

Information Societies Technology (IST) Programme

Project N°: FP6-IST- 0028097

ASTRALS



Deliverable 4.4.1

Client for adaptive H.264/SVC reception

Author(s): K. Grüneberg (HHI),
L. Celetto (STMicroelectronics),
J. Ch. How (ProVision)

Status -Version: draft version 0.5

Date: 10 October 2007

Distribution - Confidentiality: public after release

Code: ASTRALS_D4.4.1_HHI_FF_20071010.doc

Abstract:

This document describes deliverable D.4.4.2, the client for adaptive H.264/SVC reception, which is a software package.



Disclaimer

This document contains material, which is the copyright of certain ASTRALS contractors, and may not be reproduced or copied without permission. All ASTRALS consortium partners have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.

The ASTRALS Consortium consists of the following companies:

No	Participant name	Participant short name	Country	Country
1	Algosystems SA	Algosystems	Co-ordinator	Greece
2	ST Microelectronics	ST	Contractor	Italy
3	Thomson	Thomson	Contractor	France
4	Mitsubishi Electric	Mitsubishi	Contractor	UK
5	Telefonica I + D	TID	Contractor	Spain
6	ProVision Communication Technologies	ProVision	Contractor	UK
7	Fraunhofer / Heinrich Hertz Institute	HHI	Contractor	Germany
8	Institute for Infocomm Research	I ² R	Contractor	Singapore

The information in this document is provided “as is” and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.



This page has been intentionally left blank



Table of contents

Disclaimer	2
Table of contents	4
Abbreviations	5
Executive Summary	6
1. Client Implementation	7
1.1. <i>H.264/SVC network protocols</i>	7
1.1.1. FEC Engine.....	7
1.1.2. Layered Packet Re-assembling.....	8
1.2. <i>H.264/SVC Decoding</i>	9
2. Error Resilient H.264 Decoder	13
2.1. <i>H.264 Decoder Specifications</i>	13
2.2. <i>Software Architecture</i>	14
2.3. <i>Decoder Implementation</i>	15
2.3.1. Decoder Plug-in Parameters	15
2.4. <i>Decoder Structure and Interfaces</i>	15
2.4.1. Library Initialisation	16
2.4.2. Error Concealment.....	17
2.4.3. NAL Unit Parsing.....	17
2.4.4. Decoding of NAL Units.....	18
2.4.5. Data Access	18
2.5. <i>Decoder Performance</i>	19
2.6. <i>Advanced Error Concealment</i>	19
2.6.1. Temporal Error Concealment	20
2.6.2. Spatial Error Concealment.....	20
2.6.3. Mode Selection	20
2.6.4. Advanced Error Concealment Results.....	21
3. SVC Decoder	23
3.1. <i>Decoder Capabilities</i>	23
3.2. <i>Software Architecture</i>	24
3.2.1. SVC Decoder Plug-in for VLC.....	25
3.2.2. HHISVCLibW Wrapper Module.....	25
3.2.3. Decoder Core Library "svcdec"	27
3.3. <i>Profiling</i>	28
3.4. <i>Optimization</i>	29
4. Conclusion	32
5. References	33
Annex A Software Package	34



Abbreviations

AEC	<i>Advanced Error Concealment</i>
ASO	<i>Arbitrary Slice Ordering</i>
AU	<i>Access Unit (all coded media data belonging to one time instant)</i>
AVC	<i>Advanced Video Coding [1] , in this context used for the single layer encoding, in contrast to the Scalable Extensions of H.264/MPEG4-AVC</i>
BER	<i>Bit Error Rate</i>
CGS	<i>Coarse Grain Scalability</i>
CIF	<i>Common image format, resolution 352 x 288 pixels.</i>
4CIF	<i>4 times CIF format, resolution 704 x 576 pixels, equivalent to standard TV.</i>
FRExt	<i>Fidelity Range Extensions (amendment to H.264/MPEG4-AVC)</i>
IETF	<i>Internet Engineering Task Force</i>
JSVM	<i>Joint Scalable Video Model of the JVT</i>
JVT	<i>Joint Video Team of ITU and MPEG</i>
H.264	<i>Video coding standard according to ITU-T Rec. H.264 ISO/IEC 14496-10</i>
MB	<i>Macro Block</i>
MGS	<i>Medium Grain Scalability</i>
MPEG	<i>Motion Picture Experts Group</i>
NAL	<i>Network Abstraction Layer (part of H.264/MPEG4-AVC standard)</i>
QCIF	<i>Quarter CIF, resolution 176 x 144 pixels</i>
QP	<i>Quantizing Parameter</i>
RC	<i>Remote Control</i>
RFC	<i>Request For Comments (IETF standard)</i>
RTP	<i>Transport Protocol for Real-Time Applications (IETF RFC 3550)</i>
RTSP	<i>Real-Time Streaming Protocol [2]</i>
SDP	<i>Session Description Protocol (IETF RFC 2327)</i>
SVC	<i>Scalable Video Coding, here always referring to the Scalable Extensions of H.264/MPEG4-AVC (ITU-T Rec. H.264 ISO/IEC 14496-10, Amendment 3)</i>
UDP	<i>User Datagram Protocol (IETF RFC 768)</i>
VCL	<i>Video Coding Layer</i>
VLC	<i>VideoLAN Client (Open Source media player with many additional functions)</i>
VoD	<i>Video on Demand</i>
WP	<i>Work package according to the ASTRALS project work plan</i>



Executive Summary

ASTRALS (Audio-visual STReaming pLatform for domestic Leisure and Security) is focused on scalable, A/V encoding, transcoding, storage and distribution in existing households via streaming-optimised wireless links. ASTRALS motivation is to implement scalable solutions, which will enable a new beam of innovative A/V products and services including personalised, network-aware video adaptation and distribution in multiple heterogeneous terminals (from low-cost PDAs to high-end home cinemas) and intelligent surveillance, utilising a state of the art in-home network.

The center of ASTRALS A/V network architecture is the Residential Gateway (RG). Today, the wide majority of RGs are designed as multi-interface routers with low-processing capabilities. ASTRALS RG is to be enhanced with A/V and broadband wireless optimised extensions. Without excluding legacy network interfaces, ASTRALS will specify and prototype a streaming-optimised, IEEE 802.16 based, wireless platform as the major in-home distribution medium. In parallel, the RG will provide enhancements for A/V scalable/personalised encoding, transcoding, storage and distribution based on user/services profiles, content and network conditions. H.264 Advanced Video Coding (AVC) and Scalable Video Coding (SVC), A/V transcoding/trans-rating, and erosion-featured storage derived from personalised preferences, will be provided on a specialized fully programmable A/V platform. Moreover, content media adaptation and efficient transmission based on feedback from the wireless MAC and network layer will be implemented and validated.

