2010

A framework for multimedia playback and analysis of MPEG-2 videos with FFmpeg

Anand Saggi
Iowa State University

A framework for multimedia playback and analysis of MPEG-2 videos with FFmpeg

by

Anand Saggi

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science
Program of Study Committee:
Wallpak Tavanapong, Co-Major Professor
Johnny Wong, Co-Major Professor
Wensheng Zhang

Iowa State University
Ames, Iowa
2010

TABLE OF CONTENTS

ABSTRACT

Fast Forward Motion Pictures Expert Group (FFmpeg) is a well-known, high performance, cross platform open source library for recording, streaming, and playback of video and audio in various formats, namely, Motion Pictures Expert Group (MPEG), H.264, Audio Video Interleave (AVI), just to name a few. With FFmpeg current licensing options, it is also suitable for both open source and commercial software development. FFmpeg contains over 100 open source codecs for video encoding and decoding.

Given the complexities of MPEG standards, FFmpeg still lacks a framework for (1) seeking to a particular image frame in a video, which is needed for accurate annotation at the frame level for applications in fields such as medical domain, digital communications and commercial video broadcasting and (2) motion vectors extraction for analysis of motion patterns in video content. Most importantly, FFmpeg code base is not well documented, which has raised a significant difficulty for developing an extension.

As our contributions, we extended FFmpeg code base to include new APIs and libraries support accurate frame-level seek, motion vector extraction, and MPEG-2 video encoding/decoding. We documented FFmpeg MPEG-2 codec to facilitate future software development. We evaluated the performance of our implementation against a high-performance third-party commercial software development kit on videos captured from television broadcasts and from endoscopy procedures. To evaluate the usability of our libraries, we integrated them with some commercial applications. In the following sections, we will discuss our software architecture, important implementation details, performance evaluation results, and lessons learned.

ACKNOWLEDGMENTS

I am grateful to my major advisor, Dr. Wallapak Tavanapong. During the last two years, she gave me invaluable guidance and support with endless patience. Also, I would like to say thanks to Dr. Johnny Wong and Dr. Wensheng Zhang for serving on the committee and providing valuable feedbacks on this thesis. I appreciate Dr. Piet C. de Groen at Mayo Clinic Rochester for providing all the colonoscopy videos. I want to thank my colleagues Sean Stanek and Kihwan Kim, for their discussions and help. Lastly, I am forever indebted to the love and support of my parents and brother during all these years when I am far away from home.

CHAPTER 1: INTRODUCTION

MPEG standards are a set of standards developed by the Moving Pictures Expert Group (MPEG). We focus on MPEG-2, the second of the MPEG standards. MPEG-2 is widely used as a format for digital television signals that are broadcast by terrestrial (over-the-air), cable, and direct broadcast satellite TV systems. It is also used as a format for distribution of movies and other programs on DVD and similar media. As such, TV receivers, DVD players, and other equipment are often designed to support this standard. Parts 1 and 2 of MPEG-2 were developed jointly with Telecommunication Standardization Sector of International Telecommunication Union (ITU-T), and they have a respective catalog number in the ITU-T Recommendation Series. The Video section of MPEG-2, is similar to the previous MPEG-1 standard, but also includes support for interlaced video and high quality video. With some enhancements, MPEG-2 is also used in some High Definition television transmission systems.

FFmpeg is a complete, cross-platform solution to record, convert, and stream audio and video. It provides plethora of encoders, decoders, parsers, muxers, etc. FFmpeg supports several MPEG standards including MPEG-2. FFmpeg is licensed under GNU General Public License (GPL) with some portions of it (including MPEG-2) licensed under GNU Lesser General Public License (LGPL). FFmpeg is an extremely popular library accepted in the Google Summer of Code for the past three years. The library is a complete solution that meets most multimedia needs. Nevertheless, there are some important features that are missing from FFmpeg.

1.1. Problem Statement

First, FFmpeg does not provide ability to request a particular frame based on frame numbers. This feature is desirable for content-based video analysis and for users of video player applications to jump to a specific frame for annotation and comparison of content among frames. This “frame level seek” ability is important for applications used in medical

image research, video content analysis, and digital video broadcasting. FFmpeg implementation of the MPEG standard requires parsing of an entire video file to record locations (file offsets) of important frames (termed “I-frames” hereafter) before frame level seek can be performed. Parsing frame headers of a complete video file causes initial load time to be directly proportional to the length of the video. Long initial load time is a major drawback for any software requiring interactions with users.

Second, FFmpeg lacks APIs for extracting motion vectors from MPEG video files. Motion vectors are key results of motion compensation process that exploits similarity between neighboring frames for video compression by storing the location difference and the pixel value difference of similar blocks of pixels between neighboring frames instead of storing pixel values of individual frames redundantly. Motion vectors are often used for recognition of various motion patterns in a video such as camera motions and object motions. Motion estimation and compensation are essential to many modern video compression algorithms. FFmpeg internally uses eight motion estimation algorithms, but lacks the APIs for application developers to easily select the desired algorithm and extract motion vectors for subsequent motion analysis and display.

Finally and most importantly, extending FFmpeg code base to provide additional functionality is difficult and time consuming because of (1) lack of support from development community, (2) evolving APIs, and (3) lack of documentation for the FFmpeg library code. These reasons cause delay in development, integration, testing and time to market, for applications utilizing the FFmpeg library despite its flexible software licensing options. Lack of documentation for over a million lines of FFmpeg code is a major reason that deters application developers from taking full advantage of FFmpeg.

1.2 Proposed Approach

To extend the usability of FFmpeg for applications needing frame level seek, we propose an adaptation layer library, providing frame level seek based on frame numbers. This approach includes (a) a new set of APIs in addition to the current seek API based on time stamp in FFmpeg, and (b) two frame level seek algorithms for videos encoded with MPEG-2. To reduce the development, integration, testing and time to market for using

FFmpeg in multimedia applications, we propose a simple, stable set of APIs reusing what are available in FFmpeg as much as possible. The new APIs will simplify the task of application developers for integrating FFmpeg with their respective applications to support features such as encoding, transcoding, splitting video files into smaller clips, and joining different video clips into a single file.

To support research in motion analysis, we propose a motion vector extraction framework for FFmpeg to extract motion vectors from MPEG-2 videos. To support FFmpeg developers and application developers using FFmpeg in their application, we set up an FFmpeg support Web-Wiki to be used as a reference for an individual or a group trying to understand, enhance, or integrate the FFmpeg library.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 provides background on MPEG standard, FFmpeg, and other commercial codec products. Chapter 3 presents our design, architecture, and implementation of Frame Level Seek library in detail. Chapter 4 discusses the new enhanced framework for encoding, the new set of APIs provided to application developers and implementation, integration and usage of these APIs in detail. In Chapter 5, we describe the design and implementation of the motion vector extraction library. We discuss the details of the approach used by FFmpeg MPEG encoder to compute motion vectors. We present performance evaluation of our library and a third party commercial software development kit in Chapter 6. The last chapter summarizes our work and describes future extensions. We describe FFmpeg compilation and installation procedures under GPL and LGPL licenses for Microsoft Windows Operating Systems.

CHAPTER 2: BACKGROUND

This chapter provides background on principles of generic video compression and decompression and principles of MPEG-2 codec that are relevant to our work. We discuss video packets, packet header, packet start codes, and terminology used in the MPEG-2 standard henceforth. We explain different types of formats/profiles within the standard. We compare FFmpeg with various other products currently available in the market based on cost, support, feature availability, and usability.

2.1 Principles of Video Encoding and Decoding

Video encoding removes redundant and less important information from an input signal. Video decoding reconstructs an approximation or exact visual frames and audio frames from the encoded file.

Types of redundancy are as follows. (1) Spatial Redundancy: Pixel values are correlated with those of neighbors, within the same frame. The value of a given pixel is predictable to a certain extent given the values of its neighboring pixels. (2) Temporal Redundancy: Pixel values are correlated with neighbors across frames. The value of a pixel is predictable to some extent given the values of neighboring pixels from the previous or next frame. (3) Entropy Redundancy: For any non-randomized digitized signal, some code values occur more frequently than others. Entropy encoding encodes frequent values with shorter code words than those generated from rare values. (4) Psycho Visual Redundancy: Human eyes do not respond equally to all visual information. The human visual system does not rely on quantitative analysis of individual pixel values when interpreting an image – an observer searches for distinct features and mentally combines them into recognizable groupings. In this process certain information is relatively less important than other – this information is called psycho visually redundant.

Depending on the application requirements, which could range from size of encoded data, quality of audio/video, bit rate, etc., we may envisage two types of coding. (1) Lossless coding: The aim of "lossless" coding is to reduce image or video data for

storage and transmission while retaining the quality of the original images - the decoded image quality is required to be identical to the image quality prior to encoding. Examples include Huffman coding, Arithmetic coding, and Shanon-Fano coding. (2) Lossy coding: This is relevant to the applications envisioned by MPEG-2 video standards - is to meet a given target bit rate for storage and transmission. Examples include linear prediction and transform coding.

Some applications require constrained and efficient storage of videos. In these applications high video compression is achieved by degrading the video quality - the decoded image "objective" quality is reduced compared to the quality of the original images prior to encoding. The smaller the required size the higher the compression is necessary and usually more coding artifacts become visible. The ultimate aim of lossy coding techniques is to optimize image quality for a given target bit rate subject to "objective" or "subjective" optimization criteria. The degree of image degradation (both the objective degradation as well as the amount of visible artifacts) depends on the complexity of the image or video scene as much as on the sophistication of the compression technique. For simple textures in images and low motion activity a good image reconstruction with no visible artifacts may be achieved even with simple compression techniques.

Sub-sampling reduces the dimension of the input video (horizontal dimension and/ or vertical dimension) and thus the number of pixels to be encoded prior to the encoding process. For some applications video is also sub-sampled in temporal direction to reduce frame rate prior to coding. At the receiver decoded images are interpolated for display.

Specific physiological characteristics of the human eyes are utilized to remove subjective redundancy contained in the video data. For instance, the human eye is more sensitive to changes in brightness than to chromaticity changes. Therefore, pixel values are divided into YUV components (one luminance and two chrominance components). Next the chrominance components are sub-sampled relative to the luminance component with a Y: U: V ratio specific to particular applications.

A discrete cosine transform (DCT) expresses a sequence of finitely many data points in terms of a sum of cosine functions at different frequencies. It is a “lossy” compression technique. The discrete cosine transform, is often used in signal and image processing, especially for “lossy” data compression, because it has a strong "energy

compaction" property. That is, most of the signal information tends to be concentrated in a few low-frequency components of the discrete cosine transform for signals based on certain limits of Markov processes.

Motion compensation is a powerful tool to reduce temporal redundancies between frames and is used extensively as a prediction technique for temporal coding. The concept of motion compensation is based on the estimation of motion between video frames, i.e., if all elements in a video scene are approximately spatially displaced, the motion between frames can be described by a limited number of motion parameters (i.e., by motion vectors for translatory motion of pixels).

Usually both prediction errors and motion vectors are transmitted to the receiver. However, computing one motion vector per pixel is generally neither desirable nor necessary. Since the spatial correlation between motion vectors is often high, it is sometimes assumed that one motion vector is representative for the motion of a "block" of adjacent pixels. Motion vectors are used to compress video by storing the changes to an image from one frame to the next. The process is a bi-dimensional pointer that communicates to the decoder how much left or right and up or down, the prediction macro block is located from the position of the macro block in the reference frame or field. The syntax and scale of the motion vectors depend on information that is included in the picture header and picture coding extension header.

2.2 MPEG-2 Standard

MPEG-2 is a standard for "the generic coding of moving pictures and associated audio information". It describes a combination of “lossy” video compression and “lossy” audio data compression that permit storage and transmission of files using currently available storage media and transmission bandwidth. The MPEG standard makes use of the fact the human sensory system is less acutely aware of certain aspects of imagery. Therefore, some data can be removed with little or no impact to viewing experience. It also combines run-length bit compression and standard Huffman encoding techniques to take the resultant data where information has been removed, and turn it into a small bit-stream.

MPEG-2 is a packet based data encoding standard. Data is divided into packets of a specific size, which are distinguished by header information, a constant number of fixed size packets are further encapsulated into Pictures, which are further grouped into a group of pictures (GOP). A video picture is conceptually a frame. It is made up of a number of data packets, containing PTS (presentation timestamp) and DTS (decoding timestamp) values (explained later in this section) in the picture header. To reach a particular frame, the PTS and DTS values for the first and the last data packets forming that frame are required, as there is no concept of frame numbers in the MPEG standard, but PTS and DTS information fields.

The two different timestamps, PTS and DTS, are needed because of the presence of three different types of frames in an MPEG encoded video: I-frame, P-frame, and B-frame. Intra frame, also called I-frame, requires no other frames for decoding. P-frame or Predicted frame is deduced from the previous frame (I or P) and cannot be decoded if the decoder has not decoded the previous frames. B-frame or Bi-Predictive frame is decoded from the previous and next I-frames or P-frames. Since B-frames depend on both past and future pictures, the decoder needs future I-frames or P-frames before B-frames can be decoded.

PTS could be thought of as display frame number. It is a 33-bit number coded in three separate fields. It indicates the intended time of presentation in the system target decoder of the presentation unit that corresponds to the first access unit that commences in the packet. In simple words it is the time at which the decompressed packet will be presented to the user. PTS value must be larger or equal to DTS value for each frame since a frame cannot be displayed before the frame is decoded.

DTS is a 33-bit number coded in three separate fields. It indicates the intended time of decoding in the system target decoder of the first access unit that commences in the packet that is the decoding frame number, the time at which the packet is decompressed. According to the MPEG standards, DTS shall appear in a packet header if and only if the following two conditions are met.

- PTS is present in the packet header
- The decoding time differs from the presentation time.

PTS may be present in any packet header with the following exception. If no access unit commences in the packet data, PTS shall not be present in the packet header. If a PTS is present in a packet header it shall refer to the presentation unit corresponding to the first access unit that commences in the packet data. The PTS and DTS fields are generated by the encoding system.

PTS and DTS fields are used for synchronization of audio and video in addition to a clock reference field. The use of a common time base, the system time clock (STC), to unify the measurement of the timing of coded data and the timing of the presentation of the data (the PTS and DTS fields), ensures correct synchronization. PTS and DTS fields are not necessarily encoded in each picture or audio packet unit. However, these fields are required to occur with intervals not exceeding 0.7 seconds. In video streams, for I-frames and P-frames, the DTS values are nominally equal to the PTS value minus the number of picture periods of video reordering delay multiplied by the picture period.

Table 1: Sample PTS/ DTS Values

Input Picture Index and Type (in coded order)	End-of-picture delivery time (msec)	Decoding / Presentation time (msec)
	0	----
1I	109	210/250
4P	178	250/370
2B	194	290
3B	211	330
7P	280	370/490
5B	297	410
6B	313	450
10P	382	490/610
8B	399	530
9B	427	570
13I	548	610/730

Another important concept in MPEG video encoding is interlacing, which uses two fields to create a frame. One field contains all the odd lines in the image; the other contains all the even lines of the image. A PAL based television display, for example, scans 50 fields every second (25 odd and 25 even). The two sets of 25 fields work together to create a full frame every 1/25th of a second, resulting in a display of 25 frames per second. Such a

scan of every second line is called interlacing. An interlaced video reduces the signal bandwidth by a factor of two, for a given line count and a refresh rate. The human visual system averages the rapidly displayed still pictures into moving picture images. So interlace artifacts (described below) are not usually objectionable when viewed at the intended field rate, on an interlaced video display.

The process of converting interlaced video into progressive video is called de-interlacing. If done poorly, de-interlacing can introduce image degradation. De-interlacing requires the display to buffer one or more fields and recombine them into a single frame. In theory this would be as simple as capturing one field and combining it with the next field to be received, producing a single frame. However, the originally recorded signal was produced as a series of fields and any motion of the subjects during the short period between the fields are encoded into the display. When combined into a single frame, the slight differences between the two fields due to this motion resulting in a "combing" effect where alternate lines are slightly displaced from each other. Most de-interlacing techniques can be broken up into three different groups, all using their own exact techniques. The first group is called Field Combination De-Interlacers because they take the even and odd fields and combine them into one image or frame, which is then displayed. The second group is called Field Extension De-Interlacers because each field (with only half the lines) is extended to the entire screen to make a frame. The third type uses a combination of both groups and fall under the banner of motion compensation.

2.3 Packet Structures in MPEG Streams

Container or a wrapper class is a specification that dictates how data is stored (not encoded) within a file and how much metadata is effectively stored whereas no specific codification of the data is implied or specified. The most relevant family of wrappers is, in fact, to be found among multimedia file formats, where the audio and/or video streams can effectively be coded with hundreds of different alternative algorithms, whereas they are stored in fewer file formats. In this case the algorithm (or algorithms, as in the case of mixed audio and video contents in a single video file format) used to actually store the data is called a codec, e.g., AVI, MPEG (ES), MPEG (TS), Mp4, MOV, etc.

An elementary stream within a container class contains only one kind of data, e.g., audio, video or closed caption. An elementary stream is often referred to as "elementary" "data", "audio", or "video" streams. The format of the elementary stream depends upon the codec or data carried in the stream. A video elementary stream contains compressed video frames, plus sequence headers, group-of-picture (GOP) headers, and other data needed to decode the stream.

Table 2: Header for MPEG-2 Video Elementary Stream

Field Name	Size (bits)	Description
Start code	32	0x000001B3
Horizontal size	12	Width
Vertical size	12	Height
Aspect ratio	4	
Frame rate code	4	
Bit rate	18	Actual bit rate = bit rate * 400
Marker bit	1	Always 1
VBV buf size	10	Size of video buffer verifier = 16*1024*vbv buf size
Constrained parameters flag	1	
Load intra quantizer matrix	1	If bit set then intra quantizer matrix follows, otherwise use default values.
Intra quantizer matrix	0 or 512	
Load non intra quantizer matrix	1	If bit set then non intra quantizer matrix follows.

A Packetized Elementary Stream (PES) defines how elementary streams are arranged in packets within a MPEG program stream. The elementary stream is packetized inside PES packet by encapsulating sequential data bytes of the elementary stream. A typical method of transmitting elementary stream data from a video or audio encoder is to first create PES packets from the elementary stream data and then to encapsulate these PES packets inside Program Stream (PS).

An elementary stream is broken up into packets of variable length, forming a PES. Each PES packet includes a header. In many applications, the audio and video are multiplexed, thus combining the two elements. Packetized and multiplexed elementary streams may take the form of single program streams. The PES header contains

information about the content of the data bytes, allowing a decoder to process the packets. The PES packets can be of variable length, typically up to 64 Kbytes, but they can be longer. The important note is that if information carried in the header is corrupted, the entire PES packet is lost.

Table 3: PES Packet Header

Name	Size (Bits)	Description
Packet start code prefix	3 bytes	0x000001
Stream id	1 byte	Examples: Audio streams (0xC0-0xDF), Video streams (0xE0-0xEF)
PES Packet length	2 bytes	Can be zero. If the PES packet length is set to zero, the PES packet can be of any length. A value of zero for the PES packet length can be used only when the PES packet payload is a video elementary stream.
Optional PES header	Variable length	
Stuffing bytes	Variable length	
Data		

Table 4: Optional PES Header

Name	Size (Bits)	Description
Marker bits	2	10 binary or 0x2 hex
Scrambling control	2	00 implies not scrambled
Priority	1	
Data alignment indicator	1	1 indicates that the PES packet header is immediately followed by the video start code
Copyright	1	1 implies copyrighted
Original or Copy	1	1 implies original
PTS DTS indicator	2	11 = both present, 10 = only PTS
Additional copy info flag	1	
CRC flag	1	
PES header length	8	Gives the length of the remainder of the PES header
Optional fields	Variable length	Presence is determined by flag bits above
Stuffing Bytes	Variable length	0xff

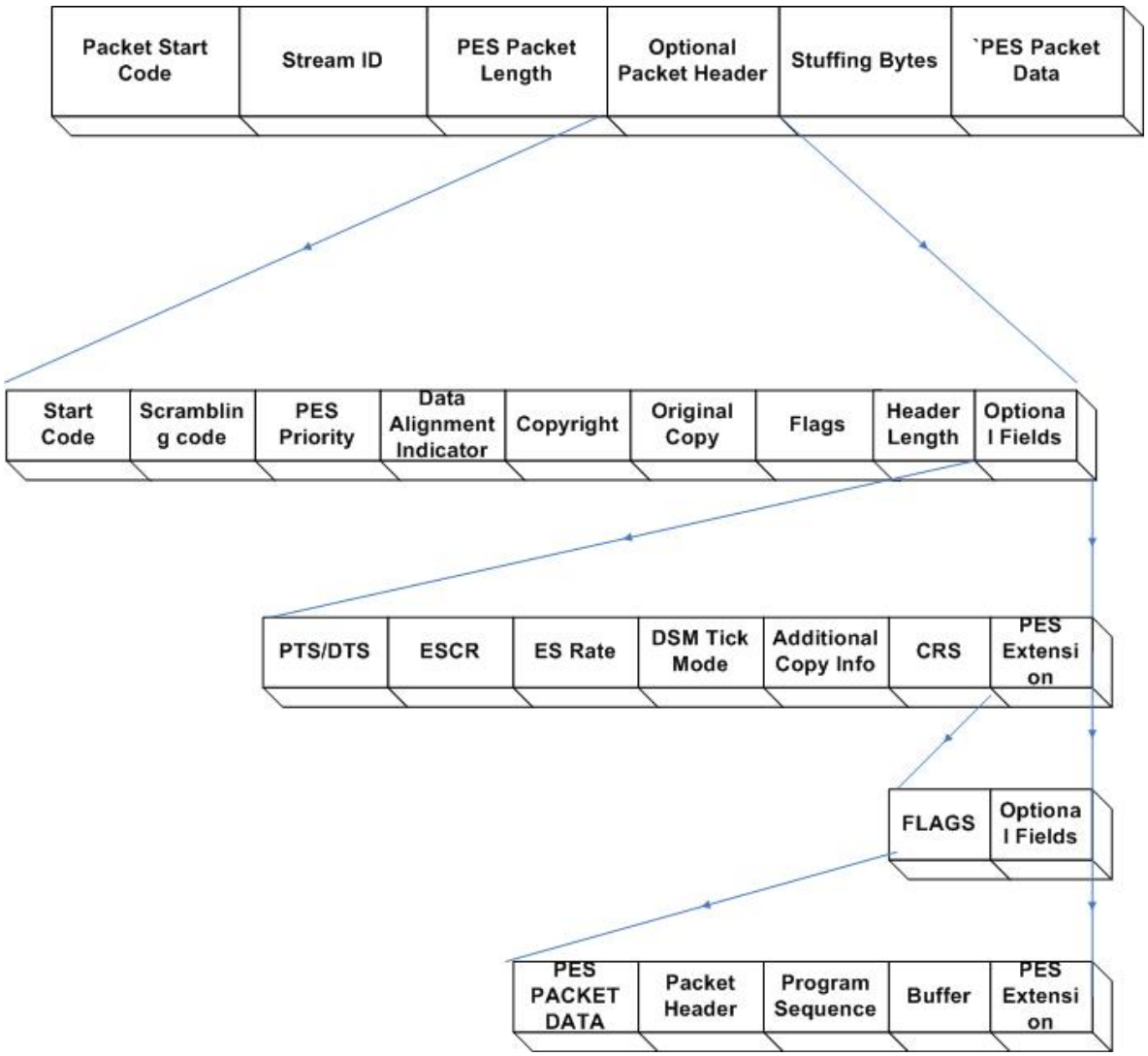

Figure 1: PES Packet Header Structure

Program stream (PS) is a container format for multiplexing digital audio, video and more. Program streams are created by combining one or more PES with a common time base into a single stream.

Table 5: Program Stream Packet Header Format

Name	Size (Bits)	Description
Sync bytes	32	0x000001BA
Marker bits	2	01b
System clock [32…30]	3	System Clock Reference
Marker bit	1	1 Bit always set.
System clock [29…15]	15	System clock bits 29 to 15
Marker bit	1	1 Bit always set.
System clock [14…0]	15	System clock bits 14 to 0
Marker bit	1	1 Bit always set.
SCR extension	9	
Marker bits	2	11 Bits always set.
Reserved	5	Reserved for future use
Stuffing length	3	
Stuffing bytes	8*stuffing length	
System header (optional)	0 or more	If system header start code follows: 0x000001BB

In a video stream based on the MPEG standard, the highest syntactic structure of the coded video bit stream is the video sequence. A video sequence commences with a sequence header, which may optionally be followed by a group of pictures header and then by one or more coded frames. The order of the coded frames in the coded bit stream is the order in which the decoder processes them, but not necessarily in the correct order for display. The video sequence is terminated by a sequence end code. At various points in the video sequence a particular coded frame may be preceded by either a repeat sequence header or a group of pictures header or both. Program streams have variable size records and minimal use of start codes, which would make over the air reception difficult, but has less overhead. The program stream coding layer allows only one program of one or more elementary streams to be combined into a single stream, in contrast to a transport stream, which allows multiple program streams.

Transport streams offer features for error correction for transportation over unreliable media. Transport streams are used in broadcast applications such as DVB (Digital Video Broadcasting). Transport streams are contrasted with MPEG Program Stream (PS), designed for more reliable media such as DVDs.

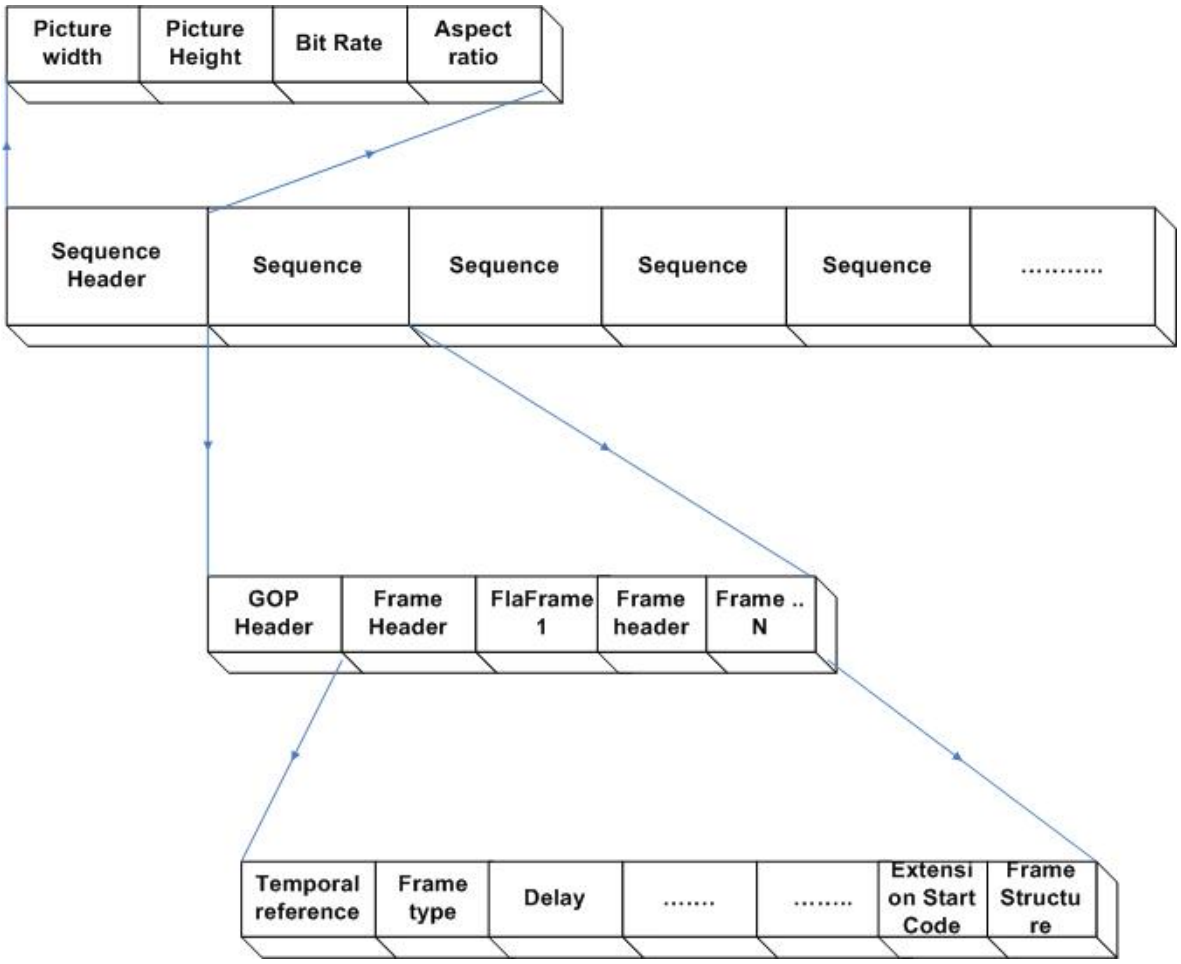

Figure 2: Structure of Coded Video Data

Start or end codes are specific bit patterns that do not otherwise occur in the video stream. Each start code consists of a start code prefix followed by a start code value. The start code prefix is a string of twenty three bits with the value zero followed by a single bit with the value one. The start code prefix is the bit string '0000 0000 0000 0000 0000 0001'. Start code is an eight bit integer that identifies the type of the start code. Most types of start code have just one start code value. All start codes shall be byte aligned. This shall be achieved by inserting bits with the value zero before the start code prefix such that the first bit of the start code prefix is the first (most significant) bit of a byte.

