrate scalable configurations, a base-layer ES and one or more enhancement-layers ES(s) are used. The initial ES Descriptor contains an AudioSpecificConfig element, which

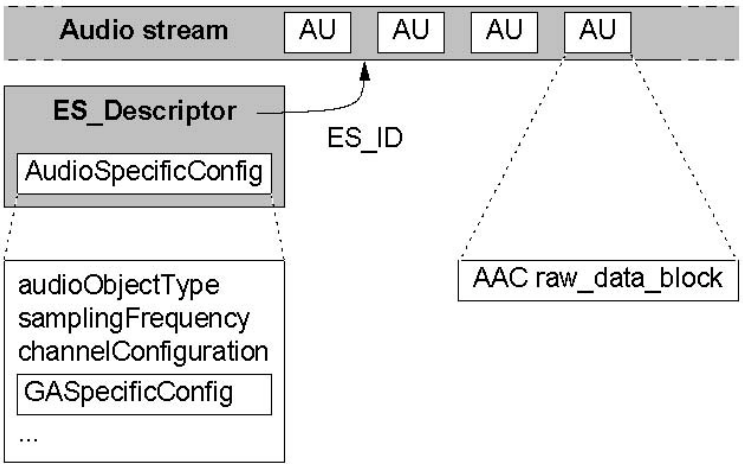


Fig. 3.5: MPEG4 Elementary Stream (ES) conveying an audio object (from [29]).

carries the audio object type (AOT) and further information required to instantiate the required audio decoder. Using the ES ID, the ES Descriptor points to the stream of the actual compressed audio data, which is a sequence of socalled access units (AU), i.e., decodable bitstream frames.

3.4 MPEG-4 Video Coding Overview (from [14])

3.4.1 Structure of 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. Fig. 3.6 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

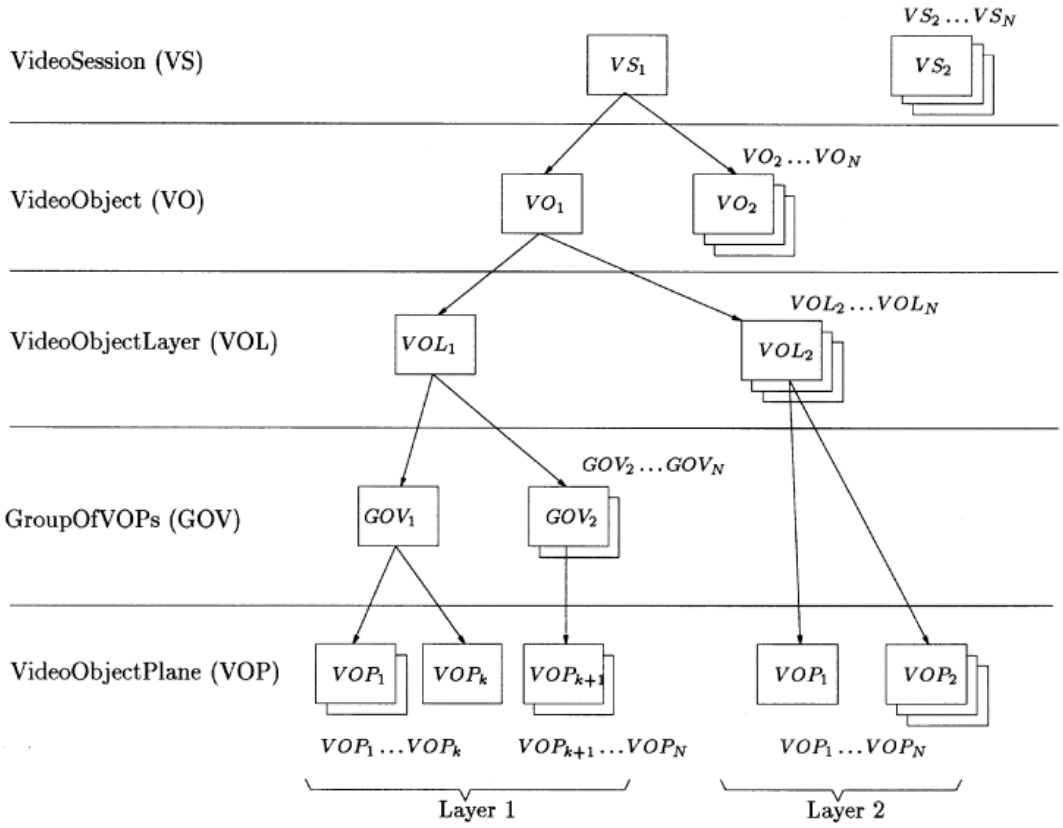


Fig. 3.6: Logical structure of coded video data (from [15]).

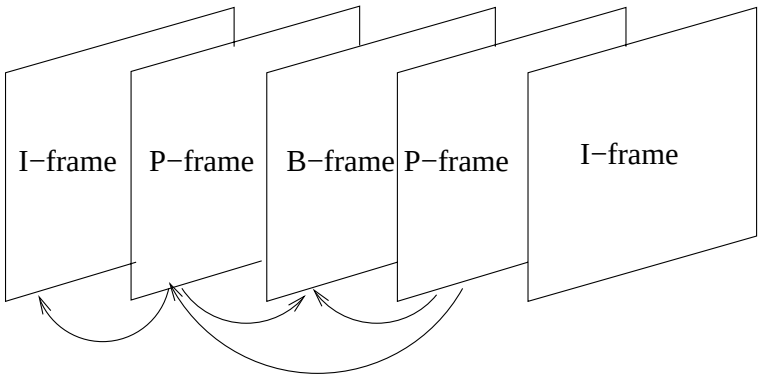


Fig. 3.7: Types of VOP.

objects. The complete MPEG-4 scene which may contain any 2-D or 3-D natural or synthetic objects.

- 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 3.7 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).
    4. A Sprite (S) VOP is a VOP for a sprite object or a VOP which is coded

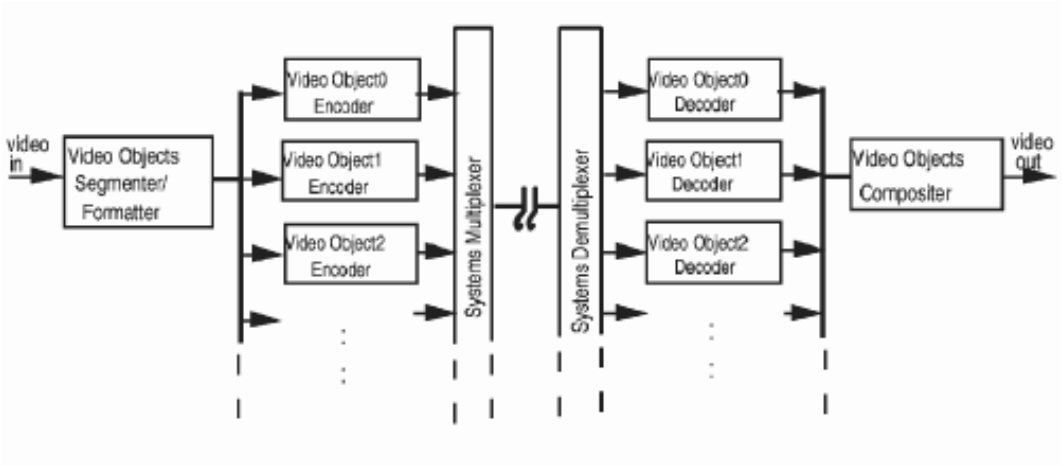


Fig. 3.8: High level structure of VO based encoder (from [13]).

using prediction based on global motion compensation from a past reference VOP.

3.5 MPEG-4 Video Texture Coding (from [13], [16] and [15])

Fig. 3.8 shows a high level logical structure of a VO based encoder. Fig. 3.9 presents the internal structure of the VO encoder. The main components are VO segmenter/formatter, VO encoders, system multiplexer/demultiplexer, VO decoders and VO compositor.

3.5.1 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

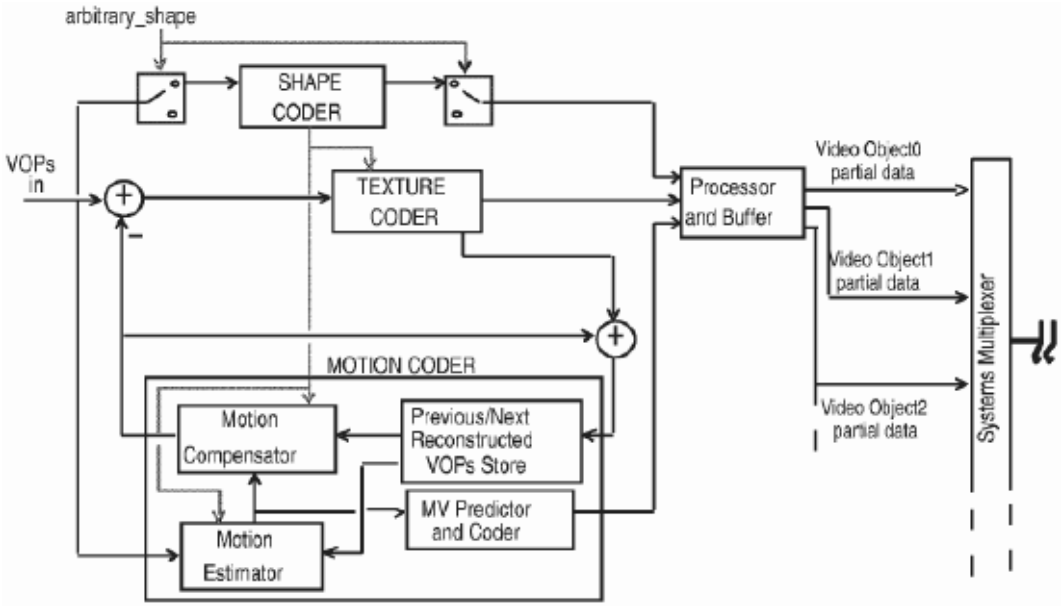


Fig. 3.9: Detailed structure of VO encoder (from [13]).

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.

3.5.2 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.

3.5.3 Motion Coder

There are four types of VOPs (see Fig. 3.7 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.

3.5.4 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.

3.6 MPEG-4 Video Encoder Optimization for Intel's MMX Technology

3.6.1 Overview of Intel's MMX Technology (from [17], [18] and [19])

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

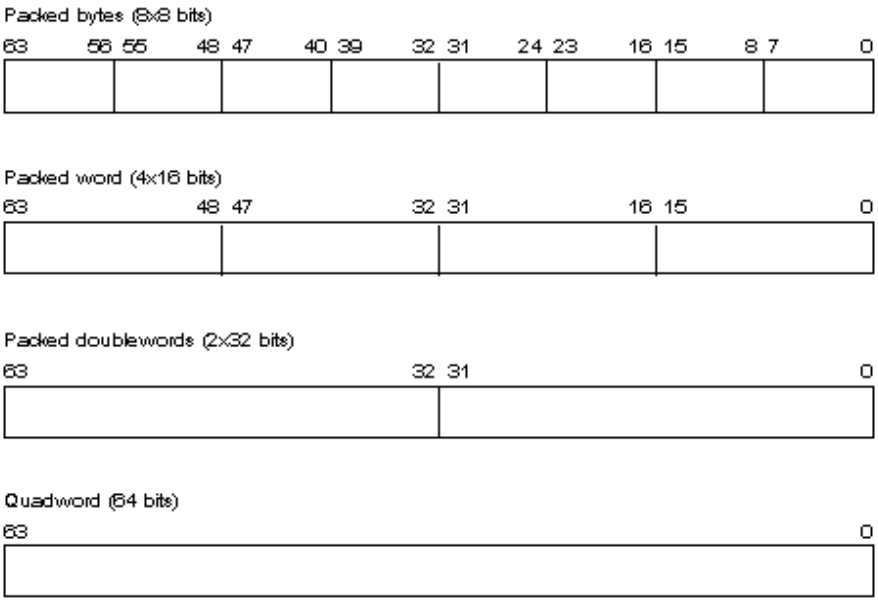


Fig. 3.10: MMX packed data types (from [17]).

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.

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

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.

MMX Data Types

The MMX technology introduced the following four new 64-bit data types as illustrated in Fig. 3.10:

- 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.

MMX Registers

The MMX register set consists of eight 64-bit registers as shown in Fig. 3.11, 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.

3.6.2 Introduction to the MMX Instruction Set

This section provides an overview of MMX instruction groups. Detailed information can be found in [19]. The MMX instructions are grouped into the following categories:

- Data transfer
- Arithmetic
- Comparison

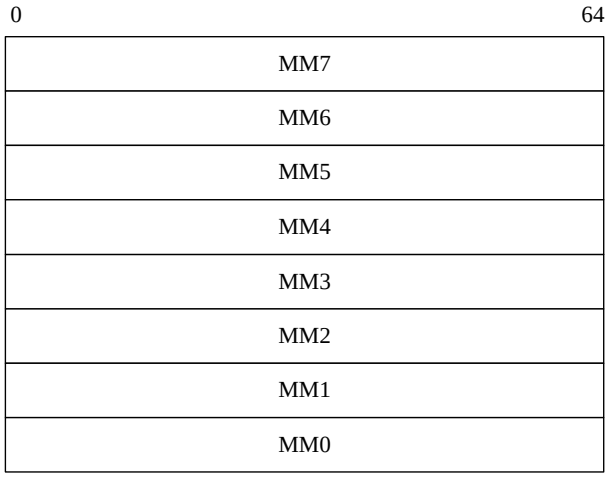


Fig. 3.11: MMX register set.

- 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

Table 3.1: MMX Instruction Set Summary [3].