This document describes the software package (ASTRALS Deliverable 4.4.2) which implements both the H.264/MPEG4-AVC and the SVC decoder client. The client implementation is based on the VideoLAN Client (VLC) open source project in combination with proprietary libraries developed within Task 4.4 of WP4.

This document is organized as follows. The first chapter describes the transport layer plug-in for VLC which is responsible for the session control through RTSP and for data packet transmission through RTP/UDP. The RTSP plug-in provided by ASTRALS is able to open several RTP sessions in parallel in order to support a Layered Multicast approach which is a completely new feature. Different clients can connect to a different number of sessions, and by this means a terminal can choose the operation point at which the scalable video stream is decoded. The interface between VLC and its plug-ins is described at the end of the first chapter.

The next chapter is dedicated to the error resilient H.264/MPEG-4 AVC decoder. It specifies the available features, describes the internal architecture and lists the parameters that can be chosen dynamically. Additionally, software interface and its use are described in detail. Finally, results are shown both regarding decoding speed and the performance of the advanced error concealment.

The third chapter shows the architecture of the SVC decoder plug-in and the features supported by the SVC decoder library. The SVC decoder is initialized by the Sequence Parameter Sets (SPS) and Picture Parameter Sets (PPS) which are transmitted through RTSP. If the scalable video stream is partially received, the highest layer is decoded. Through the VLC client interface, the number of layers to be decoded can be restricted, and the data rate consumed by the decoder can be monitored.



1. Client Implementation

This chapter describes the client implementation based on the open source VLC project, using proprietary plug-ins for the H.264/MPEG4-AVC and SVC video decoders.

1.1. H.264/SVC network protocols

The VLC provides a framework to receive the bit stream from the network and provide it to the appropriate decoder.

For unicast sessions, the session set-up and control is obtained through the RTSP protocol. The RTSP implementation supports the receiving of the *Sequence Parameter Sets* (SPS) and *Picture Parameter Sets* (PPS) through this reliable channel.

For multicast sessions, the session set-up is obtained by means of the *Session Announcement Protocol* (SAP) that delivers the vital session information through the SDP message. The SAP receiving module will decode and provide the SPS and PPS information to the decoder.

The RTP-protocol plug-in receives the bit stream and delivers it to the decoder. The plug-in implements the decoding of the RTP protocol and the specific fields reserved for the H.264 implementation and delivers to the decoder the received bit stream in chunks that can be properly decoded.

The original implementation of the RTP and RTSP protocols at the receiver was based on the live.com libraries. The plug-in was calling the library to handle the session and a function call-back `StreamRead()` was handling the delivery of the received multimedia information to the appropriate decoder. We modified the original live.com RTSP implementation in order to support the multicast streaming and implemented a RTP protocol for the receiver in order to handle the layered multicast streaming and FEC decoding inside the library we provided. The library internals will buffer some packets and exploit the FEC information to recover the missing packets. Several RTP session may be registered in a single RTP handler, so that layered multicast packet reordering will be performed inside the library and the reordered packets will be delivered to the decoder as chunks of bit stream pertaining to a single frame.

1.1.1. FEC Engine

The FEC engine is implemented inside the STM library. The FEC encoding engine duplicates the incoming RTP packets in an internal buffer and when a certain threshold is reached, it generates the FEC packets and frees the buffered memory. The generated FEC packet resembles for many aspects the packets generated with the recommendation RFC2733 [3].

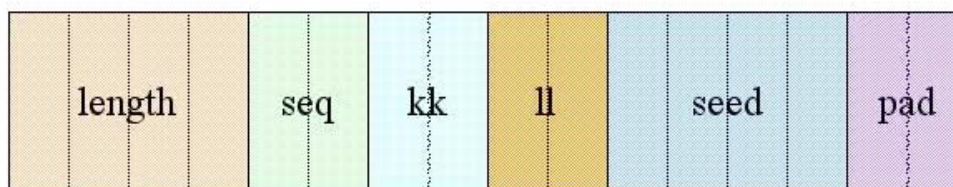


Figure 1: FEC packet header (each box is one byte)

Figure 1 shows the details of the header of the FEC packet. The *length* field contains the length of the information in the packet, the *seq* field the minimum sequence number, the *seed* field the seed for the FEC engine, followed by two extra padding bytes that are empty. The *kk* and *ll* fields show the amount of the RTP packets in the buffer and how many FEC packets to expect respectively. As these



two parameters may vary in general to match the changing network conditions, they are inserted in the FEC packet header to help the FEC decoder task.

The FEC decoder buffers the packet at the receiver side. The FEC decoder runs as a separate thread in the library, to avoid buffer overflows in the UDP kernel queue due to long decoding times in some cases. At the receiver, the packets are buffered in a queue. They cannot be delivered to the application until the related FEC packets are received and – should any packet loss occur – the packet recovery process is performed. This causes a delay associated to buffering, when considering the end-to-end delay.

Outside the FEC decoding engine, the packet delivery preserves a constant pace, making this process transparent to the following blocks (the layered multicast and the decoder). The packets pertaining to a certain decoded FEC array, are not delivered in a burst, but as soon as another packet enters the queue to fill in the next FEC matrix to be decoded. For any further information about the FEC implementation, please refer to [4].

1.1.2. Layered Packet Re-assembling

In layered coding, the multimedia information relative to one multimedia stream can be delivered using multiple RTP streaming sessions as shown in Figure 2. Dependencies between different media flows are signalled in the SDP according to a new approach currently discussed in IETF[5]. Due to the characteristics of the networks and unreliability of the UDP transmission, RTP packets coming from different paths may either arrive out-of-order or not arrive at all at the receiver point. H.264 RTP specifications[6] provide means for a proper reordering, using the *Decoding Order Numbers* (DONs).

DONs are assigned to NAL units pertaining to a multimedia stream, when the data is encapsulated in an RTP packet. DON values allow NAL unit reordering when decoding a multimedia stream, as it allows reordering NAL units coming in parallel from different sources into a unique stream.