Table 6: Start Codes

Name	Start Code Value (hex)
picture_start_code	00
slice_start_code	01 through AF
user_data_start_code	B2
sequence_header_code	B3
sequence_error_code	B4
extension_start_code	B5
sequence_end_code	B7
group_start_code	B8

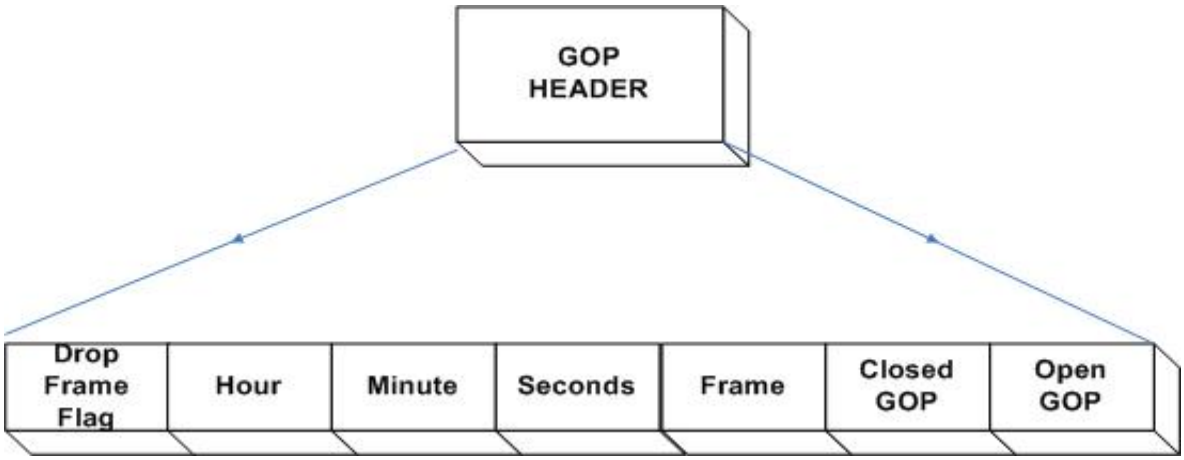

Figure 3: Structure of GOP Header

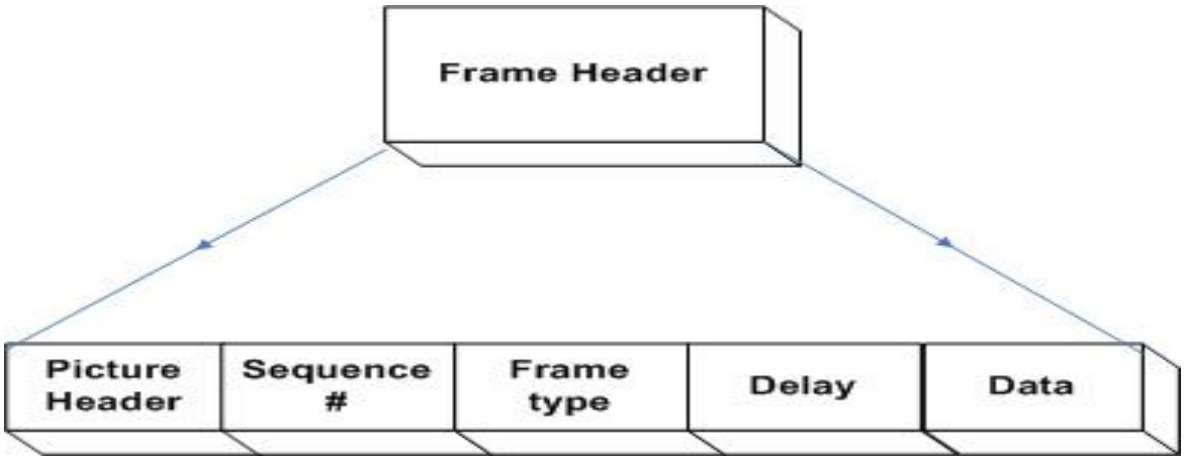

Figure 4: Structure of Frame Header

2.4 MPEG-2 Profiles and Levels

MPEG-2 defines 4 profiles and 4 levels. The profile defines the color space resolution and scalability of bits stream. The level defines the maximum and minimum of image resolutions, Y (Luminance) samples per second, the number of video and audio layers supported for scalable profiles, and the maximum bit rate per profile. The video decoder will depend on its availability and need to handle a particular bit stream. Table 7 describes the four available profiles currently supported by FFmpeg.

Table 7: MPEG Profiles and Levels

Abbr.	Name	Frame rates (Hz)	Picture Coding Types	Chroma Format	Max horizontal resolution	Max vertical resolution	Max bit rate (Mbit/s)
LL	Low Level	23.976, 24, 25, 29.97, 30	I, P	4:2:0	352	288	4
ML	Main Level	23.976, 24, 25, 29.97, 30	I, P, B	4:2:0	720	576	15
H-14	High 1440	23.976, 24, 25, 29.97, 30, 50, 59.94, 60	I, P, B	4:2:0	1440	1152	60
HL	High Level	23.976, 24, 25, 29.97, 30, 50, 59.94, 60	I, P, B	4:2:2 or 4:2:0	1920	1152	80

2.5 Other Products Comparison

Many products are currently available in the market to support video editing, playback and streaming needs. These products have good performance and product support. Considering specific requirements of frame level seek, ease of usability and motion vector extraction, we found four other products comparable with the current FFmpeg library. Except from FFmpeg, none of these are open source. Table 8 shows comparison of FFmpeg with these other products.

Table 8: Other Product Comparison

Abbr.	Name	Operating Systems Supported	License Cost	Video Format Supported	Multi Threaded Application Support
FFmpeg	MPEG-1,2, AVI	Windows, Macintosh, Linux	$00.00	SD, HD	YES
MainConcept	MPEG-1,2	Windows	$519.00	SD, HD	YES
LEADTOOLS MPEG-2	MPEG-1,2	Windows, Macintosh	$400.00	SD, HD	YES
Etymonix SoftReel	AVI	Windows	$20.00	SD, HD	NO
Real Magic NetStream 2000	MPEG-1,2, AVI	Windows	$99.99	SD	NO

2.6 FFmpeg

FFmpeg is an open source Linux library for recording, streaming and playing multimedia. It is written in GNU-C and it's licensed under GPL, but certain parts of it are licensed under LGPL, which makes it a suitable candidate for a low cost - high performance multimedia library. It is a complete, cross-platform solution to record, convert and stream audio and video data.

There are numerous projects known to incorporate work from FFmpeg in Entertainment and Healthcare industry. VLC Media Player, Mplayer, Dr.DivX, Frogger, KMediaFactory, PlayStation Portable Video Converter, PSP Media Player, Quick View

Pro, WMA codec for Mac OS X, Xine, etc are currently using the FFmpeg framework for building media players, video editing software and video streaming. AMIDE, a Medical Imaging data analyzer and EM-Manual both use FFmpeg libraries for their audio/video encoding, decoding and analyzing features. FFmpeg has been successfully ported to all Operating Systems including: Linux (all flavors), UNIX, Windows and MAC.

FFmpeg framework also provides a command line tool, called **ffmpeg** to convert one video file format to another. It can also be used for grabbing and encoding in real time from a TV card. **ffserver,** a FFmpeg tool is an HTTP (RTSP is being developed) multimedia streaming server for live broadcasts. It can also time shift live broadcast. **ffplay,** is a simple media player based on SDL and the FFmpeg libraries.

FFmpeg is licensed under the GPL or LGPL depending on the choice of configuration options. Using FFmpeg or its constituent libraries, we must adhere to the terms of the license in question. FFmpeg can be hooked up with a number of external libraries to add support for more formats. FFmpeg is freely downloadable from SVN and the directory structure of the distribution is explained in Figure 5.

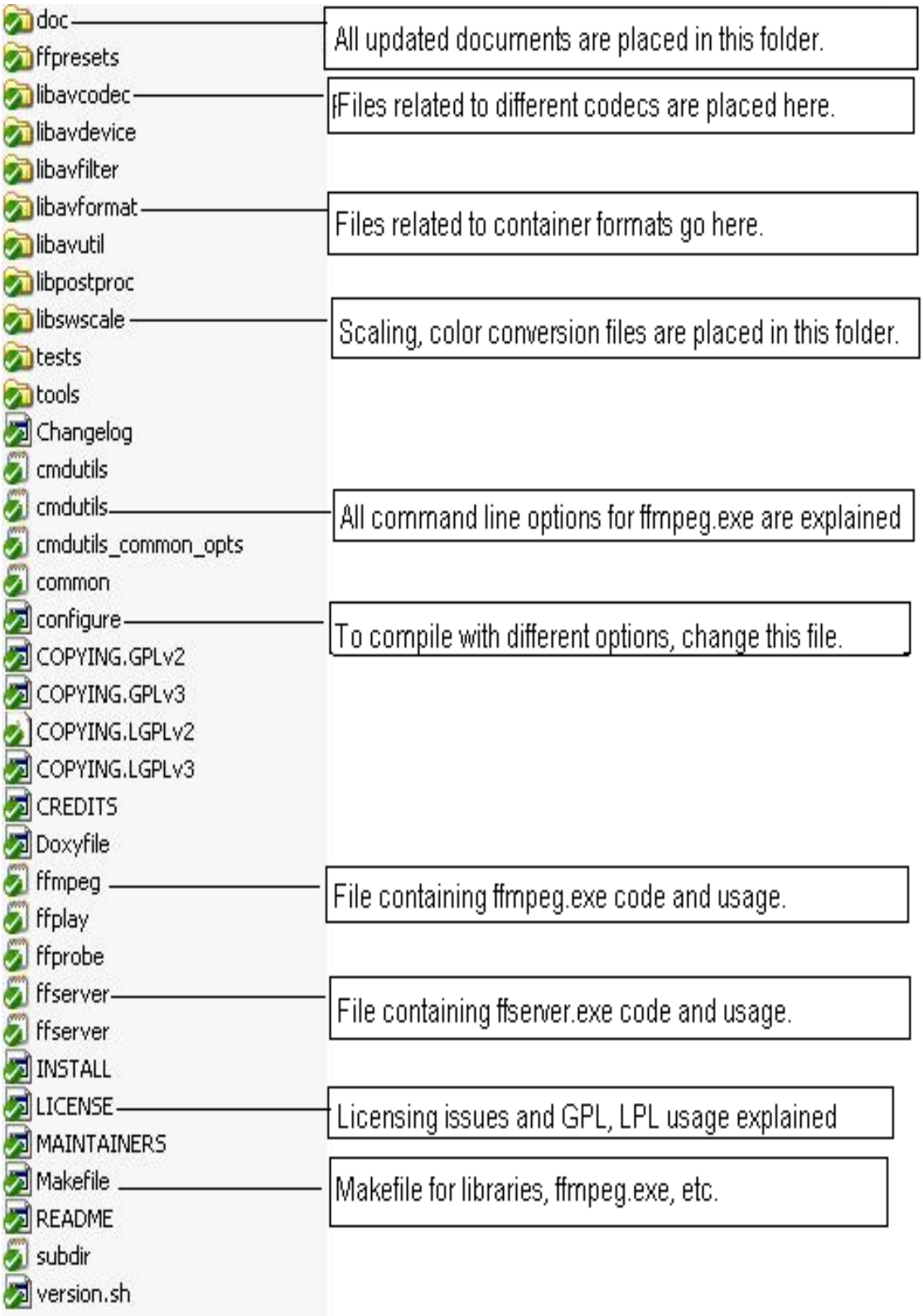

Figure 5: FFmpeg Directory Structure

CHAPTER 3: FRAME LEVEL SEEK LIBRARY

In the last chapter, we described background of MPEG-2 standard and FFmpeg code base. We now present our new Frame Level Seek library developed to further enhance FFmpeg usability. In particular, we describe the architecture of the Frame Level Seek library, its detailed design and APIs for application developers. We describe implementation details of two different approaches and contrast their advantages and disadvantages.

3.1 Limitations of FFmpeg Seek

FFmpeg provides only a timestamp-based seek API *av_seek_frame (AVFormatContext *, int stream_index, int64_t timestamp, int flags).* This API accepts timestamp as input, and seeks to its nearest I-frame. For MPEG, these frames are I-frames. The internal data structures required for accessing the I-frames are populated, with DTS value, PTS value, the current stream pointer, etc. after a call to this API. There is no decoding of data packets involved to perform seek except for the data packets comprising the last frame. The application has to completely decode the packets which comprise the frame pointed to by the timestamp passed and displays it. The API is flexible enough to be used for all registered container classes and codecs. Its functionality is to dynamically select an appropriate API amongst internal seek APIs previously registered for the particular container/codec at the time of configuration. The internal APIs can be based on the byte position, timestamp, etc. The API gets called based on the type of container and codec the video file is encoded with. This current API lacks support for seeking to a specific inter-coded frame (P-frame or B-frame).

3.2 Frame Level Seek Library

Currently there is no support for seeking to a specific P-frame or B-frame in MPEG container in the FFmpeg library. Frame level seek is useful for applications in the field of

image processing, video broadcasting, medical image research, etc. This has motivated us to provide the Frame Level Seek library extending the current FFmpeg code base.

We implement two Frame Level Seek libraries: Slow Frame Level Seek (FAS_SLOW) and Fast Frame Level Seek (FAS_FAST). The major difference between the two libraries lies in the algorithm used to obtain the location of the target frame in the input video file. FAS_SLOW first generates a *seek table* filled with the location of every I-frame in the video. In this process, it uses FFmpeg *av_seek_frame()* which decodes the picture header of every single frame. After the index table is built for the entire stream, seeking to a particular frame is accomplished by searching the seek table for the closest I-frame prior to the target frame. Additional computation is done to seek to the target frame if the frame is not an I-frame. FAS_SLOW therefore has high upfront video load time; however, it is flexible to handle different GOP structures in the same video file. Also it can support Frame level seek with audio stream included in the file. FAS_FAST version generates a seek table on the fly. It uses the header information of first and last GOP and video stream metadata to compute the timestamp of the nearest I-frame before using *av_seek_frame()* to seek to that timestamp. Subsequent decoding is performed to compute the location in the file of the target frame if it is not an I-frame.

Table 9: Major Differences between FAS_Slow and FAS_Fast

	Features	FAS Slow	FAS Fast
1.	Handle Open GOP	Yes	No
2	Handle In- consistent GOP	Yes	No
3.	Load Time	High	Negligible
4.	Accuracy	High	Marginally less
5.	Generate seek table	Yes	Partially

Both versions are part of the FAS layer which is distributed in the form of dlls in Windows system. The APIs exposed to the application are consistent and do not undergo any changes with incremental releases which makes it easier for application integration.

3.3 Design of Frame Level Seek Library

The Frame Level Seek library lies in-between the FFmpeg middleware and the application layer as shown in Figure 6. Application uses public APIs exposed by the Frame Level Seek library, which uses references to the underlying data structures provided by FFmpeg and implements its own data structures. The APIs exposed to application do not maintain internal data structures, but calls private Frame Level Seek functions to perform the actual work.

Frame Level Seek library implements its own seek index table to provide frame accurate seek. The functions provided to access the seek index table are called by public Frame Level Seek APIs and functions provided to maintain the seek index table are called by private Frame Level Seek functions. We mainly focus on the two FFmpeg libraries (1) libavformat.dll and (2) libavcodec.dll. These two libraries provide most of the functions and data structures required for frame level-seek. FFmpeg library works as a middleware between the application and the native operating system. It easily accesses the native system calls to interact with the operating system.

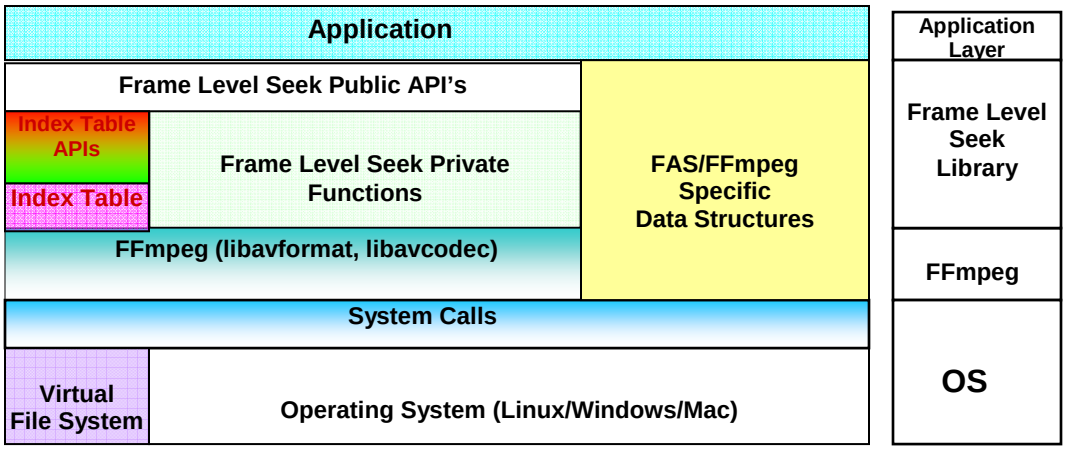

Figure 6: Frame Level Seek Library Layer

The major design decisions for Frame Level Seek library are (1) using static internal frame buffers, (2) using most of the underlying FFmpeg APIs, (3) generation of the seek table to keep track of offsets of I-Frames, (4) providing different sets of APIs for the

index table maintenance and access, and (5) separating public APIs to access underlying data structures and private functions to modify them.

- Using static internal buffers allocated during initialization gives two inherent advantages over flexibility of providing memory management to the application layer: (1) Reduced memory fragmentation caused by avoiding calls to *malloc()* and *free()* from the application layer, and (2) increased performance by reducing the overhead of memory allocation during runtime.
- Reusing the underlying FFmpeg APIs is advantageous since (1) we do not need to redo the code, which reduces implementation time and (2) the APIs are tested and used by the FFmpeg development community. We use ^{FAS} to denote Frame Level Seek library function and ^{FFmpeg} to denote FFmpeg library function hereafter.
- Since it is not possible to seek to a particular P-frame or B-frame in the current FFmpeg implementation, offsets of I-frames are maintained in the seek index table to support this feature.
- The APIs to access and maintain the seek index table for bookkeeping tasks are implemented in a separate module that can be used in both Frame Level Seek Fast and Slow libraries.
- Separation of visibility in APIs on the basis of its nature, mainly maintenance and access, provides security inherently. All public APIs are used by the application to access the internal data structures. The data structures are maintained by private functions, which cannot be called by the application.

3.3.1 State Machine of FFmpeg Frame Level Seek Library

We maintain five states in the frame level seek library as shown in Figure 7. These states are as follows: (1) *Initialize*, when all the data structures are initialized, memory allocation of internal data structures is completed, or registration of requisite codecs is done, etc.,(2) *Idle*, which is the state after initialization, or when a seek or a play command is expected, or after the data has been sent to the application for display.,(3) *Seeking*, which involves two sub states: seeking to a GOP and seeking to a specific frame.,(4) *Frame Extraction*, after seeking to a frame, it has to be converted into the requisite format for

displaying., and(5) *Display*, when a frame is sent to the application for display. After that, internal buffers are flushed and the system again enters the idle state.

We implemented Frame Level Seek library using one thread. This is because frame numbers are not explicitly coded in the video. They are obtained sequentially by incrementing a counter by one after a frame is found. It is possible to use a multithreading approach with a number of threads; each processing its own non-overlapping portion of a file to generate it's seek table. However, this approach is complex as the separate seek tables need to be merged into one table at the end. Furthermore, it has to handle the case where a partition does not start or end at the GOP boundary. Lastly, different threads access disk in a non-linear manner can incur significant disk head movement. The extra time to handle these issues may offset the time gained from multithreading. Therefore, we use a one thread sequential design.

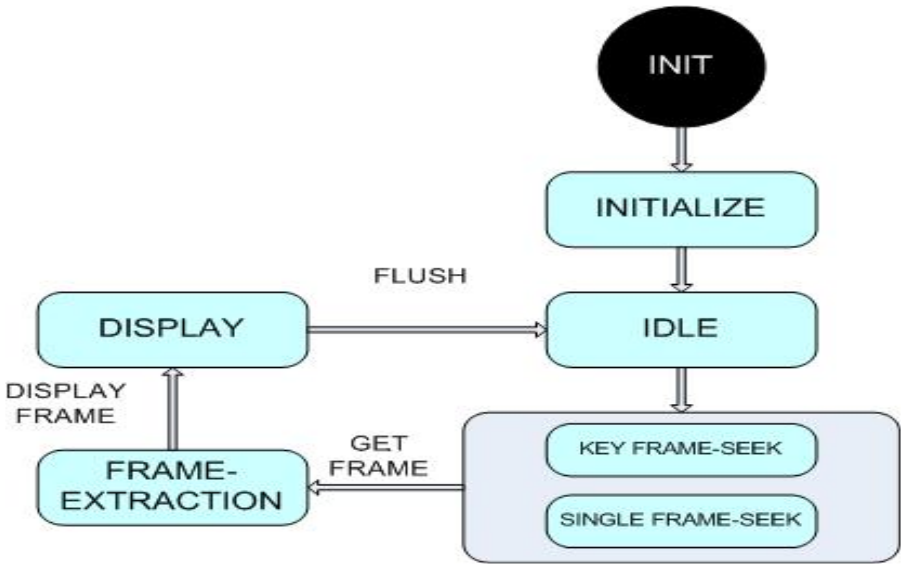

Figure 7: Frame Level Seek State Machine

The major data structures implemented for Frame Level Seek are (1) $fas\_context\_type^{FAS}$ (2) $fas\_vid\_info\_type^{FAS}$ (3) $seek\_table\_type^{FAS}$ (4) $seek\_entry\_type^{FAS}$. The $fas\_context\_type^{FAS}$ structure can be divided into four categories.

Table 10: fas_context_type Structure Members

#	Category	Members
1	FFmpeg Reference	*format_context, codec_context, stream_idx.*
2	Memory Management	*frame_buffer, rgb_frame_buffer, rgb_buffer, deinterlace_buf, raw_frame_buffer*
3	Seek Table Maintenance	*current_dts, previous_dts, keyframe_packet_dts, first_dts*
4	Playback	*current_frame_index, seek_table, vid_info, rgb_already_converted, is_video_active, is_frame_available*

Table 11: fas_vid_info_type Structure Member Description

#	Member	Description
1	*frame_rate*	Frame rate of a stream
2	*img_convert_ctx*	Information for scaling
3	*Fas_timestamp*	Start time of the stream
4	*fas_frame_count*	Exact frame count of a stream
5	*Fas_gop_size*	Actual GOP size
6	*fas_offset*	GOP size without I frames
7	*fas_frame_buffer_count*	Number of extra frames which are not flushed (still in the buffer)
8	*key*	Number of packets in a I-frame
9	*is_hd*	True is video is High Definition; false otherwise
10	*flush*	Flag set to true if flush packets are in the last GOP

3.4 Frame Level Seek Library File Structure

Frame Level Seek (Fast or Slow) code is currently distributed in 6 files, ffmpeg_fas.c, ffmpeg_fas.h, inttypes.h, private_errors.h, seek_indices.c and seek_indices.h.

Files ffmpeg_fas.c and ffmpeg_fas.h expose most APIs required by the application layer.

Files private_errors.h and inttypes.h are responsible for portability between C99 (supported on Linux) and ANSI-C (supported by Windows).

Files seek_indices.c and seek_indices.h provide APIs for the seek table generation, seek table maintenance, and book-keeping tasks.

3.5 Seek Table

The seek table is used in both Frame Level Seek fast and slow libraries. The slow version generates the complete seek table before playback, whereas the fast version partially generates the table reducing the upfront load time. The seek table is a simple structure consisting of the field types shown in Table 12. It has a pointer to an array of elements of type *seek_entry_type* shown in Table 13. The APIs for maintaining and accessing the seek table are divided into categories and shown in Table 14. The Fast Frame level seek (FAS_FAST) accesses only DTS value of the frames in first and the last GOP and generates on the fly the DTS value of the rest of the frames in between. On the contrary, the Slow Frame Level Seek (FAS_SLOW) library accesses the entire seek table linearly to jump to an I-frame prior to the requested frame. The seek table is initialized with a size of 100 fields, but if at any moment during playback a need for more space for fields is felt, the size is made two times of the size at that instant.

Table 12: Description of seek_table_type Structure Member

#	Member	Description
1	*Seek_entry_type*	Reference to the seek table
2	*Completed*	Flag set to true if the table is complete
3	*Num_frames*	Number of frames in the stream
4	*Num_entries*	Current number of frames entered
5	*Allocated_size*	Size of the seek table in bytes

Table 13: Description of the seek_entry_type Structure Member

#	Member	Description
1	*display_index*	Index of a frame in a display order
2	*first_packet_dts*	DTS of the first packet of a frame
3	*last_packet_dts*	DTS of the last packet of a frame

Table 14: API's Exposed by Seek Index Table Library; depreciated function denoted by ^

#	Category	API	Return Value	Parameter's
1	Initialization	*seek_init_table ()*	*seek_table_type*	*int*
2		*Seek_release_table ()*	*void*	*seek_table_type**
3	Seeking	*seek_copy_table ()*	*seek_table_type*	*seek_table_type*
4		*seek_append_table_entry ()*	*seek_error_type*	*seek_table_type, seek_entry_type*
5		*seek_get_nearest_entry ()^*	*seek_error_type*	*seek_table_type, seek_entry_type, int*
6	Support	*compare_seek_tables ()*	*int*	*seek_table_type, seek_table_type*
7		*seek_show_table ()*	*seek_error_type*	*seek_table_type*
8		*seek_show_raw_table ()*	*seek_error_type*	*FILE *, seek_table_type*
9		*read_table_file ()*	*seek_table_type*	*char **

The APIs in the initialization category are for initialization and de-allocation of data structures. We describe the usage of important APIs from different categories: (1) *error_type seek_append_table_entry (seek_table_type *table, seek_entry_type entry)* ^{FAS} is for appending a new entry into the seek table. This API requires pointer to the table allocated during opening of file and an entry value denoted by a structure containing 3 objects *int display_index* ^{FAS}, the *frame_index* ^{FAS} of the I-frame in the video stream, *int64_t first_packet_dts* ^{FAS}, denoting the first packet of the I-frame, *int64_t last_packet_dts* ^{FAS} denoting the last packet of I-frame is primarily responsible for all the bookkeeping tasks required by FFmpeg FAS (Fast). This API returns *seek_error_type* ^{FAS}. (2) *seek_error_type private_resize_table (seek_table_type *table, int new_size)* ^{FAS} to resize the table. The current policy is to double the seek table every time we run out of space for entry. (3) *char * seek_show_table (seek_table_type)* is part of the support category. It can be used for debugging to compare the seek table generated with actual DTS/PTS values of I-frames in the video.

3.6 API Implementation Details

Currently there are fifteen APIs exposed by the Frame Level Seek library. These can be divided into three main categories: (1) Initialization APIs for opening a video, closing a video, registration of codecs, container classes, muxers, demuxers, etc.,(2) Seeking APIs for frame level seek using the FFmpeg middleware. These APIs are for extracting frame data in the internal memory of Frame Level Seek library but not to display, which is the task of the application using the APIs, and (3) Support APIs for accessing the internally maintained data structures. These APIs are not responsible for data maintenance adhering to the design policy.

Table 15: APIs of the Frame Level Seek Library

#	Category	API	Return Value	Parameter's
1	Initialization	*fas_initialize()*	*Void*	*Void*
2		*fas_open_video()*	*fas_error_type*	*fas_context_type*, char *
3		*fas_close_video()*	*fas_error_type*	*fas_context_type**
4	Seeking	*fas_frame_available()*	*fas_boolean_type*	*fas_context_type**
5		*fas_get_frame_index()*	*uint64_t*	*fas_context_type**
6		*fas_step_forward()*	*fas_error_type*	*fas_context_type**
7		*Fas_get_frame()*	*fas_error_type*	*fas_context_type**
8		*fas_seek_to_frame()*	*fas_error_type*	*fas_context_type*, uint target index*
9	Support	*fas_get_frame_count()*	*uint64_t*	*fas_context_type**
10		*fas_get_frame_rate()*	*Float*	*fas_context_type**
11		*fas_get_bit_rate()*	*Int*	*fas_context_type**
12		*fas_get_codec_type()*	*Int*	*fas_context_type**
13		*fas_get_current_width ()*	*Int*	*fas_context_type**
14		*fas_error_type*	*char **	*Int*
15		*fas_get_current_height()*	*Int*	*fas_context_type**

3.6.1 Frame Level Seek Library (Fast)

The fast Frame Level Seek library (FAS_FAST) is developed to reduce the upfront load time of a video file, which is a major drawback of the slow Frame Level Seek library (FAS_SLOW). FAS_SLOW before opening a video file generates the entire seek table containing the DTS value of I-frames, their display index, etc. but FAS_FAST does calculation of the seek index on the fly. It first decodes the first GOP and last GOP to compute the GOP size. As we make sure that GOP is closed and its size remains the same throughout the content, we calculate the DTS value of I-frames from first till last GOP on the fly. The calculation of DTS value without decoding the video stream headers eliminates the upfront load time. The requirements for FAS_FAST to accurate frame seek without upfront load time is as follows: (1) Video stream must consist of only closed group of pictures (GOP) of the same size throughout the video, (2) The maximum number of B-frames in a GOP should be consistent, and (3) A video file contains only a single video stream. The entire framework uses underlying APIs provided by FFmpeg, implements certain APIs of its own, and exposes these to the application. The library is initialized by calling *fas_initialize()*^{FAS} before any other API is called. This API is a wrapper over the underlying *av_register_all ()*^{FFmpeg} API, which registers the codecs, parsers, muxers, etc, and streaming protocol support: gopher, HTTP, etc, specified during the initialization of the application. If required to initialize the specified components individually which reduces the memory footprints, *void avcodec_register(AVCodec *)*^{FFmpeg,} performs the required job.

The next step is to open the video file by calling *fas_open_video()*^{FAS} ,which extracts the required information from the video stream and populates the *fas_context_type* ^{FAS} structure. This API acts as a wrapper API over six FFmpeg APIs and also some utility functions developed for this purpose. All the private functions have been declared static and start with private_* keyword to distinguish them from exposed APIs to application. The pseudo code is shown in Figure 8.

fas_open_video (fas_context_ref*, uint64_t target_index){

1. Initialize a seek table
2. Allocate memory for context
3. Fill data in context
4. Search for video stream
5. Display information
6. Allocate internal buffer memory
7. Find and open the corresponding decoder
8. Fill references in context
9. Extract the first frame
10. Save DTS of frames in first GOP in the Index table
11. Compute and save exact frame count and dts of last GOP of frames
12. Return

}

Figure 8: Pseudo Code for fas_open_video()FAS_FAST API

Seeking to a particular frame the application is required to call fas*_error_type fas_seek_to_frame (fas_context_ref*, uint64_t target_index)* [FAS], which jumps to the frame given by *target_index* [FAS]. This API performs the task of seeking to a frame and decoding the frame in the internal buffer. The pseudo code for this API is described in Figure 9.

fas_seek_to_frame (fas_context_ref*, uint64_t target_index) {

1. Get current frame index.
2. Is Target index in First or Last GOP?
    a. YES:
        i. Seek to frame through Index table.
        ii. Goto 4.
    b. NO:
        i. Is Target Index in current GOP?
            1. NO:
                a. Compute timestamp of nearest Index Frame less than Target Index.
                b. Jump to Nearest Index Frame less than Target Index through timestamp computed.
                c. Goto 1.
            2. YES:
                a. Jump (Target Index – Current Frame Index) times frames.
3. Is Current Frame Index equal to Target Index?
    a. NO:
        i. Is Current Frame Index greater than Target Index?
            1. YES:
                a. Return FAS_SEEK_ERROR.
            2. NO:
                a. Goto 1.
    b. YES:
        1. Goto 4.
4. Decode the frame and store it in internal buffer.
5. Return FAS_SUCCESS.