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		

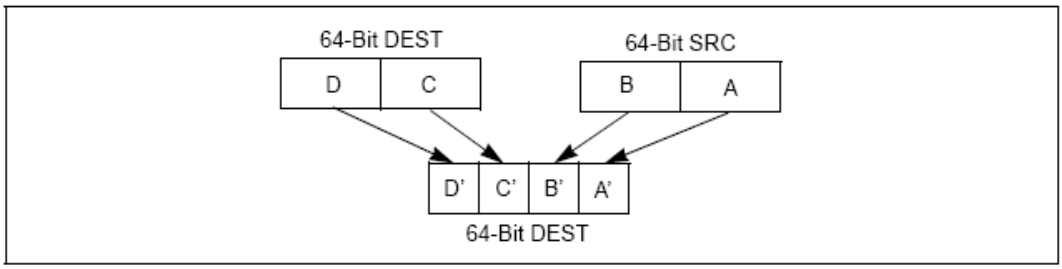


Fig. 3.12: PACKSSDW instruction operation using 64-bit operands ([19]).

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.

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 Fig. 3.12 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.

Logical Instructions

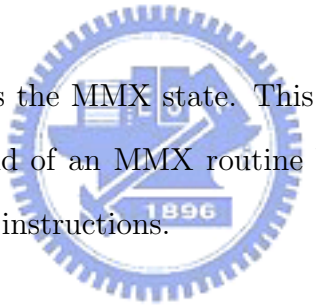
The logical instructions perform bitwise logical operations on 64-bit quantities. For example, we can generate a zero register in MM0 by using "PXOR mm0, mm0."

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.



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.

3.6.3 SSE and SSE2, Later Extensions of MMX Technology (from [3])

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

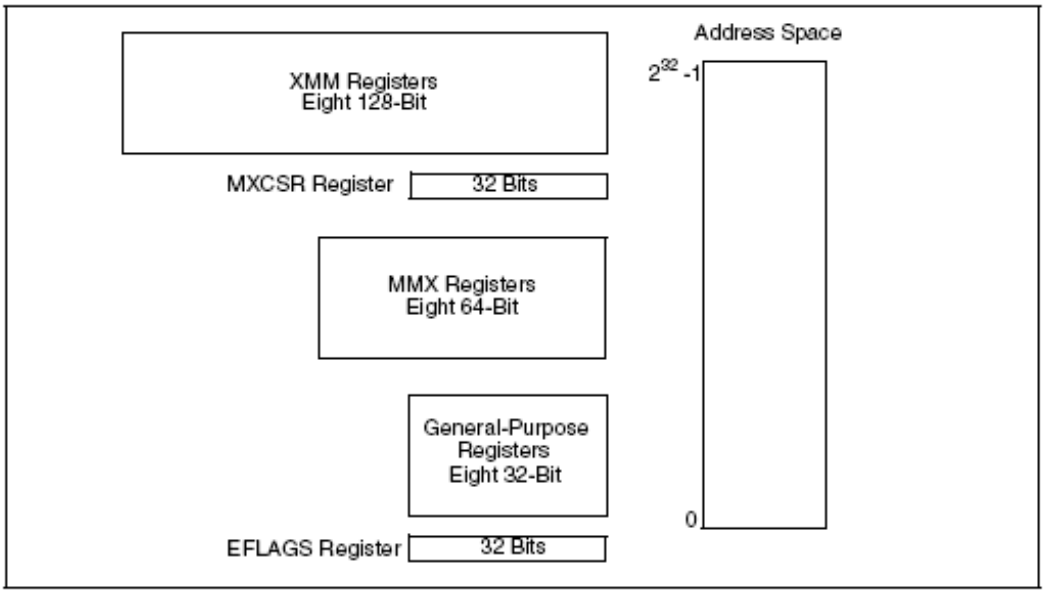


Fig. 3.13: SSE execution environment (from [18]).

- 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.

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 [18]

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 [18].

3.7 Microsoft MPEG-4 Visual Reference Software (from [31])

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.

The source files and directories are arranged as shown in Table 3.2, and Table 3.3 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 employ an optimized version of this software in our system [3].

3.8 Code Acceleration of MPEG-4 Encoder

Several methods are exploved in [3] to accelerate the encoder. One of them is to use the MMX technology to modify the most computation-intensive operations of the encoder while the other is at the algorithm level, modifying the video coding algorithm for decreasing of complexity.

3.8.1 Example of Optimization

Fig. 3.14 shows that the most computation is spent on functions relating to motion estimation. The major functions of motion estimation are summarized in Table 3.4 and we also show the percentage complexity of each function with respect to the encoder and to the motion estimation. Therefore, we take blkmatch16 optimization

Table 3.2: Source Files and Directories Arrangement of MPEG-4 Video Reference Software (from [31])

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()

as the example [3].

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. The hotspots of blkmatch16 are shown in Fig. 3.15.

As we can see, the most complexity is to calculate SAD (sum of absolute differences) at integer pixel displacements. The modified code that uses MMX instructions for the SAD kernel is shown in Fig. 3.16.

The major instruction used 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

Table 3.3: Funtionalities of the Microsoft MPEG-4 Video Reference Software (from [31])

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.

Table 3.4: Major Functions of Motion Estimation (from [3])

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%

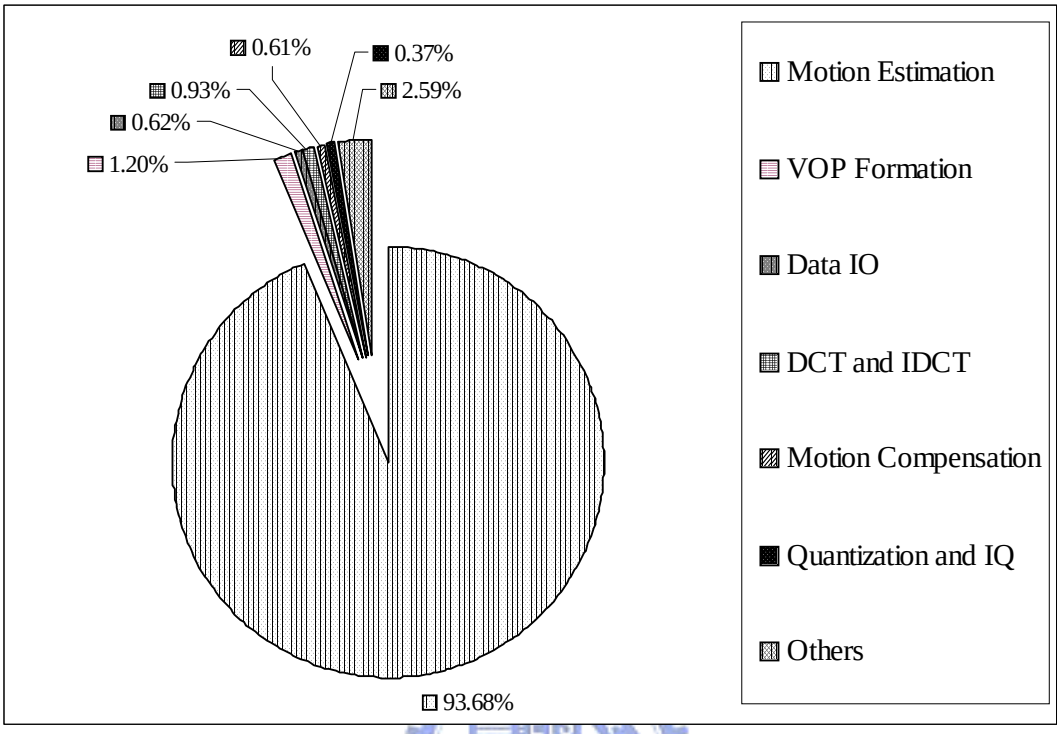


Fig. 3.14: Breakdown of execution time in Microsoft MPEG-4 Visual Reference Software (from [3]).

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
}	

Fig. 3.15: Code segment of hotspots of blkmatch16 (from [3]).

```
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;
}
```

Fig. 3.16: Revised code segment of SAD kernel of integer pixel motion search (from [3]).

in the destination operand. 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 [20], this premature breakout mechanism will decrease the efficiency. But experiment reported in [3] shows 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.

3.9 Conclusion in Optimization

The results after optimization are shown in Figs 3.17 and 3.18 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.

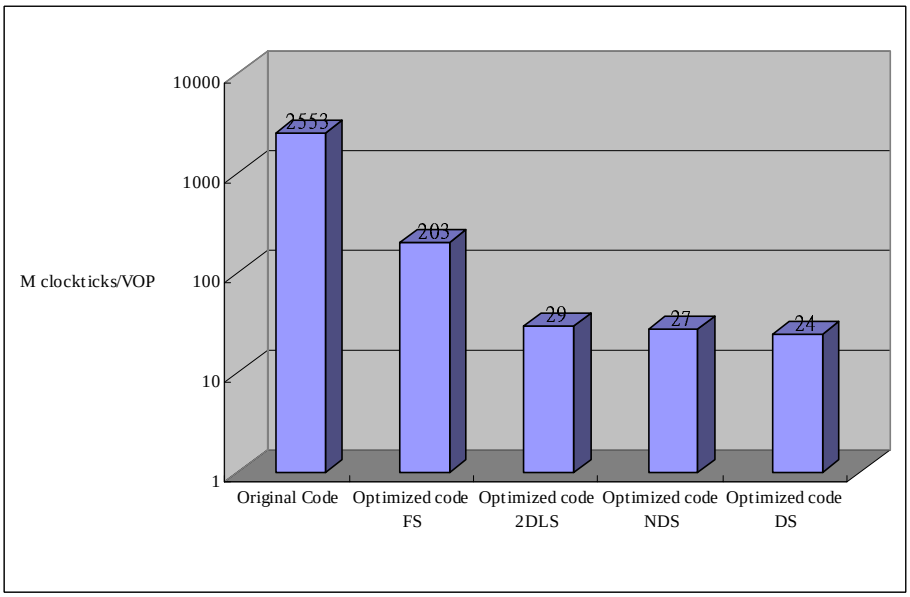


Fig. 3.17: Comparison between original reference software and optimized code in execution time for motion estimation (from [3]).

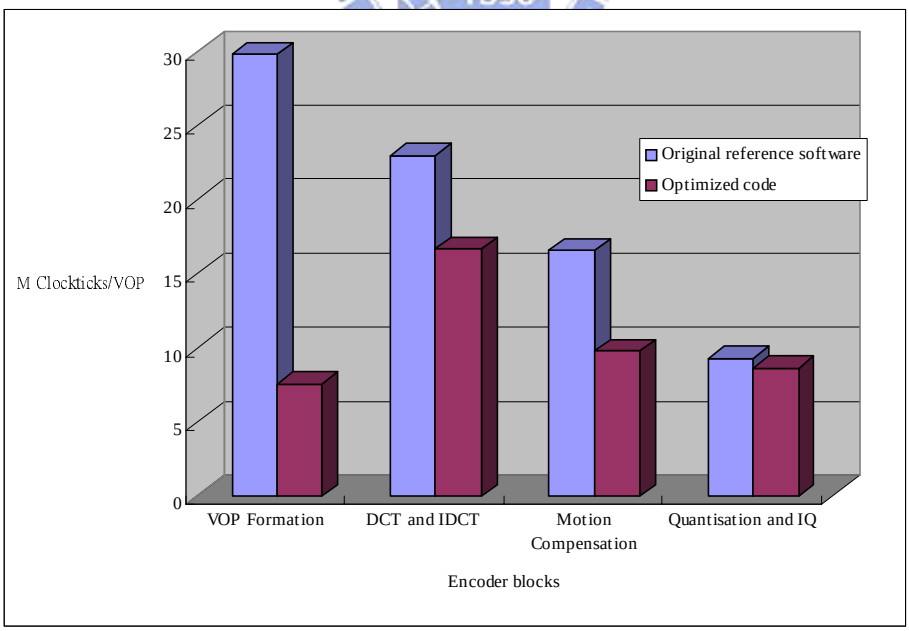


Fig. 3.18: Comparison between original reference software and optimized code in execution time for other encoder blocks (from [3])

Table 3.5: Clockticks Complexity Analysis of MPEG-4 Encoder

Function	Clockticks	Ratio in total system (%)
whole system	50097	100
apply (belonged to CBlockDCT)	3411	7
blkmatchForShape	2230	5
quantizeInterDCTcoefMPEG	1934	4
biInterpolateY	1418	3
blkmatch16WithShape	1353	3
blockmatch8	1166	3

3.10 Performance Analysis of MPEG-4 Encoder for Videoconference

Because we use binary shape coding in our videoconference system, the complexity breakdown is a little different from the above. Fig. 3.19 illustrates this statistics. We see that the complexity relatively evenly distribute over many functions. There is no major bottleneck in the MPEG-4 encoder. However, since MPEG-4 video encoding occupies a significant proportion of the computational complexity of the overall videoconference system, we check to see whether additional optimization can be performed.

Unfortunately, using VTune to track the encoder, we find that the most time-consuming parts are already optimized by using MMX assembly code. In other parts of the MPEG-4 encoder, the time-consuming instructions not yet optimized are almost assign instructions. The optimization of these parts are left to potential future work. Table 3.5 illustrates the clockticks-based complexity analysis, where 1 clockticks represents 1ms of execution time. We catch the scene of our lab as the input to collect the data and there are totally 830 frames in this simulation.