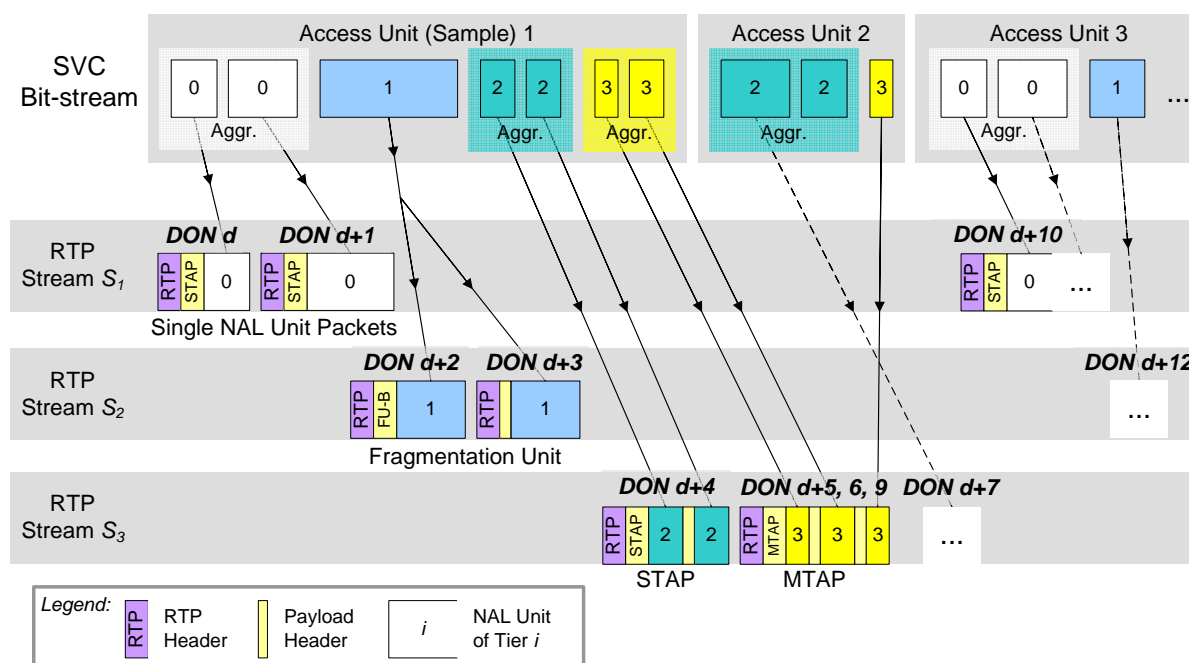


Figure 2: Example of SVC transmission as Layered Multicast

Reordering of the NAL units is quite a complex task. Some buffering is required, as packets coming from different RTP sessions do not arrive at the same instant. Moreover, FEC buffering is increasing



the variance of the overall timings for packet arrivals. For example, buffering a certain number of RTP packets at each layer may result in different buffering time for each layer, as for instance the base layer may contain several large NAL units which are split in *Fragmentation Units* (FU), while at upper layers the *Single Time Aggregation Packet* (STAP) mechanism may assemble a number of NAL units in a single RTP packet.

A proper buffering may solve the problem. Packets coming from each RTP layer are placed in a circular queue and indexed through their DON number. After some buffering time, the received packets are flowed to the application. Should any packet arrive later, it will be discarded, as it will not be used any more by the application. A proper trade-off between longer buffering time to ensure that all the packets are reordered correctly and a minimum end-to-end delay in the application should be sought.

1.2. H.264/SVC Decoding

The VLC provides standard means of creating plug-ins for decoders. The basic rule is that a chunk of bit stream is provided to the decoder. When the decoder is ready with a decoded picture it returns a non-NULL pointer containing the data of the decoded image. Chunks of bit stream are properly formatted to ease the decoding task. This may be done by means external to the VLC implementation (for instance, the File Format or the received RTP packets contain information formatted at image level) or internally by a specific plug-in, called packetizer.

The chunk of bit stream is contained in the `block_t` structure (see Table 1 for details). This structure contains a chain of packets to be decoded and timing fields. The `i_dts` field contains the time for decoding the current packet of bit stream. The `i_pts` field contains the time for presenting the decoded image to the display. The two time instants may differ when bi-directionally predicted frames are present as the reference image for backward prediction should be available prior to decoding the current B image.

<i>Type</i>	<i>Field name</i>	<i>Description</i>
<code>block_t</code>	<code>p_next</code>	Pointer to the next block structure in a double-linked list
<code>block_t</code>	<code>p_prev</code>	Pointer to the previous block structure in a double-linked list
<code>uint32_t</code>	<code>i_flags</code>	Attributes of the current block, regarding its contents (I,P,B frame bit stream, etc.)
<code>mtime_t</code>	<code>i_pts</code>	The Program TimeStamp for audio/video synchronization
<code>mtime_t</code>	<code>i_dts</code>	The Display TimeStamp for audio/video synchronization, to be used in alternative to the previous field
<code>int</code>	<code>i_samples</code>	The audio samples contained in current bit stream and available when data is decoded
<code>int</code>	<code>i_rate</code>	
<code>int</code>	<code>i_buffer</code>	The buffer size in bytes
<code>uint8_t *</code>	<code>p_buffer</code>	The pointer to the buffer containing the bit stream and holding <code>i_bytes</code> data.
<code>void</code>	<code>(*pf_release)</code>	The pointer to the function that de-allocates the previous buffer
<code>vlc_object_t *</code>	<code>p_manager</code>	The VLC object manages the blocks reducing memory allocations/de-allocations



<i>Type</i>	<i>Field name</i>	<i>Description</i>
block_sys_t *	p_sys	Contains proprietary data of the block_t structure. Not used, and not to be used according to the comments in the code.

Table 1: The block_t structure

The decoder handler structure contains information about the decoder status. Pointer functions allow defining ad-hoc functions for allocating and free memory buffers. The buffers may be locked to be used as a reference for future image decoding, avoiding the releasing of the memory for other parts of the VLC.

<i>Type</i>	<i>Field Name</i>	<i>Description</i>
module_t *	p_module	Contains some basic information about module capabilities, to be used to choose the right functionality dynamically
decoder_sys_t *	p_sys	Contains the internal status of the decoder (its settings and variables).
es_format_t *	fmt_in	Contains the incoming data format and may be used to exchange some preliminary information on the following modules in the chain by means of the i_extra and p_extra fields.
es_format_t *	fmt_out	Contains the format of the outgoing data for reference to the following modules in the chain. This means that fmt_out will become the fmt_in of the following module in the chain.
picture_t *	(*pf_decode_video)	This function pointer contains the address of the called function and is used to decode one frame. The resulting bit stream will be delivered by means of the returning block_t * pointer.
aout_buffer_t *	(*pf_decode_audio)	This function pointer contains the address of the called function and is used to encode one audio data chunk.
subpicture_t *	(*pf_decode_sub)	This function pointer is used to decode the sub-pictures .
aout_buffer_t *	(*pf_aout_buffer_new)	This function returns a buffer where to store the decoded data for the audio decoder
void	(*pf_aout_buffer_del)	This function frees the audio buffer once its contents are used
picture_t *	(*pf_vout_buffer_new)	This function returns a buffer where to store the decoded data for the video decoder.
void	(*pf_vout_buffer_del)	This function frees the video buffer once its contents are used
void	(*pf_picture_link)	This function links the picture buffer in order to avoid it is freed. This buffer then can be used as a reference picture in the future



<i>Type</i>	<i>Field Name</i>	<i>Description</i>
void	(*pf_picture_unlink)	This function unlinks the previously linked picture buffer
subpicture_t *	(*pf_spu_buffer_new)	This function returns a buffer where to store the decoded data for the subpicture decoder.
void	(*pf_spu_buffer_del)	This function frees the spu buffer once its contents are used
decoder_owner_sys_t	p_owner	Contains the pointer to the owner of the current decoder

Table 2: The decoder_t structure

The decoding process starts when the VLC calls the decoding function, registered in the pf_decode_video() field of the decoder handler. A complying function call can be

picture_t *DecodeAU(decoder_t *p_dec, block_t **pp_block)

where the fields of decoder_t and block_t * are defined in Table 1 and Table 2. The decoding function will return a NULL pointer, if there is not any output picture to be displayed¹, a valid pointer otherwise. The picture_t structure contains the resulting decoded image.