}

Figure 9: Pseudo Code for fas_seek_to_frame() API

To extract a particular frame we execute *fas_error_type fas_get_frame (fas_context_type)* [FAS]. It is responsible for decoding the requisite amount of packets which

make a complete frame, rescale according to the parameters passed and scale them according to the requirements of application. The pseudo code is given in Figure 10.

```
fas_error_type fas_get_frame(fas_context_ref_type context) {
        1. Check if Video is High Definition?
            a. NO:
                i. De-interlace the video frame.
            b. YES:
                i. Goto 2.
        2. Scale according to the required input for application.
        3. Invert the frame, required by Windows System
        4. Return fas_error.
}
```

Figure 10: Pseudo Code for fas_get_frame() API

The internal function *static void private_fas_pre_process_video_frame(fas_context_ref* , AVPicture *, void **)* [FAS] is responsible for de-interlacing the interlaced video files generated. To close the video at any moment, the API *fas_error_type fas_close_video (fas_context_ref*)* [FAS] is called, which cleans up the memory allocated and closes all the decoders, parsers, etc opened during the lifetime of execution of the program.

3.6.2 Frame Level Seek Library (Slow)

The slow Frame Level Seek (FAS_SLOW) is developed to provide very high Frame Accuracy. It does a linear file walk, saving the I-frame number and corresponding DTS values in *seek_table* [FAS].

Generation of an entire *seek_table* [FAS] prior to opening a video file increases the upfront time to load the video for playback. The APIs exposed to the application by FAS_SLOW have the same signature as the APIs for FAS_FAST in order to maintain the scalability among them.

The major difference lies in implementation of fas*_error_type fas_open_video (fas_context_ref_type *context_ptr, char *file_path)* [FAS] API from FAS_FAST API. The pseudo code for this API is given in Figure 11 below.

fas_open_video(context*, filename){

1. Initialize seek table,
2. Allocate memory for context,
3. Fill data in context,
4. Search for video stream,
5. Display information,
6. Allocate internal buffer memory,
7. Find Decoder.
8. Set Decoder to parse headers only.
9. Open Decoder.
10. Parse Packet headers to generate Index Table.
11. Close Input file.
12. Search for video stream,
13. Find Decoder and open with regular parameters.
14. Return.

}

Figure 11: Pseudo Code for fas_open_video() FAS_SLOW API

The FAS_SLOW version of this API calls the same set of FFmpeg functions and internal private functions as FAS_FAST version before the internal memory allocation as indicated in Step 6 of Figure 11. After which this API calls the internal function *fas_error_type private_generate_index_table (fas_context_ref_type *context_ptr)* [FAS] which is responsible for opening a codec for parsing the video file headers only. The parameters passed are *hurry_up* [FFmpeg] (deprecated in the current FFmpeg release). *parse_only* [FFmpeg], which ensures that if true, only parsing is done. *skip_bottom* [FFmpeg] parameter ensures to skip the specified number of bottom strides for decoding and *skip_top* [FFmpeg] parameter ensures to skip the top strides specified for decoding. The *lowres* [FFmpeg] parameter ensures to decode packets with specified resolution. Minimum values of the parameter can be 1, which stands for resolution half the size and value 2, which stands for resolution of quarter size. This function is responsible for populating the entire *seek_table* [FAS] with the codec capabilities applied to ensure the quickest possible generation.

Since in FFmpeg, properties of a codec opened on the fly cannot be changed, i.e. since a previously opened video with *skip_top* [FFmpeg], *skip_bottom* [FFmpeg] fields set, the codec needs to be closed and a new codec opened to extract the frames. Before leaving this function, the codec and format context need to be closed.

We call the internal function *fas_error_type private_open_video (fas_context_ref_type *context_ptr, char *file_path)* [FAS], which opens the MPEG-1 or MPEG-2 codec with proper parameters in order to extract the complete frames.

Once the complete seek table is generated, the rest of the implementation of FAS _SLOW is similar to FAS_FAST. During playback, the FAS_SLOW version extracts I-frame DTS value from the generated seek table in contrast to FAS_FAST version, which computes it on fly. It then extracts the requisite frame pointed to by the DTS value from seek table in the statically allocated internal frame buffer and provides it to the application layer.

3.6.3 Frame Level Seek Library Structure

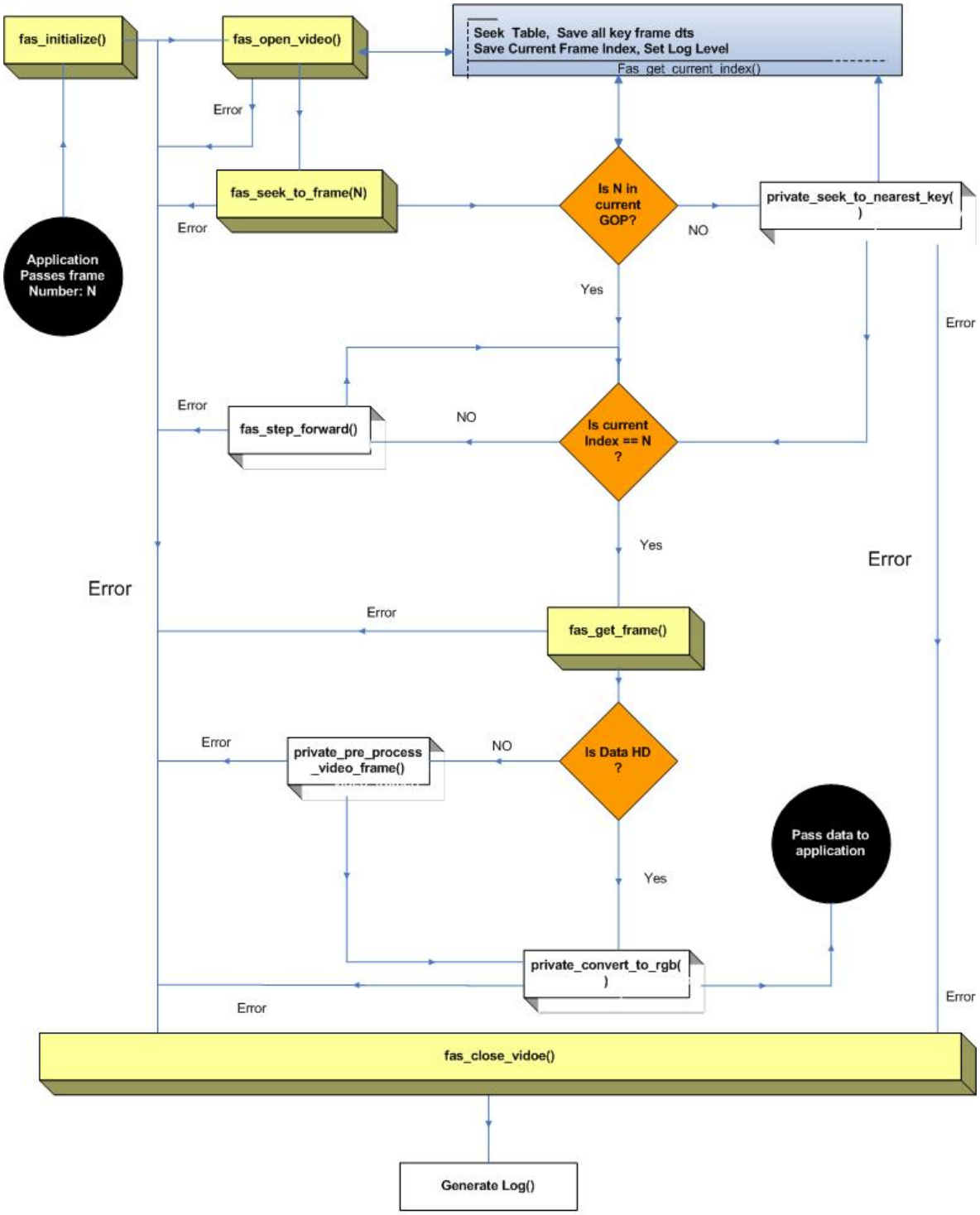

Figure 12: Frame Level Seek Library Structure Overview

3.6.4 Frame Level Seek Library Data Dependency

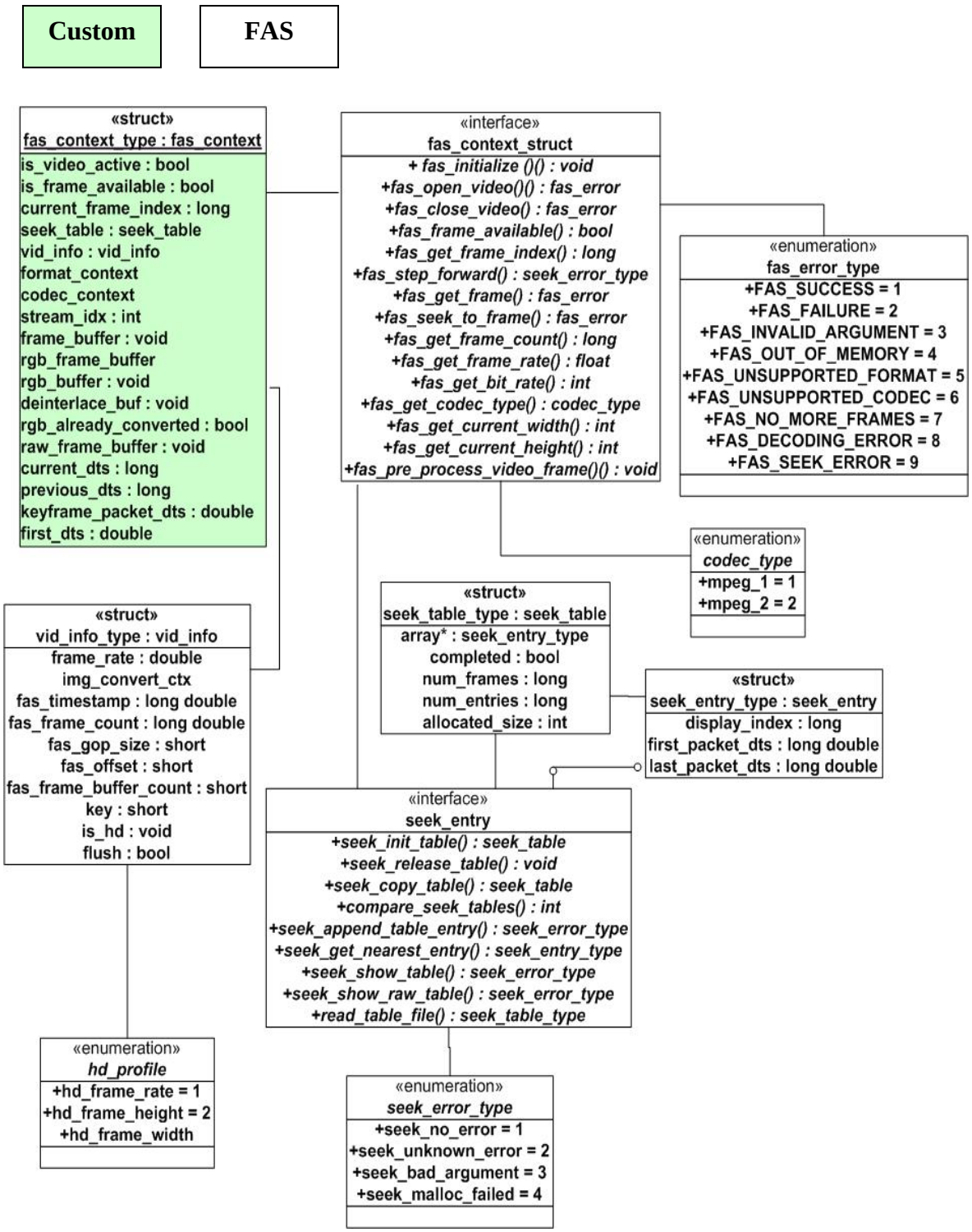

Figure 13: Frame Level Seek Library Data Dependency

CHAPTER 4: ENCODING/DECODING EXTENSION LIBRARY

FFmpeg has a wide range of open source container formats, encoders and decoders available which can be used for encoding raw data stream into various encoded streams. Preference can be specified in the *.configure* file prior to compiling the code base to include or exclude certain encoders, decoders, muxers and de-muxers reducing the library footprint. Also, certain encoders are available in the GPL licensed version only. But the MPEG-1 and MPEG-2 encoders along with MPEG container format are licensed under LGPL license.

In this chapter, we will be focusing on the FFmpeg MPEG-1 and MPEG-2 encoders along with MPEG container. Our emphasis will be on implementation and use of the encoding APIs provided to the application developer by the developed encode library, the internal state machine implemented to support this feature and the data structures maintained. We will also discuss the implementation and use of FFmpeg MPEG-2 related encoding APIs.

4.1 Limitations of FFmpeg Encoding

FFmpeg provides an API *"int avcodec_encode_video(AVCodecContext *, uint8_t *, int , const AVFrame *)*[FFmpeg]" which is responsible for encoding the data pointed to by the second parameter of type *uint8_t**. The limitations of this API specific to our requirements are (1) A complex pattern of function calls to be followed before execution of the API, (2) Loss of the last three frames during encoding due to data left in internal FFmpeg buffers, and (3) Lack of an integrated API supporting both pre-encoded and decoded input data.

4.2 Encoding Library

In the current library, we have provided a set of four APIs to application developers, eliminating the three limitations of FFmpeg encode APIs as discussed above. The set of APIs will significantly reduce the development, integration and testing time for video editing applications as it provides a flexible and simple framework. The drawback of loss of the last three frames during encoding has also been fixed. In the current distribution, support for MPEG-PS container with MPEG-1 and MPEG-2 encoder has been provided. We have also integrated this library with the FFmpeg Frame Level Seek library in an application providing support for both pre-encoded and decoded input video data.

4.3 Design of Encode Library

The library is designed to handle two types of input data: (1) reference from a memory location, and (2) pre-encoded data in the form of a video stream. The library has been designed to be compatible with both FAS_FAST and FAS_SLOW, so many of the design decisions have been taken, keeping the Frame Level Seek library in mind for integration.

The major design decisions taken are: (1) Using static input buffers for encoding., (2) Non inclusion of audio stream., (3) Keeping fixed GOP size., (4) Initializing the ‘muxer’, (5) using most of underlying FFmpeg APIs., and (6) Separation of Public APIs exposed to the application and private functions to modify internal data structures.

- Using static input frame buffers initialized during startup hinders the flexibility for the application but reduces the complexity of memory management for application, runtime overhead of dynamic memory allocation and memory fragmentation caused by continuous calls to *malloc()* and *free ()* system calls of memory buffers.
- Audio streams have not been included for encoding as no support for audio by the Frame Level Seek library makes the two compatible.
- Currently the GOP size is fixed, making it compatible with the Fast Frame Level Seek library.

- Initializing the ‘muxer’ to encode every data packet with a header for comparability with previous application (EmCapture) versions.
- Re-using most of the underlying FFmpeg API’s providing reduction in time for development, support from the FFmpeg community and tested and in-production framework. We use ^{encode} to denote encode library function and ^{FFmpeg} to denote FFmpeg library function hereafter
- The APIs exposed to applications are not responsible for maintaining the internal data structures, but call private Encode functions to perform the task.

4.3.1 Encode Library in Operating System

The library is written to provide a simple and stable adaptation layer for FFmpeg middleware and application layer, exposing a set of consistent and stable APIs. It uses many of FFmpeg APIs and acts as a wrapper for them, reducing the integration complexity for the application developer. This library is smaller than the Frame Level Seek library in memory footprint discussed in the previous chapter. Since it is not required to do book keeping of I-frames DTS value for encoding purpose, the Index table module was not embedded with this library.

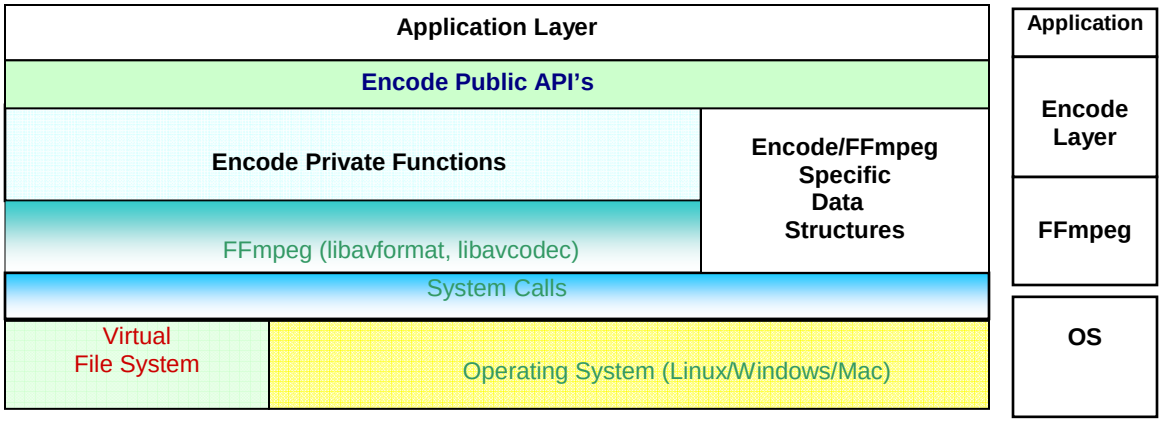

Figure 14: Encode Library placement in OS

4.3.2 Encode Library State Machine

During execution, the encoder layer can be in one of these five states, (1) *Init*, when the application layer calls the library function for passing codec type, resolution, frames per

second, etc; (2) *Initialize*, when the internal encoders, decoders etc are selected based on the parameters passed during *Init*, internal memory allocation is performed, internal data structures used are initialized, etc; (3) *Idle*, the system is in idle state expecting data from the application after initialization or after encoding data; (4) *Decode*, after the application passes data (pre-encoded or decoded) to this layer; (5) *Encode*, this state is responsible for encoding the data decoded by the decoder.

There is only one thread of execution in the Encode library. The process is designed to be linear in nature without introducing any parallelism because of the same reasons as discussed for Frame Level Seek library.

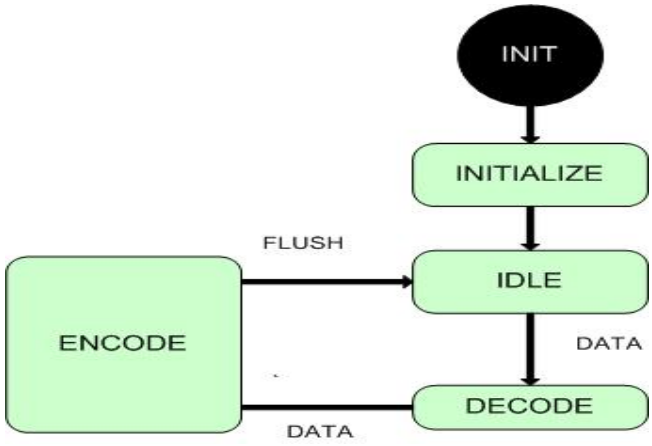

Figure 15: Encode Library State Machine

There are three major structures used in the implementation which keep reference of the FFmpeg structures and introduce some of their own member objects. These are (1) *out_context_type*, (2) *ip_context_typ,e* and (3) *dec_context_type*.

The *op_context_type* data structure is responsible for storing the encoder data members. The *ip_context_type* data structure is passed by the application to encode layer for setting the parameters for encoding. The *dec_context_type* data structure maintains the information required for decoding the input data passed. The data members are listed in Tables 16-18, respectively.

Table 16: Members of op_context_type Structure

#	Member	Description
1	*EncodeFmtCtx**	Reference to FFmpeg *AVFormatContext* structure
2	*EncCodecCtx**	Reference to FFmpeg *AVCodecContext* structure
3	*DecCtx**	Reference to Encode *dec_context_type* structure
4	*ipCtx**	Reference to Encode *ip_context_type* structure
5	*img_convert_ctx**	Reference to FFmpeg *SwsContext* structure
6	*myAVTIMEBASEQ*	Structure to store rational values
7	*St**	Reference to FFmpeg *AVStream* structure
8	*g_emulate_pts*	Counter to store number of frames
9	*Tmp_pic**	Reference to Internal Buffer
10	*Outbuf**	Reference to Internal Buffer
11	*picture_buf**	Reference to Internal Buffer
12	*Dump**	Reference to Internal log file

Table 17: Members of ip_context_type Structure

#	Member	Description
1	*mpeg_type*	1 for MPEG-1 and 2 for MPEG-2
2	*out_width*	Required width by application
3	*out_height*	Required height by application
4	*Out_bit_rate*	Required bit rate by application
5	*Out_gop_size*	Required GOP size by application
6	*out_max_b_frames*	Required Maximum B- Frames by application
7	*in_filename**	Input file name to open
8	*out_frame_num*	Frame rate numerator
9	*out_frame_den*	Frame rate denominator
10	*is_hd*	1 if video to be encoded is HD, 0 otherwise

Table 18: Members of dec_context_type Structure

#	Member	Description
1	*DecCodecCtx**	Reference to FFmpeg *AVCodecContext* structure
2	*DecPicture**	Reference to FFmpeg *AVFrame* structure
3	*Size*	Size of the decoded frame
4	*Frame*	Count of the decoded frames

4.4 Encode Library File Structure

The current implementation consists of three files, encode_ff.h, which contains the declarations of functions and all data structures. This file is also used as an interface for exporting symbols if compiled as a dynamically linked library.

All the APIs visible to the application layer are defined in the file encode_ff.c. Certain applications currently using this library are: (1) *putframes_ff.exe,* which provides support for encoding raw data from memory reference, (2) *cap.exe*, an application used for real time encoding of data captured from video card, and (3) *SEndoPaste.dll* which is built specifically to provide feature of encoding pre-encoded video data into a different format (Transcoding).

4.5 API Implementation Details

In the Encode engine there is a set of four APIs currently visible to the application developer, which includes

Table 19: API's Exported by Encode Library

#	Category	API	Return Value	Parameter's
1	Initialization	*en_init ()*	*Void*	*void*
2		*en_enc_setup ()*	*en_error_type*	*out_context_type *, ip_context_type**
3	Exiting	*en_close_vid()*	*en_error_type*	*out_context_type**
4	Encoding	*en_enc_frm ()*	*en_error_type*	*out_context_type*, AVFrame**

The *en_error_type en_enc_setup(out_context type *, ip_context_type*)* [encode] API is called after *en_init ()* [encode]. This API wraps fifteen of the underlying FFmpeg APIs and four of Encode library internal functions. The pseudo code is:

```
en_error_type en_enc_setup(out_context_type *,  ip_context_type*) {
        1. Allocate memory for out_context_type structure.
        2. Is video to be encoded in MPEG-1 format?
            i. NO:
                1. Find MPEG-2 encoder.
            ii. YES:
                1. Find MPEG-1 encoder.
        3. Allocate memory for Codec context structure.
        4. Find container for encoder.
        5. Allocate memory for Format context (container) structure.
        6. Allocate a new Video Stream.
        7. Set Codec and Format parameters according to ip_context_type structure.
        8. Initialize and open Decoder.
        9. Allocate memory for internal buffers.
        10. Open encoder.
        11. Write Header information according to the format.
        12. Return EN_SUCCESS.
    }
```

Figure 16: en_encode_setup() Pseudo Code

The return type "*en_error_type*" is defined to return only two values, *EN_FAILURE* [encode] and *EN_SUCCESS* [encode]. Appendix C has the detailed implementation of this API. To encode a specific frame, we call (c) *en_error_type en_enc_frm (en_Context_Struct*, AVFrame*)* [encode]; which takes reference to the *out_context_type* [encode] structure already allocated and reference to the *AVFrame* [FFmpeg] structure which the application has to pass. The pseudo code is given in Figure 17.

```
en_enc_frm(en_Context_Struct *op_context,AVFrame *ip_frame){
        1. Is input data in Raw format ?
            a. YES
                i. Goto 3
            b. NO
                i. Decode data in Raw Format.
        2. Scale Raw Data with input parameters.
        3. Encode data in required format.
        4. Initialize data packet.
        5. Point packet data field to encoded frame data field.
        6. Write packet to the container in appropriate stream.
        7. Return.
    `
```

Figure 17: en_enc_frame() API Pseudo Code

The API (d) *en_error_type en_close_vid (en_Context_Struct *)* [encode] is used to close all references, free allocated memory, and flush out remaining frames in the internal FFmpeg encoder buffers.

Since the encoder stores certain frames for future reference, it becomes necessary for the encoder to flush the data stored. This is achieved by calling the API int *avcodec_encode_video(AVCodecContext *, uint8_t *, int, AVFrame *)* [FFmpeg] with NULL passed as the reference frame to be encoded i.e. the fourth parameter, on success we write the frame to the stream by calling *int av_interleaved_write_frame(AVFormatContext *, AVPacket *)* [FFmpeg].

4.5.1 Encode Library Engine

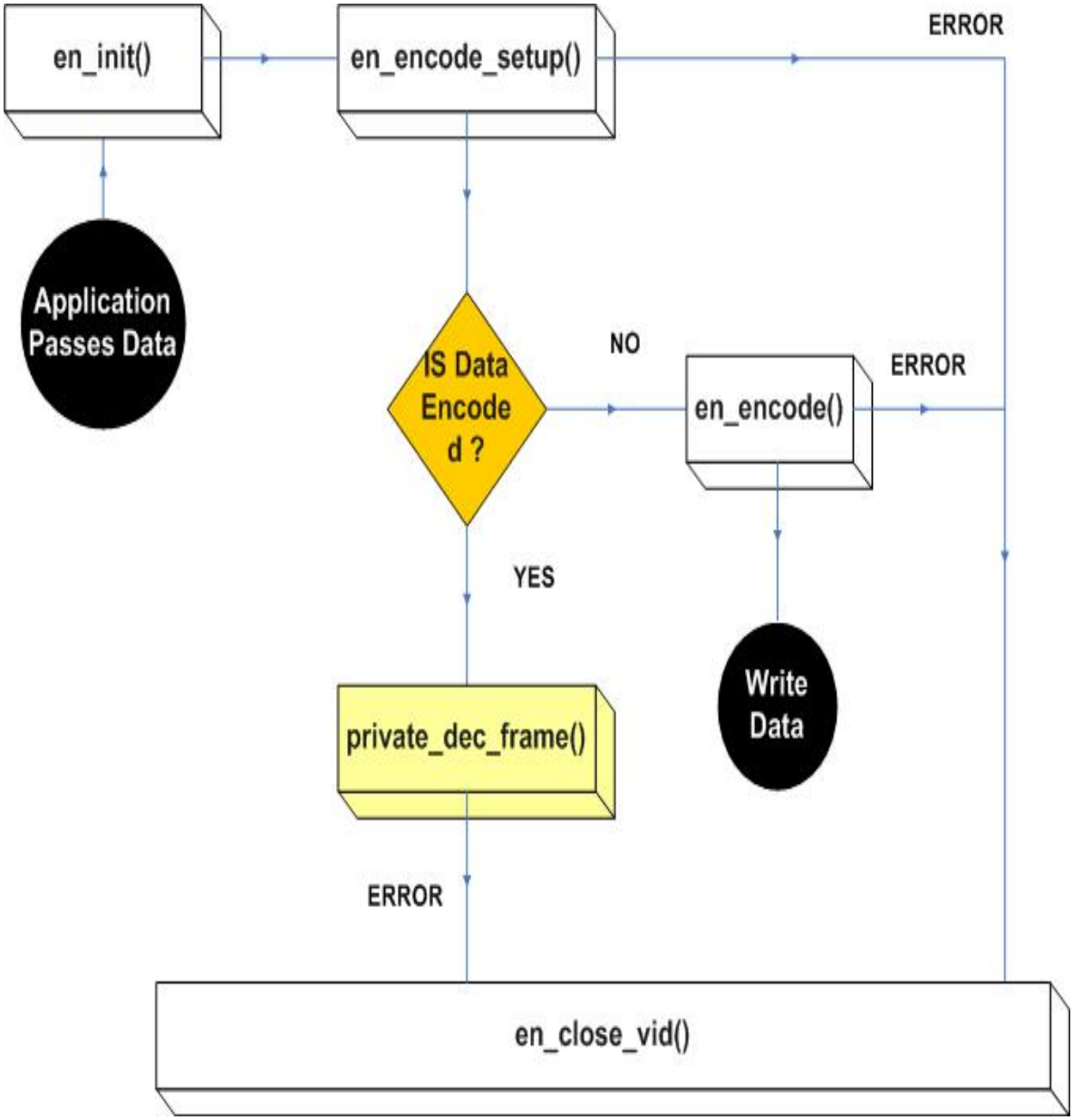

Figure 18: Encode Library Structure

4.5.2 Encode Library Data Dependency

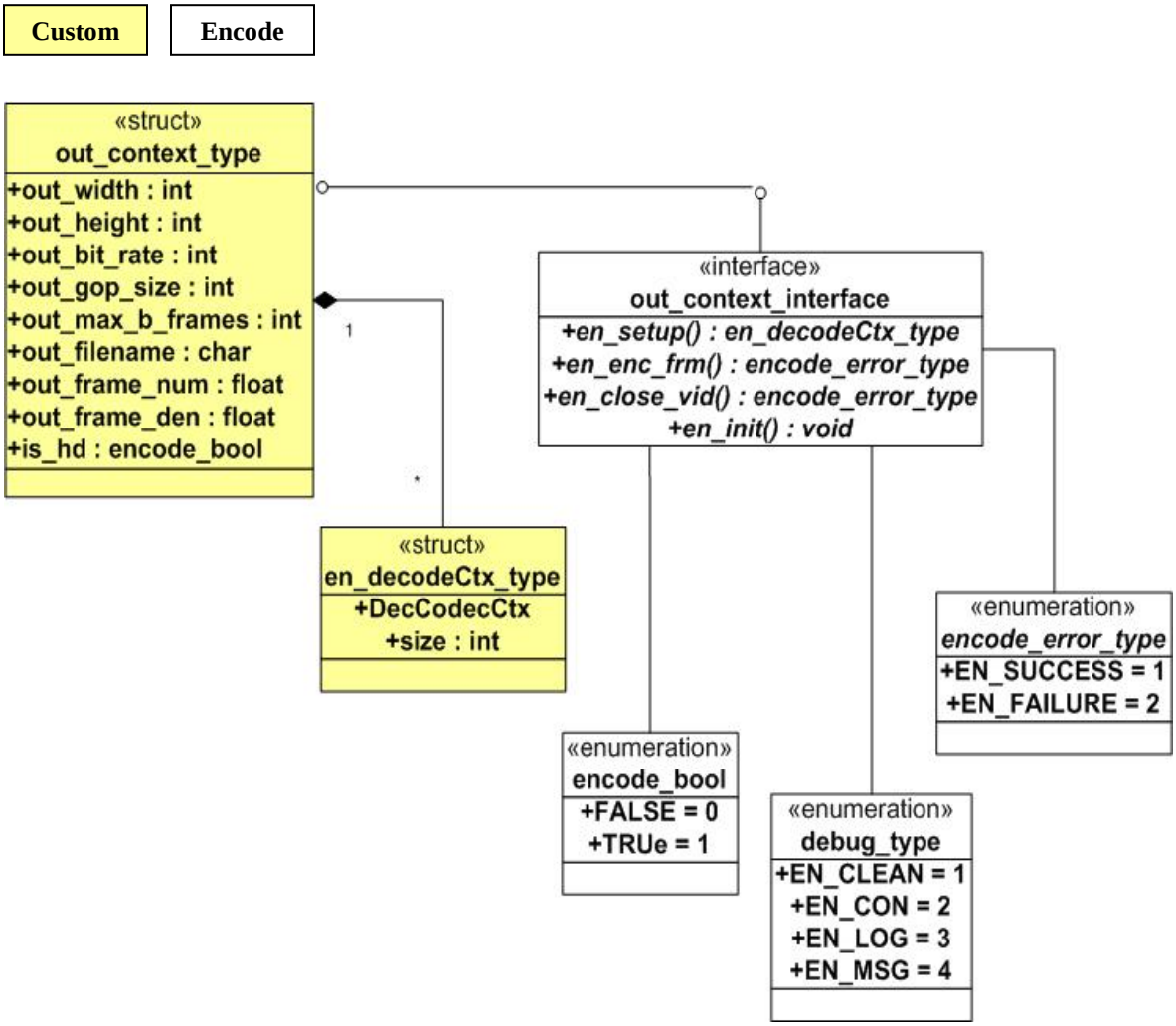

Figure 19: Encode Library Data Dependency

CHAPTER 5: MOTION VECTOR EXTRACTION LIBRARY

In this chapter we focus on the architecture, design, implementation and state machine involved in our motion vector extraction library. We focus on different algorithms implemented in FFmpeg code base for motion vector calculation for video encoded with MPEG-1 and MPEG-2 encoders. We describe the APIs exposed by this library for application developers. Next, we present the internal data structures maintained for motion vector calculation. We will also focus on the FFmpeg APIs used, their implementation, and use cases. We will use MV for Motion Vectors henceforth.