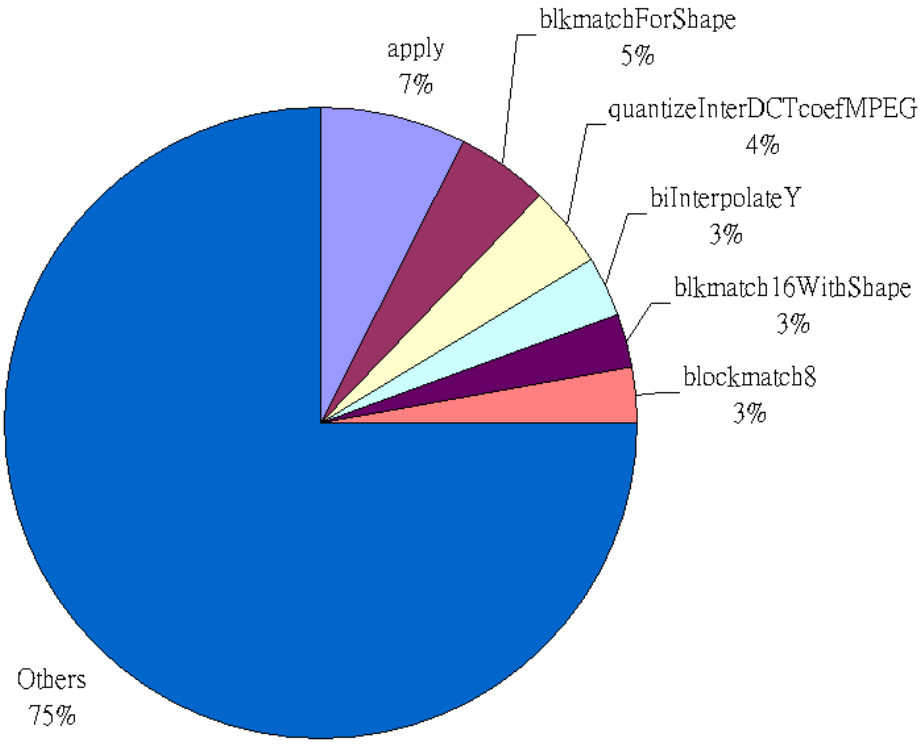


Fig. 3.19: Complexity breakdown of MPEG-4 video encoder in video conference system.

Chapter 4

Overview of RTP

4.1 Introduction (from [5])

The real-time transport protocol (RTP) provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. Those services include payload type identification, sequence numbering, timestamping and delivery monitoring. Applications typically run RTP on top of UDP to make use of its multiplexing and checksum services; both protocols contribute parts of the transport protocol functionality. However, RTP may be used with other suitable underlying network or transport protocols. RTP supports data transfer to multiple destinations using multicast distribution if provided by the underlying network.

RTP consists of two closely-linked parts:

- the real-time transport protocol (RTP), to carry data that has real-time properties; and
- the RTP control protocol (RTCP), to monitor the quality of service and to convey information about the participants in an on-going session. The latter aspect of RTCP may be sufficient for "loosely controlled" sessions, i.e., where there is no explicit membership control and set-up, but it is not necessarily intended to support all of an applications control communication requirements.

4.2 Definitions in RTP (from [5])

- RTP payload: The data transported by RTP in a packet, for example audio samples or compressed video data.

- RTP packet: A data packet consisting of the fixed RTP header, a possibly empty list of contributing sources, and the payload data. Some underlying protocols may require an encapsulation of the RTP packet to be defined. Typically one packet of the underlying protocol contains a single RTP packet, but several RTP packets may be contained if permitted by the encapsulation method.

- RTCP packet: A control packet consisting of a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type. Typically, multiple RTCP packets are sent together as a compound RTCP packet in a single packet of the underlying protocol; this is enabled by the length field in the fixed header of each RTCP packet.

- Port: The "abstraction that transport protocols use to distinguish among multiple destinations within a given host computer. TCP/IP protocols identify ports using small positive integers [21]." The transport selectors (TSEL) used by the OSI transport layer are equivalent to ports. RTP depends upon the lower-layer protocol to provide some mechanism such as ports to multiplex the RTP and RTCP packets of a session.

- Transport address: The combination of a network address and port that identifies a transport-level endpoint, for example an IP address and a UDP port. Packets are transmitted from a source transport address to a destination transport address.

- RTP session: The association among a set of participants communicating with RTP. For each participant, the session is defined by a particular pair of destination transport addresses (one network address plus a port pair for RTP and RTCP). The destination transport address pair may be common for all participants, as in the case of IP multicast, or may be different for each, as in the case of individual unicast network addresses plus a common port pair. In a multimedia session, each medium is carried in a separate RTP session with its own RTCP packets. The multiple RTP sessions are distinguished by different port number pairs and/or different multicast addresses.

- Synchronization source (SSRC): The source of a stream of RTP packets, identified by a 32-bit numeric SSRC identifier carried in the RTP header so as not to be dependent upon the network address. All packets from a synchronization source form part of the same timing and sequence number space, so a receiver groups packets by synchronization source for playback.

- Contributing source (CSRC): A source of a stream of RTP packets that has contributed to the combined stream produced by an RTP mixer (see below). The mixer inserts a list of the SSRC identifiers of the sources that contributed to the generation of a particular packet into the RTP header of that packet. This list is called the CSRC list. An example application is audio conferencing where a mixer indicates all the talkers whose speech was combined to produce the outgoing packet, allowing the receiver to indicate the current talker, even though all the audio packets contain the same SSRC identifier (that of the mixer).

- End system: An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets. An end system can act as one or more synchronization sources in a particular RTP session,

but typically only one.

- Mixer: An intermediate system that receives RTP packets from one or more sources, possibly changes the data format, combines the packets in some manner and then forwards a new RTP packet. Since the timing among multiple input sources will not generally be synchronized, the mixer will make timing adjustments among the streams and generate its own timing for the combined stream. Thus, all data packets originating from a mixer will be identified as having the mixer as their synchronization source.

- Translator: An intermediate system that forwards RTP packets with their synchronization source identifier intact. Examples of translators include devices that convert encodings without mixing, replicators from multicast to unicast, and applicationlevel filters in firewalls.

- Monitor: An application that receives RTCP packets sent by participants in an RTP session, in particular the reception reports, and estimates the current quality of service for distribution monitoring, fault diagnosis and long-term statistics. The monitor function is likely to be built into the application(s) participating in the session, but may also be a separate application that does not otherwise participate and does not send or receive the RTP data packets. These are called third party monitors.

- Non-RTP means: Protocols and mechanisms that may be needed in addition to RTP to provide a usable service. In particular, for multimedia conferences, a conference control application may distribute multicast addresses and keys for encryption, negotiate the encryption algorithm to be used, and define dynamic mappings between RTP payload type values and the payload formats they represent for formats that do not have a predefined payload type value.

4.3 RTP Fixed Header Fields (from [5])

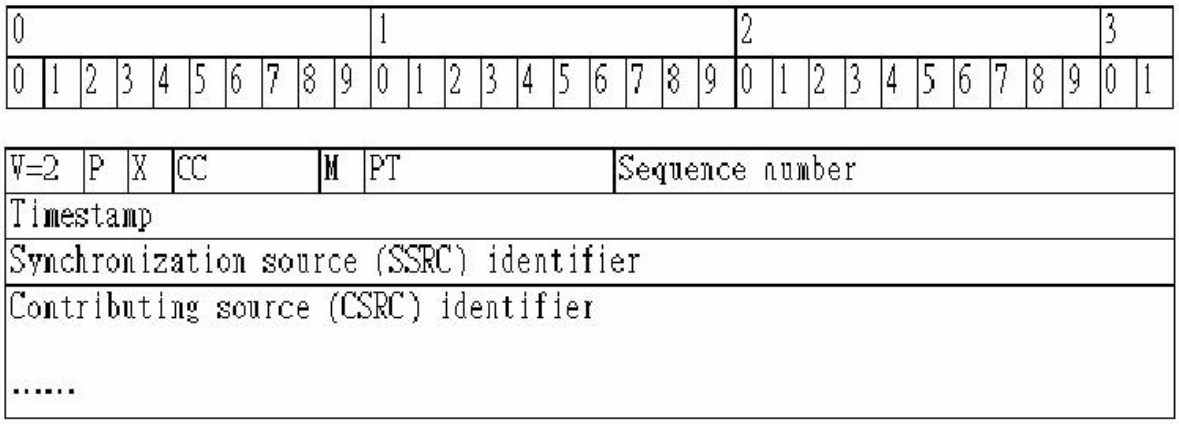


Fig. 4.1: RTP header (from[5]).

Fig. 4.1 shows the RTP header format. The first twelve octets are present in every RTP packet, while the list of CSRC identifiers is present only when inserted by a mixer. The fields have the following meaning:

- Version (V): 2 bits

  This field identifies the version of RTP. The version defined by the specification we took is two[5]. (The value 1 is used by the first draft version of RTP and the value 0 is used by the protocol initially implemented in the "vat" audio tool.)

- Padding (P): 1 bit

  If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload.

- Extension (X): 1 bit

  If the extension bit is set, the fixed header is followed by exactly one header extension.

- CSRC count (CC): 4 bits

  The CSRC count contains the number of CSRC identifiers that follow the fixed header.

- Marker (M): 1 bit

  The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream.

- Payload type (PT): 7 bits

  This field identifies the format of the RTP payload and determines its interpretation by the application.

- Sequence number: 16 bits

  The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence. Techniques for choosing unpredictable numbers are discussed in [22].

- Timestamp: 32 bits

  The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations.

- SSRC: 32 bits

  The SSRC field identifies the synchronization source. This identifier is chosen randomly, with the intent that no two synchronization sources within the same RTP session will have the same SSRC identifier.

- CSRC list: 0 to 15 items, 32 bits each

  The CSRC list identifies the contributing sources for the payload contained in this packet. The number of identifiers is given by the CC field. If there are

more than 15 contributing sources, only 15 may be identified. CSRC identifiers are inserted by mixers, using the SSRC identifiers of contributing sources.

4.4 RTP Control Protocol – RTCP (from [5])

The RTP control protocol (RTCP) is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets. The underlying protocol must provide multiplexing of the data and control packets, for example using separate port numbers with UDP. RTCP performs four functions:

1. The primary function is to provide feedback on the quality of the data distribution. This is an integral part of the RTPs role as a transport protocol and is related to the flow and congestion control functions of other transport protocols.

2. RTCP carries a persistent transport-level identifier for an RTP source called the canonical name or CNAME. Since the SSRC identifier may change if a conflict is discovered or a program is restarted, receivers require the CNAME to keep track of each participant. Receivers also require the CNAME to associate multiple data streams from a given participant in a set of related RTP sessions, for example to synchronize audio and video.

3. The first two functions require that all participants send RTCP packets, therefore the rate must be controlled in order for RTP to scale up to a large number of participants. By having each participant send its control packets to all the others, each can independently observe the number of participants. This number is used to calculate the rate at which the packets are sent.

4. A fourth, optional function is to convey minimal session control information,

for example participant identification to be displayed in the user interface. This is most likely to be useful in "loosely controlled" sessions where participants enter and leave without membership control or parameter negotiation. RTCP serves as a convenient channel to reach all the participants, but it is not necessarily expected to support all the control communication requirements of an application.

Functions 1–3 are mandatory when RTP is used in the IP multicast environment, and are recommended for all environments. RTP application designers are advised to avoid mechanisms that can only work in unicast mode and will not scale to larger numbers.

4.4.1 RTCP Packet Format

We introduce several RTCP packet types that carry a variety of control information:

- SR: Sender report, for transmission and reception statistics from participants that are active senders.
- RR: Receiver report, for reception statistics from participants that are not active senders.
- SDES: Source description items, including CNAME.
- BYE: Indicates end of participation.
- APP: Application specific functions.

Each RTCP packet begins with a fixed part similar to that of RTP data packets, followed by structured elements that may be of variable length according to the packet type but always end on a 32-bit boundary. The alignment requirement and a length field in the fixed part are included to make RTCP packets "stackable."

Multiple RTCP packets may be concatenated without any intervening separators to form a compound RTCP packet that is sent in a single packet of the lower layer protocol, for example UDP. There is no explicit count of individual RTCP packets in the compound packet since the lower layer protocols are expected to provide an overall length to determine the end of the compound packet.

Each individual RTCP packet in the compound packet may be processed independently with no requirements upon the order or combination of packets. However, in order to perform the functions of the protocol, the following constraints are imposed:

- Reception statistics (in SR or RR) should be sent as often as bandwidth constraints will allow to maximize the resolution of the statistics, therefore each periodically transmitted compound RTCP packet should include a report packet
- New receivers need to receive the CNAME for a source as soon as possible to identify the source and to begin associating media for purposes such as lip-sync, so each compound RTCP packet should also include the SDES CNAME.
- The number of packet types that may appear first in the compound packet should be limited to increase the number of constant bits in the first word and the probability of successfully validating RTCP packets against misaddressed RTP data packets or other unrelated packets.

Thus, all RTCP packets must be sent in a compound packet of at least two individual packets, with the following format recommended: Encryption prefix. If and only if the compound packet is to be encrypted, it is prefixed by a random 32-bit quantity redrawn for every compound packet transmitted.

- SR or RR: The first RTCP packet in the compound packet must always be a report packet to facilitate header validation. This is true even if no data has

been sent nor received, in which case an empty RR is sent, and even if the only other RTCP packet in the compound packet is a BYE.

- Additional RRs: If the number of sources for which reception statistics are being reported exceeds 31, the number that will fit into one SR or RR packet, then additional RR packets should follow the initial report packet.

- SDES: An SDES packet containing a CNAME item must be included in each compound RTCP packet. Other source description items may optionally be included if required by a particular application, subject to bandwidth constraints.