<i>Type</i>	<i>Field Name</i>	<i>Description</i>
video_frame_format_t	Format	The properties of the current picture
uint8_t *	p_data	Not to be touched, contains the address to the memory location of the frame-buffer
void*	p_data_orig	Pointer before memory alignment, preserved for proper memory de-allocation
plane_t	p[]	Contains the addresses to the different planes of the image
int	i_planes	Contains the number of planes for current frame
int	i_status	Picture status signals (internally) if the picture is ready to be displayed, or reserved, etc.
int	i_type	Contains information if there is direct rendering etc.
vlc_bool_t	b_slow	Contains information if the picture is in a slow memory
int	i_matrix_coefficients	Contains the YUV encoding type for current frame
int	i_refcount	Contains references for current picture (not to be accessed directly, but through APIs)
mtime_t	Date	Display date for current image (i.e. the temporal instant when the image must be displayed)

¹ The decoder may buffer some images in case of bi-directionally predicted images. Hence, even if one picture was decoded, the decoder may decide to output its contents during the following calls.



<i>Type</i>	<i>Field Name</i>	<i>Description</i>
vlc_bool_t	b_force	Force displaying current image, even if it is in the past (i.e. the decoder is not fast enough to display in proper time). Using this flag may broke audio/video synchronization.
vlc_bool_t	b_progressive	Signals if the image is a progressive or interlaced image
unsigned int	i_nb_fields	Number of displayed fields for current image
vlc_bool_t	b_top_field_first	Signals if the top field or the bottom one is the first to be decoded
picture_heap_t *	p_heap	The heap this image is attached to. All the pictures are stored in a heap in order to maintain them accessible and allocated when they are already displayed.
int	(*pf_lock)	Allows picture buffer locking before modifying memory areas
int	(*pf_unlock)	Unlocks previously allocated pictures
picture_sys_t *	p_sys	Private data of the current video frame
void	(*pf_release)	Releases the memory areas for the current frame
struct picture_t *	p_next	Pointer that references the next buffer in the FIFO queue of pictures

Table 3: The picture_t structure

When decoding H.264 bit streams, the most important information regarding decoding set-up is conveyed by means of the Sequence Parameters Sets (SPS) and Picture Parameter Sets (PPS). RFC specifications recommend sending this piece of information through a reliable channel. A possibility is using a specific parameter in the SDP message received by means of RTSP protocol. The HHI H.264/SVC decoder can either use the SPS and PPS given during the initialisation phase or wait any in-stream SPS and PPS, before starting decoding the bit stream. Hence, we can support both modalities – having the SPS, PPS received through a reliable channel during the initialisation phase or during the streaming.

While the latter option is straightforward, obtaining the SPS, PPS from the RTSP protocol requires some data passing between different modules. For this purpose, we used the `i_extra` and `p_extra` data fields in `es_format_t` structure to move SPS and PPS information from the RTSP plug-in to the decoder plug-in. In the initiation phase, the decoder checks whether this fields are non-NULL and it delivers the information to the decoder.



2. Error Resilient H.264 Decoder

The error-resilient H.264/AVC decoder that was implemented by ProVision within this work package of ASTRALS is described in this section. The software architecture of the decoder was designed to provide reliable error detection and recovery when decoding bitstreams corrupted with errors, and exhibits a high degree of robustness to both bit-errors and packet loss. The decoder was implemented for the PC platform running Linux (Ubuntu 2.6.10) and is provided both as a stand-alone library and as a plug-in for the VideoLAN (VLC) open source software.

In a real operating environment, such as the wireless platform implemented and used within ASTRALS, the bitstream received by the H.264 decoder cannot be guaranteed to be entirely error free or without any loss. All H.264 decoders must be able to decode any standard compliant error-free bitstream, and produce identical decoded frames. However, if a bitstream is corrupted with errors, the decoded frames can vary widely between decoder implementations. In the worst case scenario, an error can result in memory and data corruption and cause the decoder to crash. The architecture of the ProVision H.264 decoder was designed from the start to provide optimum decoded video quality, and maximise robustness to errors. The decoder is able to tolerate both bit-errors and packet loss, and will automatically resynchronise to the next available NAL unit. To further improve the decoded video quality in the presence of errors, an advanced error concealment algorithm has also been integrated into the decoder, such that portions of the decoded picture that are missing or in error are concealed using available information from neighbouring correctly received parts of the picture..

In the following sections, the capabilities of the H.264 decoder and the supported coding options are given, followed by a description of the decoder architecture. A detailed description of the decoder library implementation and its interface is then given. The implementation of the library as a VLC plug-in, and its integration within the VLC software, is also described. The decoding speed of the H.264 decoder is then assessed. Finally, the advanced error concealment algorithm implemented here is described and its performance when decoding a corrupted bit stream is characterized.

2.1. H.264 Decoder Specifications

The H.264 decoder conforms to the baseline profile of the H.264 video coding standard. The following coding options supported are:

- I and P slices – frame coding only
- variable block sizes – 16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4
- ¼ pixel motion estimation/compensation
- context-adaptive variable length coding (CAVLC)
- de-blocking filter
- flexible macroblock ordering (FMO) - Dispersed and Interleaved modes
- multiple reference frames (up to maximum of 16)
- Annex B bit stream

The decoder can handle picture sizes up to 4CIF resolution (720 by 576 pixels) in YCbCr format with 4:2:0 chrominance sub-sampling, and bit-rates in excess of 10 Mbps.

The decoder was designed to maximise decoded image quality in error-prone environments and supports the following robust features:

- can decode bit streams corrupted with both bit-errors and packet loss
- error-detection based on H.264 bit stream syntax
- error-recovery through resynchronisation to the next NAL unit
- identification of missing or corrupted MBs within each decoded picture
- concealment of missing or corrupted MBs using an advanced error concealment algorithm



2.2. Software Architecture

The H.264 video decoder software architecture was designed for fast and efficient video decompression, and for maximum robustness in the decoding of incomplete or corrupted bitstreams. The software architecture enables error detection based on invalid syntax and coding parameters, and provides graceful error recovery and re-synchronisation. This enables the decoder to maximise decoded video quality even when the bit stream is corrupted with errors. The architecture of the decoder is shown in Figure 3.