5.1 Limitations of Motion Vector Extraction in FFmpeg

Currently, there is no provision in FFmpeg distribution to expose Motion Vector (MV) calculated during encoding and decoding to the application. FFmpeg is designed to initialize an encoder or decoder during runtime. The upper layer (Management), depending upon the parameters passed by the application, chooses the specific encoder or decoder from a registered list. This list depends upon the parameters passed during compilation in the *.configure* file.

The encoder chosen is then initialized and all the relevant data structures are populated. Since the calculation of MV is an integral part of MPEG standard, there are certain other codecs such as Real Media, Quick time, and Metroska, which have their own algorithms for reducing redundant data. So the entire framework for MV calculation is initialized at runtime when the MPEG codec is chosen to be used.

5.2 Motion Vector Extraction Library

The MV library is designed to compute MVs from a video previously encoded with MPEG encoder during decoding. We re-calculate MVs from the decoded frame data. This

framework is capable for computing MVs for an inter-coded frame (P-frame or B-frame). Figure 20 gives a high level view of the MPEG standard for video compression.

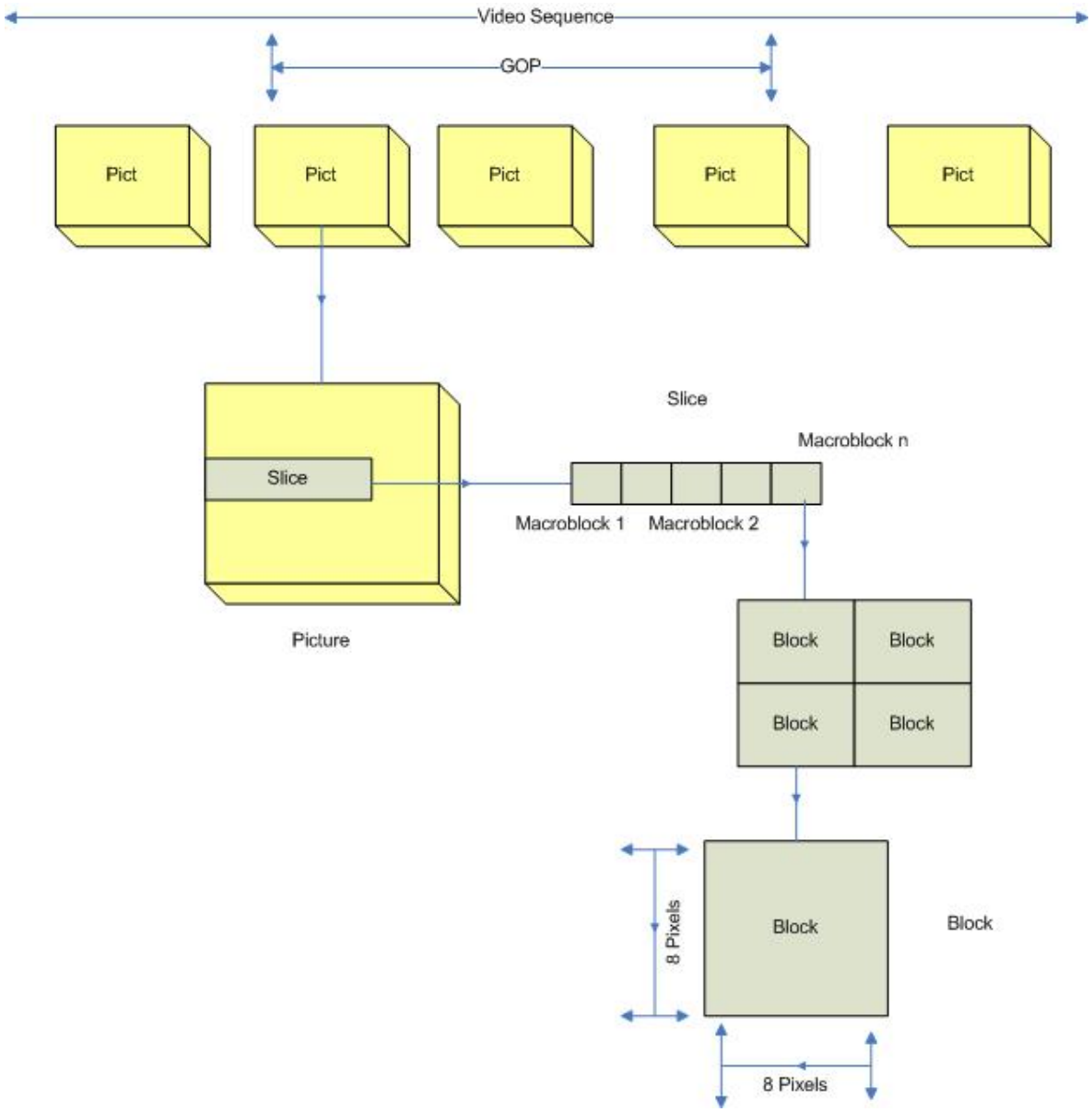

Figure 20: High Level Structure of an MPEG Video Sequence

The various MV search algorithms implemented in FFmpeg are (1) ME_ZERO: no search, that is use 0, 0 vectors whenever one is needed, (2) ME_FULL, (3) ME_LOG, (4) ME_PHODS, (5) ME_EPZS: enhanced predictive zonal search, (6) ME_X1: reserved for experiments, (7) ME_HEX: hexagon based search, (8) ME_UMH: uneven multi-hexagon search, (9) ME_ITER: iterative search, (10) ME_TESA: transformed exhaustive search

algorithm, see Appendix B for reference. We use ME_EPZS MV estimation algorithm in the encode library as it is supported by both H.264 and MPEG-2 decoders for playback.

5.3 Design of Motion Vector Extraction Library

The MV extraction library is designed to extract MVs from files containing video data. This library acts as an adaptation layer between the application and FFmpeg library. Its placement is similar to the encode library described in the previous chapter. The library has been designed to be compatible with both FAS_FAST and FAS_SLOW. As a result, many of the design decisions have been taken keeping the Frame Level Seek library in mind for integration. We took the following design decisions.

1. Using static input frame buffers initialized during startup hinders the flexibility for the application but reduces the (1) complexity of memory management for applications, (2) runtime overhead of dynamic memory allocation, and (3) memory fragmentation caused by continuous calls to *malloc()* and *free ()* system calls of memory buffers.
2. Reusing most of the underlying FFmpeg APIs provides reduction in development time and support from the FFmpeg community. We use ^{mv} to denote Motion Vector extraction library function and ^{FFmpeg} to denote FFmpeg library functions hereafter.
3. Separation of public APIs that uses private functions to update internal data structures for ease of maintenance.

5.3.1 Motion Vector Extraction Library State Machine

The MV library consists of five internal states as shown in Figure 21.

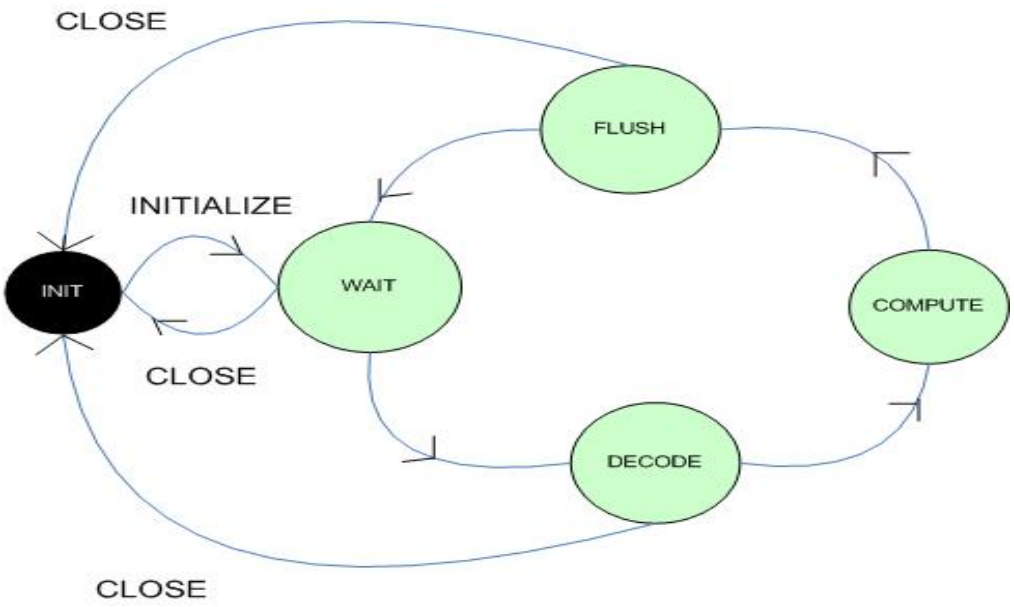

Figure 21: State Machine for the Motion Vector Extraction Library

(1) The library starts in the *init* state in which the internal buffers are initialized or released, (2) *Wait* state can be reached either after the *Flush* state or after the *init* state before any frame is decoded, (3) *Decode* state is reached when the application decoded a frame from the video calls the MV extraction function. The library can return to the *init* state from this state if there are no more frames available for decoding, (4) *Compute* state provides the framework of calculation of MVs and cannot be reached from the *init* state, and (5) *Flush* state can be reached after the MVs are calculated and the framework needs to flush internal buffers for further computation. The library can either go to the *wait* state when there is a delay by the application to provide the next frame for decoding, or the *init* state when there are no more frames available.

The MV extraction library provides a set of four APIs (Table 20) to the application developer for integration. It uses the MPEG decoder internally to decode the video stream into frames to get data for re-calculation of MVs. The FFmpeg API *int avcodec_decode_video(AVCodecContext *, AVFrame *, int *, const uint8_t *, int)* is used by the MV extraction library for decoding purpose. It also implements two structures (1) *mv_param_struct* and (2) *mv_context_struct* for book keeping, implementing the internal state machine, and decoding and parameter passing from the application. These are explained in Table 21 and Table 22, respectively.

Table 20: API's Exported by Motion Vector Extraction Library

#	Category	API	Return Value	Parameter's
1	Initialization	mv_initialize ()	Void	Void
2		mv_open_video ()	mv_error_type	mv_context_type**, mv_parameter_type*
3	Exiting	mv_close_video ()	void	mv_context_type*
4	Extraction	mv_extract ()	mv_error_type	mv_context_type*

Table 21: mv_context_struct Data Members

#	Member	Description
1	format_context*	Reference to FFmpeg AVFormatContext structure
2	codec_context*	Reference to FFmpeg AVCodecContext structure
3	frame_buffer*	Reference to FFmpeg AVframe structure
4	current_frame_index	Current frame number decoded
5	stream_idx	Video stream index
6	is_video_active	True if video is active, false otherwise
7	is_frame_available	True if there are frames left, false otherwise
8	out_file_handle*	Reference to a file handle to store MV
9	mv_type	Type of MV to extract

Table 22: mv_param_struct Data Members

#	Member	Description
1	in_file_path*	Reference to a file handle to extract MV from
2	out_file_path*	Reference to a file handle to store MV to
3	mv_type_char*	Reference to the type of frame to extract MV from

5.4 Motion Vector Extraction Library File Structure

The library source code is divided into 3 files. (1) ffmpeg_mv_extract.h, (2) ffmpeg_mv_extract.c, and (3) mv_extract_example.c. The ffmpeg_mv_extract.h contains all the global declarations and macro definitions. This file acts as the interface for the motion vector extraction library as it exports all the API signatures. The motion vector

extraction implementation is in ffmpeg_mv_extract.c, which also contains the code for decoding frames. Last, mv_extract_example.c is a small command-line application provided along with the library for testing purposes.

5.5 API Implementation Details

For *mv_error_type mv_open_video (mv_context_ref_type *, mv_param_struct)*[MV], *void mv_initialize ()*[MV], and *void mv_close_video (mv_context_type *)* [MV], refer to Fast Frame Level Seek library APIs *fas_open_video()*[FAS], *fas_initialize()*[FAS] , and *fas_close_video()*[FAS] , respectively since they are similar.

The *mv_error_type mv_extract (mv_context_type *)* API performs re-calculation of motion vectors from the decoded frame data. This API uses six private MV extraction library functions implemented and calls four FFmpeg APIs mainly exposed to application for decoding and flushing purposes. The pseudo code for this library is shown in Figure 22.

```
mv_error_type mv_extract (mv_context_type *) {
    1. Is frame available?
        a. YES
            i. Extract next frame in internal buffer.
            ii. Go to 2.
        b. NO
            i. Go to 3.
    2. Is it an I-frame?
        a. YES
            i. Flush internal buffers.
            ii. Go to 1.
        b. NO
            i. Get Frame Type (P or B).
            ii. Get Macro Block Height.
            iii. Get Macro Block Width.
            iv. Get Macro Block Stride.
            v. Compute Motion Vector Stride.
            vi. Get Direction for Motion Vector calculation.
            vii. Compute Pix Fmt. Type.
            viii. Is there a motion vector for this coordinate?
                1. NO
                    a. Compute Block Size.
                    b. Fill default values for no motion
                       vectors.
                    c. Save Motion Vectors.
                2. YES
                    a. Compute Block Size.
                    b. Compute Motion Vectors.
                    c. Save Motion Vectors.
    3. Free buffers and close the video.
}
```

Figure 22: Motion Vector Extraction Pseudo Code

When we extract a frame, the frame is stored in an array of pointers to integers *(uint8_t *data [4]* [FFmpeg]), in which the 4 pointers represent four data planes. The fourth data plane (i.e., *data [3]*) is the Alpha plane. This data is stored in *AVframe* [FFmpeg] structure which is stored as a reference in *frame_buffer** [FFmpeg] and in the *mv_context_struct*[MV]. To compute macro blocks, we need to first compute the interpolation precision, denoted by *shift* [MV] (*shift* [MV] *= 1 + s->quarter_sample* [MV]). The value of *quarter_sample* [MV] can be either 0 or 1. As MVs always have a fractional-sample precision, e.g. a vector of (0.5, 0) indicates translating the video by half a pixel (interpolating between samples in the reference frame), where *quarter_sample* of 1 means the precision of 0.25, and 0 means the precision of 0.5. The numbers in *motion_val[]*[FFmpeg] (Motion Vector table) are fixed-point representations of those fractional vectors.

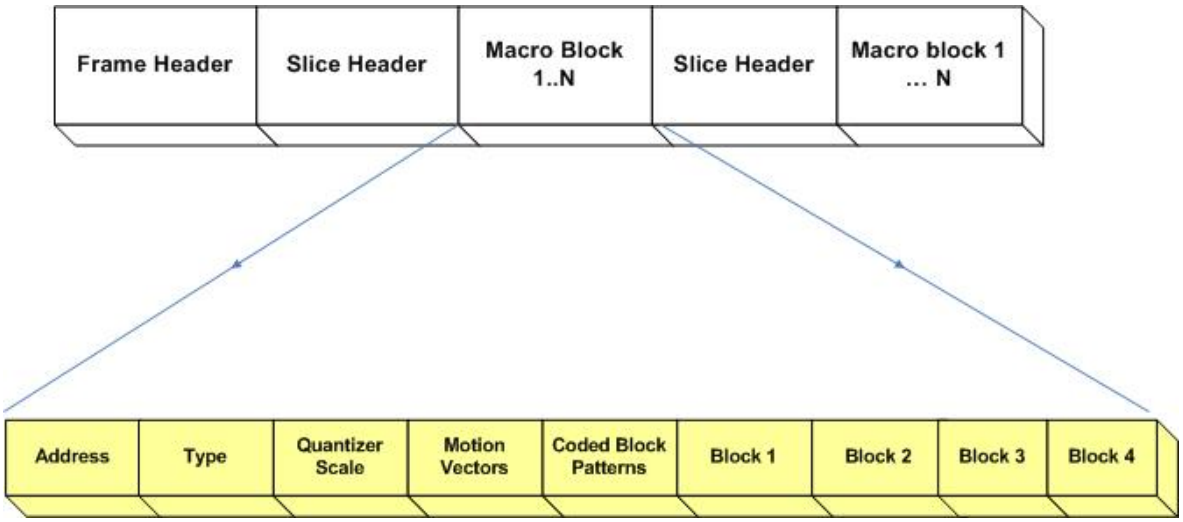

Figure 23: Macro Block Structure

The motion estimation algorithms are specified by the field *me_method* [FFmpeg]. The various search algorithms supported by FFmpeg are listed in Section 5.2. To get motion vectors of a compressed frame like P-frame, the previous reference frame is reconstructed. We get the referenced frame from *int8_t *ref_index [2]* [FFmpeg] given in the *AVframe* [FFmpeg] structure which is an array of 2 pointers, populated as required. The motion vectors are stored in *int16_t (*motion_val [2]) [2]* [FFmpeg] table which is an array of 2 pointers to an array of 2 integers. This array is a 3 dimensional array, which is accessed using *mv [arg1][arg2][arg3]* [FFmpeg].

Table 23: Description of Motion Vector Buffer

#	Argument	Value	Description
1	arg1	0	Visualizes forward predicted MVs of P frames
		1	Visualizes forward and backward predicted MVs of B frames
2	arg2	MV_TYPE_16X16	1 MV for the whole macro block
		MV_TYPE_8X8	4 MVs for one macro block
		MV_TYPE_8X16	2 MVs, one per 8x16 block
		MV_TYPE_16X8	2 MVs, one per 16x8 block
3	arg3	0	X Direction
		1	Y Direction

```
1.  mv_sample_log2^MV = 4- pict->motion_subsample_log2^FFmpeg
2.  mb_height^MV = video_height/16
3.  mb_width^MV = video_width/16
4.  0 < mb_y^MV <  mb_height^MV
5.  0 < mb_x^MV < mb_width^MV
6.  sx^MV = mb_x^MV *16 + 8;
7.  sy= mb_y^MV *16 + 8;
8.  xy^MV = (mb_x^MV + mb_y^MV *mv_stride^MV) << mv_sample_log2^MV;
9.  mx^MV = (pict->motion_val^FFmpeg [direction^FFmpeg][xy][0]>>shift^FFmpeg) + sx;
10. my^MV = (pict->motion_val[direction^FFmpeg][xy][1]>>shift^FFmpeg) + sy;
```

Figure 24: Motion Vector Calculation Code Snippet

The output of the motion vectors extraction library is given in the following format.

mb_size : sx, sy, dx, dy: pix_fmt, frame_ type: frame_num

where *mb_size* is the macro block size. The top-left corner of a frame has the coordinate (0,0). Fields *sx* and *sy* are the absolute *x* and *y* pixel coordinates of the current frame; *dx* and *dy* are the absolute *x* and *y* coordinates of the matching block in the referenced frame; *pix_fmt* can either be *CHROMA_420, CHROMA_422,* or *CHROMA_444* depending on the codec pixel format, which can be PIX_FMT_YUVJ422P or PIX_FMT_YUV420P in case of MPEG-2, See the Appendix B; *frame_ type* is the type of the frame which can be either P or B. A P-frame has inter-coded blocks with forward predicted motion vectors or intra-coded blocks. A B-frame has inter-coded blocks with backward predicted motion vectors in

addition to the type of blocks in P-frames. Last, *frame_num* is the frame number of the current frame in the display order.

Table 24: Description of Motion Vector Parameters

#	Member	Value	Macro Block Size
1	mv_sample_log2	0	16x16
		1	8X8
		2	4X4
		3	2X2
2	sx		Left X offset value of a macro block
3	sy		Top Y offset value for a macro block

5.5.1 Motion Vector Library Engine

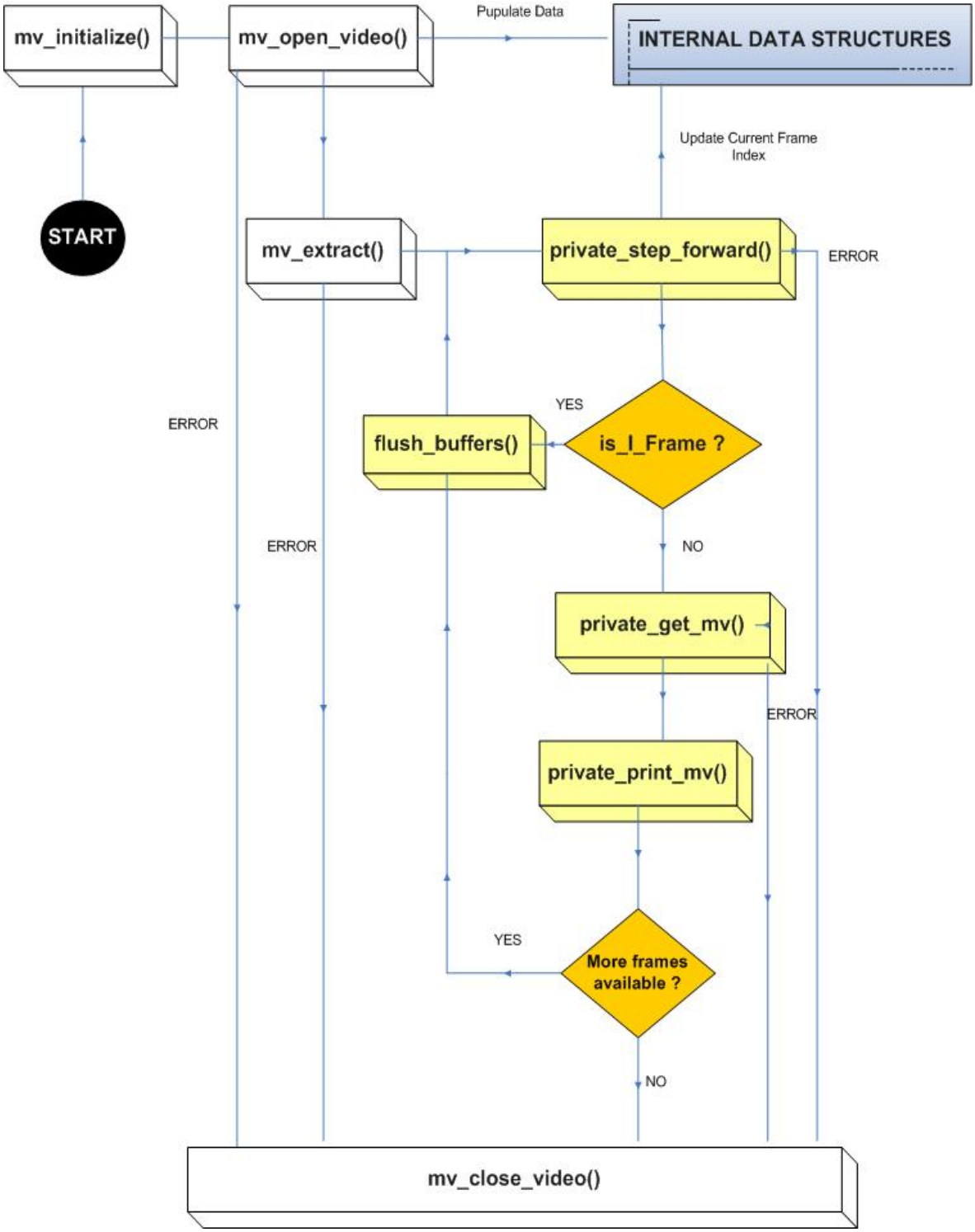

Figure 25: Engine of the Motion Vector Extraction Library

5.5.2 Motion Vector Library Data Dependency

Custom **MV Library**

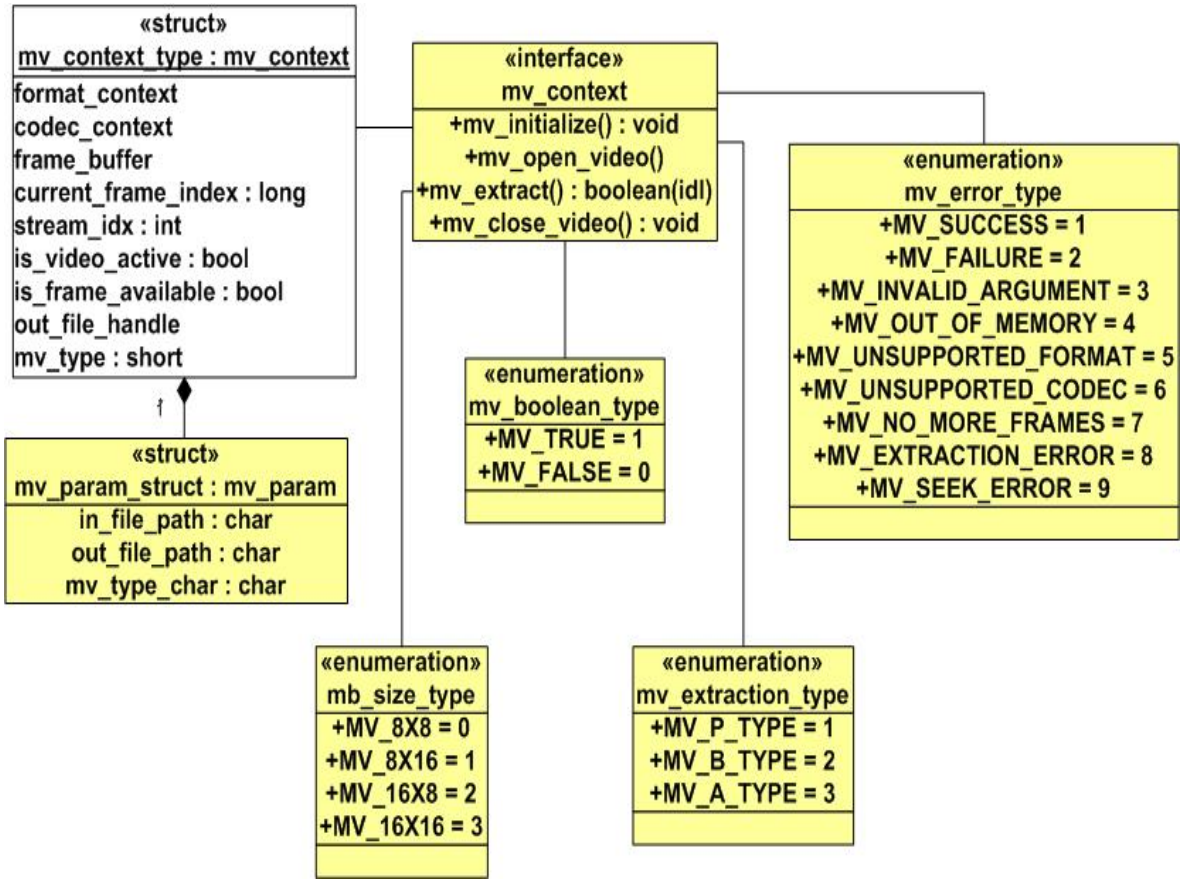

Figure 26: Motion Vector Extraction Library Data Dependency

CHAPTER 6: EXPERIMENTATION AND PERFORMANCE MEASUREMENT

This chapter presents results of various tests conducted to measure stability, performance, and usability of (1) Frame Level Seek (FAS_FAST) Library, (2) Frame Level Seek (FAS_SLOW) Library, (3) Encode Library, and (4) Motion Vector Extraction Library. We discuss various applications that were integrated with these libraries to test their stability and usability. We describe the test environments and the properties of videos taken from five different categories. We show various performance measurement comparisons between the FFmpeg library and the MainConcept library (referred as MC* henceforth). The performance metrics are memory utilization, CPU utilization, PSNR comparison, and upfront load time which is the wait time before subsequent functions are ready for use.