- BYE or APP: Other RTCP packet types, including those yet to be defined, may follow in any order, except that BYE should be the last packet sent with a given SSRC/CSRC. Packet types may appear more than once.

It is advisable for translators and mixers to combine individual RTCP packets from the multiple sources they are forwarding into one compound packet whenever feasible in order to amortize the packet overhead. An example RTCP compound packet as might be produced by a mixer is shown in Fig. 4.2 If the overall length of a compound packet would exceed the maximum transmission unit (MTU) of the network path, it may be segmented into multiple shorter compound packets to be transmitted in separate packets of the underlying protocol. Note that each of the compound packets must begin with an SR or RR packet.

An implementation may ignore incoming RTCP packets with types unknown to it. Additional RTCP packet types may be registered with the Internet Assigned Numbers Authority (IANA).

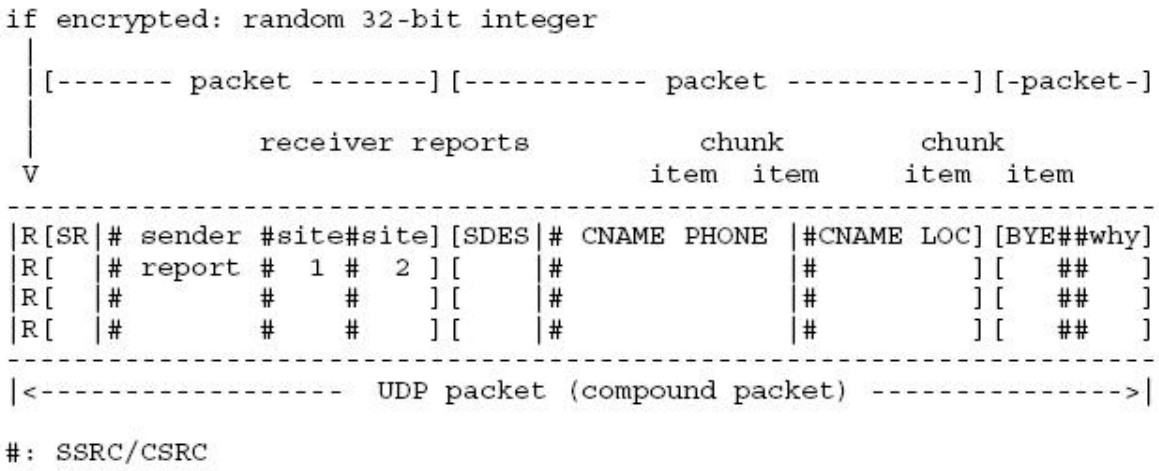


Fig. 4.2: Example of an RTCP compound packet (from [5]).

4.5 Introduction to the JRTPLib Software (from [32])

The JRTPLib software is a public source, and an object-oriented library written in C++ which aims to help developers in using the Real-time Transport Protocol (RTP). This library is developed within JThread library, which is a public source, too. JThread contains only two classes, namely JThread and JMutex. JThread represents a thread and JMutex a mutex. The thread class only contains very basic functions, such as to start or kill a thread. The goal of JThread is to make use of threads easy on different platforms, like MS-Windows or Unix platform [33].

The goal of JRTP library is to make user send and receive data using RTP without worrying about SSRC collisions, scheduling and transmitting RTCP data. The user only needs to provide the library with the payload data to be send and the library gives the user access to incoming RTP and RTCP data. Table 4.1 shows the class and the simple descriptions below, and [32] show the more detail list.

Table 4.1: JRTPLib Classes

Class name	Descriptions
RTPLibraryVersion	show the version of JRTP library.
RTPTime	This class is used to specify wallclock time, delay intervals ...etc.
RTPRandom	The RTPRandom class can be used to generate random numbers.
RTCPSDESInfo	The class is a container for RTCP SDES information.
RTPTransmitter	The abstract class specifies the interface for actual transmission components.
RTPUDPv4Transmitter	Inherit the RTPTransmitter interface and implements a transmission component which user UDP over IPv4 to send and receive RTP and RTCP data.
RTPUDPv6Transmitter	Inherit the RTPTransmitter interface and implements a transmission component which user UDP over IPv6 to send and receive RTP and RTCP data.
RTPTransmissionParams	An abstract class which will have a specific implementation for a specific knod of transmission component.
RTPUDPv4TransmissionParams	Represents the parameters used by the UDP over IPv4 transmission component.
RTPUDPv6TransmissionParams	Represents the parameters used by the UDP over IPv6 transmission component.
RTPTransmissionInfo	An abstract class which will have a specific implementation for a specific kind of transmission component.
RTPUDPv4TransmissionInfo	Give some additional information about the UDP over IPv4 transmission component.
RTPUDPv6TransmissionInfo	Give some additional information about the UDP over IPv6 transmission component.
RTPAddress	An abstract class which is used to specify destinations, multicast groups etc.
RTPIPv4Address	This class is used by the UDP over IPv4 transmission component.
RTPIPv6Address	This class is used by the UDP over IPv6 transmission component.
RTPRawPacket	Be used by the transmission component to store the incoming RTP and RTCP data in.
RTPPacket	Be used to parse a RTPRawPacket instance if it represents RTP data.
RTCPCompoundPacket	Describe an RTCP compound packet.
RTCPPacket	A base class for specific types of RTCP packets.
RTCPSRPacket	Describe an RTCP sender report packet.
RTCPRRPacket	Describe an RTCP receiver report packet.

RTCPSDESPacket	Describe an RTCP SDES packet.
RTCPAPPPacket	Describe an RTCP APP packet.
RTCPUnknownPacket	The class does not have any extra member functions besides the ones it inherited.
RTCPCompoundPacketBuilder	Be used to construct an RTCP compound packet.
RTPSources	Represent a table in which imformation about the participating sources is kept.
RTPSourceData	Contain all information about a member of the session
RTPPacketBuilder	This class can be used to build RTP packets and is a bit more high-level than the RTPPacket class.
RTCPPacketBuilder	Be used to build RTCP compound packets. This class is more high-level than the RTCPCOmpoundPacketBuilder class.
RTPCollisionList	Represent a list of address from which SSRC collisions were detected.
RTCPScheduler	Determine when RTCP compound packets should be sent.
RTCPSchedulerParams	Describe the parameters to be used by the scheduler.
RTPSessionParams	Describe the parameters for to be used by an RTPSession instance.
RTPSession	Handle the RTCP part completely internally and make user focus on sending and receiving the actual data.

4.6 Construction of RTP Connection

Before using RTP on the network, we should clean up and set up the network interface, as illustrated in Fig. 4.3. One has to create an RTPSession object first. The constructor of the RTPSession class takes a parameter of type RTPTransmitter::Transmission Protocol which defaults to RTPTransmitter::IPv4UDPProto. This means that the UDP over IPv4 transmission component will be used. Hence we create an RTPSessionParams object and an RTPUDPv4TransmissionParams object at the same time as shown in Fig. 4.4.

Now we list and interpret the instructions in Figs. 4.3 and 4.4 below [32]:

- WSADATA wsaData: The WSADATA structure is used to store Windows Sockets initialization information.
- WORD wVersionRequested = MAKEWORD(2,2): The MAKEWORD macro

creates a WORD value by concatenating the specified values. The first parameter 2 represents we specify the low-order byte 2, and the second parameter 2 represents we specify the high-order byte 2, too. This instruction is to suit the parameter of the next instruction, WSAStartup.

- WSAStartup(wVersionRequested, &wsaData): The WSAStartup function initiates use of WS2_32.DLL by a process. In this condition, it means the application supports only version 2.2 of Windows sockets.

- RTPSessionParams sessParams: Create an object that describes the parameters to be used by an RTPSession instance.

- sessParams.SetOwnTimestampUnit(1.0 / 1.0): Set the timestamp unit for our own data. The timestamp unit is defined as a time interval in seconds divided by the number of samples in that interval.

- sessParams.SetUsePollThread(1): The syntax of this instruction is "int SetUsePollThread(bool usethread)." If usethread is true, the session will use a poll thread to automatically process incoming data and to send RTCP packets when nece ssary.

- sessParams.SetMaximumPacketSize(MAX_PACKET_SIZE): Set the maximum allowed packet size for the session.

- RTPUDPv4TransmissionParams transParams: The RTPTransmissionParams class represents the parameters used by the UDP over IPv4 transmission component.

Fig. 4.5 shows how we construct an RTP session. Before we consider the RTPUDPv4TransmissionParams object as the factor of RTPSession object, we should be careful of the bounded ports which are used by other programs. Therefore we

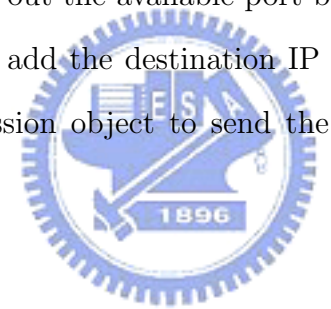
```
void network_initialize()
{
        //MUST call WSAStartup() to use WS2_32.DLL
        WSADATA wsaData;
        WORD wVersionRequested = MAKEWORD( 2, 2 );
        WSAStartup(wVersionRequested, &wsaData);      //WSACleanup called in OnDestroy
}
```

Fig. 4.3: Related code of network initialization.

```
RTPSessionParams sessParams;
sessParams.SetOwnTimestampUnit(1.0 / 1.0);
sessParams.SetUsePollThread(1);     //background thread to call virtual callbacks - set by
                                      default, but just to be sure
sessParams.SetMaximumPacketSize(MAX_PACKET_SIZE);
                                    //setup transmission parameters
RTPUDPv4TransmissionParams transParams;
```

Fig. 4.4: Related code of RTP parameters construction.

write a while loop to help find out the available port-base automatically. After setting up the port-base, we can add the destination IP into the RTPSession object. Then we can use the RTPSession object to send the data by using the "rtpSession.SendPacket" function.



```
int status;
do{
        transParams.SetPortbase(SERVER_PORT);
        status = rtpSession.Create(sessParams, &transParams);
        SERVER_PORT += 2;
}while (status<0);

unsigned long intIP = inet_addr(MCAST_IP);
_ASSERT(intIP != INADDR_NONE);
intIP = ntohl(intIP);   //put in host byte order
RTPIPv4Address rtpAddr(intIP, MCAST_PORT);
status = rtpSession.AddDestination(rtpAddr);
```

Fig. 4.5: Related code of RTPSession construction.

Chapter 5

Integration of Videoconference Transmitter

5.1 Introduction

Fig. 5.1 shows the thread view of the overall system. We have introduced the video segmentation, the MPEG-4 encoder, and the RTP protocol before. Now we need to integrate them with some additional functions such as video and audio capture and the AAC encoder. Besides, we also should modify the inputs and outputs of the functions stated above from files to memories, speed them up, and do some other trivia.

5.2 Video Capture

5.2.1 Video for Windows

In this system, the input image is captured by a digital camera. To control the operation of capturing, a standard video capturing method, named VfW (abbreviation of Video for Windows), in the Microsoft OS is adopted. Video for Windows version 1.0 was released in November 1992 for the Windows 3.1 operating system and was optimized for capturing movies to disk [23]. This SDK provides applications with a simple, message-based interface to access video and waveform-audio

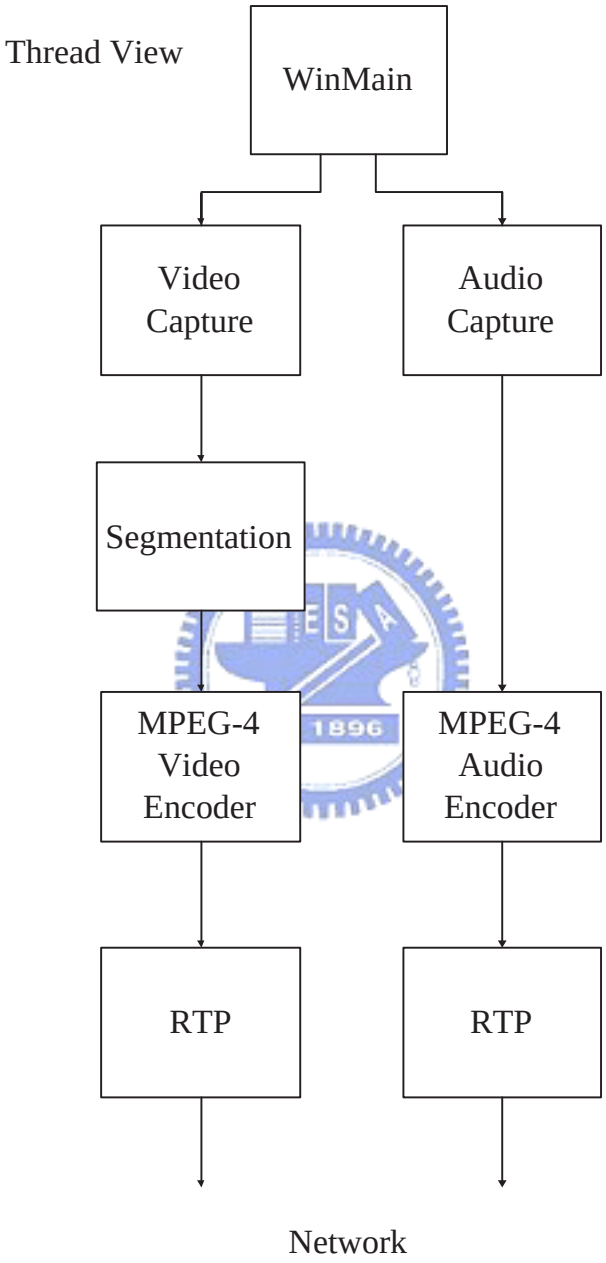


Fig. 5.1: Thread view of the overall system.