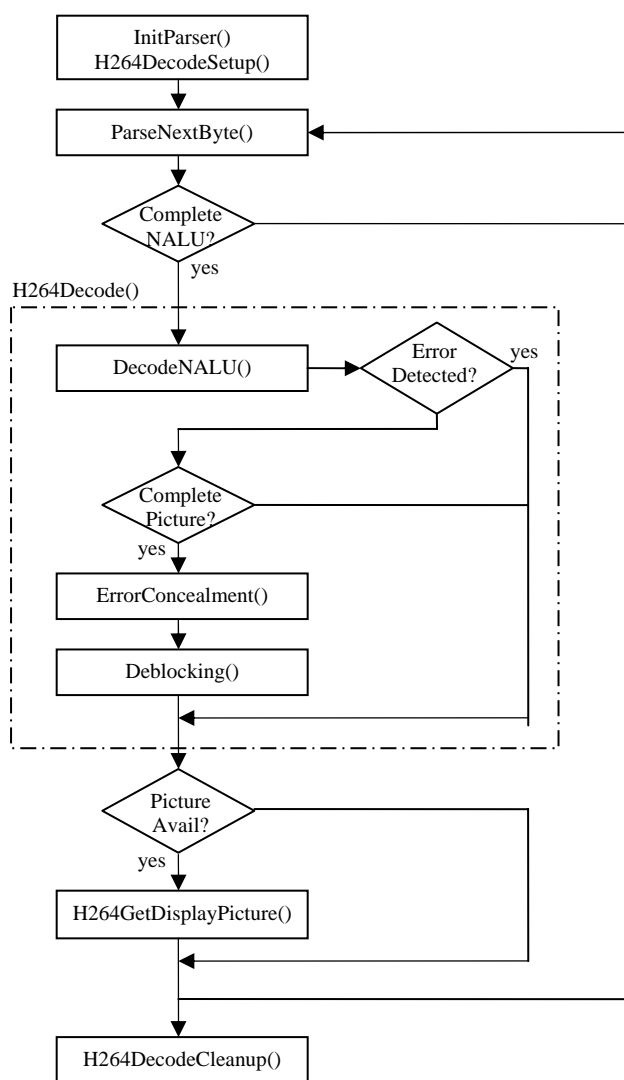


Figure 3: H.264 Decoder Architecture.

The decoder operates on a NAL unit basis, i.e. the received bit stream is first parsed into individual NAL units, and each NAL unit is then passed to the NALU decoding module for decoding. The NALU decoding module operates as a state-machine, where the internal state represents a specific stage in the decoding process. The internal state of decoder is updated as each NAL unit is decoded and the decoding module keeps its internal state between calls.

The decoding of a NAL unit by the NALU decoding module is always sequential, i.e. slice level information, e.g. SPS, PPS and slice headers, are decoded first, followed by MB level information,



e.g. MB type, motion vectors, CBP and quant, followed by the decoding of VLC level information, e.g. transform coefficients.

The H.264 bit stream syntax is checked at every stage by the NALU decoding module. Whenever an invalid syntax is detected, it is assumed that the bit stream has been corrupted, and resynchronisation is performed at the slice level. This enables reliable detection of bit stream corruption due to bit errors or packet losses.

Error concealment is an integral part of the decoder, and is performed on the missing or corrupted MBs in a picture before the de-blocking filter is applied. The decoder maintains the status of every MB in a picture in a MB status map. Before decoding of a picture begins, all MBs are marked as corrupted, and the status of each decoded MB is updated as decoding progresses. When the end of a decoded picture is reached, i.e. when the start of a new picture is detected, all the MBs in the current picture that are still marked as corrupted are assumed to be either missing or lost. If a NAL unit is lost or missing, all the MBs in that NALU will still be marked as corrupted. The missing MBs can then be concealed using a suitable error concealment algorithm prior to the decoding of the next picture.

2.3. Decoder Implementation

The H.264 decoder has been implemented as a shared library under Linux.

An H.264 decoder plug-in for use with the VideoLAN streaming software was also implemented as part of this work. The pvdec decoder plug-in uses the H.264 decoder shared library and the source code for the pvdec decoder plug-in is given in the file pvdec.c.

2.3.1. Decoder Plug-in Parameters

The following parameters to the pvdec plug-in can be specified:

sout-pvdec-ecmethod	Error concealment mode that is used in the decoder for concealment of corrupted or missing MBs 0 = replace with green 1 = previous frame 2 = advanced error concealment
sout-pvdec-showerrormbs	If set, corrupted or missing MBs that have been detected and concealed with be highlighted with a black border in the decoded picture

2.4. Decoder Structure and Interfaces

The H.264 decoder library consists of the following structures and functions which are defined in h264dec.h, and its associated header files:

- Data structures
 - JPC_H264Decoder
 - NalUnit
 - NalParser



- Initialisation functions
 - void InitNalUnit(NalUnit * nal, unsigned char * buf, int bufsize)
 - int H264DecodeSetUp(JPC_H264Decoder *h264Dec)
 - int H264DecodeCleanUp(JPC_H264Decoder *h264Dec)
 - int H264DecodeSetErrorConcealment(JPC_H264Decoder *h264Dec, int concealment, int showErrorMBs);
- Decoding function
 - int H264Decode(JPC_H264Decoder *h264Dec, NalUnit *nal, int *message)
- Data Access
 - int H264GetDisplayPicture(JPC_H264Decoder *h264Dec, unsigned char **pDispPicture, int *width, int *height);
- NAL parsing utilities
 - void InitParser (NalParser * parser, NalUnit * nal);
 - int ParseNextByte (NalParser * parser, NalUnit * nal, unsigned char abyte);

The JPC_H264Decoder structure defines all the data elements required by the H.264 decoder. Each declaration of a JPC_H264Decoder variable represents an instance of the H.264 decoder. Several instances of the H.264 decoder can be used in a single application by declaring more than one variable of type JPC_H264Decoder.

The main H.264 decoding function is H264Decode(), which operates on integral NAL units. The bit stream must be parsed into NAL units prior to calling H264Decode(). The NAL unit parsing is done through the NalParser structure. The helper function ParseNextByte() is provided, which analyses the bitstream byte by byte, and whose return value indicates if a complete NAL unit is available. The NAL unit is stored in a NALUnit structure..

When a complete NAL unit has been parsed, the NALUnit variable is passed to H264Decode() as a parameter. H264Decode() decodes the NAL unit and returns. The return value of H264Decode() indicates if a complete picture has been decoded. If a complete picture has been decoded, H264GetDisplayPicture() is called to retrieve a pointer to the decoded picture.