6.1 Experimental Environment

The comparison experiments were conducted on two different sets of libraries, FFmpeg and MC* both built for Windows. FFmpeg libraries have been built using the latest distribution, version 20536. The experiments were performed on two different Windows machines.

Table 20: Hardware Profiles of Test Machines

#	Machine	Operating System	CPU	CPU Address bus	Freq (GHz)	RAM GB.
1	Inslab.cs.iastate.edu	Windows-XP SP3	Intel-Xeon	32 bit	3.60	2
2	Bigvision.cs.iastate.edu	Window-7 Professional	Intel-Xeon	64 bit	4.00	6

We used twenty-five different videos from the following categories. The videos in categories 2, 3, 4 and 5 were captured using a DeckLink video capture card. The properties of these videos are described in Table 21. All videos were captured without audio stream because the FFmpeg Frame Level Seek library currently does not support seek in videos

with audio data. Colonoscopy videos were captured from routine colonoscopy procedures using a WinFast TV Expert 2000 capturing card.

Table 21: Description of Test Videos

#	Profile	Name	Duration (Sec)	Dimension (Pixels) Width x Height	Frames
1	Fujinon Colonoscopy (1)	Fuji_1.mpg	812	720 x 480	24,429
2		Fuji_3.mpg	1,326	720 x 480	39,748
3		Fuji_9.mpg	2,147	720 x 480	64,346
4		Fuji_36.mpg	1,125	720 x 480	32,987
5		Fuji_53.mpg	1,944	720 x 480	40,303
6	Animation (2)	Simpsons_1.avi	1,315	624 x 352	32,896
7		Simpsons_2.avi	1,295	624 x 352	32,391
8		Simpsons_3.avi	1,284	624 x 352	32,124
9		Simpsons_4.avi	1,284	624 x 352	32,124
10		Simpsons_5.avi	1,281	624 x 352	32,136
11	Low Motion (3)	Tonight_1.avi	2,685	624 x 352	64,638
12		Tonight _2.avi	2,685	624 x 352	64,645
13		Tonight _3.avi	2,590	624 x 352	64,786
14		Tonight _4.avi	2,685	624 x 352	64,442
15		Tonight _5.avi	2,680	624 x 352	64,296
16	High Motion (4)	Soccer_1.avi	7,689	640x368	77,590
17		Soccer_2.avi	3,214	640x368	70,414
18		Soccer_3.avi	3,273	640x368	34,404
19		Soccer_4.avi	6,227	640x512	39,976
20		Soccer_5.avi	5,984	640x512	20,020
21	High Definition (5)	Dsp_Hsw_1.avi	2,494	1280x720	20,020
22		Dsp_Hsw_2.avi	2,540	1280x720	20,020
23		Dsp_Hsw_3.avi	2,488	1280x720	20,020
24		Dsp_Hsw_4.avi	2,488	1280x720	20,020
25		Dsp_Hsw_5.avi	2,483	1280x720	20,020

Profiling was performed with AMD's Codeanalyst---a GUI-based code profiler for x86-based machines. It is a statistical profiler that profiles based on sampled data and it does not affect the execution speed. It is also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. Since most of the code base is in C and

C++, this application was chosen to perform the profiling due to the following reasons. (1) It is C++ based; (2) It supports Windows environment; (3) It is free; and (4) It performs statistical profiling. Virtual Dub, an open source Windows based software, was used to extract frames from the captured videos. All the captured videos are uncompressed videos in AVI format, except for colonoscopy videos that are in MPEG-2 format.

Frames from the raw videos were extracted in PNG format and were used for comparison between FFmpeg and MC* MPEG encoder. The non-HD frames were encoded with properties: bit rate of 80,000 bps (bits per second), default dimensions, maximum number of B- frames of 1, GOP size of 15, closed GOP, and the frame rate of 30 fps. The HD frames were encoded with properties: bit rate of 80,000 bps, default dimension, maximum number of B-frames of 1, a GOP size of 15 frames, closed GOP, and the frame rate of 60 fps.

Codeanalyst requires that libraries to be profiled be built in a debug mode. Therefore, we built our libraries for performance comparison in a debug mode. However, we do not have MC* library in the debug mode. Therefore, specific details like function call graphs and memory level instruction calls could not be gathered when using MC* library. MC* could not perform seeking and encoding for High Definition (HD) videos, so the comparison results for HD videos only contain profiling data of FFmpeg in Figures 17, 18 ,22, 23, 24 ,25 and 26.

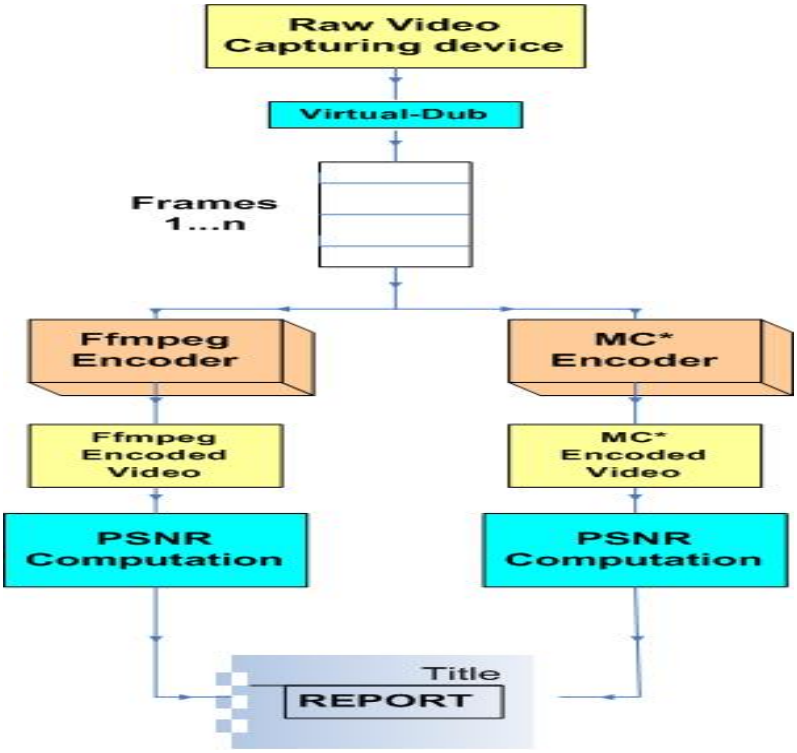

Figure 16: Report Generation Procedure

6.2 Performance of Frame Level Seek FFmpeg

We used an application called getframes that decodes an MPEG-2-video into a series of images. We built two versions of getframes: one using our FFmpeg extended library (FAS_FAST) and another using MC* library. We ran the profiler tool to profile FFmpeg Frame level seek library on videos from each category. The highest number of function calls made was to the avcodec.dll library at 4.6% of all function calls. This library contains all the functions related to decoding packets into frames. After that, among the FFmpeg libraries, swscale.dll^{FFmpeg} had the highest number of calls at 1.8%, with negligible calls to avcodec.dll and avutil.dll^{FFmpeg}. In the avcodec.dll^{FFmpeg}, most CPU cycles were utilized by tasks to decode the intra blocks in a frame, *ff_mpeg_decode_block_intra()*^{FFmpeg}, an FFmpeg API called by Frame level seek library. After that, the FFmpeg de-interlacing function, *avpicture_deinterlace()*^{FFmpeg} was most CPU centric. This is also an FFmpeg function called through FAS_FAST.

Comparing the performance of both the FFmpeg and MC* MPEG-2 decoders, the FFmpeg decoder is faster than MC* on both 32-bit and 64-bit x86 test machines as shown in Figure 27 for one video of each category. Table 26 shows the detailed comparison results on the 32-bit test machine.

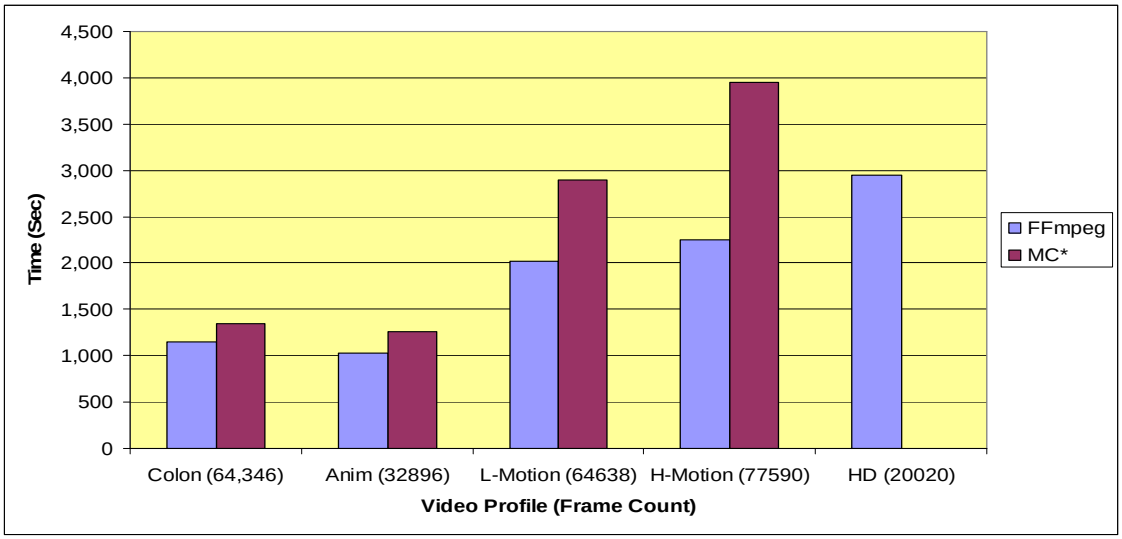

Figure 17: Frame Extraction Time Comparison between MC* and FFmpeg

6.3 Performance of FFmpeg Encode Library

For this test, we extracted frames from raw videos into a series of PNG images, after which they were encoded into an MPEG-2 video with an encoder application called "putframes". We built two versions of putframes: one using our FFmpeg extended library and another using MC* library. We ran the profiler tool to profile the two versions of the putframes application. The results are as follows. The highest percentage of function calls of 29.4% was made to the FFmpeg swsscale.dll *FFmpeg* library. This library contains all the functions related to decoding the packets into frames. Among the remaining FFmpeg libraries, avcodec.dll *FFmpeg* was called the highest number of times, 16.8%, with negligible calls to avformat.dll *FFmpeg* on both test machines.

In the swscale.dll ^{FFmpeg}, *sws_format_name()*^{FFmpeg}, an FFmpeg API called by *en_enc_frame()*^{FAS} took the most CPU time, followed by the FFmpeg motion search API, *ff_epzs_motion_search()*^{FFmpeg}, which is also called through *ff_enc_frame()*^{FAS}.

Comparing the performance of both the FFmpeg and MC* MPEG-2 encoders, the FFmpeg encoder performs faster than MC* on a 32-bit x86 architecture, whereas the MC* encoder performs faster than FFmpeg encoder on a 64 bit x86 architecture. Comparison results between the two libraries are shown in Table 29.

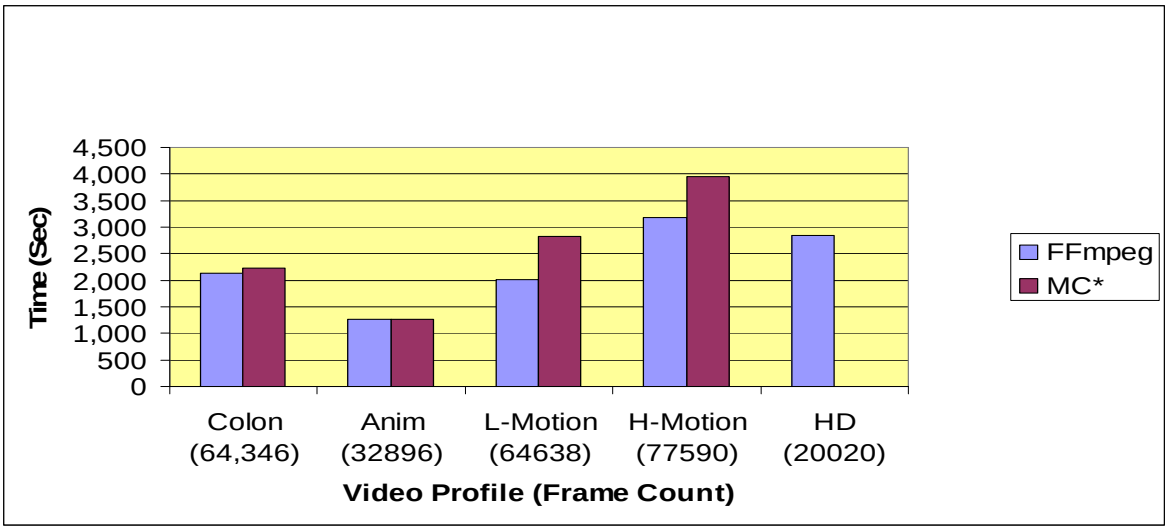

Figure 18: Comparison between FFmpeg and MC*Encoding Time

6.4 PSNR Comparison between MC* and FFmpeg

Video quality is a characteristic of a video passed through a video transmission or processing system, a formal or informal measure of perceived video degradation (typically, compared to the original video). Video processing systems may introduce some distortion or artifacts in the video signal. Video quality of codec is very important. Measurement of video quality can be made in terms of objective measurement or subjective measurement.

For objective measure of quality of a digital video processing system (e.g. video codec like DivX, XviD), the signal-to-noise ratio (SNR) and peak signal-to-noise ratio (PSNR) between the original video signal and signal passed through this system are often used. PSNR is the most widely used objective video quality metric. It describes the ratio

between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. Because many signals have a very wide dynamic range, PSNR is usually expressed in terms of the logarithmic decibel scale. To compute PSNR, we first need to calculate the mean squared error (MSE) between the original video sequence and the reconstructed sequence. For video sequences, MSE is simply the average squared pixel-by-pixel difference, which is a measure of the noise power introduced. Given the MSE, PSNR is defined as PSNR = 10 log (v^2/MSE) where the log is to base 10 and *v* is the maximum intensity value of the video signal. For example, for 8-bit intensity values, *v* = 255. The higher PSNR indicates a better performance. However, PSNR does not reflect subjective quality of videos as perceived by humans.

We computed PSNR values using the static build of FFmpeg. Comparing videos encoded with both FFmpeg and MC* encoders, FFmpeg performs better in all our test categories. The results are depicted in Figures 19-23.

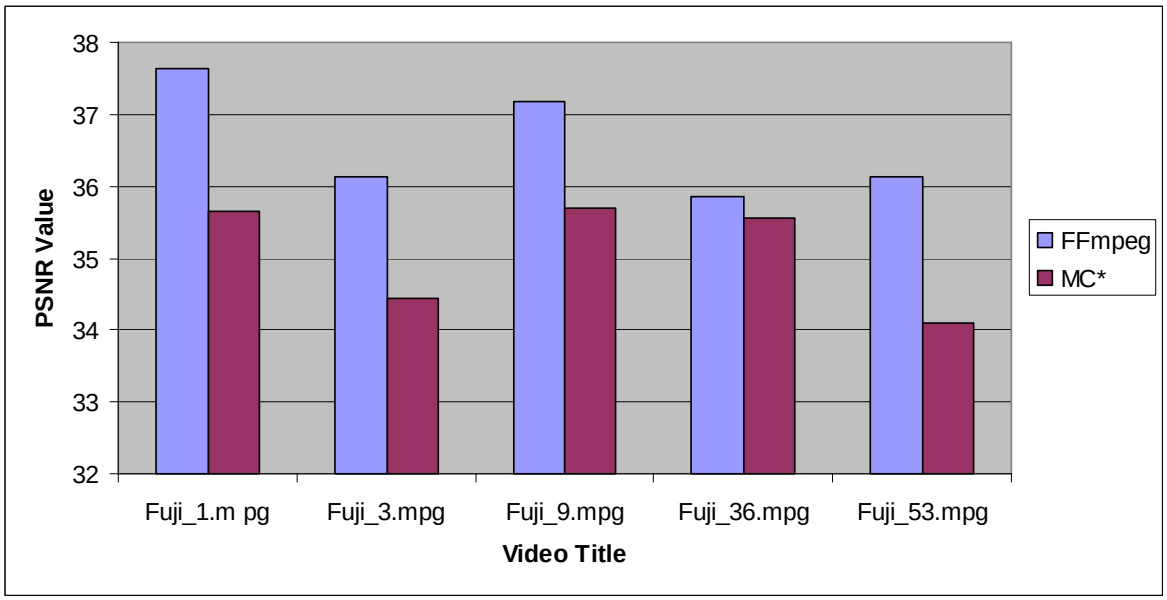

Figure 19: PSNR Comparison for Colonoscopy Videos

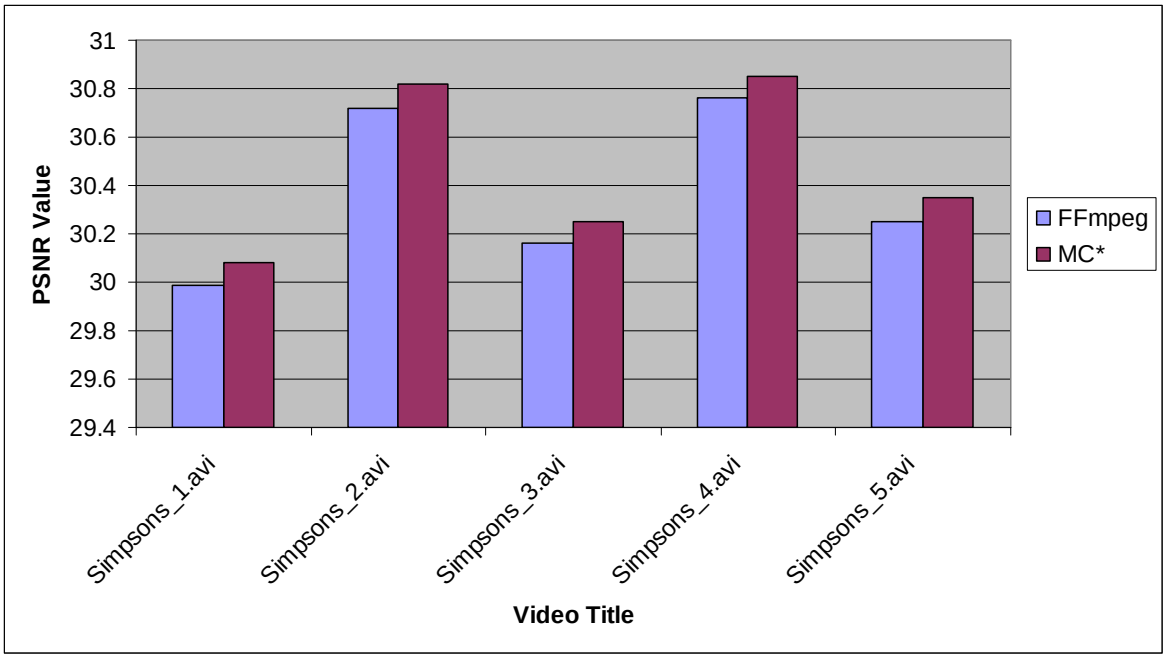

Figure 20: PSNR Comparison for Animated Videos

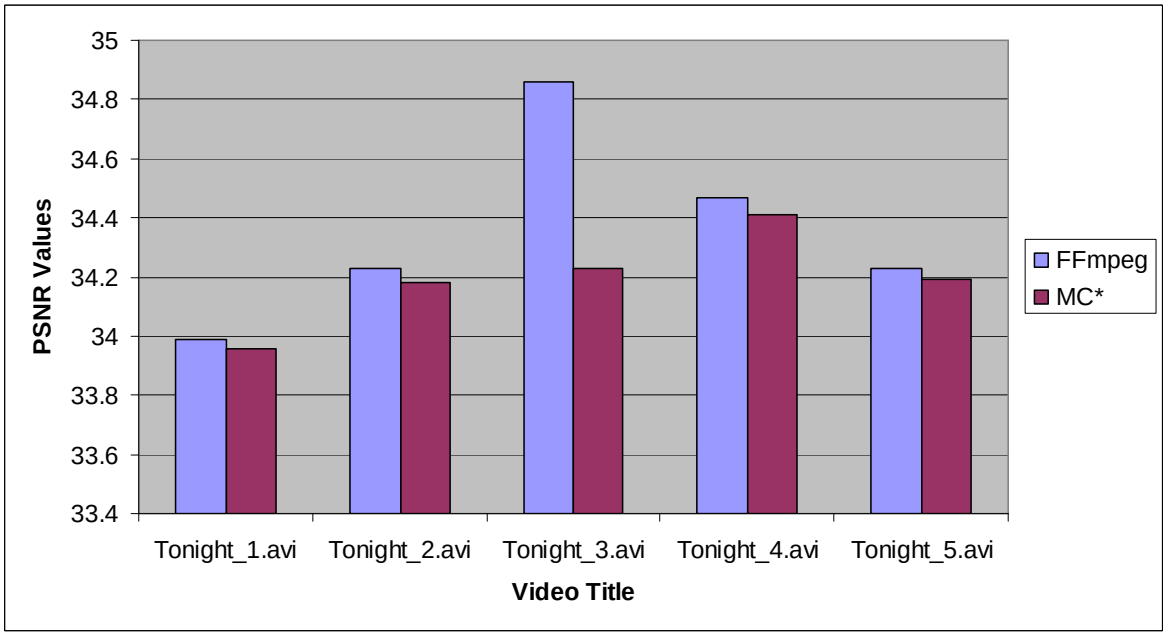

Figure 21: PSNR Comparison for Low Motion Videos

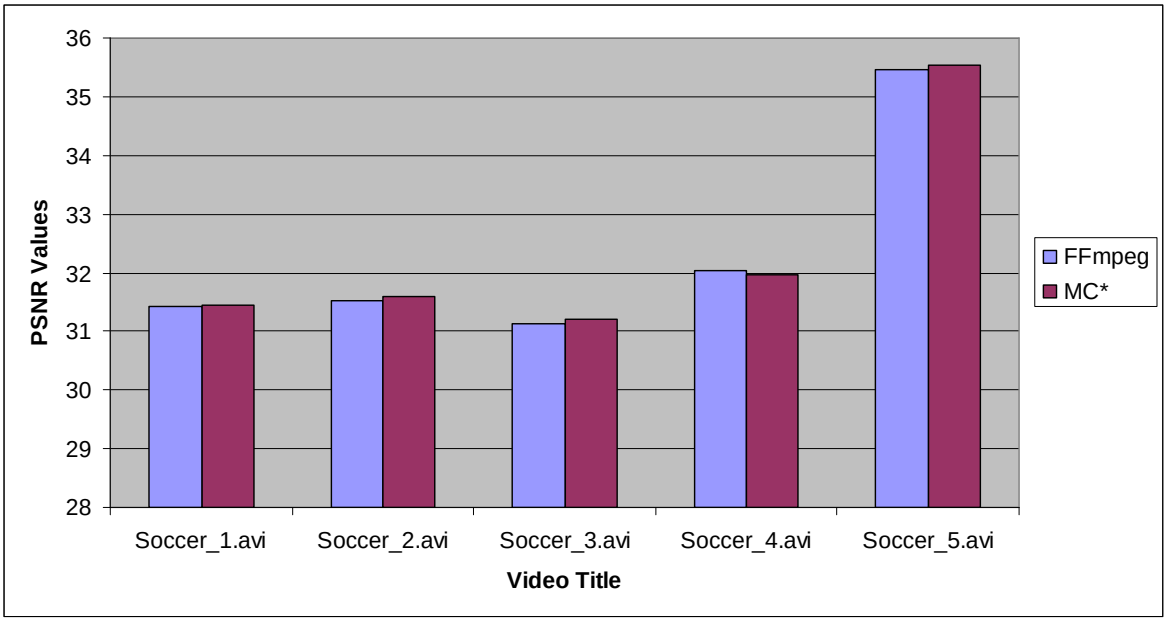

Figure 22: PSNR Comparison for High Motion Videos

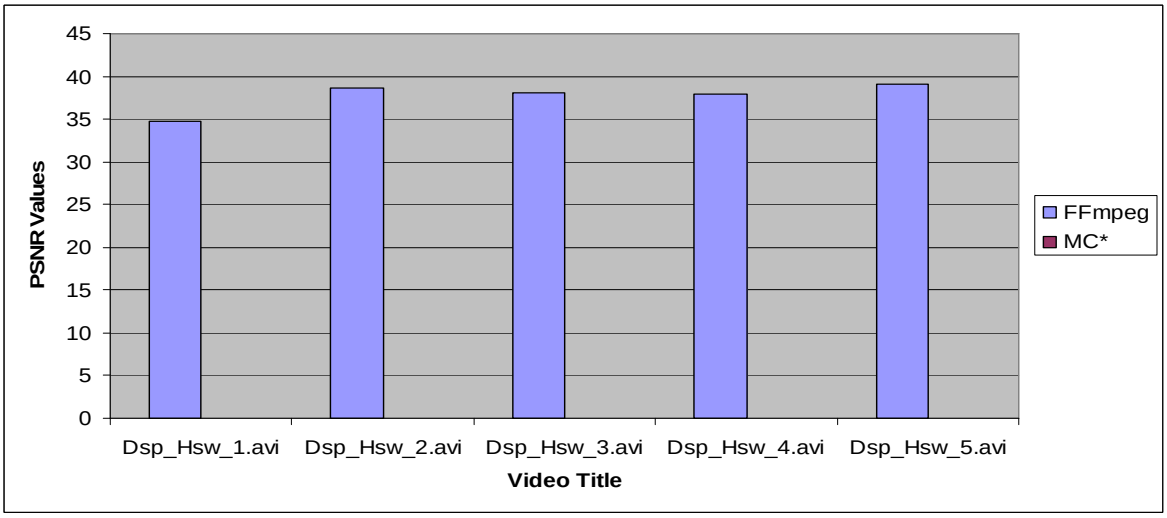

Figure 23: PSNR for High Definition Videos

6.5 Frame Level Seek and Encoding Comparison between MC* and FFmpeg

Comparing with FFmpeg, the MC* library mcmpeg.dll, containing functionalities for the decoder was used 43% of the time. FFmpeg utilized less memory compared to MC*, both in encoding and extracting frames from an MPEG-2 encoded video. Also FFmpeg

utilized less CPU cycles as compared to MC* for both test machines. The results are listed in Table 26 and Table 29 in the Appendix C, respectively.