RIFF	RIFF universal file header
AVI	AVI file header
Hdrl	Header list
Avih	AVI header
Strl	List of stream header for each stream in the AVI file
Strh	Video or audio stream header
Strf	Video or audio stream format
JUNK	Used to align data
##wb	Audio frame data
##db	Video frame data

Fig. 5.2: AVI header (from[2]).

acquisition hardware and to control the process of streaming video capture to disk. Besides, VfW helps with connectivity to device driver and retrieve the capability and information of it.

5.2.2 AVI Format

In VfW, AVI is the mostly used format. The captured raw frame is embedded in an AVI file which can be extracted for segmentation input.

AVI stands for Audio-Video Interleaved. Fig. 5.2 shows the hierarchical structure. Refer to header file (vfw.h in Visual C++) for complete information about parsing AVI file. To extract video data, we use a file parser that simply locate $\#\#db$ and copy suitable length of data following that.

5.2.3 Implementation of Capture

The implementation of capture is aided by a free application called AVICap from [24]. It contains three steps:

1. Create capture handle:

   An AVICap capture window handles the details of streaming audio and video capture to AVI files and it provides a flexible interface for applications. The video capture can be added to application by the code shown in Fig. 5.3.

- capCreateCapturewindow: The capCreateCaptureWindow function creates a capture window. The syntax is shown below:

  HWND VFWAPI capCreateCaptureWindow(

  LPCSTR lpszWindowName,

  DWORD dwStyle,

  int x,

  int y,

  int nWidth,

  int nHeight,

  HWND hWnd,

  int nID);

  In the above, lpszWindowName represents the the name used for the capture window. dwStyle represents Window styles used for the capture window. Here we assign the window as a child window and is visible initially. x and y represent the x and y coordinates of the upper left corner of the capture window. nWidth and nHeight represent the width and the height of the capture window. hWnd represents the handle to the window. Finally, nID represents the window identifier.

- capDriverConnect(hwndC,0): The capDriverConnect macro connects a capture window to a capture driver.

- capDriverGetCaps(hwndC, &caps, sizeof(caps)): The capDriverGetCaps macro returns the hardware capabilities of the capture driver currently connected to a capture window. caps here is a pointer to the CAPDRIVERCAPS structure to contain the hardware capabilities.

2. Parameter modification:

   After initializing driver window handler, some fundamental parameters should

```
//Create the capture window
hwndC = capCreateCaptureWindow("Video Capture
Window",WS_CHILD | WS_VISIBLE,0,0,352,288,hwnd, 0);
// Connect the capture window to the driver
capDriverConnect(hwndC, 0);
// Get the capabilities of the capture driver
capDriverGetCaps(hwndC, &caps, sizeof(caps));
```

Fig. 5.3: Related code for creating a capture window.

```
// Set the preview rate in milliseconds
capPreviewRate(hwndC,30);
// Start previewing the image from the camera
capPreview(hwndC, TRUE);
```

Fig. 5.4: Related code for parameter modification.

be confirmed to ensure captured data fit system requirement, such as the code shown in Fig. 5.4.

- capPreviewRate(hwndC,30): The capPreviewRate macro sets the frame display rate in preview mode. We set the frame display rate to 30 here.
- capPreview(hwndC,TRUE): The capPreview macro enables or disables preview mode. In preview mode, frames are transferred from the capture hardware to system memory and then displayed in the capture window.

3. Capture operation:

   In this step, we start to capture image from digital camera and the related code is shown in Fig. 5.5. Here, the captured image in AVI format is stored in a buffer and then the required video data are extracted from the buffer. Finally, the extracted data are sent to the module doing video segmentation.

5.2.4 Modified Video Capture Process

The general method of video capture supplied by VFW is simple, but the principle of it is not suitable for real-time video capture. AVICap, by default, routes video and audio stream data from a capture window to a file named CAPTURE.AVI in

```
char filename[] = "c:\\buffer.avi" ;
unsigned char Y_Component[352*288*3/2];
capFileSetCaptureFile(hwndC,filename);
FptrIn = fopen(filename,"rb");
capCaptureSingleFrameOpen(hwndC);
capCaptureSingleFrame(hwndC);
capCaptureSingleFrameClose(hwndC);
fseek(FptrIn,0xa08,SEEK_SET);
fread(Y_Component,1,352*288*3/2,FptrIn);
```

Fig. 5.5: Related code for capture operation.

the root directory of the current drive. However, what we want is to get the video stream from memory directly. Therefore, the speed of video capture would be higher because of reduced operations [25]. Fig. 5.6 gives a simple diagram illustrating the two ways of doing video capture.

We could use capture services without writing the data to a disk file by using the WM_CAP_SEQUENCE_NOFILE message (or the capCaptureSequenceNoFile macro). However, this message is useful only in conjunction with callback functions that allow the application to use the video and audio data directly. Callback function interacts with the running thread. Besides, it would catch the data from the running thread. We could use multi-thread to replace the functionality of the callback function, but the capture thread here only allows the callback function. This is because the priority of capture thread is highest. Hence we cannot run another thread when running the capture thread. Figs. 5.8 and 5.9 show the declaration of callback function.

Before using the callback function, we should use WM_CAP_SET_CALLBACK_VIDEOSTREAM message or capSetCallbackOnVideoStream macro to set the callback function in the application. The prototype of WM_CAP_SET_CALLBACK_VIDEOSTREAM is shown in Fig. 5.9. AVICap calls this procedure during streaming capture when a video buffer is filled. fpProc is the pointer to the video-stream callback function. Specify NULL for this parameter to

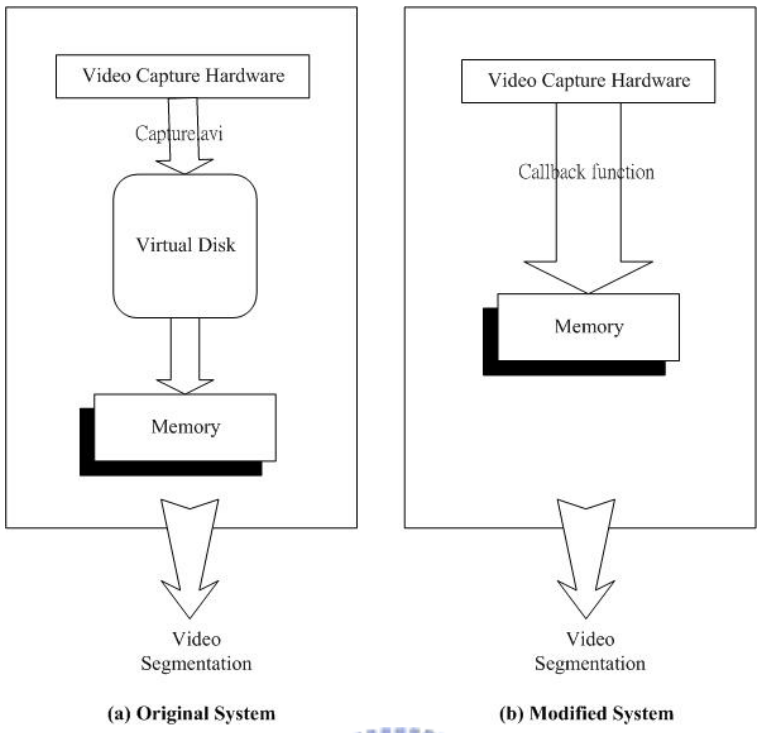


Fig. 5.6: Simple diagram explaining two ways of video capture.

disable a previously installed video-stream callback function. The message would return TRUE if successful or FALSE if streaming capture or a single-frame capture session is in progress.

However, what we want to keep in mind is that:

- The capture window calls the callback function before writing the captured frame to disk. This allows applications to modify the frame if desired. If a video stream callback function is used for streaming capture, the procedure must be installed before starting the capture session and it must remain enabled for the duration of the session. It can be disabled after streaming capture ends.

- We could view the process of video capture as a thread of the program. However, unfortunately, the priority of the thread about video capture without writing into disk is the highest of everything! It is quite different compared

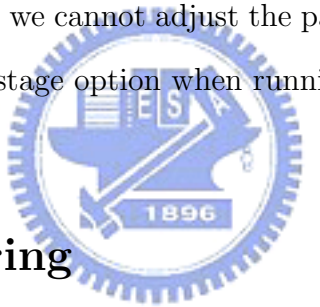
```
capCaptureSequenceNoFile(hwndC);
//Capture Without Using Disk Storage
capSetCallbackOnVideoStream(hwndC,videohanlder);
//Set a callback function
```

Fig. 5.7: Related code for video capture without writing the data to a disk file.

```
LRESULT CALLBACK videohanlder(HWND
window,LPVIDEOHDR videohdr){
return 0;
}
//Declaration of callback function.
```

Fig. 5.8: Related code of declaration of callback function.

with the general video capture program. The priority of the general video capture program is not higher than the main SDK function. Hence we could still receive another message during capturing video. Contrarily, the main SDK function could not receive any message during capturing video without writing into disk. That means we cannot adjust the parameter, such as the thread and the noise estimation stage option when running the capture process.



5.3 Audio Capturing

5.3.1 The WAV Format

The WAV format is the most common format used in capturing audio. MCI (Media Control Interface) supplies three methods to capture audio. Besides the lowest-level method, the other two methods would capture the audio stream and generate it as a .wav file. Hence we use Table 5.1 to introduce the format of the .wav file [26].

- ChunkID: Contains the letters “RIFF” in ASCII form.

```
WM_CAP_SET_CALLBACK_VIDEOSTREAM
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);
```

Fig. 5.9: Related code of prototype of set callback function.

Table 5.1: .WAV File Format

Offset	Size	Name
0	4	“ChunkID”
4	4	“ChunkSize”
8	4	“Format”
12	4	“Subchunk1ID”
16	4	“Subchunk1Size”
20	2	“AudioFormat”
22	2	“NumChannels”
24	4	“SampleRate”
28	4	“ByteRate”
32	2	“BlockAlign”
34	2	“BitsPerSample”
36	4	“Subchunk2ID”
40	4	“Subchunk2Size”
44	*	“Data”

- ChunkSize: 36 + SubChunk2Size, or more precisely: 4 + (8 + SubChunk1Size) + (8 + SubChunk2Size). This is the size of the rest of the chunk following this number. This is the size of the entire file in bytes minus 8 bytes for the two fields not included in this count: ChunkID and ChunkSize.
- Format: Contains the letters “WAVE.”

The “WAVE” format consists of two subchunks: “fmt” and “data”; the “fmt” subchunk describes the sound data’s format.

- Subchunk1ID: Contains the letters “fmt”.
- Subchunk1Size: 16 for PCM. This is the size of the rest of the subchunk which follows this number.
- AudioFormat: PCM = 1 (i.e., linear quantization). Values other than 1 indicate some form of compression.
- NumChannels: Mono = 1, Stereo = 2, etc.

```
//audio declaration
MCI_GENERIC_PARMS
mciGeneric;
MCI_OPEN_PARMS    mciOpen;
MCI_RECORD_PARMS
mciRecord;
MCI_SAVE_PARMS    mciSave;
```

Fig. 5.10: Related code of audio declaration.

- SampleRate: 8000, 44100, etc.
- ByteRate: $= SampleRate * NumChannels * BitsPerSample/8$.
- BlockAlign: $= NumChannels * BitsPerSample/8$.
- BitsPerSample: 8 bits = 8, 16 bits = 16, etc.

The "data" subchunk contains the size of the data and the actual sound.

- Subchunk2ID: Contains the letters "data".
- Subchunk2Size: $= NumSamples * NumChannels * BitsPerSample/8$
- Data: The actual sound data.

5.3.2 Implementation of Audio Capture

Fig. 5.10 introduces the declaration of the audio MCI class; Fig. 5.11 introduces the record of audio stream; and Fig. 5.12 introduces the saving of audio stream.

- MCI_GENERIC_PARMS mciGeneric: The structure contains the handle of the window that receives notification messages.
- MCI_OPEN_PARMS mciOpen: The structure contains information for the MCI_OPEN command. The MCI_OPEN command means that we initialize a device or file. The structure of MCI_OPEN_PARMS is shown below:

```
//Open waveform audio
mciOpen.dwCallback = 0;
mciOpen.wDeviceID = 0;
mciOpen.lpstrDeviceType = TEXT ("waveaudio");
mciOpen.lpstrElementName = TEXT ("");
mciOpen.lpstrAlias = NULL;
dwError = mciSendCommand (0, MCI_OPEN, MCI_WAIT | MCI_OPEN_TYPE |
MCI_OPEN_ELEMENT, (DWORD) (LPMCI_OPEN_PARMS) &mciOpen);

//Save the Device ID
wDeviceID = mciOpen.wDeviceID ;

//Begin recording
mciRecord.dwCallback = (DWORD) hwnd;
mciRecord.dwFrom = 0;
mciRecord.dwTo = 0;
mciSendCommand
(wDeviceID,MCI_RECORD,MCI_NOTIFY,(DWORD)(LPMCI_RECORD_PARMS) &mciRecord);
```

Fig. 5.11: Related code of audio record.

```
//Stop recording
mciGeneric.dwCallback = 0;

mciSendCommand (wDeviceID, MCI_STOP , MCI_WAIT, (DWORD) (LPMCI_GENERIC_PARMS)
&mciGeneric);

//Save the file
mciSave.dwCallback = 0;
mciSave.lpfilename = szFileName;

mciSendCommand (wDeviceID, MCI_SAVE, MCI_WAIT | MCI_SAVE_FILE, (DWORD)
(LPMCI_SAVE_PARMS) &mciSave);

//Close the waveform device
mciSendCommand (wDeviceID, MCI_CLOSE, MCI_WAIT, (DWORD) (LPMCI_GENERIC_PARMS)
&mciGeneric);
```

Fig. 5.12: Related code of audio saving.

typedef struct{

DWORD_PTR dwCallback;

MCIDEVICEID wDeviceID;

LPCSTR lpstrDeviceType;

LPCSTR lpstrElementName;

LPCSTR lpstrAlias;

}MCI_OPEN_PARMS;

In the above, dwCallback represents the low-order word specifies a window handle used for the MCI_NOTIFY flag, wDeviceID represents the identifier returned to application, lpstrDeviceType represents the name or the constant identifier of the device type, lpstrElementName represents the device element (often a path), and lpstrAlias represents the optional device alias.