Nal unit parsing is done outside of H264Decode() for maximum flexibility, enabling the library to be used for any decoding scenario. If the NAL parsing was done inside H264Decode(), the application would have to guarantee that any packet of data passed to H264Decode() did not contain more than one complete picture, since the application can access a decoded picture only when H264Decode() returns. Having H264Decode() operate on NAL units facilitates error detection and error-concealment, as well as ensuring that the application remains responsive and operates in real-time.

2.4.1. Library Initialisation

At the beginning of the application, three variables of type JPC_H264Decoder, NalParser and NalUnit, respectively, must be declared.

```
NalUnit nalu;  
NalParser nalParser;  
JPC_H264Decoder H264Dec;
```



These variables must then be initialised before being used.

```
InitNalUnit(&nalu, streambuf, bufsize);  
InitParser (&nalParser, &nalu);  
H264DecodeSetUp(&H264Dec);
```

The parameters to `InitNalUnit()` are a pointer to the `NalUnit` to be initialised, an unsigned char buffer pointer and the size of the buffer. The buffer will be used to store the raw byte stream payload (RBSP) of the NAL units, and must be larger than the maximum NAL unit size.

The parameters to `InitParser()` are a pointer to the `NalParser` to be initialised, and a pointer to the `NalUnit` where the parsed bitstream will be stored. The `NalUnit` must have been initialised with `InitNalUnit()` already.

`H264DecodeSetUp()` initialises the `H264Dec` variable with all the default values, and allocates all the memory buffers required.

2.4.2. Error Concealment

The error concealment algorithm used to conceal missing or corrupted MBs can be selected using

```
int H264DecodeSetErrorConcealment(JPC_H264Decoder *h264Dec,  
int concealment, int showErrorMBs);
```

The selected concealment algorithm depends on the value of `concealment`.

<code>concealment = 0</code>	No concealment. Missing MBs are replaced with green.
<code>concealment = 1</code>	Previous frame concealment.
<code>concealment = 2</code>	Advanced error concealment EECMS concealment.

If `showErrorMBs` is set, the missing MB in a picture will be highlighted with a black border in the reconstructed picture.

`H264DecodeSetErrorConcealment()` returns `JPC_H264DEC_OK` if successful.

`H264DecodeSetErrorConcealment()` can be called at any time after `H264DecodeSetUp()` to set the concealment mode. The chosen concealment mode will be applied to any future pictures decoded by `H264Decode()` and obtained with `H264GetDisplayPicture()`.

2.4.3. NAL Unit Parsing

The NAL unit parser must be initialised with the function

```
void InitParser (NalParser * parser, NalUnit * nal);
```

The parameter `nal` must be a `NalUnit` that has already been initialised.

The parsing of an H.264 Annex B bitstream into NAL units is done using the function

```
int ParseNextByte (NalParser * parser, NalUnit * nal, unsigned char abyte);
```



ParseNextByte() is called sequentially for each byte of the bitstream, with the byte value contained in the parameter abyte. The return value depends on the status of the bitstream parsing. A positive or zero value indicates that a complete NAL unit is available in nal. A value of -1 indicates that no complete NAL unit is available so far. A negative value less than -1 indicates that an error condition in the NAL parsing has occurred. The NalUnit and NalParser should be initialised following the detection of an error condition by ParseNextByte().

2.4.4. Decoding of NAL Units

Once a complete NAL unit is available, as indicated by a positive return value of ParseNextByte(), the NAL unit is decoded using

```
int H264Decode(JPC_H264Decoder *h264Dec, NalUnit *nal, int *message)
```

- nal is the NalUnit variable that contains the NAL unit that was parsed from the bitstream.
- h264Dec is the particular instance of the H.264 decoder that is being used.
- message is reserved for future use and must be NULL.

The possible return values of H264Decode() and their meaning are:

JPC_H264DEC_OK	nal was decoded ok
JPC_H264DEC_CALLBACK	Decoding of nal has not been completed. H264Decode() should be called again
JPC_H264DEC_DISPLAY	nal was decoded ok and a complete picture is available to be displayed.
JPC_H264DEC_CALLBACK_AND_DISPLAY	A complete picture is available to be displayed, but decoding of nal has not been completed. H264Decode() should be called again.
JPC_H264DEC_ERROR	An error condition was detected while decoding nal.

The call-back procedure is required because the end of a picture may not always be detected until the start of the NAL unit belonging to the next picture. When that happens, the availability of the complete picture for display needs to be signalled before decoding of the next picture begins. The call-back procedure also facilitates error-recovery in the event of bit stream corruption.

2.4.5. Data Access

The availability of a complete decoded picture is indicated by the return value of H264Decode(). The decoded picture can be accessed before the next call to H264Decode() using

```
int H264GetDisplayPicture(JPC_H264Decoder *h264Dec,  
                        unsigned char **pDispPicture, int *width, int *height);
```

H264GetDisplayPicture() returns JPC_H264DEC_OK if successful. In case of success, pDispPicture will contain a pointer to the decoded picture in YUV 4:2:0 planar format with 8-bits per sample. The samples are ordered in raster-scan order starting from the top-left corner of the picture. The parameters width and height will contain the width and height of the luminance component of the picture in pixels. There is a one frame buffer in the decoder to take into account the



possibility of pictures being re-ordered in the decoder. Therefore, when decoding the very first frame of a sequence, no decoded picture is returned. After that, a decoded picture will be available every time a complete frame is decoded.

2.5. Decoder Performance

The H.264 decoder was benchmarked on a Toshiba laptop with Intel Core 2 CPU, T7200@2 GHz, 1 GB RAM, running Ubuntu 6.10, kernel 2.6.17.

The decoding speed was measured for 250 frames of the bus and foreman test sequences at CIF resolution at various bit-rates ranging from 256 to 2 Mbps. The results are shown in Table 4 below. The time required to decode each frame was measured using the `gettimeofday()` utility in Linux, and does not include time required for file access. The bit streams were generated using a baseline profile encoder with sub 16x16 block size, de-blocking filter, and quarter-pel motion estimation accuracy.

At 2 Mbps, the decoder can operate at more than four times faster than real-time. This is relatively fast, especially considering that the main objective of this work was to develop a robust H.264 decoder to be used within ASTRALS, and relatively less time was spent on speed optimisation. The speed of the decoder can be further improved by careful optimisation of the code to minimize memory usage, reduce code size, and making use of specific multimedia instructions available on most modern processors. However, the development of a fully optimised decoder is beyond the scope of this work.

Test Sequence	Decoding speed /fps
bus, 242 kbps	210.3
bus, 544 kbps	164.6
bus, 1023 kbps	134.1
bus, 2100 kbps	102.1
foreman, 242 kbps	220.6
foreman, 477 kbps	174.2
foreman, 990 kbps	133.9
foreman, 2102 kbps	100.8

Table 4: Decoder performance

2.6. Advanced Error Concealment

In a real operating environment, such as the wireless ASTRALS platform, the bit stream received by the H.264 decoder cannot be guaranteed to be entirely error free. When a bit stream is corrupted with errors, the decoder will decode and reconstruct as much of the picture as possible. Missing or corrupted areas of the picture are identified by the decoder. Error concealment is the process of estimating or replacing those areas such that the impairment due to the errors is minimized, and the reconstructed picture is visually acceptable.