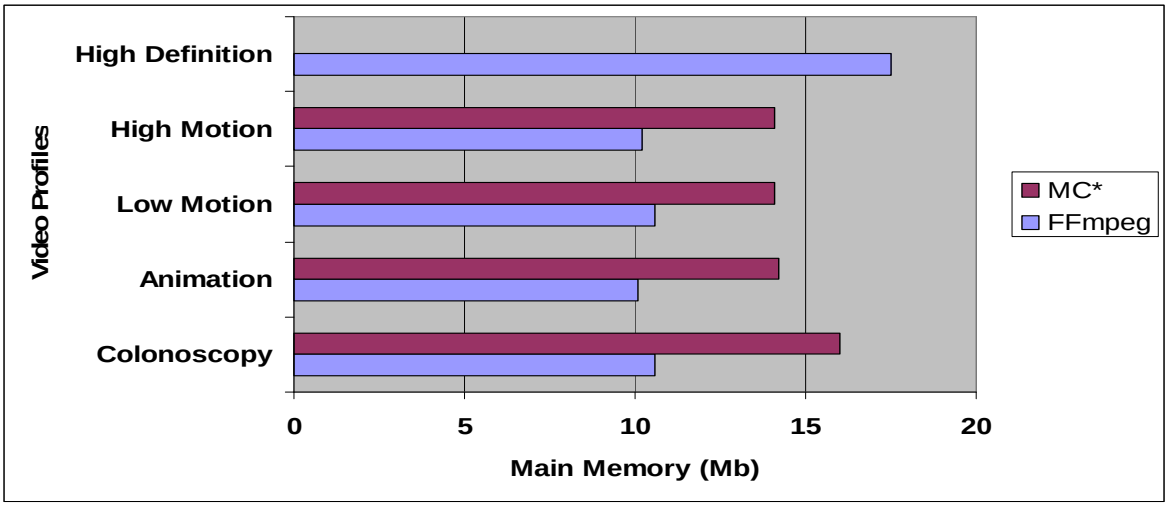

Figure 24: Comparison of Memory Utilization between Two Versions of putframes

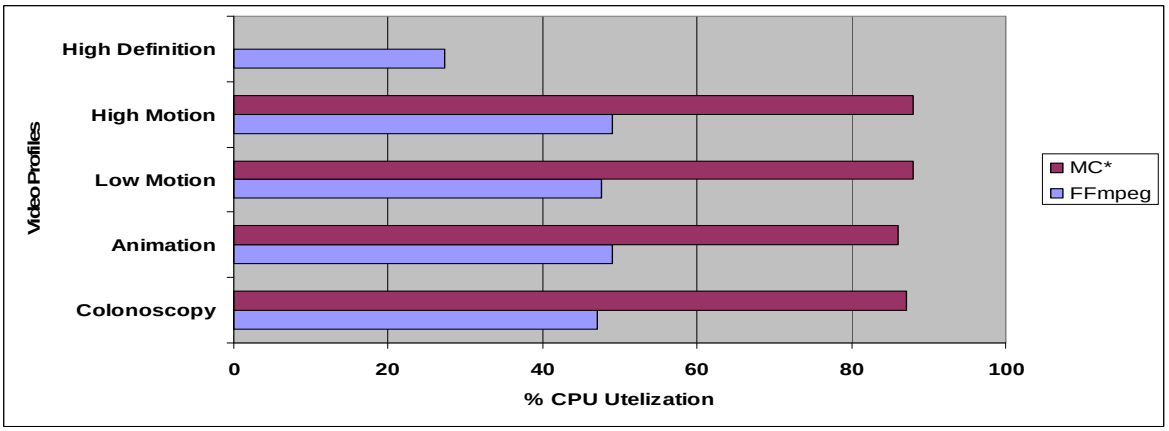

Figure 25: Comparison of CPU Utilization between Two Versions of putframes

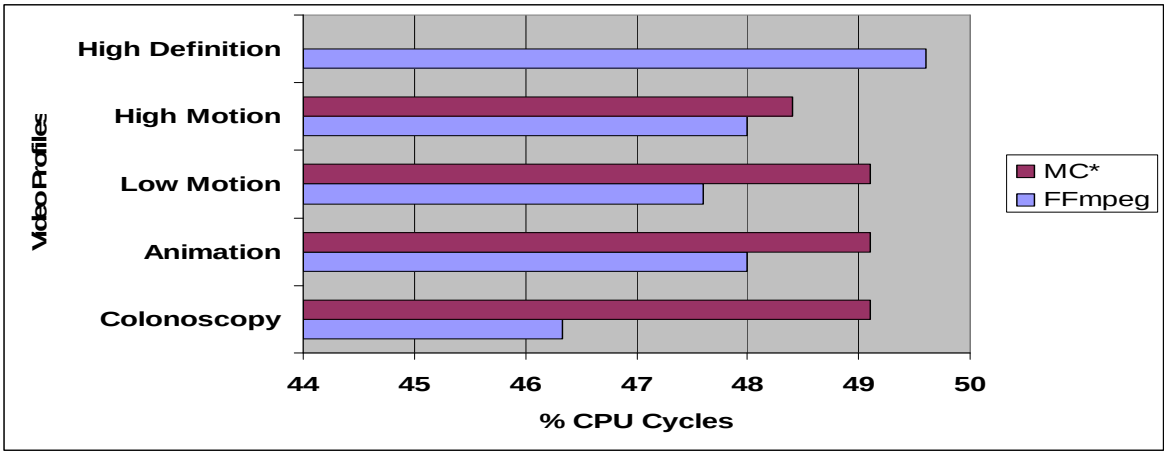

Figure 26: Comparison of CPU Utilization between Two Versions of getframes

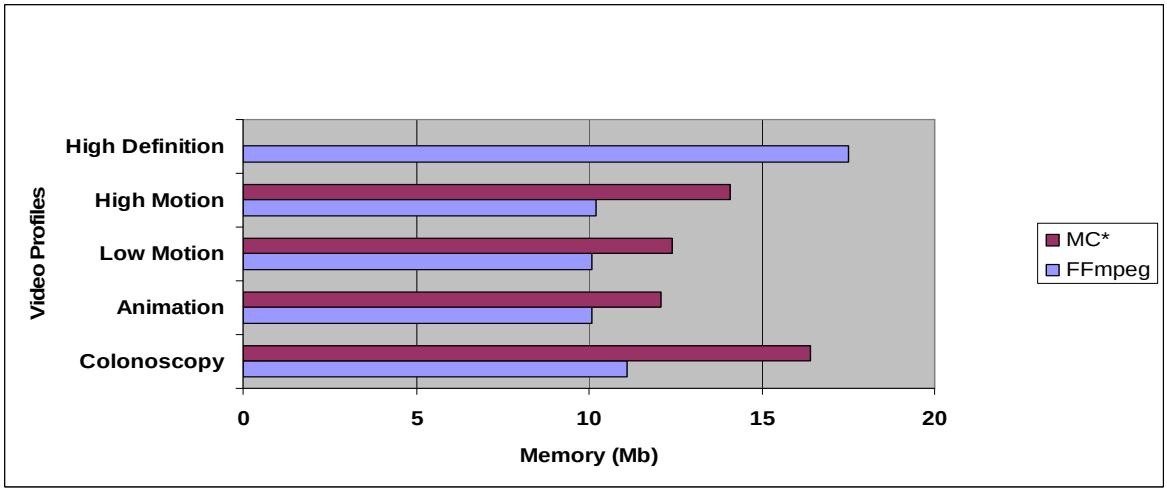

Figure 27: Comparison for Memory Utilization between Two Versions of getframes

6.6 Upfront Load Time Comparison between FFmpeg Frame Level Seek Fast and Slow

As discussed in Chapter 3, we developed two different versions of FFmpeg Frame level seek: Fast (FAS_FAST) and Slow (FAS_SLOW). FAS_SLOW does generation of a seek table before opening a video file, enabling it to seek to a particular frame but increasing the load time as compared to FAS_FAST because it has to decode the header of every I-frame present in the video stream. The FAS_FAST does a calculation of the seek table on the fly, which makes the load time negligible, but it is restricted to a certain type of

GOPs. Table 27 in the Appendix C shows the average time comparison between the FAS_FAST and FAS_SLOW to load the test video files.

We measured the time taken to generate the seek table generation for each file as the performance metric. Since FAS_SLOW computes DTS value for all I-frames before opening a file, it takes longer time to load the file. The FAS_FAST and FAS_SLOW gave the same results for seek comparison in case the test videos were encoded with specific parameters as discussed in Section '3.6.1'. The accuracy of frame level seek was tested by extracting the frames through "getframes" built for both FAS_FAST and FAS_SLOW versions. The time calculated is the average upfront load time of videos from each category. Figure 28 shows the time to build the seek table used by FAS_SLOW.

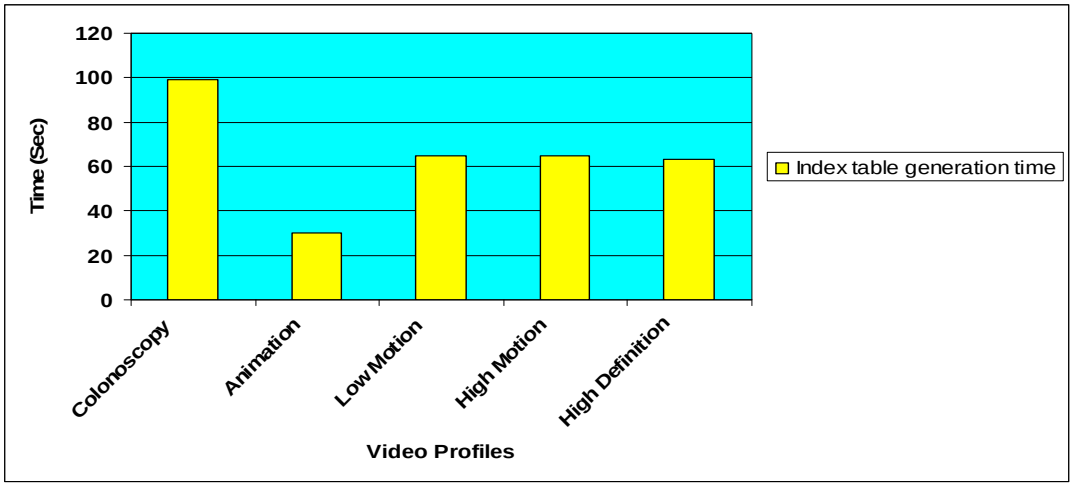

Figure 28: Time to Build Seek Table of FAS_SLOW

6.7 Integration with Applications

The FFmpeg Frame level seek Fast and Encode libraries were built as separate dlls for Windows platform compatible with both 32 bit and 64 bit architectures. We successfully integrated these libraries with six different applications (1) *SvideoPlayer Non Real Time;* (2) *SvideoPlayer Real Time;* (3) *SendoPaste;* (4) *EmCapture;* (5) *getframes_ff, and* (6) *putframes_ff.*

- *SvideoPlayer* is a Java based video player that calls our underlying Frame level seek library to play video streams. *SvideoPlayer* provides interface for playing a media

file up to 32x fast forward and 2x reverse. It also provides an interface for specifying the frame number to jump to. It uses the APIs exposed by Frame Level Seek library as discussed in Section 3.4 to perform these tasks. Figure 29 shows the user interface that calls SVideoPlayer APIs. The real-time version differs from the non-real time version as it can play a video stream while it is being captured whereas the non-real time version can only play videos that have already been captured and saved into a file.

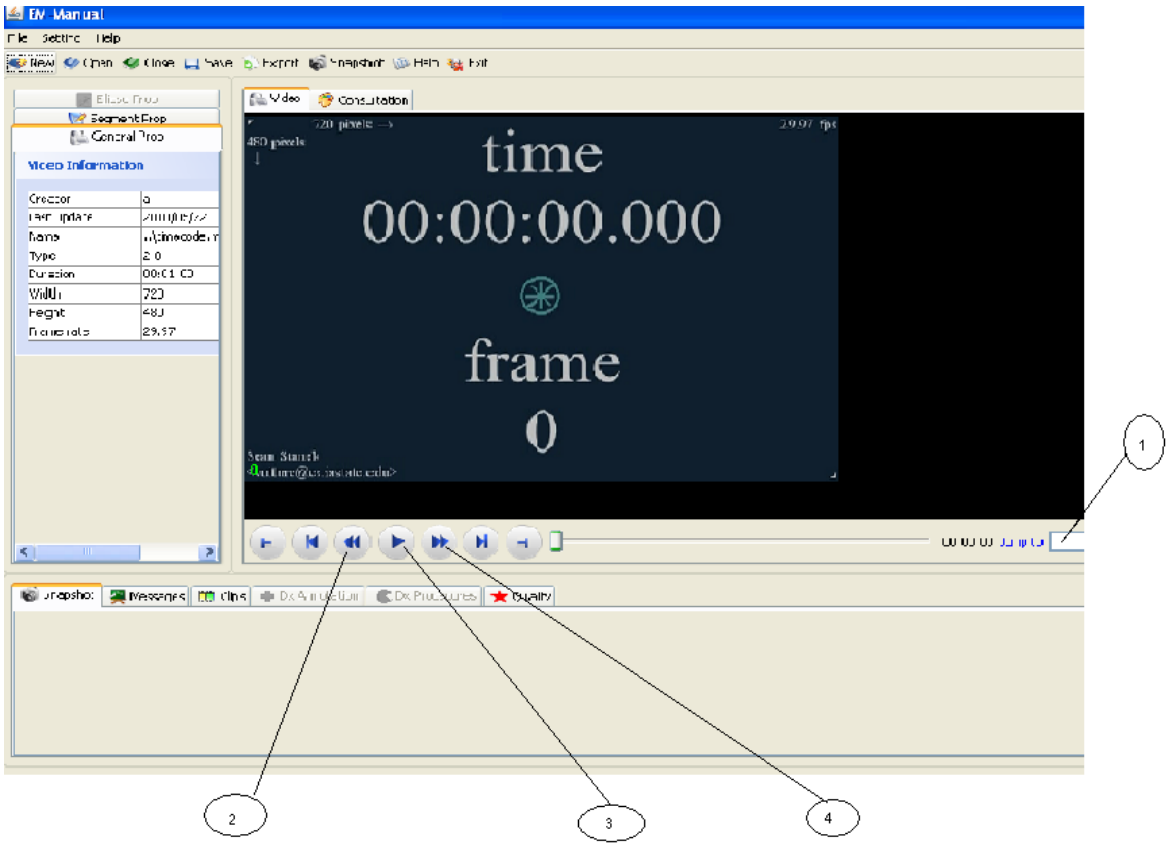

Figure 29: SvideoPlayer Components: (1) Frame number Box (2) Trick Play, supporting -2x (3) Play (4) Trick Play supporting +2x, +4x, +8x, +16x and +32x.

- *SendoPaste* is another dynamic linked library written in Java that supports (a) cutting a video file into smaller clips; (b) joining different video clips; and (c) transcoding a video clip into MPEG-1 or MPEG-2 format.

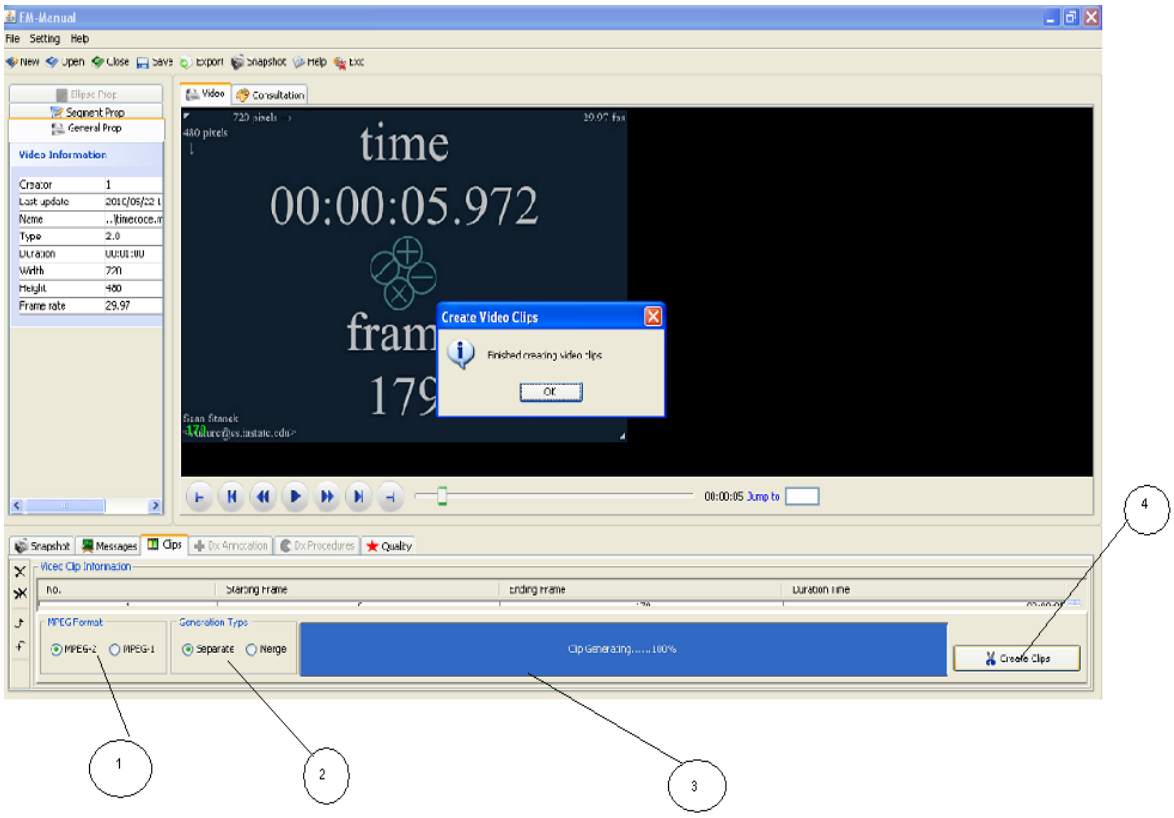

Figure 30: SendoPaste Components; (1) Codec selection option box (2) Merge option selection box (3) Status bar (4) Clip generation button.

- *EmCapture* is an application that captures video from a video capturing device and encoding it into MPEG-1 or MPEG-2 video files. It uses the Encode library.
- *getframes_ff* is a command line application to extract frames from a video file in JPEG, PNG, BMP and GIF format. This application uses FAS_FAST and FAS_SLOW libraries.
- *putframes_ff* is a command line application providing feature to encode frames available in JPEG, PNG, BMP and GIF format into an MPEG-2 video file according to the parameters passed. This application uses the Encode library with this application.

CHAPTER 7: CONCLUSION AND FUTURE WORK

We have discussed the extended FFmpeg library that supports frame accurate seek, encoding, decoding, and motion vector extractions for MPEG-2 videos. We reported performance of the extended FFmpeg library compared to a third party commercial software development kit. We successfully integrated the library with six existing applications.

Major challenges faced during the design of the three libraries were separation of functionality on the basis of internal private functions and external exposed APIs. We learnt the fundamentals of identifying and separating individual modules in large software on the basis of functionality and policy.

Large amount of memory fragmentation and reduction in performance of the library when used for a very long time lead us to the decision of using static internal buffers instead of dynamic allocation.

Practicing software engineering principle of following a specific pattern of nomenclature for APIs, data members and data structures from the beginning helped in reducing development time.

Testing the APIs individually, after integration, as a product and after integration with the application exposed over 20 bugs/issues. Nevertheless, there are still some limitations to be addressed as future work.

- The current FFmpeg Frame Level Seek Fast library supports video streams only. Future work should support MPEG-2 files with audio.
- The library currently supports closed GOP only. Future work should also support open GOP.
- From the analysis of PSNR for FFmpeg and MC*, the overall quality provided by FFmpeg is better than that of MC* except the first and the last GOPs. Further investigation is needed.
- The fast rewind algorithm used in *SvideoPlayer* is not very efficient, resulting in 50% much higher number of frames extracted compared to the fast forward play. Due to this, the CPU and memory utilization increases significantly during the fast rewind in comparison to normal or fast forward playback.

- Lack of debugging framework for Windows dlls was a major hindrance during development. GDB, a Linux based debugger was used to debug issues and then cross compile the library for windows platform. It is a bottleneck in the development process. Integrating “Wascana" to view debug prints can be a very helpful tool, reducing the cross-compilation time during development.
- The biggest challenge faced during development was lack of documentation and support. Since the major area of focus was MPEG-1 and MPEG-2 codec and MPEG-PS format, extension of documentation beyond this area of interest is still missing.

APPENDIX A: FFmpeg COMPILATION

This chapter provides an introduction to three different licenses: (1) GPL, (2) LGPL, and (3) FFmpeg. Next, we describe all the steps taken to port the FFmpeg library (which is natively built under C99) from Linux environment to Windows environment (Microsoft Windows Studio C++). A detailed description of all the built libraries, their specific tasks in the software stack are also be discussed. A brief discussion on various bugs (both in FFmpeg and the new libraries) which were fixed during the development of the new libraries are provided.

A.1 GPL License

GNU stands for GNU general public license. It is a free, copy left license for software and other kinds of works. The licenses for most software and other practical works are designed to take away freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee freedom to share and change all versions of a program--to make sure it remains free software for all its users. The Free Software Foundation uses GNU General Public License for most of the software; it applies also to any other work released this way by its authors.

Proprietary software developers have the advantage of money; free software developers need to make advantages for each other. Using the ordinary GPL for a library gives free software developers an advantage over proprietary developers: a library that they can use, while proprietary developers cannot use it.

A.2 LGPL License

LGPL stands for Lesser General Public License. It is used by a few (not by any means all) GNU libraries. The latest version is version 3. Using the Lesser GPL permits use of the library in proprietary programs; using the ordinary GPL for a library makes it available only for free programs. Using the ordinary GPL is not advantageous for every

library. There are reasons that can make it better to use the Lesser GPL in certain cases. The most common case is when a free library's features are readily available for proprietary software through other alternative libraries. In that case, the library cannot give free software any particular advantage, so it is better to use the Lesser GPL for that library.

A.3 FFmpeg License

FFmpeg is licensed under the GNU Lesser General Public License (LGPL) version 2.1 or later. However, FFmpeg incorporates several optional parts and optimizations that are covered by the GNU General Public License (GPL) version 2 or later. If those parts get used the GPL applies to all of FFmpeg. Using the GNU GPL will require that all the released improved versions be free software. To generate the LGPL compliant libraries, we are required to, follow certain steps, including:

(1) Compile FFmpeg without "**--enable-gpl**" and without "**--enable-nonfree**". (2)Use dynamic linking (on windows, this means linking to dlls) for linking with FFmpeg libraries. (3) Distribute the source code of FFmpeg, no matter if you modified it or not.
(4) make sure the source code corresponds exactly to the library binaries you are distributing. (5) Run the command "svn diff . libswscale > changes.diff" in the root directory of the FFmpeg source code to create a file with only the changes. (6) Explain how you compiled FFmpeg, for example the configure line, in a text file added to the root directory of the source code. (6) Use tar ball or a zip file for distributing the source code.
(7) Host the FFmpeg source code on the same web server as the binary you are distributing. (8) Add "This software uses code of FFmpeg licensed under the LGPLv2.1 and its source can be downloaded here" to every page in your website where there is a download link to your application. (9) Mention "This software uses libraries from the FFmpeg project under the LGPLv2.1" in your program "about box". (11) Mention in your EULA that your program uses FFmpeg under the LGPLv2.1. (12) If your EULA claims ownership over the code, you have to explicitly mention that you do not own FFmpeg, and where the relevant owners can be found. (13) Remove any prohibition of reverse engineering from your EULA. (14) Do not misspell

FFmpeg (two capitals F and lowercase "MPEG") (15) Do not rename FFmpeg dlls to some obfuscated name, but adding a suffix or prefix is fine (renaming "avcodec.dll" to "MyProgDec.dll" is not fine, but to "avcodec-MyProg.dll" is). (16) go through all the items again for any LGPL external library you compiled into FFmpeg (for example LAME). (17) make sure your program is not using any GPL libraries (notably libx264).

A.4 Doxygen documentation

Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D. It is an industry standard for documenting large scale projects.

It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code. Doxygen can also extract code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Visualization of the relations between the various elements by means of dependency graphs, inheritance diagrams, and collaboration diagrams, can be generated automatically.

Doxygen is developed under Linux and Mac OS X, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are also available. Many of the open source libraries like GNU Standard C++ Library, KDE, MySQL, Samba, etc are documented with Doxygen.

Since there is no documentation available for FFmpeg except from some mailing lists of m-player, we used Doxygen to document some major useful parts of FFmpeg.
Documents have been generated for on-line documentation browser (in HTML) which will be used as a Wiki and can be updated from different users.

A.5 Porting FFmpeg on Windows

FFmpeg does not build under Microsoft Visual Studio C++ compiler because MSVC++ does not adhere to C99, which FFmpeg developers do. Thus, the entire build process of FFmpeg was done with MSys+MinGW.

The libraries created by FFmpeg with MinGW are usable just like any other library (either static or shared), with a C99 compiler. With a modified build framework to behave like a C99 system, it was possible to use FFmpeg libraries with MSVC++.

However, it was not possible to debug the libav* libraries, since MSVC++ does not recognize the debug symbols generated by GCC.

A.5.1 Installing MSys

Get packages (1) MSYS-1.0.11.exe and (2) coreutils-5.97-MSYS-1.0.11-snapshot.tar.bz2. The first step is to install MSYS.exe in C:\msys\ directory. Then, it is required to unpack the file and copy coreutils-5.97\bin\pr.exe to C:\msys\bin\.

A.5.2 Installing MinGW

The following packages are required: (1) GNU Binutils BIN v2.20.1, (2) MinGW Runtime DLL v3.18, (3) MinGW Runtime DEV v3.18 , (4) MinGW API for MS-Windows DEV v3.14 to install MinGW, (5) GCC Version 4, and (5) GCC Full v4.4.0. Extract these packages to C:\MinGW\. These files in C:\MinGW\bin\ need to be renamed,

c++-sjlj.exe to c++.exe, cpp-sjlj.exe to cpp.exe, g++-sjlj.exe to g++.exe and gcc-sjlj.exe to gcc.exe

A.5.3 Downloading FFmpeg from SVN

Install TortoiseSVN, the source code for FFmpeg, which can be downloaded from “svn://svn.mplayerhq.hu/ FFmpeg /trunk”

A.5.4 Configuration for dynamic linked libraries under LGPL

The build is done in MSys command line. Go to the directory where FFmpeg is located. Run the command

"**. /configure --enable-shared --disable-static --enable-memalign-hack**" The --enable-memalign-hack option is necessary for FFmpeg to run MMX and SSE-optimized code on Windows. If there are no errors, run make and make install.

A.6 Build description

FFmpeg builds 17 libraries after compilation. The following 4 libraries were used (1) avcodec-52.dll (2) avformat-52.dll (3) avutil-50.dll and (4) swscale-0.dll. The numbers in the name are associated with the version of distribution. Complete list of the libraries built are described in the table below.

Table 22: FFmpeg Library Size

Index	Library	Size(bytes)
1	avcodec-52.32.0.dll	4,962,816
2	avcodec-52.36.0.dll	4,962,816
3	avcodec-52.dll	4,962,816
4	avcodec.dll	4,962,816
5	avdevice-52.2.0.dll	10,752
6	avdevice-52.dll	10,752
7	avdevice.dll	10,752
8	avformat-52.36.0.dll	729,088
9	avformat-52.dll	729,088
10	avformat-52.39.0.dll	729,088
11	avformat.dll	729,088
12	avutil-50.3.0.dll	75,264
13	avutil-50.dll	75,264
14	avutil.dll	75,264
15	swscale-0.7.1.dll	154,624
16	swscale-0.dll	154,624
17	swscale.dll	154,624

A.6.1 libavformat.dll and libavcodec.dll description

Many video file formats (AVI being a prime example) do not actually specify which codec(s) should be used to encode audio and video data; they merely define how an audio and a video stream (or potentially, several audio/video streams) should be combined into a single file. This is why sometimes, when an AVI file is opened, only sound is heard, but no picture - because the right video codec isn't installed. Thus, libavformat deals with parsing video files and separating the streams contained in them, and libavcodec.dll deals with decoding raw audio and video streams.

A.6.2 libswscale.dll description

FFmpeg has recently added a new interface, to handle image scaling. This new interface is more modular and faster. This library performs highly optimized image scaling and color space/pixel format conversion operations.

A.6.3 libavutil.dll description

It is a library containing functions for simplifying programming, including random number generators, data structures, mathematics routines and much more.

A.6.4 libavdevice.dll description

It is a library containing input and output devices for grabbing from and rendering to many common multimedia input/output software frameworks, including Video4Linux, Video4Linux2, VfW, and ALSA.

A.7 Bug Reports and Issues faced in Open Source Programming

Over 20 issues/bugs were encountered during development, integration and testing of the three modules with the applications, which were fixed and tested successfully.

Table 23: Bugs List

Bug #	Bug List For EM-Manual, FFmpeg	Status	Effected Libs.
1	Memory corruption due to FFmpeg free	Closed	Em-capture, libavcodec
2	Crash in video close	Closed	Em-capture
3	MPEG-1,2 Selection	Closed	SendoPaste
4	Delay opening a video file	Closed	SendoPaste/SVideo Player
5	Delay encoding a video file	Closed	SendoPaste
6	Crash on Video re-open	Closed	SVideoPlayer
7	HD Support	Closed	SendoPaste/SVideo Player
8	De-interlace Support	Closed	SVideoPlayer
9	Rewind algorithm not correct	Closed	SVideoPlayer
10	Jitter in Video Display	Closed	SVideoPlayer
11	Video Freeze after long playback	Closed	SVideoPlayer
12	Generated clip frame loss	Closed	SendoPaste
13	HD Opening Crash	Closed	SVideoPlayer
14	HD Inaccurate Aspect Ratio	Closed	SVideoPlayer
15	HD crash after re-open NON-HD video	Closed	SVideoPlayer
16	Inverted Saved Image	Closed	SVideoPlayer
17	Inaccurate last frame jump	Closed	SVideoPlayer
18	Jump to end Bug	Closed	SVideoPlayer
19	Merged Video inaccurate frame seek	Closed	SendoPaste
20	Frame Accuracy	Open	SVideoPlayer

APPENDIX B: DEFINITIONS

B.1 AC coefficient:

Any DCT coefficient for which the frequency in one or both dimensions is non-zero

B.2 B-picture, Bi-directionally predictive-coded picture

A picture that is coded using motion compensated prediction from past and/or future reference fields or frames

B.3 Backward compatibility

A newer coding standard is backward compatible with an older coding standard if decoders designed to operate with the older coding standard are able to continue to operate by decoding all or part of a bit stream produced according to the newer coding standard.

B.4 Backward motion vector

A motion vector that is used for motion compensation from a reference frame or reference field at a later time in display order

B.5 Backward prediction

Prediction from the future reference frame (field)

B.6 Base layer

First, independently decodable layer of a scalable hierarchy

B.7 Bit stream, stream

An ordered series of bits that forms the coded representation of the data

B.8 Bit rate

The rate at which the coded bit stream is delivered from the storage medium to the input of a decoder

B.9 Block

An 8-row by 8-column matrix of samples, or 64 DCT coefficients (source, quantized or de-quantized)

B.10 Byte aligned

A bit in a coded bit stream is byte-aligned if its position is a multiple of 8-bits from the first bit in the stream.

B.11 Byte

A sequence of 8-bits

B.12 Channel

A digital medium that stores or transports a bit stream constructed according to this specification.

B.13 Chrominance format

The format defines the number of chrominance blocks in a macro block.

B.14 Chrominance component

A matrix, block or single sample representing one of the two color difference signals related to the primary colors in the manner defined in the bit stream. The symbols used for the chrominance signals are Cr and Cb.

B.15 Coded B-frame

A B-frame picture or a pair of B-field pictures

B.16 Coded frame

A coded frame is a coded I-frame, a coded P-frame, or a coded B-frame

B.17 Coded I-frame

An I-frame picture or a pair of field pictures, where the first field picture is an I-picture and the second field picture is an I-picture or a P-picture

B.18 Coded P-frame

A P-frame picture or a pair of P-field pictures

B.19 Coded picture

A coded picture is made of a picture header, the optional extensions immediately following it, and the following picture data. A coded picture may be a coded frame or a

coded field. Coded video bit stream is a coded representation of a series of one or more pictures as defined in this specification.

B.20 Coded order

The coded order is the order in which the pictures are transmitted and decoded. This order is not necessarily the same as the display order of the pictures.

B.21 Coded representation

A data element as represented in its encoded form.

B.22 Coding parameters

Coding parameters are user-definable parameters that characterize a coded video bit stream. Bit streams are characterized by coding parameters. Decoders are characterized by the bit streams that they are capable of decoding.

B.23 Component

A matrix, block or single sample from one of the three matrices (luminance and two chrominance) that make up a picture

B.24 Compression

Reduction in the number of bits used to represent an item of data

B.25 Container or Wrapper Format

It is a meta-file format whose specification describes how data and metadata are stored (not coded). A program able to identify and open a container file might not be able to decode the contained data. This may be caused by the opening program lacking the required decoding algorithm, or the meta-data not providing enough information.