- MCI_RECORD_PARMS mciRecord: The structure contains positioning information for the MCI_RECORD command. The MCI_RECORD command means that we start recording from the current position or from one specified location to another specified location. The structure of MCI_RECORD_PARMS is shown below:

  typedef struct {

  DWORD_PTR dwCallback;

  DWORD dwFrom;

  DWORD dwTo;

  } MCI_RECORD_PARMS;

  In the above, dwCallback represents the low-order word specifies a window handle used for the MCI_NOTIFY flag, dwFrom represents the position to play from, and dwTo represents the position to play to.

- MCI_SAVE_PARMS mciSave: The structure contains the filename information

for the MCI_SAVE command. The MCI_SAVE command saves the current data of MCI_RECORD from memory into file. The structure of MCI_SAVE_PARMS is shown below:

typedef struct {

DWORD_PTR dwCallback;

LPCSTR lpfilename;

} MCI_SAVE_PARMS;

In the above, dwCallback represents the low-order word specifies a window handle used for the MCI_NOTIFY flag, and lpfilename represents the name of the file to save.

- mciSendCommand: The most important MCI function that we use is mciSendCommand. The structure of mciSendCommand is "MCIERROR mciSendCommand(MCIDEVICEID IDDevice, UINT uMsg, DWORD fdwCommand, DWORD_PTR dwParam);" The first parameter, IDDevice, is the device identifier of the MCI device that is to receive the command message. The second parameter, uMsg, is the command message. The third parameter, fdwCommand, is the flags for the command message. The fourth parameter, dwParam, is a pointer to a structure that contains parameters for the command message. The function would return zero if successful or an error otherwise. The low-order word of the returned DWORD value contains the error return value. If the error is device-specific, the high-order word of the return value is the driver identifier; otherwise, the high-order word is zero.

- mciGetErrorString: Another important Mci function is mciGetErrorString. The mciGetErrorString function retrieves a string that describes the specified MCI error code. The prototype of mciGetErrorString is "BOOL mciGetErrorString(DWORD fdwError, LPTSTR lpszErrorText, UINT cchErrorText);"

fdwError is the error code returned by the mciSendCommand or mciSendString function. lpszErrorText is the pointer to a buffer that receives a null-terminated string describing the specified error. Finally, cchErrorText is the length of the buffer, in characters, pointed to by the lpszErrorText parameter. The function would return TRUE if successful or FALSE if the error code is not known.

5.4 Modification of the Video Segmentation Method

5.4.1 Optimization with MMX Instructions

The video segmentation system is not the bottleneck of the whole videoconference system (we would discuss the performance of the whole videoconference system in the next chapter.). Hence we do not have too much needs to optimize video segmentation, but we still try to find the most significant function of the video segmentation system we get and optimize it.

From VTune, We could get the code sections that is the bottleneck of the video segmentation system. It is shown in Fig. 5.13. The function deals with the conflict of the temporal foreground and the temporal background. When one mask views a pixel as foreground but the other views it as background, the mask to propagate is set to 128, the medium value.

After optimization, we get the modified function in Fig. 5.14. The algorithm is introduced as follows. First, we make all the bytes in register mm0 0, and all the bytes in register mm1 1. Registers mm0 and mm1 are the criterion of the mask backID[ ] and mask[ ]. In the same way, we define all the bytes of the register result 128 and view the register result as the criterion of 128. Next, we move the data of backID[ ] and mask[ ] to mm2 and mm3 respectively. Then we could compare mm2 with mm0 and set the byte of mm0 is 1 if the byte of mm2 is 0; we also compare mm3

```
//************************************
// COMBINE BACKGROUND and FOREGROUND
//************************************

for( uv1=0;uv1<frame_size;uv1++)
{
        if((backID[uv1]==0)&&(mask[uv1]==255))
        {   backID[uv1]=128 ;  }   //partial wrong back
}
```

Fig. 5.13: Section of code that we would like to optimize with MMX.

with mm1 and set the byte of mm1 is 1 if the byte of mm3 is 0. Next, we compare mm0 with mm1 and set the byte of mm0 is 1 if the byte of mm0 and mm1 is equal. In the step, we could imply the function "(backID[uv1]==0)&&(mask[uv1]==255)."

Third, we use the function "pand" to AND all the bytes in mm0 and mm4. That means that if elements in backID[ ] and mask[ ] are 0 and 255 at the same time, the element in mm0 would be set to 128. However, if the elements in backID[ ] and mask[ ] aren't 0 and 255 in the same time, the element in mm0 would be set to 0.

Finally, we add the byte of mm0 to the byte mm2 which means backID[ ]. That means that if the element in the backID[ ] is conflict with the element in the mask[ ], it would be set to 128. If not, nothing happens in the backID[ ].

In the program, the most MMX instruction we used is "pcmpeqb mm, mm/m64." The last letter "b" just means that we use byte-based operations in the register. The instructor means we compare packed byte in MMX register/memory with packed byte in MMX register for equality. Fig. 5.15 shows an example using word-based operations.

With the same idea, "pxor mm, mm/m64" means we XOR 64 bits from MMX register/memory to MMX register; "pand mm, mm/m64" means we AND 64 bits from MMX register/memory to MMX register; "paddusb mm, mm/m64" means we add unsigned packed byte from MMX register/memory to unsigned packed byte in MMX register and saturate; "movq mm(mm/64), mm/64(mm)" means we move 64 bits from MMX register/memory to MMX register or vice versa [28].

```
DWORD64 result =    0x8080808080808080;
for(uv1=0;uv1<frame_size;uv1+=8)
{
        __asm
        {
                pusha
                mov eax,backID
                mov ebx,mask
                add eax,uv1
                add ebx,uv1
                pxor mm0,mm0    //backIDstd
                pcmpeqd mm1,mm1    //maskstd
                movq mm4, result

                movq mm2,[eax]
                movq mm3,[ebx]
                pcmpeqb mm0,mm2
                pcmpeqb mm1,mm3
                pcmpeqb mm0,mm1

                pand mm0,mm4
                paddusb mm2,mm0
                movq [eax],mm2

                popa
                emms
         }
}
```

Fig. 5.14: Code in Fig. 5.13 after optimization using MMX instructions.

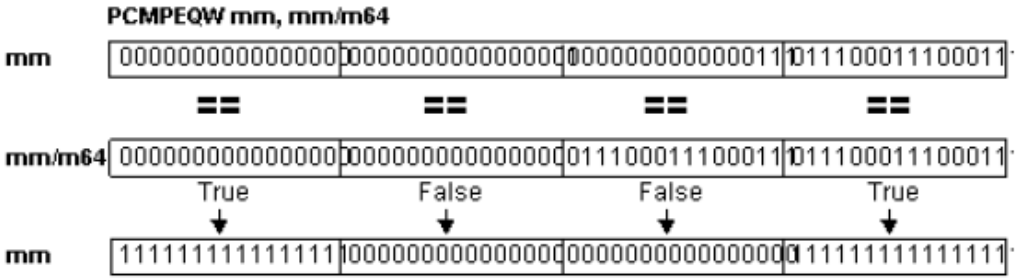


Fig. 5.15: Example of pcmpeqw instruction.

5.5 Integration with the MPEG-4 Video Encoder into the System

The edition of Microsoft MPEG-4 visual reference software we get was already reasonably optimized [3]. Hence we need to change the input and output of the software from files to segmentation output and RTP input, respectively.

5.5.1 Modification of Input to the MPEG-4 Video Encoder

Initially, because of the need of videoconference, we choose the "Binary Shape Object Coding" mode. Binary shape coding compresses a binary mask that defines a foreground video video object. There are two modes to binary shape coding. In the first, the mask shape is compressed on its own, with no other data present. This is *shape-only* mode. In the second, the video object texture (YUV data) is also compressed.

To use binary shape coding, it is necessary to supply a texture file (".yuv" file) and a segmentation mask file (".seg" file). This is true even if shape-only mode is used; the texture file is read, but the information is ignored. Two ways of encoding binary shape sequences are possible. In the first, a segmentation file is created which contains a binary mask, with either 0 (background) or 255 (mask) for each pixel. A number of these files can then be used to specify each object in the same sequence. The second method of encoding binary shape sequences is to have a common segmentation file that contains all the segmentation information.

For the need of the video conference system, we choose the non-shape-only mode and encode with a common segmentation file.

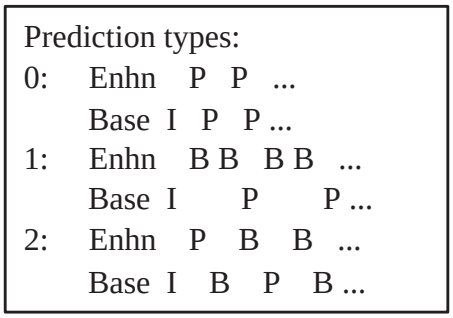


Fig. 5.16: Prediction types in a VOP.

In the program, the input/output processes are included in the class "CSessionEncoder." The data loading is mainly processed in the function "Bool CSessionEncoder:loadData (UInt iFrame, FILE* pfYuvSrc, FILE* pfSegSrc, FILE** ppfAuxSrc, PixelC pxlcObjColor, CVOPU8YUVBA* pvopcDst, CRct& rctOrg, const VOLMode& volmd)." To facilitate memory based operation, we change the call sequence to "Bool CSessionEncoder::loadData (UInt iFrame, FILE* pfYuvSrc, FILE* pfSegSrc, FILE** ppfAuxSrc, PixelC pxlcObjColor, CVOPU8YUVBA* pvopcDst, CRct& rctOrg, const VOLMode& volmd ,unsigned char* buff,unsigned char* seg_buff,long int &count,long int &seg_count)." The difference between these is that the parameters of the later function increase to include buff, seg_buff, and seg_count, where buff is the pointer to the memory we put the image data, seg_buff is the pointer to the memory we put the final image mask, and seg_count is an integer which helps us count the segmentation frame number.

In addition, we need to take care of the issue of frame sequencing. Fig. 3.7 shows a VOP sequence of "I P B P ..." in the base layer, and "P B B ..." in the enhance layer. Some other choices are shown in Fig. 5.16. We could assign the prediction types into the parameter file to the MPEG-4 encoder. The MPEG-4 encoder always reads I-frame at first, P-frame in the next, and B-frame in the last. That means the input of video stream would not be sequential, and we should adjust it when we change file-based input to memory-based input.

```
initVOEncoder (rgpvoenc, iVO, rgpostrmTrace);
rgpostrm [BASE_LAYER]->write (rgpvoenc [BASE_LAYER]->pOutStream ()->str (),
                              rgpvoenc [BASE_LAYER]->pOutStream ()->pcount ());
                            //VO and VOL header
```

Fig. 5.17: Related code of output of VO and VOL header.

```
rgpostrm [BASE_LAYER]->write (rgpvoenc [BASE_LAYER]->pOutStream ()->str (),
                              rgpvoenc [BASE_LAYER]->pOutStream ()->pcount ());
                            //write sprite unit
```

Fig. 5.18: Related code of output of MPEG-4 video encoder.

5.5.2 Modification of the Output to the MPEG-4 Video Encoder

The MPEG-4 video encoder would generate VO and VOL header in the beginning based on the parameter the user types in. The generater of VO and VOL header is included in the function "initVOEncoder" which is in "CSessionEncoder," too. Fig. 5.17 shows the related code of VO and VOL header. All the output header information, including data and length, is put in CVideoObjectEncoder::pOutStream.

Figs. 5.18 and 5.19 show the related code of compressed data output for the modes we choose to use. Unlike VO and VOL header, the compressed data would be output to the .cmp file. Fig. 5.18 describe the write sprite unit, and would be called one time, following the VO and VOL header. Fig. 5.19 is called every time when MPEG-4 encoder compress one VOP inside. We could easily change the goal of rgpostrm to put the data in the memory or could directly consider the pOutStream ()->str () as what we want.

```
rgpostrm [iLayer]->write (rgpvoenc [iLayer]->pOutStream ()->str (),
                          rgpvoenc [iLayer]->pOutStream ()->pcount ());
```

Fig. 5.19: Related code of output of MPEG-4 video encoder.

5.6 Integration with the MPEG-4 Audio Encoder

MCI deals with the most audio capture problems and what we do about audio capture is just to get the .wav file. We can not modify the software from file-based output to memory-based output. Hence we could just only send the .wav file to faac software, which the MPEG-4 audio encoder we get also owns file-based input. It is a quite simple process, so we do not describe deeply.

Chapter 6

Experimental Results

In this chapter, we show the performance of the whole videoconference system, and also show the performance of the components, in particular video capture and video segmentation.

6.1 Performance of Video Capture

As discussed in the last chapter, AVICap, by default, routes video and audio stream data from a capture window to a file named CAPTURE.AVI in the root directory of the current drive. If we would like to use the data, we must to get the data from the CAPTURE.AVI file (Fig. 5.6). Now we discuss the performance of the original and the modified method.