In order to do so, typical concealment methods exploit the correlation that exists between a particular MB and its adjacent MBs in the same or previous frame. In temporal error concealment methods, the reference frame and motion vector of missing MBs is estimated, and the resulting motion



compensated MB is used to replace the missing MB. Methods that use spatial concealment use the spatially adjacent MBs in the same frame to estimate the missing MB.

The advanced error concealment method implemented in the decoder is capable of both spatial and temporal concealment, and uses a mode decision algorithm to determine the best mode on MB basis. This is described next. Additional information about the error concealment algorithm is given in [7].

2.6.1. Temporal Error Concealment

The temporal concealment method used begins by forming a list of suitable motion vector (MV) candidates for the MB to be concealed. For each candidate MV, an error measure known as the external boundary matching error (EBME) is calculated. The MV candidate with the lowest EBME is chosen and the motion compensated MB is used to conceal the missing MB.

The MV candidate list consists of the following 18 MVs:

- Zero MV
- MV of collocated MB in most recent previous decoded frame
- MVs of collocated 8-neighbour MBs in most recent previous decoded frame
- MVs of 8-neighbour MBs in current frame

The most recent previous decoded reference frame is used as the reference frame for each of the 18 candidate MVs.

The EBME for each candidate MV is the SAD between the two-pixel wide boundary of MBs adjacent to the missing MB in the current frame, and the collocated two-pixel wide boundary of MBs adjacent to the replacement MB in the reference frame.

$$EBME = \sum_{i=1}^2 \sum_{x=x_0}^{x_0+15} \left[\left| p_{x,y_0-1} - p_{x+v_x,y_0-i+v_y}^r \right| + \left| p_{x,y_0+15+i} - p_{x+v_x,y_0+15+i+v_y}^r \right| \right] \\ + \sum_{i=1}^2 \sum_{y=y_0}^{y_0+15} \left[\left| p_{x_0-i,y} - p_{x_0-i+v_x,y+v_y}^r \right| + \left| p_{x_0+15+i,y} - p_{x_0+15+i+v_x,y+v_y}^r \right| \right]$$

where (x_0, y_0) is the position of the top-left pixel of the missing MB, (v_x, v_y) is the candidate MV being evaluated, and $p_{x,y}$ and $p_{x,y}^r$ are the pixel values of the current and reference frames, respectively, at pixel position (x, y) .

2.6.2. Spatial Error Concealment

When spatial error concealment is chosen, the boundary pixels of adjacent MBs are used to replace the missing pixels with weighted averages of the boundary pixels. In particular, bi-linear interpolation is used.

In addition to bi-linear interpolation, a more complex algorithm based on edge detection and directional interpolation is proposed in [7]. However, this was not implemented here because of additional complexity and time constraints.

2.6.3. Mode Selection

Temporal concealment is generally more effective than spatial concealment as it exploits the correlation that exists between consecutive frames, and for typical sequences, temporal correlation between MBs is generally higher than spatial correlation. However, in specific cases such as scene changes and sequences with irregular motion or uncovered objects, temporal correlation may be very low. In those circumstances, spatial concealment is more effective.



An adaptive mode selection algorithm is used on a MB basis to decide whether to use spatial or temporal concealment. For each MB, temporal concealment as described in the previous section is first performed and the best candidate MV for concealment is determined. A temporal activity measure, Act_{temp} is then calculated. Act_{temp} is defined as the mean squared error between the eight-connected MBs surrounding the missing MB in the current frame, and the eight-connected MBs surrounding the replacement MB in the reference frame. This is compared with a spatial activity measure, Act_{spat} , defined as the variance of the eight-connected MBs surrounding the MB being concealed in the current frame.

$$Act_{spat} = E\left[\left(p_{x,y} - \mu\right)^2\right]$$

$$Act_{temp} = E\left[\left(p_{x,y} - p_{x+v_x, y+v_y}^r\right)^2\right]$$

for all pixel positions (x,y) belonging to the eight-connected MB surrounding the missing MB.

If $(Act_{temp} < Act_{spat})$ or $(Act_{spat} < 3)$ use temporal concealment

Else use spatial concealment

2.6.4. Advanced Error Concealment Results

The standard test sequences bus and foreman were encoded at approximately 500, 1000 and 2000 kbps. 250 frames at CIF resolution were used, and a fixed quantization parameter was used for all frames. Only I and P frames were used, with one I frame every 25 frames, and the maximum slice size was limited to 500 bytes.

Simulations with random bit-errors at various error rates were performed, and corrupted slices were discarded, i.e. the simulations assume that each slice is transmitted as individual packets, and the transmission channel suffers from packet erasures. For each bit stream being evaluated, a random error pattern was generated, and the resulting bit stream with the slice erasures was then decoded. This was repeated 100 times for each bit stream, and the PSNR of the decoded frames was averaged over the 100 runs. It is assumed that the sequence and picture parameter sets are always correct, e.g. they can be either transmitted out-of-band or pre-defined at the decoder.

Simulations results were obtained for the advanced error concealment (AEC) algorithm described in this document, and also for previous frame error concealment (PFC) where missing MBs are assumed to have zero motion vectors, and are concealed with the co-located MB from the most recent previous decoded reference frame. The results are shown in Figure 4.

The use of AEC greatly improves the decoded picture PSNR, even at very low error rates. At a BER of 5×10^{-7} , which is equivalent to a packet error rate (PER) of approximately 0.2% (with 500 bytes/packet), AEC provides a gain of up to 2.3 dB for bus at 2 Mbps compared to PFC. As the BER increases, the quality with PFC falls very sharply, especially at higher rates, such that at BERs above 5×10^{-6} , the decoded video quality is similarly bad for all rates, and the video is virtually unwatchable. By contrast, with AEC, the video quality degrades much more slowly, and can still provide acceptable video quality even at BER of 10^{-5} and above. Moreover, with AEC, a higher bit-rate video always results in better average PSNR than a lower bit-rate video, at all the BERs evaluated here.

It is generally accepted that objective video quality measures like PSNR may not always correlate well with subjective video quality measures. Informal subjective assessments of the decoded video sequences confirm that AEC provides a considerable improvement compared to PFC.