B.26 Constant bit rate coded video

A coded video bit stream with a constant bit rate

B.27 Constant bit rate

Operation where the bit rate is constant from start to finish of the coded bit stream

B.28 Data element

An item of data as represented before encoding and after decoding

B.29 DC coefficient

The DCT coefficient for which the frequency is zero in both dimensions

B.30 DCT coefficient

The amplitude of a specific cosine basis function

B.31 Decoder input buffer

The first-in first-out (FIFO) buffer specified in the video buffering verifier

B.32 Decoder

An embodiment of a decoding process

B.33 Decoding (process)

The process defined in this specification that reads an input coded bit stream and produces decoded pictures or audio samples.

B.34 De-quantization

The process of rescaling the quantized DCT coefficients after their representation in the bit stream has been decoded and before they are presented to the inverse DCT.

B.35 Discrete cosine transform (DCT)

Either the forward discrete cosine transform or the inverse discrete cosine transform. The DCT is an invertible, discrete orthogonal transformation.

B.36 Display aspect ratio

The ratio height/width (in SI units) of the intended display

B.37 Display order

The display order dictates the order in which the decoded pictures are displayed. Normally this is the same order in which they were presented at the input of the encoder.

B.38 Display process

The (non-normative) process by which reconstructed frames are displayed

B.39 Editing

Editing is the process by which one or more coded bit streams are manipulated to produce a new coded bit stream. Conforming edited bit streams must meet the requirements defined in this specification.

B.40 Encoder

An embodiment of an encoding process

B.41 Encoding (process)

A process, not specified in this specification that reads a stream of input pictures or audio samples and produces a valid coded bit stream as defined in this specification.

B.42 Fast forward playback

The process of displaying a sequence, or parts of a sequence, of pictures in display-order faster than real-time

B.43 Fast reverse playback

The process of displaying the picture sequence in the reverse of display order faster than real-time

B.44 Field

For an interlaced video signal, a “field” is the assembly of alternate lines of a frame. Therefore an interlaced frame is composed of two fields, a top field and a bottom field.

B.45 Flag

A one bit integer variable which may take one of only two values (zero and one)

B.46 Forward motion vector

A motion vector that is used for motion compensation from a reference frame or reference field at an earlier time in display order

B.47 Forward prediction

Prediction from the past reference frame (field)

B.48 Frame

A frame contains lines of spatial information of a video signal. For progressive video, these lines contain samples starting from one time instant and continuing through successive lines to the bottom of the frame. For interlaced video a frame consists of two fields, a top field and a bottom field. One of these fields will commence one field period later than the other.

B.49 Frame-based prediction

A prediction mode using both fields of the reference frame

B.50 Frame period

The reciprocal of the frame rate

B.51 Frame rate

The rate at which frames are be output from the decoding process

B.52 Future reference frame (field)

A future reference frame (field) is a reference frame (field) that occurs at a later time than the current picture in display order.

B.53 Frame reordering

Frame reordering is the process of reordering the reconstructed frames when the coded order is different from the display order. Frame reordering occurs when B-frames are present in a bit stream. There is no frame reordering when decoding low delay bit streams.

B.54 Group of pictures (GOP)

A notion defined only in ISO/IEC 11172-2 (MPEG-1 Video). In this specification, a similar functionality can be achieved by the mean of inserting group of pictures headers.

B.55 Header

The header is a block of data in the coded bit stream containing the coded representation of a number of data elements pertaining to the coded data that follow the header in the bit stream.

B.56 Interlace

The property of conventional television frames where alternating lines of the frame represent different instances in time. In an interlaced frame, one of the fields is meant to be displayed first. This field is called the first field. The first field can be the top field or the bottom field of the frame.

B.57 I-picture, intra-coded picture

A picture coded using information only from itself

B.58 Intra coding

Coding of a macro block or picture that uses information only from that macro block or picture

B.59 Layer

In a scalable hierarchy denotes one out of the ordered set of bit streams and (the result of) its associated decoding process (implicitly including decoding of all layers below this layer).

B.60 Luminance component

A matrix, block or single sample representing a monochrome representation of the signal and related to the primary colors in the manner defined in the bit stream. The symbol used for luminance is Y.

B.61 Macro block

The four 8 by 8 blocks of luminance data and the two (for 4:2:0 chrominance format), four (for 4:2:2 chrominance format) or eight (for 4:4:4 chrominance format) corresponding 8 by 8 blocks of chrominance data coming from a 16 by 16 section of the luminance component of the picture. Macro block is sometimes used to refer to the sample data and sometimes to the coded representation of the sample values and other data elements defined in the macro block header of the syntax defined in this part of this specification. The usage is clear from the context.

B.62 Motion compensation

The use of motion vectors to improve the efficiency of the prediction of sample values. The prediction uses motion vectors to provide offsets into the past and/or future

reference frames or reference fields containing previously decoded sample values that are used to form the prediction error.

B.63 Motion estimation

The process of estimating motion vectors during the encoding process.

B.64 Motion vector

A two-dimensional vector used for motion compensation that provides an offset from the coordinate position in the current picture or field to the coordinates in a reference frame or reference field.

B.65 Non-intra coding

Coding of a macro block or picture that uses information both from itself and from macro blocks and pictures occurring at other times.

B.66 P-picture, predictive-coded picture

A picture that is coded using motion compensated prediction from past reference fields or frame

B.67 Parity (of field)

The parity of a field can be top or bottom.

B.68 Past reference frame (field)

A past reference frame (field) is a reference frame (field) that occurs at an earlier time than the current picture in display order.

B.69 Picture

A Picture is a source, coded or reconstructed image data. A source or reconstructed picture consists of three rectangular matrices of 8-bit numbers representing the luminance and two chrominance signals. For progressive video, a picture is identical to a frame, while for interlaced video, a picture can refer to a frame, or the top field or the bottom field of the frame depending on the context.

B.70 Picture data

In variable-bitrate video, picture data is defined as all the bits of the coded picture, all the header(s) and user data immediately preceding it if any (including any stuffing

between them) and all the stuffing following it, up to (but not including) the next start code, except in the case where the next start code is an end of sequence code, in which case it is included in the picture data.

B.71 Prediction

The use of a predictor to provide an estimate of the sample value or data element currently being decoded

B.72 Prediction error

The difference between the actual value of a sample or data element and its predictor

B.73 Profile

A profile is defined as subset of the syntax of this specification. In this specification, the word “profile” is used as defined above. It should not be confused with other definitions of “profile”.

B.74 Progressive

The property of film frames where all the samples of the frame represent the same instances in time.

B.75 Peak signal-to-noise Ratio (PSNR)

PSNR is the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. Because many signals have a very wide dynamic range, PSNR is usually expressed in terms of the logarithmic decibel scale.

B.76 Quantization matrix

A set of sixty-four 8-bit values used by the de-quantizer

B.77 Quantized DCT coefficients

DCT coefficients before de quantization; a variable length coded representation of quantized DCT coefficients is transmitted as part of the coded video bit stream.

B.78 Quantizer scale

A scale factor coded in the bit stream and used by the decoding process to scale the de-quantization

B.79 Reconstructed frame

A reconstructed frame consists of three rectangular matrices of 8- bit numbers representing the luminance and two chrominance signals. A reconstructed frame is obtained by decoding a coded frame.

B.80 Reconstructed picture

A reconstructed picture is obtained by decoding a coded picture. A reconstructed picture is either a reconstructed frame (when decoding a frame picture), or one field of a reconstructed frame (when decoding a field picture). If the coded picture is a field picture, then the reconstructed picture is the top field or the bottom field of the reconstructed frame.

B.81 Reference field

A reference field is one field of a reconstructed frame. Reference fields are used for forward and backward prediction when P-pictures and B-pictures are decoded. Note that when field P-pictures are decoded, prediction of the second field P picture of a coded frame uses the first reconstructed field of the same coded frame as a reference field.

B.82 Reference frame

A reference frame is a reconstructed frame that was coded in the form of a coded I-frame or a coded P-frame. Reference frames are used for forward and backward prediction when P-pictures and B-pictures are decoded.

B.83 Re-ordering delay

A delay in the decoding process that is caused by frame reordering

B.84 Reserved

The term “reserved” when used in the clauses defining the coded bit stream indicates that the value may be used in the future for ISO/IEC defined extensions.

B.85 RGB Color Format

Different intensities of red, green, and blue are added to generate various colors.

RGB is not a uniform color space. RGB is not efficient since it uses equal bandwidth for each color component. However, human eye is most sensitive to green, less sensitive to red, and least sensitive to blue.

B.86 Sample aspect ratio, SAR

This specifies the relative distance between samples. It is defined (for the purposes of this specification) as the vertical displacement of the lines of luminance samples in a frame divided by the horizontal displacement of the luminance samples. Thus its units are (meters per line) / (meters per sample).

B.87 Scalability

Scalability is the ability of a decoder to decode an ordered set of bit streams to produce a reconstructed sequence. Moreover, useful video is output when subsets are decoded. The minimum subset that can thus be decoded is the first bit stream in the set which is called the base layer. Each of the other bit streams in the set is called an enhancement layer. When addressing a specific enhancement layer, “lower layer” refer to the bit stream which precedes the enhancement layer.

B.88 Saturation

Limiting a value that exceeds a defined range by setting its value to the maximum or minimum of the range as appropriate

B.89 Skipped macro block

A macro block for which no data is encoded

B.90 Slice

A consecutive series of macro blocks which are all located in the same horizontal row of macro blocks. It is a sequence of consecutive rows in an image. Slices can be bottom to top or top to bottom.

B.91 Signal-to-noise Ratio (S.N.R.)

It is a measure used to quantify how much a signal has been corrupted by noise. It is defined as the ratio of signal power to the noise power corrupting the signal. A ratio higher than 1:1 indicates more signal than noise.

B.92 Source, input

Term used to describe the video material or some of its attributes before encoding.

B.93 Spatial prediction

Prediction derived from a decoded frame of the lower layer decoder used in spatial scalability

B.94 Start codes (system and video)

Start codes are 32-bit codes embedded in that coded bit stream that are unique. They are used for several purposes including identifying some of the structures in the coding syntax. Stuffing (bits) and stuffing (bytes) are code-words that may be inserted into the coded bit stream that are discarded in the decoding process. Their purpose is to increase the bit rate of the stream which would otherwise be lower than the desired bit rate.

B.95 Temporal prediction

Prediction derived from reference frames or fields other than those defined as spatial prediction

B.96 Temporal scalability

A type of scalability where an enhancement layer also uses predictions from sample data derived from a lower layer using motion vectors. The layers have identical frame size, and chrominance formats, but can have different frame rates.

B.97 Tristimulus Theorem

Any color can be obtained by mixing three primary colors in an appropriate proportion. Primary colors cannot be obtained by mixing the other two primary colors. Examples of primary colors are red, green, and blue. Three primary colors are sufficient to represent all colors since there are three types of color receptors in a human eye.

B.98 Variable bit rate

Operation where the bit rate varies with time during the decoding of a coded bit stream

B.99 Variable length coding (VLC)

A reversible procedure for coding that assigns shorter code-words to frequent events and longer code-words to less frequent events

B.100 Video sequence

It is the highest syntactic structure of coded video bit streams in MPEG. It contains a series of one or more coded frames.

B.101 YUV Color Format

Luminance is closely related to brightness whereas chrominance is related to hue and saturation. Thus YUV uses luminance and color-differencing signals. It encodes a color image or video taking human perception into account, allowing reduced bandwidth for chrominance components, thereby typically enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using a "direct" RGB-representation. Y' stands for the luma component (the brightness) and U and V are the chrominance (color) components. YUV is the basic color used by the NTSC, PAL, and SECAM composite color TV standards.

B.102 ME_ZERO

This motion vector estimation algorithm internally uses no motion vector search but uses a 0,0 vector whenever one is needed.

B.103 ME_FULL

This motion vector estimation algorithm supports H.264 codec.

B.104 ME_LOG

This motion vector estimation algorithm instead of performing full motion vector search, it performs search of log 2 block sizes.

B.105 ME_EPZS

This motion vector estimation algorithm uses enhanced predictive zonal search method.

B.106 ME_X1

This motion vector estimation algorithm is reserved for experimentation at the moment.

B.107 ME_HEX

This motion vector estimation algorithm performs hexagon based search for motion vectors.

B.108 ME_UMH

This motion vector estimation algorithm performs uneven multi-hexagon based search for motion vectors.

B.109 ME_ITER

This motion vector estimation algorithm performs iterative motion vector search.

B.110 ME_TESA

This motion vector estimation algorithm performs a transformed exhaustive for motion vectors.

B.111 YCbCr Color Format

Most image compression standards adopt this color format as an input image signal. Human does not recognize chrominance details as in luminance details. Sub-sampling format: J: a: b Ex. 4:4:4, 4:4:0, 4:2:2, 4:2:0, 4:1:1, 4:1:0 defined in terms of a reference region of J pixels wide and 2 pixels high.

- a: number of chroma pixels (Cr and Cb) taken in the first row
- b: number of chroma pixels taken in the second row
- Cr and Cb are sub sampling at the same location

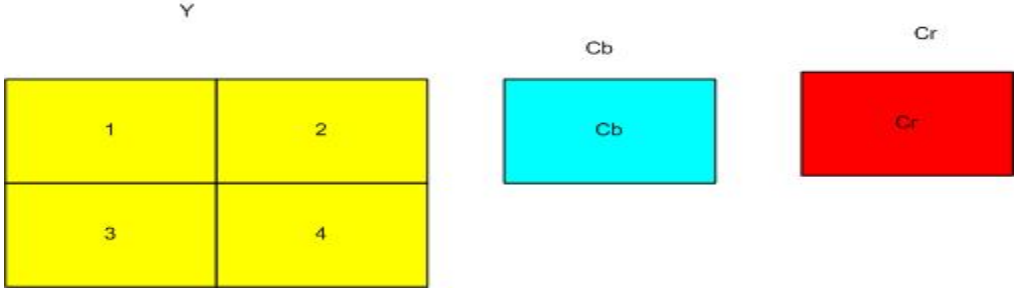

Figure 31: YCbCr Color Format

APPENDIX C: API IMPLEMENTATION DETAILS

C.1 FFmpeg Frame Level Seek APIs

- ***fas_error_type fas_open_video (fas_context_ref_type *, char *)***

To initialize the Frame level seek library, this API is called.Initialization phase starts with allocating index table with the Index table API call *seek_init_table (int)* [FAS], initializing the seek table with appropriate size passed in the parameter field. *int av_open_input_file(AVFormatContext **, const char *, AVInputFormat *, int , AVFormatParameters *)* [FFmpeg] is responsible for opening the input file and extracting the requisite information from the first parameter which is a reference to AVFormatContext structure, member of *fas_context_type* [FAS] structure.

AVFormatContext [FFmpeg] structure contains information for the container of the opened stream; it has members like number of media streams in the file denoted by *nb_streams* [FFmpeg], filename of the input file denoted by filename [FFmpeg], pointers to different input streams which are stored in an array of pointers denoted by *AVStream *streams [MAX_STREAMS]* [FFmpeg] (MAX_STREAMS [FFmpeg] being the maximum number of streams supported by FFmpeg, which is 20), position of the first frame given by int64_t *start_time* [FFmpeg]; duration of the stream populated in duration [FFmpeg], size of input file in bytes given by int64_t *file_size* [FFmpeg], total stream Bit-rate in bit/s, given by int *bit_rate* [FFmpeg] (This value can be 0 if FFmpeg can not compute Bit-Rate), *Codec-ID* of video stream required to open video decoder video_*codec_id* FFmpeg; Codec-ID of Audio Stream required by audio decoder given by *audio_codec_id* [FFmpeg]; Codec-ID of subtitle stream required to open sub-title decoder *subtitle_codec_id* [FFmpeg]. The library file avformat-52.dll contains the functions required to extract all the information and maintain this structure.

After identifying the required information associated with the container class, the next step is to extract the information associated with the incoming stream, which is pointed to by the *AVStream *streams [MAX_STREAMS]* [FFMPEG] in *AVFormatContext* [FFmpeg] structure. This API reads packets from specified media file to get stream information. It is useful for file formats with no headers such as MPEG. This API also computes the real

frame-rate. The logical file position is not changed by this API; examined packets are buffered for later processing. It returns a value greater than or equal to 0 if successful, *AVERROR* [FFmpeg] on error. Since a container can contain 20 different streams, the stream number associated with first video stream is computed and stored in the *stream_idx* [FFmpeg] member of *fas_context_type* [FAS] structure eliminating overhead for future references.

The *AVStream* [FFmpeg] structure is associated with every stream which contains the requisite information necessary for stream playback. Some of the members of this structure are, *index* [FFmpeg] which is the stream index in *AVStream *streams [MAX_STREAMS]* [FFmpeg] array, *AVCodecContext* * [FFmpeg], pointer to the codec context for this stream, frame_rate [FFMPEG], giving the frame rate of this stream, first DTS [FFmpeg] for the packet and the PTS [FFmpeg] information, quality factor of the stream given by *quality* [FFmpeg], number of frames given by nb_frames [FFmpeg] (0 if unknown), aspect ratio of frame *sample_aspect_ratio* [FFmpeg], Parser context denoted by *AVCodecParserContext* * [FFmpeg]. All the functionality required for the maintenance of the codec and the related data structures is provided in avcodec-52.dll.

The initialization API is responsible for allocation of all the internal buffers required during the lifetime of execution according to the design policy.

The API *AVCodec *avcodec_find_decoder (enum Codec-ID)* [FFmpeg] is responsible for returning a handle to the decoder passed in the parameter. It returns a NULL in case of failure. Opening of a codec is performed by the API *int avcodec_open(AVCodecContext *, AVCodec *).* This API internally does some type checking before calling the *AVCodec init ()*[FFmpeg] API which is provided for each codec specifically. All the initialization of internal data members is done in *void private_fill_vid_info (fas_context_ref*)* [FAS] function call.

In FFmpeg FAS (Fast) DTS of frames in first and last GOP are stored in the seek_table [FAS] which is a member of *fas_context_type* [FAS] . DTS of frames in first GOP is computed by internal function *private_complete_seek_table (fas_context_ref*)* [FAS], returning *FAS_ERROR_TYPE* [FAS]. Function *private_compute_frame_count (fas_context_ref*)* [FAS] does dual purpose of computing frame count and populating *seek_table* [FAS] with DTS of frames in last GOP. It does this by seeking to the last GOP, counting the number of frames in last GOP and populating the *seek_table* [FAS] with DTS values of frames. Successful call to this API returns *FAS_SUCESS* [FAS].

- ***fas_error_type fas_seek_to_frame (fas_context_ref_type, uint64_t)***

This API performs seek to a particular frame buffer passed by the application in the input video stream. It accomplishes this by comparison of the *target_index* [FAS] with the current index returned by *uint64_t fas_get_frame_index (fas_context_ref*)* [FAS.] Current implementation depends on the seek table generated during opening of video file for seeking to a frame in the first and last GOP. If the target index is greater than first GOP or less than the last GOP, we use the internal function *fas_error_type.*

fas_seek_to_nearest_key (fas_context_ref, uint64_t target_index)* [FAS] which does the task of seeking in the video stream to nearest I-frame with DTS value less than the *target_index* [FAS] and updating the current frame index in *fas_context_type* [FAS,] the first step of computation of timestamp for nearest I-frame from the *target_index* [FAS] is to compute the delta value which is modulo of the target index with GOP size.

Time Stamp = (uint64_t) (context->format_context->streams[context->stream_idx] ->time_base.den*(target_index - (delta)))/fas_get_frame_rate(context); This value allows us to call FFmpeg seek API based on timestamps int *av_seek_frame(AVFormatContext *s, int stream_index, int64_t timestamp, int flags)* [FFmpeg], where parameter 1 is the stream index computed during opening of file, timestamp value should be pre-computed and flags define the direction in which we need to seek, the direction is based on timestamps, it can be *AVSEEK_FLAG_BACKWARD* [FFmpeg] which makes the stream to seek in decreasing time stamps value from the current value otherwise its *AVSEEK_FLAG_FORWARD* [FFmpeg]. The value of *frame_index* [FAS] field is updated in the *fas_context_type* [FAS] accordingly.

On success *fas_error_type fas_step_forward (fas_context_ref*)* [FAS] is called number of times equal to the frame count less than the *target_index* [FAS]. This API is also responsible for updating any seek table entries as the decoding of video packets is done in this API which makes it to compute the DTS of a particular frame and then depending upon if it's a I-frame, it's value is inserted in the seek table.

The internal function *avcodec_decode_video (codec_context*, frame_buffer*, frameFinished*, packet.data, packet.size)* [FFmpeg] provided by the FFmpeg library is responsible for decoding the video packets. Parameter *codec_context* [FFmpeg] is a pointer to the codec context of video stream; *frame buffer*[FAS] is a temporary storage which is

allocated statically; the value of *frameFinished* [FFmpeg] is updated internally by this API when it decodes data requisite for a complete frame; *packet_data* [FFmpeg] represent the packet read by the FFmpeg function int *av_read_frame(fas_context_ref*, packet*)* [FFmpeg] which is not responsible for decoding the data but filing and initializing all the internal data structures required; *packet_size* [FFmpeg] varies according to the stream to be decoded. After the frame is decoded, it is checked against if it is a I-frame and its value is updated in the *seek _table.*

C.2 FFmpeg Encode APIs

- ***en_error_type en_enc_setup(en_Context_Struct **,ip_Context_Struct*)***

This API is called with an *ip_Context_Struct* [encode] initialized as the second parameter by the application specifying the video parameters to encode the video. The parameters to be specified are described in Table 17.

The first parameter *en_Context_Struct* [encode] is a double reference to *out_context_type* [encode], which stores references to both FFmpeg defined data types and Encode defined data types. See Table 16 for reference.

This API is responsible to initialize all the data structures and allocate buffers which will be required by both the encoder and decoder during the execution of application. All the internal function calls start with private_* which is the same pattern followed in Frame Level Seek.

This API internally calls *private_en_init_context()*[encode] function which does the memory allocation for context. Finding an encoder from available options is performed via the API *AVcodec *avcodec_find_encoder(Codec_ID)* [FFmpeg], where *Codec_ID* [FFmpeg] is an enumeration for storing names of all the encoders available. Note that since it's a static list, some of these encoders may not be available depending on the license and the settings in the .configure file, refer chapter 9.6 for more details. The registered codec's are stored in a linked list, *first_avcodec* [FFmpeg] being the head and *CodecID* [FFmpeg] is a field in *AVCodec* [FFmpeg] structure storing the name of that codec. *AVCodecContext *avcodec_alloc_context ()*[FFmpeg] allocates a reference to *AVCodecContext* [FFmpeg] and sets its fields to contain default values. The resulting structure can be de-allocated by simply calling *av_free()*[FFmpeg]. On

success, it returns an *AVCodecContext* [FFmpeg] reference pointer populated with default values or NULL on failure.

The AVOutputFormat **guess_format (const char *, const char *, const char *)* [FFmpeg], API is used to get a reference to the *AVFormatContext* [FFmpeg] structure which takes a short name as the first parameter, filename as the second and mime type as the third. At least one of these should be passed in the parameter and any one match will result in success. Formats are stored in a linked list with *first_oformat*[FFmpeg] being the head of registered formats.

The API *AVStream *av_new_stream(AVFormatContext *, int)* [FFmpeg] adds a new stream to a media file. This takes a reference to *the AVformatContext*[FFmpeg] as the first parameter, which should be previously allocated and a file-format-dependent stream ID as the second parameter. It also initializes the new stream to be used directly. Before opening a codec for encoding, we need to specify a minimum set of parameters for the codec, which are passed by the application as reference to the structure *ip_context_ptr* [encode]. This API is also responsible to allocate and open the decoder.

The internal function static *en_error_type private_en_dec_setup(en_Context_Struct *)*[encode], performs the initialization and memory allocation for the decoder. The encoder requires some scratch buffer of the size of the frame to be encoded. As the frame size depends on the pixel format used for encoding, the API *int avpicture_get_size(enum PixelFormat, int, int)* [FFmpeg] exposed by FFmpeg performs this calculation for us, which takes Pixel Format as the first parameter, width and height of the frame as second and third parameters respectively.

Function *int avpicture_fill (AVPicture *, uint8_t *, enum PixelFormat, int, int)* [FFmpeg] is responsible for filling the *AVPicture*[FFmpeg] fields. *AVPicture* [FFmpeg] structure acts as a reference to the data of a frame. Depending on the specified picture format, one or multiple image data pointers and line sizes are initialized. If a planar format is specified, several pointers will be pointing to the specific picture planes. This structure has two fields namely, *uint8_t *data [4]* [FFmpeg] and *int linesize [4]* [FFmpeg]; the *int *data [4]* [FFmpeg] *field* [FFmpeg] pointers to all the four planes of a frame, which are stored in the second parameter of the API *avpicture_fill ()*[FFmpeg]. The second parameter, *int linesize [4]* [FFmpeg] is responsible for storing the width of the planes pointed to by the data array. The third parameter of the

API is responsible for storing the pixel format of the frame, fourth and fifth refers to the width and height of the frame respectively. This API returns the size of the buffer allocated on success or -1 on failure.

*SwsContext *sws_getContext (int srcW, int srcH, enum PixelFormat srcFormat, int dstW, int dstH, enum PixelFormat dstFormat, int flags, SwsFilter *srcFilter, SwsFilter *dstFilter, const double *param)* [FFmpeg] API is a wrapper over the scaling functionality provided by the library. Refer Chapter 9.2 for its implementation details. It takes the source width, height, and pixel format as the first three parameters and destination width, height and pixel format as the next three, respectively. The rest of parameters are used to define which algorithm and parameters specific to the algorithm are to be used. It returns a reference to the SwsContext structure in case of success or a NULL reference in case of failure. This structure is freed by calling the *void sws_freeContext (struct SwsContext *)* [FFmpeg] API, and is used by *int sws_scale (struct SwsContext *, uint8_t* srcSlice[], int srcStride[], int srcSliceY, int srcSliceH, uint8_t* dst[], int dstStride[])*[FFmpeg] API.

This API scales the image slice in a source slice and puts the resulting scaled slice. See Chapter 9.1 for details in the destination image The first parameter is reference to the allocated context from *sws_getContext ()*[FFmpeg] API. Parameter *srcSlice* [FFmpeg] is the array containing the pointers to the planes of the source slice. Parameter *srcStride* [FFmpeg] is the array containing the strides for each plane of the source image. Parameter *srcSliceY* [FFmpeg] is the position in the source image of the slice to process, that is the number (counted starting from zero) in the image of the first row of the slice. Parameter *srcSliceH* [FFmpeg] is the height of the source slice, which is the number of rows in the slice. Parameter dst is the array containing the pointers to the planes of the destination image. Parameter *dstStride* [FFmpeg] is the array containing the strides for each plane of the destination image.

Various options available for scaling algorithms are (1) SWS_FAST_BILINEAR, (2) SWS_BILINEAR, (3) SWS_BICUBIC, (4) SWS_X, SWS_POINT, (5) SWS_AREA, (6) SWS_BICUBLIN, (7) SWS_GAUSS, (8) SWS_SINC, (9) SWS_LANCZOS, and (10) SWS_SPLINE. Quality and speed of these scaling algorithms are listed in the order from lower quality (fast speed) to higher quality (slow speed): SWS_FAST_BILINEAR, SWS_BILINEAR, SWS_BICUBIC, SWS_X, SWS_POINT, SWS_AREA, SWS_BICUBLIN, SWS_GAUSS, SWS_SINC, SWS_LANCZOS, and SWS_SPLINE.

In the current implementation, we used SWS_BICUBIC[FFmpeg] as the scaling algorithm as it maintains a balance between speed and quality.

- ***en_error_type en_enc_frm(en_Context_Struct*,AVFrame*)***

To encode raw input data by specified encoder, we call this API. It first scales the input frame according to the *img_convert_ctx*[encode] initialized previously in *en_enc_setup()*[encode] . It then calls int *avcodec_encode_video(AVCodecContext *, uint8_t *, int, const AVFrame *)*[FFmpeg] to encode the frame data according to the codec opened. It encodes the video frame passed as a reference in the fourth parameter into a buffer passed as a reference in the second parameter. The first parameter is a reference to the *AVCodecContext* [FFmpeg] already allocated and the third parameter specifies the maximum size of the output buffer, i.e. the maximum size of encoded frame in bytes. A negative return value specifies an error during execution where zero specifies that the frame has been saved for future reference i.e. it will be used as a reference to encode future frames. A positive return value signifies the number of bytes used from the input buffer. This API internally calls *int (*encode) (AVCodecContext *, uint8_t *, int, void *)*[FFmpeg], an API registered with the codec.

The encoded data is to be put into a video stream. The data is put into packets and then written into the stream according to the format. We use the API *void av_init_packet(AVPacket *)*[FFmpeg] to initialize optional fields of a packet with the default values. The API *int av_interleaved_write_frame(AVFormatContext *, AVPacket *)*[FFmpeg] performs the task of writing the packets to the stream according to the format.

The packet must contain one audio or video frame. If the packets are already correctly interleaved, the application should call *av_write_frame ()*[FFmpeg] instead as it is slightly faster in this scenario. It is essential to keep in mind that completely non-interleaved input will need huge amounts of memory to interleave if used with this API, so it is preferable to interleave at the de-muxer level. The return value is less than zero in case of an error; otherwise it is zero.

C.3 Overview of swsscale.dll

The swscale.dll library is responsible for image scaling, color conversion, etc. This is distributed in the form of libswscale.dll. This module can be divided into two paths: (1) Main Path and (2) Special Conversion as described in Figure 31.