Fig. 6.1 shows the performance of different situations. First, we focus on the original model and the modified model. These two curves are obtained using the same platform (AMD Athlon XP 3200+, 2.21 GHz, 480 MB RAM). The average of the original model is 10.93367fps, and the variance is 0.440914. The average of the modified model is 30.01967, and the variance is 2.040659. From Fig. 6.1, though the variance of the modified model is bigger, the stability is acceptable. Note that the average efficiency is increased to about 300%!

Now we turn to the modified model and the modified model on another computer.

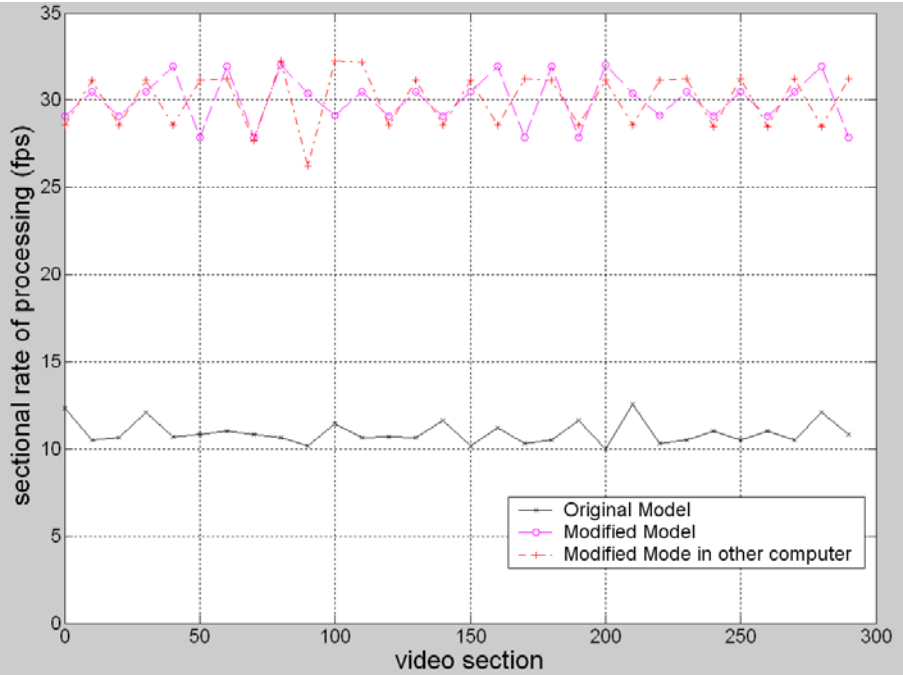


Fig. 6.1: Performance of video capture.

This other computer is Intel Centrino Pentium M 1.5 GHz, with 512 MB DDR RAM. The average is 30.045fps which is a little higher, and the variance is 2.664964, which is also a litter higher. Because the two computers are quite different, we suspect that the limit to the video capture frame rate is bounded by the web camera (capture device).

Fig. 6.2 shows the results using two different web cameras. Web camera 1 is Logitech's QuickCam Pro4000, and web camera 2 is Logitech's QuickCam Express. Computer 1 is AMD Athlon XP 3200+, 2.21GHz, 480 MB RAM, and computer 2 is Intel Centrino Pentium M 1.5GHz, 512 MB DDR RAM. We see that the factor impacting frame rate does not reside in the personal computer, but in the capture device.

Because the bottleneck of video capture depends on the capture device, but not the software or the personal computer, Hence we can not use VTune to analyze the clockticks to do in-deep research.

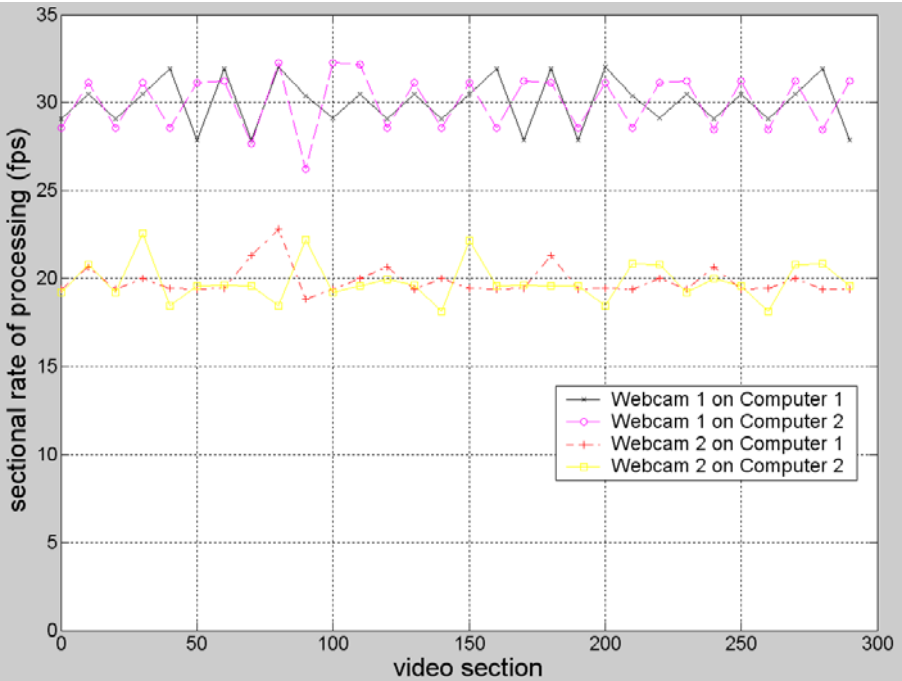


Fig. 6.2: Using different web camera on different computers.

6.2 Performance of Video Segmentation System

Fig. 6.3 shows the simulation result with and without use of MMX instructions. The simulation is run using the original video capture model, in the hope that it will not have significant impact on the relative reult. Table 6.1 shows the result we get from VTune software, where 1 clocktick represents 1ms in execution time. The data of Table 6.1 comes from the data without MMX instructions in Fig. 6.3. Fig. 6.4 illustrates the relative computing time of the modules use of MMX instructions. Table 6.2 gives us the clockticks analysis. We catch the scene of our lab as the input to collect the data and there are totally 480 frames in this simulation.

We find the mean sectional rate of processing without MMX code is 7.523103 and the variance is 0.018376; the mean sectional rate of processing with MMX code is 8.132414 and the variance is 0.11366. The sectional rate of processing efficiency is enhanced by $(8.132414 - 7.523103)/7.523103 = 8\%$. It seems quite low comparing with MPEG-4 video encoder optimization using MMX. There are two main reasons:

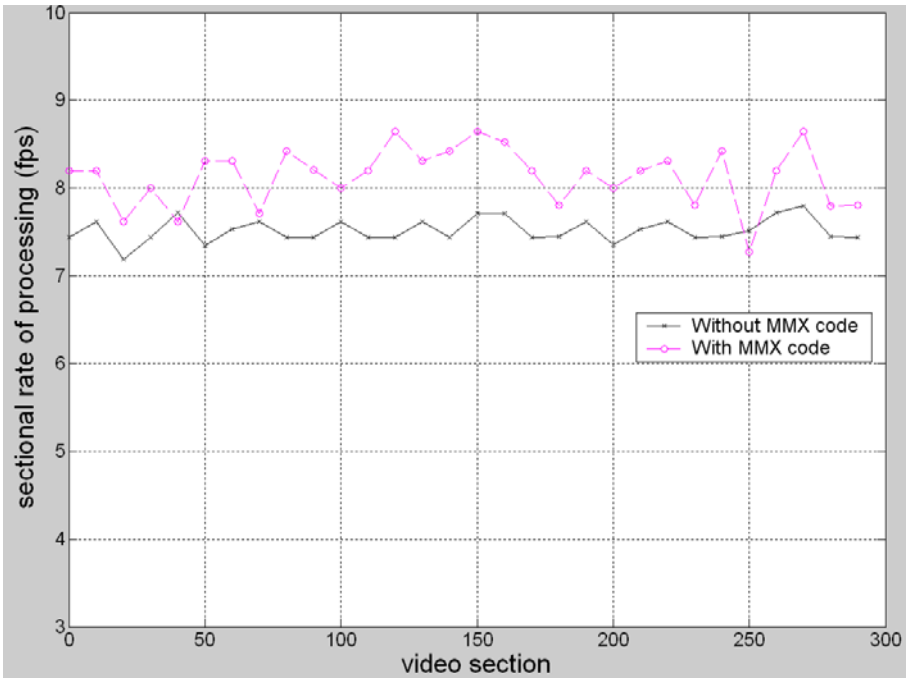


Fig. 6.3: Comparison between use and not use of MMX instructions.

Table 6.1: Simulation Result of Optimization

Function	Original code clockticks	Modified code clockticks	Total software clockticks	Speedup
Combine background and foreground	497	84	8040	593.25%

1. The video segmentation system does not have a few critical bottlenecks. From VTune software, we could find that the clockticks of all major functions are almost about 400 to 500 with the total clockticks being 8040. Most of them are due to assign functions, such as int x = 300, bool y = true, etc. It takes effort to optimize these functions and we leave it to potential future work.

2. We have not used the capture with modified method yet. This could impact the simulation results deeply. We will consider this issue in the whole system performance analysis.

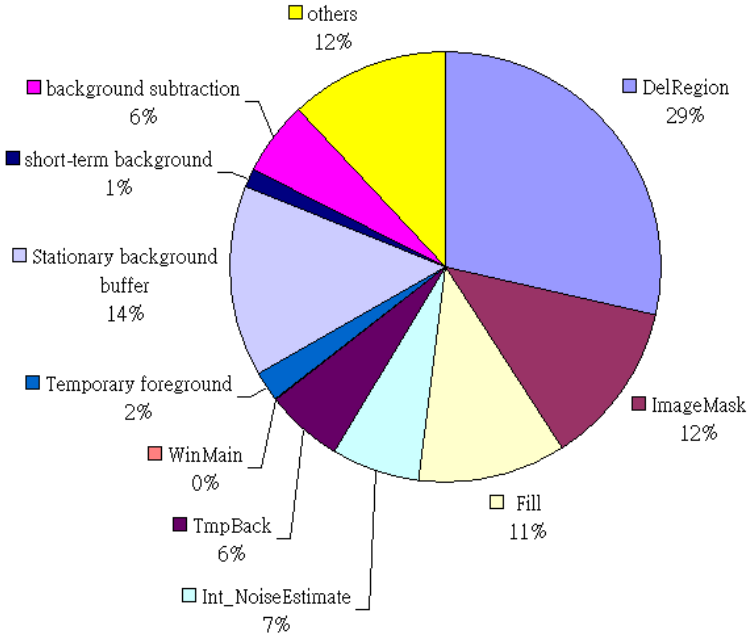


Fig. 6.4: Pie chart when using MMX code.

Table 6.2: Clockticks Analysis with Use of MMX Instructions

Function	Clockticks	Ratio(%)
whole system	8040	100
DelReigon	2295	29
Stationary background buffer	1162	14
ImageMask	981	12
Fill	908	11
Int_NoiseEstimate	538	7
TmpBack	451	6
background subtraction	450	6
Temporary foreground	195	2
short-term background	96	1

6.3 Overall System Performance

Now we turn to the overall videoconference system, which include all the modified methods and the audio encoder system. Fig. 6.5 shows the complexity breakdown we get with the VTune software; Fig. 6.6 shows the sectional rate of processing of the system. The mean sectional rate of processing is 10.69767fps, and the variance is 0.054012. Table 6.3 shows the clockticks analysis. We catch the scene of our lab as the input to collect the data and there are totally 710 frames in this simulation. We divide the functions into MPEG-4 video encoder, MPEG-4 audio encoder, capture, segmentation, RTP, and C++ library. Obviously, we can see one thing from the above data: MPEG-4 audio encoder, capture, and RTP occupy only a little time in the whole program.

Tables 6.4 and 6.5 shows more details about the MPEG-4 audio encoder and the RTP of the overall system. We catch the scene of our lab as the input to collect the data and there are totally 4150 frames in this simulation. However, as we stated before, because video and audio capture functions are bounded by the capture device, we cannot get the clockticks information.

Compared with the video segmentation system, we are surprised that the sectional rate of processing would still maintain at 10.7 even though we add many functions inside, such as audio capture, audio encoder, video encoder, and RTP functions. We could ascribe it mainly to the modified video capture method. With so many functions inside the system, the sectional rate of processing finally is not bounded by the capture device, but by the MPEG-4 video encoder instead.

We could realize it more clearly from Figs. 6.7 and 6.8. Table 6.6 shows the clockticks analysis. We catch the scene of our lab as the input to collect the data and there are totally 3640 frames in this simulation. Without the MPEG-4 video encoder inside, the sectional rate of processing seems to be bounded still by the video capture device. Hence we could affirm the whole bottleneck of the video

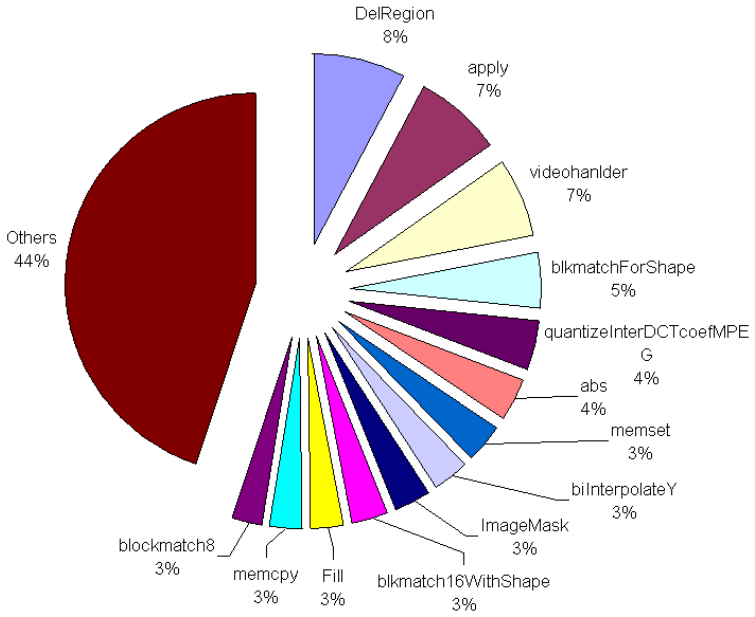


Fig. 6.5: Complexity breakdown of overall system.

conference is the MPEG-4 video encoder. Unfortunately, from Fig. 6.5, there are not clearly identifiable bottleneck functions inside the MPEG-4 video encoder. Continue optimization of the MPEG-4 video encoder is left to potential future work.