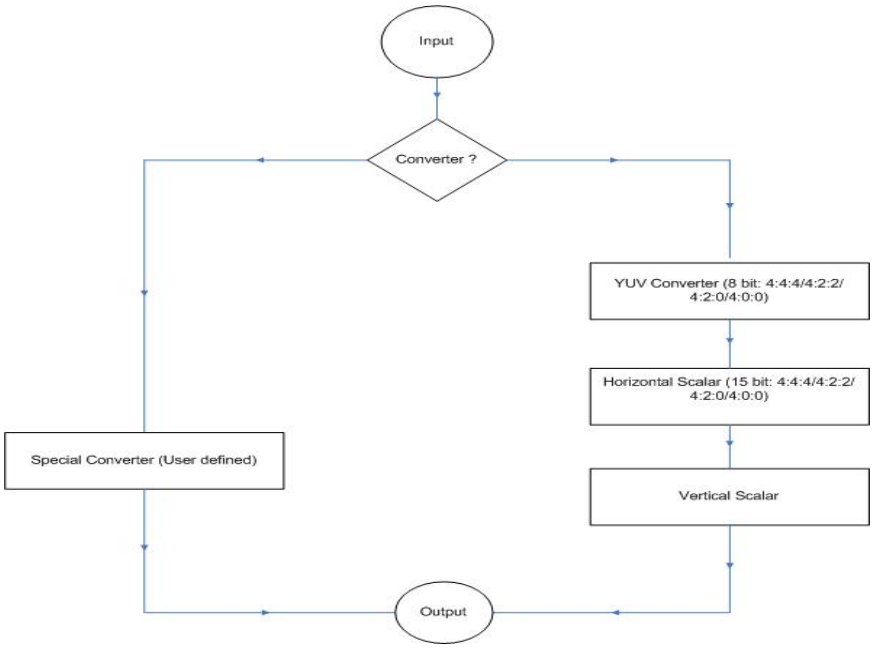

Figure 32: SwsScale Library Structure

Each side must be capable of handling slices, that is, consecutive non-overlapping rectangles of dimension (0, slice_top) - (picture_width, slice_bottom). Special converters generally are unscaled converters of common formats, like YUV 4:2:0/4:2:2 to RGB15/16/24/32. Though it could also in principle contain scalers optimized for the following specific common cases.

Main path: The main path is used when no special converter can be used. The code is designed as destination line pull architecture. That is, for each output line the vertical scaler pulls lines from a ring buffer. When the ring buffer does not contain the wanted line, then it is pulled from the input slice through the input converter and horizontal scaler. The result is also stored in the ring buffer to serve future vertical scaler requests. When no more output can be generated because lines from a future slice would be needed, then all

remaining lines in the current slice are converted, horizontally scaled and put in the ring buffer. (This is done for luma and chroma, each with possibly different numbers of lines per picture.)

YUV Converter: When the input to the main path is not planar 8 bits per component YUV or 8-bit gray, it is converted to planar 8-bit YUV. Two sets of converters exist for this currently: One performs horizontal downscaling by 2 before the conversion; the other leaves the full chroma resolution, but is slightly slower. The scaler will try to preserve full chroma when the output uses it. It is possible to force full chroma with SWS_FULL_CHR_H_INP even for cases where the scaler thinks it is useless.

Horizontal scaler: There are several horizontal scalers. A special case worth mentioning is the fast bilinear scaler that is made of runtime-generated MMX2 code using specially tuned pshufw[FFmpeg] instructions. The remaining scalers are specially-tuned for various filter lengths. They scale 8-bit unsigned planar data to 16-bit signed planar data. Future >8 bits per component inputs will need to add a new horizontal scaler that preserves the input precision.

Vertical scaler and output converter: There is a large number of combined vertical scalers and output converters like unscaled output converters, unscaled output converters that average 2 chroma lines, bilinear converters, arbitrary filter length converters and Plain C 8-bit 4:2:2 YUV to RGB converters. RGB with less than 8 bits per component uses dither to improve the subjective quality and low-frequency accuracy.

After the scaling structure is assigned, and the encoder parameters are set, we *call int avcodec_open (AVCodecContext *, AVCodec*)* [FFmpeg], taking the parameters in the form of references to previously allocated codec and *AVCodecContext* [FFmpeg]. Which internally calls *int (*init) (AVCodecContext *)*[FFmpeg] function, initialization API for the specific codec passed as the second parameter. To write to the specified file in the container format, FFmpeg requires the file to be opened with *int url_fopen(ByteIOContext **s, const char *filename, int flags)* [FFmpeg] API. To write header for the specified media stream, FFmpeg provides *av_write_header (AVFormat*)*[FFmpeg], which has a return type void and reads the filename from the previously assigned name.

C.4 De-interlacing implementation in Frame Level Seek

De-interlace mechanism in Frame Level Seek is implemented in the private API *private_fas_pre_process_video_frame(fas_context_ref_type *, AVPicture *, void **)*[FAS]; Currently it can be managed with *'DO_DEINTERLACE* [FAS]*'* Flag which is specified in FFmpeg _fas.h file as a global constant. The frames will be de-interlaced only if this Flag is set. This API acts as a wrapper over the exposed FFmpeg API *int avpicture_deinterlace(AVPicture *dst, const AVPicture *src, enum PixelFormat , int width, int height)* [FFmpeg]. The first parameter of FFmpeg API is a scratch buffer which will contain the de-interlaced frame data which is allocated as *deinterlace_buf* [FAS] during the opening of the video. The second parameter is the source frame which needs to be de-interlaced. The FFmpeg API works only with PIX_FMT_YUV420P, PIX_FMT_YUV422P, PIX_FMT_YUV444P, PIX_FMT_YUV411P and PIX_FMT_GRAY8 pixel formats. For other formats, it will return value less than 0 to indicate a failure. The third and fourth parameters are frame width and height, respectively.

This API internally uses two FFmpeg APIs depending on whether we need to de-interlace in-place or not. These are (1) *deinterlace_bottom_field_inplace(uint8_t *src1, int src_wrap, int width, int height)* [FFmpeg] and (2) *deinterlace_bottom_field(uint8_t *dst, int dst_wrap, const uint8_t *src1, int src_wrap, int width, int height)* [FFmpeg]. The *uint8_t *src1* is the source plane which can be Y, Cb or Cr; *src_wrap* signifies the width of the data field. The *uint8_t *dst* in case of the second API gives the destination where the de-interlaced data will be copied. Width and height are the dimensions of the frame. The API returns value 0 in case of success and -1 in case of failure. In case the FFmpeg API *avpicture_deinterlace()*[FFmpeg] fails, we do not proceed further, which will cause that particular frame to be written without being de-interlaced.

APPENDIX D: ADDITIONAL EXPERIMENTAL RESULTS

Table 24: Performance of FFmpeg FAS Library

Video Profile	Video	Hardware Profile	CPU (%) Cycles	Mem (Mb)	Time (Sec)
1	Fuji_1.mpg	Inslab	46.34	10.6	792
		Bigvision	12.6	9.6	732
	Fuji_3.mpg	Inslab	46.34	10.6	1,530
		Bigvision	12.6	9.6	1,172
	Fuji_9.mpg	Inslab	46.34	10.6	2,124
		Bigvision	12.6	9.6	1,803
	Fuji_36.mpg	Inslab	46.34	10.6	1,341
		Bigvision	12.6	9.6	994
	Fuji_53.mpg	Inslab	46.34	10.6	1,698
		Bigvision	12.6	9.6	1,090
2	Simpsons_1.avi	Inslab	48	10.1	1,261
		Bigvision	12.6	9.2	1,201
	Simpsons_2.avi	Inslab	48	10.1	1,276
		Bigvision	12.6	9.2	1,208
	Simpsons_3.avi	Inslab	48	10.1	1,287
		Bigvision	12.6	9.2	1,156
	Simpsons_4.avi	Inslab	48	10.1	1,198
		Bigvision	12.6	9.2	1,223
	Simpsons_5.avi	Inslab	48	10.1	1,201
		Bigvision	12.6	9.2	1,152
3	Tonight_1.avi	Inslab	47.6	10.6	2,020
		Bigvision	11.1	9.8	2,001
	Tonight_2.avi	Inslab	47.6	10.6	2,061
		Bigvision	11.1	9.8	2,049
	Tonight_3.avi	Inslab	47.6	10.6	2,044
		Bigvision	11.1	9.8	2,941
	Tonight_4.avi	Inslab	47.6	10.6	2,110
		Bigvision	11.1	9.8	2,060
	Tonight_5.avi	Inslab	47.6	10.6	2,344
		Bigvision	11.1	9.8	2,040

Table 24: (continued)

Video Profile	Video	Hardware Profile	CPU (%) Cycles	Mem (Mb)	Time (Sec)
4	Soccer_1.avi	Inslab	48	10.2	3,180
		Bigvision	11.9	9.8	1,830
	Soccer_2.avi	Inslab	48	10.2	3,720
		Bigvision	11.9	9.8	1,674
	Soccer_3.avi	Inslab	48	10.2	1,861
		Bigvision	11.9	9.8	1,723
	Soccer_4.avi	Inslab	49	10.2	3,746
		Bigvision	11.9	9.8	1,601
	Soccer_5.avi	Inslab	49	10.2	1,765
		Bigvision	11.9	9.8	1,567
5	Dsp_Hsw_1.avi	Inslab	49.6	17.5	2,941
		Bigvision	12.7	17.4	2,901
	Dsp_Hsw_2.avi	Inslab	49.6	17.5	2,840
		Bigvision	12.7	17.4	2,763
	Dsp_Hsw_3.avi	Inslab	49.6	17.5	2,903
		Bigvision	12.7	17.4	2,897
	Dsp_Hsw_4.avi	Inslab	49.6	17.5	2,884
		Bigvision	12.7	17.4	2,883
	Dsp_Hsw_5.avi	Inslab	49.6	17.5	2,967
		Bigvision	12.7	17.4	2,901

Table 25: Performance of FFmpeg Encode Library

Video Profile	Video	Hardware Profile	CPU (%) Cycles	Mem. (Mb)	Time (FF)
1	Fuji_1.mpg	Inslab	47.1	11.1	1,154
		Bigvision	9.1	10.4	1,014
	Fuji_3.mpg	Inslab	47.1	11.1	2,580
		Bigvision	9.1	10.4	2,411
	Fuji_9.mpg	Inslab	47.1	11.1	3,480
		Bigvision	9.1	10.4	2,973
	Fuji_36.mpg	Inslab	47.1	11.1	1,346
		Bigvision	9.1	10.4	1,301

Table 25: (continued)

Video Profile	Video	Hardware Profile	CPU (%) Cycles	Mem. (Mb)	Time (FF)
2	Simpsons_1.avi	Inslab	49	10.1	1,026
		Bigvision	9.1	9.3	914
	Simpsons_2.avi	Inslab	49	10.1	948
		Bigvision	9.1	9.3	910
	Simpsons_3.avi	Inslab	49	10.1	940
		Bigvision	9.1	9.3	921
	Simpsons_4.avi	Inslab	49	10.1	1,021
		Bigvision	9.1	9.3	906
	Simpsons_5.avi	Inslab	49	10.1	1,045
		Bigvision	9.1	9.3	917
3	Tonight_1.avi	Inslab	47.6	10.1	2,020
		Bigvision	10.1	8.6	2,001
	Tonight_2.avi	Inslab	47.6	10.1	2,061
		Bigvision	10.1	8.6	2,049
	Tonight_3.avi	Inslab	47.6	10.1	2,044
		Bigvision	10.1	8.6	2,941
	Tonight_4.avi	Inslab	47.6	10.1	2,110
		Bigvision	10.1	8.6	2,060
	Tonight_5.avi	Inslab	47.6	10.1	2,344
		Bigvision	10.1	8.6	2,040
4	Dsp_Hsw_1.avi	Inslab	27.4	17.5	2,768
		Bigvision	12.7	17.4	2,675
	Dsp_Hsw_2.avi	Inslab	27.4	17.5	3,181
		Bigvision	12.7	17.4	2,987
	Dsp_Hsw_3.avi	Inslab	27.4	17.5	2,169
		Bigvision	12.7	17.4	2,042
	Dsp_Hsw_4.avi	Inslab	27.4	17.5	3,300
		Bigvision	12.7	17.4	3,124
	Dsp_Hsw_5.avi	Inslab	27.4	17.5	2,220
		Bigvision	12.7	17.4	1,995

Table 26: Comparison of Performance between getframes_ff and getframes_mc

Prof	Video (.mpg)	Hardware Profile	CPU (%) Cycles (FF)	CPU (%) Cycles (MC*)	Mem. (Mb) (FF)	Mem. (Mb) (MC*)	Time (FF)	Time (MC*)
1	Fuji_1.mpg	Inslab	46.34	49.1	10.6	16	792	1,254
		Bigvision	12.6	14.8	9.6	10.1	732	1,165
	Fuji_3.mpg	Inslab	46.34	49.1	10.6	16	1,530	1,720
		Bigvision	12.6	14.8	9.6	10.1	1,172	1,341
	Fuji_9.mpg	Inslab	46.34	49.1	10.6	16	2,124	2,221
		Bigvision	12.6	14.8	9.6	10.1	1,803	2,398
	Fuji_36.mpg	Inslab	46.34	49.1	10.6	16	1,341	1,743
		Bigvision	12.6	14.8	9.6	10.1	994	1,150
	Fuji_53.mpg	Inslab	46.34	49.1	10.6	16	1,698	1,654
		Bigvision	12.6	14.8	9.6	10.1	1,090	1,201
2	Simpsons_1.avi	Inslab	48	49.1	10.1	14.2	1,261	1,263
		Bigvision	12.6	12.6	9.2	13.1	1,201	1,206
	Simpsons_2.avi	Inslab	48	49.1	10.1	14.2	1,276	1,266
		Bigvision	12.6	12.6	9.2	13.1	1,208	1,211
	Simpsons_3.avi	Inslab	48	49.1	10.1	14.2	1,287	1,301
		Bigvision	12.6	12.6	9.2	13.1	1,156	1,241
	Simpsons_4.avi	Inslab	48	49.1	10.1	14.2	1,198	1,234
		Bigvision	12.6	12.6	9.2	13.1	1,223	1,201
	Simpsons_5.avi	Inslab	48	49.1	10.1	14.2	1,201	1,221
		Bigvision	12.6	12.6	9.2	13.1	1,152	1,194
3	Tonight_1.avi	Inslab	47.6	49.1	10.6	14.1	2,020	2,821
		Bigvision	11.1	12.1	9.8	12.8	2,001	2,101
	Tonight_2.avi	Inslab	47.6	49.1	10.6	14.1	2,061	2,124
		Bigvision	11.1	12.1	9.8	12.8	2,049	2,311
	Tonight_3.avi	Inslab	47.6	49.1	10.6	14.1	2,044	1,987
		Bigvision	11.1	12.1	9.8	12.8	2,941	2,816
	Tonight_4.avi	Inslab	47.6	49.1	10.6	14.1	2,110	2,760
		Bigvision	11.1	12.1	9.8	12.8	2,060	2,202
	Tonight_5.avi	Inslab	47.6	49.1	10.6	14.1	2,344	2,801
		Bigvision	11.1	12.1	9.8	12.8	2,040	2,100

Table 26: (continued)

Prof	Video (.mpg)	Hardware Profile	CPU (%) Cycles (FF)	CPU (%) Cycles (MC*)	Mem. (Mb) (FF)	Mem. (Mb) (MC*)	Time (FF)	Time (MC*)
4	Soccer_1 .avi	Inslab	48	48.4	10.2	14.1	3,180	3,950
		Bigvision	11.9	12	9.8	14.7	1,830	2,115
	Soccer_2 .avi	Inslab	48	48.4	10.2	14.1	3,720	4,203
		Bigvision	11.9	12	9.8	14.7	1,674	1,952
	Soccer_3 .avi	Inslab	48	48.4	10.2	14.1	1,861	2,033
		Bigvision	11.9	12	9.8	14.7	1,723	1,801
	Soccer_4 .avi	Inslab	49	48.4	10.2	14.1	3,746	3,991
		Bigvision	11.9	12	9.8	14.7	1,601	1,932
	Soccer_5 .avi	Inslab	49	48.4	10.2	14.1	1,765	1,954
		Bigvision	11.9	12	9.8	14.7	1,567	1,596
5	Dsp_Hs w_1.avi	Inslab	49.6	N/A	17.5	N/A	2,941	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,901	N/A
	Dsp_Hs w_2.avi	Inslab	49.6	N/A	17.5	N/A	2,840	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,763	N/A
	Dsp_Hs w_3.avi	Inslab	49.6	N/A	17.5	N/A	2,903	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,897	N/A
	Dsp_Hs w_4.avi	Inslab	49.6	N/A	17.5	N/A	2,884	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,883	N/A
	Dsp_Hs w_5.avi	Inslab	49.6	N/A	17.5	N/A	2,967	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,901	N/A

Table 27: Comparison between FFmpeg and MC* PSNR Values

Prof	Video	Encoder	Y	U	V	PSNR
1	Fuji_1.m pg	FFmpeg	36.72	40.84	40.07	37.64
		MC*	34.60	39.62	38.74	35.66
	Fuji_3.mpg	FFmpeg	34.72	40.14	42.18	36.14
		MC*	34.68	39.11	40.33	34.45
	Fuji_9.mpg	FFmpeg	36.29	39.83	39.76	37.18
		MC*	34.67	39.02	38.90	35.69

Table 27: (continued)

Prof	Video	Encoder	Y	U	V	PSNR
1	Fuji_36.mpg	FFmpeg	34.59	41.61	40.11	35.86
		MC*	34.46	39.90	38.68	35.56
	Fuji_53.mpg	FFmpeg	35.02	40.31	39.41	36.12
		MC*	32.83	39.23	38.59	34.09
2	Simpsons_1.avi	FFmpeg	28.65	34.22	37.53	29.99
		MC*	28.71	34.61	37.84	30.08
	Simpsons_2.avi	FFmpeg	29.49	34.54	36.42	30.72
		MC*	29.55	34.90	36.74	30.82
	Simpsons_3.avi	FFmpeg	28.89	34.19	36.14	30.16
		MC*	28.95	34.58	36.48	30.25
	Simpsons_4.avi	FFmpeg	29.47	34.93	37.00	30.76
		MC*	29.53	35.24	37.30	30.85
	Simpsons_5.avi	FFmpeg	28.98	34.28	36.31	30.25
		MC*	29.05	34.69	36.63	30.35
3	Tonight_1.avi	FFmpeg	32.82	38.14	38.31	33.99
		MC*	32.78	38.14	38.32	33.96
	Tonight_2.avi	FFmpeg	33.00	38.70	38.97	34.23
		MC*	32.94	38.66	38.98	34.18
	Tonight_3.avi	FFmpeg	33.64	39.11	39.83	34.86
		MC*	33.05	38.27	38.76	34.23
	Tonight_4.avi	FFmpeg	33.33	38.21	38.64	34.47
		MC*	33.27	38.19	38.63	34.41
	Tonight_5.avi	FFmpeg	33.11	37.76	38.42	34.23
		MC*	33.06	37.76	38.47	34.19
4	Soccer_1.avi	FFmpeg	30.03	37.07	38.09	31.43
		MC*	30.05	37.38	38.33	31.46
	Soccer_2.avi	FFmpeg	30.15	36.71	38.64	31.53
		MC*	30.21	36.86	38.74	31.60
	Soccer_3.avi	FFmpeg	29.75	36.61	37.92	31.14
		MC*	29.82	36.75	38.06	31.21
	Soccer_4.avi	FFmpeg	31.03	37.24	39.16	32.03

Table 27: (continued)

Prof	Video	Encoder	Y	U	V	PSNR
4	Soccer_4.avi	MC*	31.05	37.11	39.03	31.96
	Soccer_5.avi	FFmpeg	34.31	38.90	40.02	35.45
		MC*	34.40	39.02	40.03	35.53
5	Dsp_Hsw_1.avi	FFmpeg	37.53	44.67	44.28	34.67
		MC*	N/A	N/A	N/A	N/A
	Dsp_Hsw_2.avi	FFmpeg	37.29	44.65	43.80	38.63
		MC*	N/A	N/A	N/A	N/A
	Dsp_Hsw_3.avi	FFmpeg	36.62	45.57	45.20	38.11
		MC*	N/A	N/A	N/A	N/A
	Dsp_Hsw_4.avi	FFmpeg	36.51	43.62	43.62	37.87
		MC*	N/A	N/A	N/A	N/A
	Dsp_Hsw_5.avi	FFmpeg	37.68	45.00	45.75	39.08
		MC*	N/A	N/A	N/A	N/A

Table 28: Comparison of Load Time between FFmpeg FAS Fast and Slow

Video Profile	Video	Hardware Profile	FAS Fast Time (Sec)	FAS Slow Time (Sec)
1	Fuji_1.mpg	Inslab	0	42
		Bigvision	0	38
	Fuji_3.mpg	Inslab	0	86
		Bigvision	0	64
	Fuji_9.mpg	Inslab	0	155
		Bigvision	0	144
	Fuji_36.mpg	Inslab	0	61
		Bigvision	0	59
	Fuji_53.mpg	Inslab	0	109
		Bigvision	0	108
2	Simpsons_1.avi	Inslab	0	44
		Bigvision	0	34
	Simpsons_2.avi	Inslab	0	26
		Bigvision	0	21
	Simpsons_3.avi	Inslab	0	26

Table 28: (continued)

Video Profile	Video	Hardware Profile	FAS Fast Time (Sec)	FAS Slow Time (Sec)
2	Simpsons_2.avi	Bigvision	0	21
	Simpsons_3.avi	Inslab	0	26
		Bigvision	0	27
	Simpsons_4.avi	Inslab	0	27
		Bigvision	0	24
	Simpsons_5.avi	Inslab	0	26
		Bigvision	0	22
3	Tonight_1.avi	Inslab	0	65
		Bigvision	0	62
	Tonight_2.avi	Inslab	0	66
		Bigvision	0	55
	Tonight_3.avi	Inslab	0	65
		Bigvision	0	59
	Tonight_4.avi	Inslab	0	64
		Bigvision	0	68
	Tonight_5.avi	Inslab	0	65
		Bigvision	0	60
4	Soccer_1.avi	Inslab	0	91
		Bigvision	0	86
	Soccer_2.avi	Inslab	0	94
		Bigvision	0	87
	Soccer_3.avi	Inslab	0	52
		Bigvision	0	52
	Soccer_4.avi	Inslab	0	50
		Bigvision	0	48
	Soccer_5.avi	Inslab	0	36
		Bigvision	0	33
5	Dsp_Hsw_1.avi	Inslab	0	64
		Bigvision	0	43
	Dsp_Hsw_2.avi	Inslab	0	62
		Bigvision	0	53
	Dsp_Hsw_3.avi	Inslab	0	61

Table 28: (continued)

Video Profile	Video	Hardware Profile	FAS Fast Time (Sec)	FAS Slow Time (Sec)
5	Dsp_Hsw_3.avi	Bigvision	0	59
	Dsp_Hsw_4.avi	Inslab	0	64
		Bigvision	0	61
	Dsp_Hsw_5.avi	Inslab	0	64
		Bigvision	0	48

Table 29: Comparison of Performance between FFmpeg and MC* Encoder

Prof	Video	Hardware Profile	CPU (%) Cycles (FF)	CPU (%) Cycles (MC*)	Mem. (Mb) (FF)	Mem. (Mb) (MC*)	Time (FF)	Time (MC*)
1	Fuji_1.mpg	Inslab	47.1	87.1	11.1	16.4	1,154	1,344
		Bigvision	9.1	14.5	10.4	14.1	1,014	850
	Fuji_3.mpg	Inslab	47.1	87.1	11.1	16.4	2,580	2,901
		Bigvision	9.1	14.5	10.4	14.1	2,411	2,249
	Fuji_9.mpg	Inslab	47.1	87.1	11.1	16.4	3,480	3,896
		Bigvision	9.1	14.5	10.4	14.1	2,973	2,601
	Fuji_36.mpg	Inslab	47.1	87.1	11.1	16.4	1,346	1,664
		Bigvision	9.1	14.5	10.4	14.1	1,301	1,214
	Fuji_53.mpg	Inslab	47.1	87.1	11.1	16.4	2,349	2,612
		Bigvision	9.1	14.5	10.4	14.1	2,111	2,097
2	Simpsons_1.avi	Inslab	49	86	10.1	12.4	1,026	1,263
		Bigvision	9.1	17	9.3	11.7	914	910
	Simpsons_2.avi	Inslab	49	86	10.1	12.4	948	1,206
		Bigvision	9.1	17	9.3	11.7	910	923
	Simpsons_3.avi	Inslab	49	86	10.1	12.4	940	1,301
		Bigvision	9.1	17	9.3	11.7	921	911
	Simpsons_4.avi	Inslab	49	86	10.1	12.4	1,021	1,234
		Bigvision	9.1	17	9.3	11.7	906	904
	Simpsons_5.avi	Inslab	49	86	10.1	12.4	1,045	1,221
		Bigvision	9.1	17	9.3	11.7	917	907

Table 29: (continued)

Prof	Video	Hardware Profile	CPU (%) Cycles (FF)	CPU (%) Cycles (MC*)	Mem. (Mb) (FF)	Mem. (Mb) (MC*)	Time (FF)	Time (MC*)
3	Tonight_1.avi	Inslab	47.6	88	10.1	12.4	2,020	2,896
		Bigvision	10.1	12.3	8.6	11	2,001	1,704
	Tonight_2.avi	Inslab	47.6	88	10.1	12.4	2,061	2,611
		Bigvision	10.1	12.3	8.6	11	2,049	1,741
	Tonight_3.avi	Inslab	47.6	88	10.1	12.4	2,044	2,467
		Bigvision	10.1	12.3	8.6	11	2,941	1,801
	Tonight_4.avi	Inslab	47.6	88	10.1	12.4	2,110	2,881
		Bigvision	10.1	12.3	8.6	11	2,060	1,754
	Tonight_5.avi	Inslab	47.6	88	10.1	12.4	2,344	2,924
		Bigvision	10.1	12.3	8.6	11	2,040	1,796
4	Soccer_1.avi	Inslab	49	88	10.2	14.1	2,252	3,950
		Bigvision	11.9	12.4	9.8	13.6	2,431	2,431
	Soccer_2.avi	Inslab	49	88	10.2	14.1	2,756	2,809
		Bigvision	11.9	12.4	9.8	13.6	2,418	2,400
	Soccer_3.avi	Inslab	49	88	10.2	14.1	1,654	1,650
		Bigvision	11.9	12.4	9.8	13.6	1,472	1,201
	Soccer_4.avi	Inslab	49	88	10.2	14.1	1,891	2,114
		Bigvision	11.9	12.4	9.8	13.6	1,601	1,493
	Soccer_5.avi	Inslab	49	88	10.2	14.1	1,147	1,316
		Bigvision	11.9	12.4	9.8	13.6	1,097	965
5	Dsp_Hsw_1.avi	Inslab	27.4	N/A	17.5	N/A	2,768	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,675	N/A
	Dsp_Hsw_2.avi	Inslab	27.4	N/A	17.5	N/A	3,181	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,987	N/A
	Dsp_Hsw_3.avi	Inslab	27.4	N/A	17.5	N/A	2,169	N/A
		Bigvision	12.7	N/A	17.4	N/A	2,042	N/A
	Dsp_Hsw_4.avi	Inslab	27.4	N/A	17.5	N/A	3,300	N/A
		Bigvision	12.7	N/A	17.4	N/A	3,124	N/A
	Dsp_Hsw_5.avi	Inslab	27.4	N/A	17.5	N/A	2,220	N/A
		Bigvision	12.7	N/A	17.4	N/A	1,995	N/A

Table 30: FFmpeg Motion Vector Extraction Library Performance

Video Profile	Video	Hardware Profile	Time (Sec)
1	Fuji_3.mpg	Inslab	24
		Bigvision	15
	Fuji_9.mpg	Inslab	36
		Bigvision	30
	Fuji_36.mpg	Inslab	123
		Bigvision	109
	Fuji_53.mpg	Inslab	98
		Bigvision	76
2	Simpsons_1.avi	Inslab	86
		Bigvision	71
	Simpsons_2.avi	Inslab	92
		Bigvision	73
	Simpsons_3.avi	Inslab	85
		Bigvision	71
	Simpsons_4.avi	Inslab	88
		Bigvision	72
3	Tonight_1.avi	Inslab	69
		Bigvision	62
	Tonight_2.avi	Inslab	66
		Bigvision	61
	Tonight_3.avi	Inslab	72
		Bigvision	63
	Tonight_4.avi	Inslab	72
		Bigvision	61
4	Soccer_1.avi	Inslab	107
		Bigvision	96
	Soccer_2.avi	Inslab	114
		Bigvision	102
	Soccer_3.avi	Inslab	106
		Bigvision	93
	Soccer_4.avi	Inslab	108
		Bigvision	95

BIBLIOGRAPHY

[1] ISO/IEC 11172-1 1993, Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s — Part 1: Systems.

[2] ISO/IEC 11172-2 1993, Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s — Part 2: Video.

[3] ISO/IEC 11172-3 1993, Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s — Part 3: Audio.

[4] Huffing Sun, Wilson Kwok, Max Chien, and C. H. John Ju (1997). MPEG Coding Performance Improvement by Jointly Optimizing Coding Mode Decisions and Rate Control. IEEE Transactions on Circuits and Systems for Video Technology, 7 (3), 449-458.

[5] Kevin McGuinness, Gordon Keenan, Tomasz Adamek, Noel O'Connor. A Framework and User Interface for Automatic Region Based Segmentation Algorithms.

[6] FFmpeg Multimedia System (10/2010): *http://* FFmpeg*.mplayerhq.hu/.*

[7] FFmpeg for Windows Help (09/2010). *http://* FFmpeg*.arrozcru.org/.*

[8] FFmpeg development discussions and patches (10/2010). *https://lists.mplayerhq.hu/mailman/listinfo/FFmpeg-devel*

[9] Martin Böhme. How to write a Video Player in less than 1000 lines (02/2009). *http://dranger.com/FFmpeg/tutorial01.html.*

[10] Martin Böhme. Using libavformat and libavcodec (06/2009). *http://www.inb.uni-luebeck.de/~boehme/using_libavcodec.html.*

[11] Doxygen: Source code documentation generator tool (07/2010). *http://www.stack.nl/~dimitri/doxygen/.*

[12] AMD CodeAnalyst Performance Analyzer (06/2010). *http://developer.amd.com/cpu/codeanalyst/Pages/default.aspx*

[13] FFmpeg demuxer how to (10/2010). *http://wiki.multimedia.cx/index.php?title*=FFmpeg_*demuxer_howto*

[14] GNU General Public License (09/2010). *http://www.gnu.org/licenses/gpl.html*

[15] Wascana Desktop Developer (09/2010). *http://wascana.sourceforge.net/*

[16] Alexis M. Tourapis, (2002) Enhanced predictive zonal search for single and multiple frame motion estimation