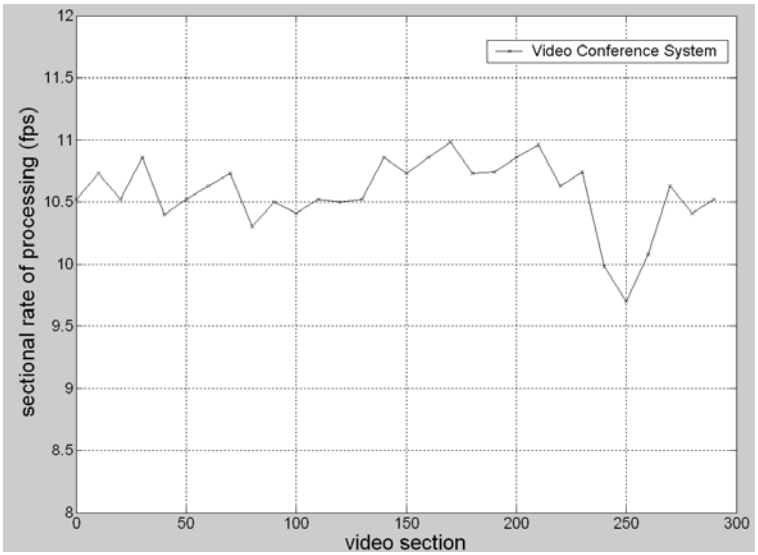


Fig. 6.6: Whole system sectional rate of processing analysis.

Table 6.3: Clockticks Analysis of Overall System

Function	classification	Clockticks	Ratio(%)
whole system		47385	100
DelReigon	segmentation	3637	8
apply	MPEG-4 video encoder	3411	7
videohandler	segmentation	3088	7
blkmatchForShape	MPEG-4 video encoder	2230	5
quantizeInterDCTcoefMPEG	MPEG-4 video encoder	1934	4
abs	C++ library	1685	4
memset	C++ library	1552	3
bilnterpolateY	MPEG-4 video encoder	1418	3
ImageMask	segmentation	1369	3
blkmatch16WithShape	MPEG-4 video encoder	1353	3
Fill	segmentation	1311	3
memcpy	C++ library	1282	3
blockmatch8	MPEG-4 video encoder	1166	3

Table 6.4: Clockticks Analysis of Overall System about MPEG-4 Audio Encoder

Function	Clockticks	Ratio(%)
whole system	582,545	100
faacEncEncode	232	0.04
uc2s_array	45	0.01
sf_read_short	1	0
pcm_read_uc2s	1	0
faacmain	0	0
faacEncOpen	0	0
read_fmt_chunk	0	0
psf_sprintf	0	0
wav_close	0	0

Table 6.5: Clockticks Analysis of Overall System about RTP

Function	Clockticks	Ratio(%)
whole system	582,545	100
PollSocket	54	0.01
CalculateDeterminist	0	0
FillInSDES	0	0
GetRTCPPort_NBO	0	0

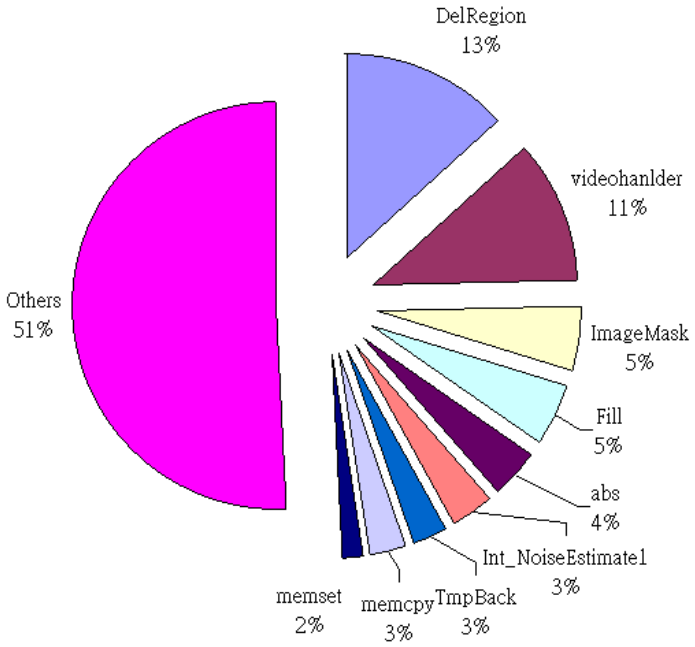


Fig. 6.7: Whole system performance analysis without MPEG-4 video encoder inside.

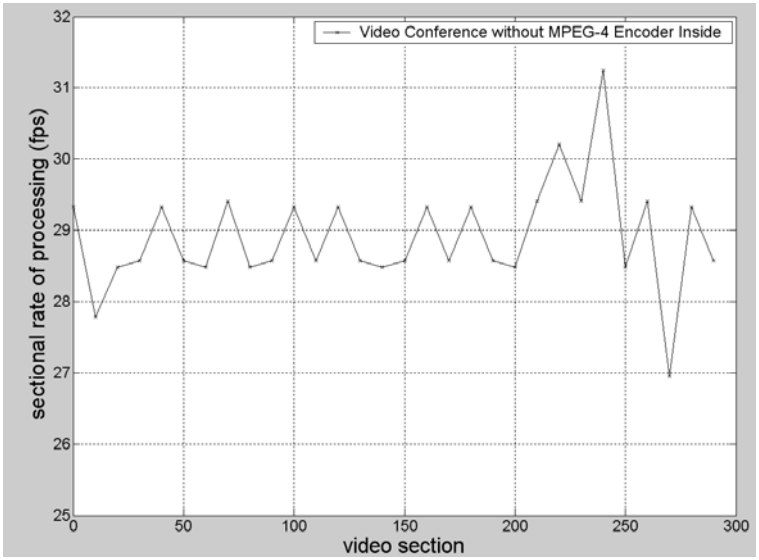


Fig. 6.8: Whole system sectional rate of processing analysis without MPEG-4 video encoder inside.

Table 6.6: Clockticks Analysis of Overall System Without MPEG-4 Encoder

Function	classification	Clockticks	Ratio(%)
whole system		47244	100
DelReigon	segmentation	5379	11
videohandler	segmentation	3088	7
ImageMask	segmentation	2416	5
Fill	segmentation	2329	5
abs	C++ library	1852	4
Int_NoiseEstimate1	segmentation	1486	3
TmpBack	segmentation	1349	3
memcpy	C++ library	1275	3
memset	C++ library	880	2

Chapter 7

Conclusion and Future Work

We developed and implemented of an video conference system on personal computer. We use the function supplied by SDK library to capture video and audio stream. Using video segmentation to take the foreground and compressed the data in MPEG-4 video encoder. In the audio aspect, we just send the audio stream into MPEG-4 audio encoder. Finally, delivering the compressed data in the network through RTP.

Getting video capture, the video segmentation system, and the optimized MPEG-4 video encoder, we modified the video capture not to write directly into virtual disk but memory instead. Speeding up the video segmentation by modify the noise stage estimation, correct a bug about U and V components, cancel the residual interface, and speed it up with MMX instructions. We also modify the input and output of MPEG-4 video encoder to make it suitable for video conference system. Besides, we integrate some new system into the video conference, such as audio capture, MPEG-4 audio encoder, and RTP programs.

Doing the job of integration, we also should do the setting of environment. Changing the C program file into C++ program file, and integrating the library of visual 6.0 edition into the visual .NET edition. Moreover, combining the Intel compiler into .NET edition, too.

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

1. Speeding up the frame rate.

   Though we spend many time into speeding up, the frame rate, about 10, is still not satisfied. After analyzing the whole video conference system, we know the bottleneck is MPEG-4 video encoder. There's two ways to go. One is keeping optimizing and the other is to change another edition or using other encoder.

2. Making the segmentation system more suitable

   The segmentation system we get sometimes doesn't work well. For example, when we take something with lattice as the view. The segmentation system would always consider it as the foreground. We should add more function to avoid these condition.

3. Promoting the stability.

   The program isn't very stable, especially when we communicate the decoder. We should add sleep instruction then the decoder could receive the data and decoder it. If not, the decoder receive nothing from the network. Besides, when the decoder interrupts, sometimes the encoder would interrupts, too. I think we should try to correct these bugs.

4. Promoting the compatibility of the environment.

   The compatibility of the environment isn't very good, too. Give an example involve Intel compiler. Every time when we open an project, we would change change condition of the item "EnableWPO" of Intel Specific(R) account. This is because the compatibility between .NET and Intel compiler isn't perfect enough. We should think another method to integrate environment to approve the condition.

Bibliography

[1] C.-K. Chien, "A multimpoint videoconference receiver for MPEG-4 objected-based video," M.S. thesis, Department of Electrical Engineering, National Chaio Tung University, Hsinchu, Taiwan, R.O.C., June 2005.

[2] Y.-H. Lin, "Real-time video segmentation based on background modeling for videoconferencing," M.S. thesis, Department of Electrical Engineering, National Chaio Tung University, Hsinchu, Taiwan, R.O.C., June 2004.

[3] M.-Y. Liu, "Real-time implementation of MPEG-4 video encoder using SIMD-enhanced Intel processor," M.S. thesis, Department of Electronics Engineering, National Chaio Tung University, Hsinchu, Taiwan, R.O.C., July 2004.

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

[5] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," RFC 1889, Audio-Video Transport Working Group, GMD Fokus, Precept Software, Inc., Xerox Palo Alto Research Center, and Lawrence Berkeley National Laboratory, Jan. 1996.

[6] T. Aach, A. Kaup, and R. Mester, "Statistical model-based change detection in moving video," *Signal Processing*, vol. 31, pp. 165–180, Mar. 1993.

[7] Y.-H. Jan and D. W. Lin, "Video segmentation with extraction of overlaid objects via multi-tier spatio-temporal analysis," *Int. J. Elec. Eng.,* vol. 11, no.3, pp.205–217, Aug. 2004.

[8] T. Meier and K. N. Ngan, "Video segmentation for content-based coding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 9, no. 8, pp. 525–538, Dec. 1999.

[9] J. F. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Machine Intell.,* vol. 6, pp. 679–698, Nov. 1986.

[10] "Canny operator code," http://ouray.cudenver.edu/ na0alber/DataCompressionPaper.htm.

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

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

[13] 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.

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

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

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

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

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

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

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

[21] D. E. Comer, *Internetworking with TCP/IP, vol. 1.* Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[22] Eastlake, D., Crocker, S., and J. Schiller, "Randomness recommendations for security", RFC 1750, DEC, Cybercash, MIT, Dec. 1994.

[23] "Video for Windows," http://msdn.microsoft.com/library/psdk/multimed/avifile_8dgz.htm.

[24] "VidCap: Full-featured video capture application," http://msdn.microsoft.com/library/devprods/vs6/visualc/vcsample/vcsmpvidcap.htm.

[25] "Capture without using disk storage," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_capture_without_using_disk_storage.asp.

[26] "Microsoft RIFF," http://netghost.narod.ru/gff/graphics/summary/micriff.htm.

[27] "Recording with a waveform-audio device," http://msdn.microsoft.com/library/default.asp?url=/library /en-us/multimed/htm/_win32_recording_with_a_waveform_audio_device.asp

[28] Intel, *Architecture MMX Technology — Programmers Reference Manual.* 1996.

[29] Martin Wolters, Kristofer kjorling, Daniel Homm and Heiko Purnhagen, "A closer look into MPEG-4 high efficiency AAC," presented at the 115th Convention., NY, USA, Oct. 2003.

[30] Pourmohammadi-Fallah, Y. Asrar-Haghighi, K. and Alnuweiri, H, "Internet delivery of MPEG-4 object-based multimedia," *IEEE Trans. Multimedia,* vol. 10, issue 3, pp. 68–78, July 2003.

[31] Microsoft, *ISO/IEC 14496 (MPEG-4) Video Reference Software User Manual.* Oct. 2004.

[32] "JRTPLIB 3.1.0," http://research.edm.luc.ac.be/jori/jrtplib/jrtplib.html.

[33] " JThread," http://research.edm.luc.ac.be/jori/jthread/jthread.html.

[34] "AudioCoding.com," http://www.audiocoding.com/

作者簡歷

蔡鎮宇，民國七十年四月出生，民國九十二年六月畢業於國立交通大學電子工程學系，並於同年九月進入國立交通大學電子研究所系統組就讀，從事多媒體系統方面相關研究。民國九十四年六月取得碩士學位，碩士論文題目為『採用以 MPEG-4 物件形式視訊編碼之視訊會議傳送端之整合』。