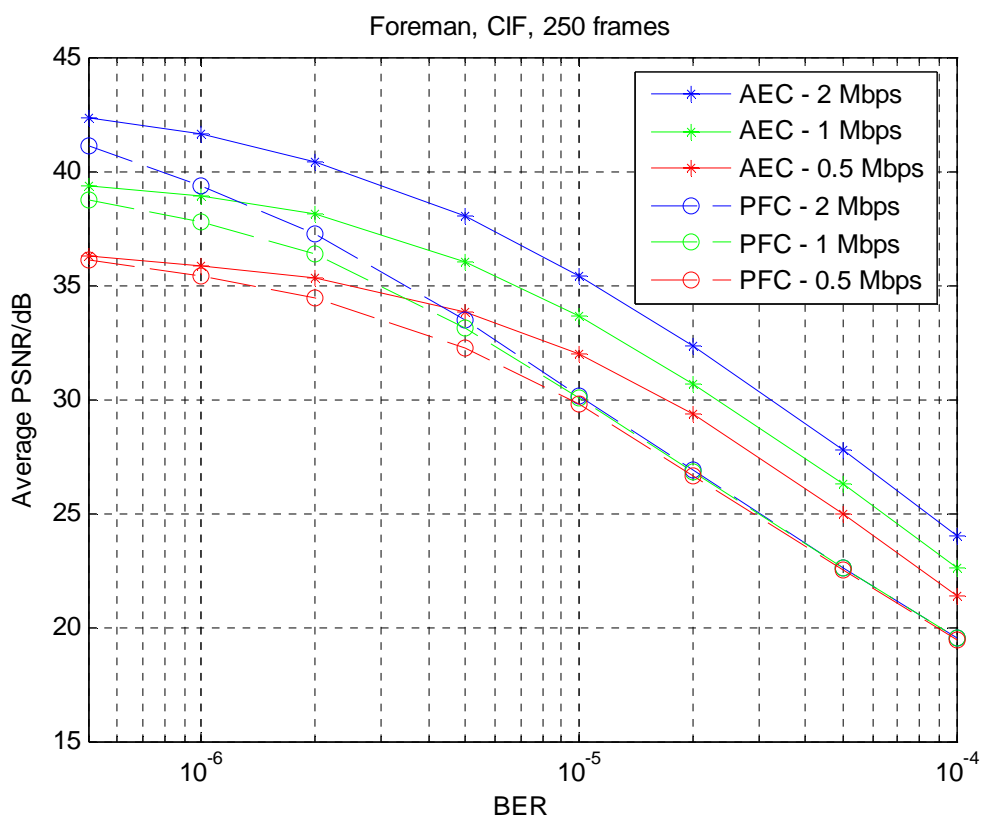
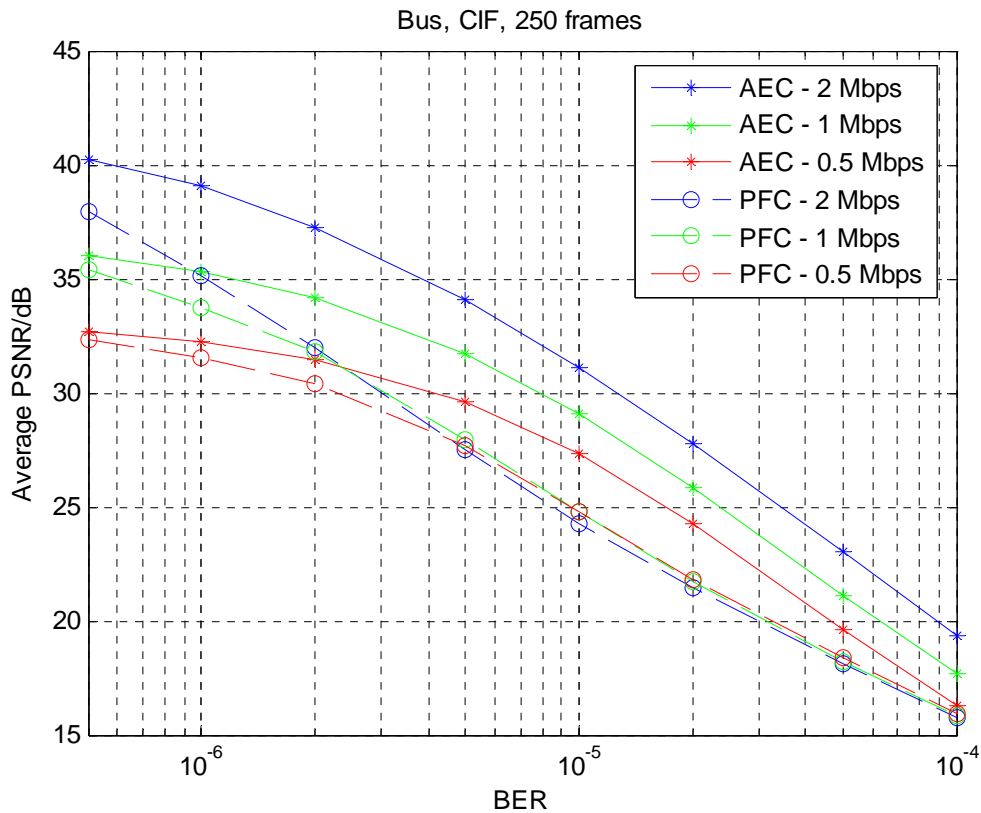


Figure 4: AEC vs PFC for Bus (top) and Foreman (bottom) sequences.



3. SVC Decoder

The SVC decoder development started from an HHI H.264/MPEG-4 AVC decoder for baseline profile. Its SVC functionality is compatible with version 8.7 of the JSVM reference codec[8]. Since that version, standardization went on, but it had been necessary to freeze the ASTRALS decoder specification at that version in order to finish the implementation and optimization on time for this deliverable.

3.1. Decoder Capabilities

The SVC decoder capabilities implemented in ASTRALS are shown in Table 5. They follow the definition of profile A during the 22nd JVT Meeting [9].

Tool	SVC A	Comments
Slices	I, P, EI, EP	Ok
PR slice motion refinement	N	Ok
AR-PR slices	N	Ok
FGS coding mode	N	Ok
Interlace	N	Ok
CAVLC	Y	Ok
CABAC	Y*	Ok
Deblocking filter	Y	Ok
Deblocking filter (upsampling)	Y	Ok
Constrained intra prediction flag below top layer	1	Ok
Arbitrary slice order	N	Ok
Number slice groups > 1	Y	Ok
Slice group map type	2	
Resolution factors 2, 1.5	Y	realized by ESS, non-optimized functions (for factor 2 simple study done)
ESS (any factor)	N	Ok
ESS aligned crop window	Y	Ok
EIDR	Y	Ok
IROI	N	Ok
Fragmented PR slice	N	Ok
CGS with varying quality levels (MGS)	Y	Ok

* activation of the tool is subjected to levels definition



Tool	SVC A	Comments
Weighted prediction	Y	Ok
Use base representation flag	Y	Ok
Adaptive transform block size	Y*	No
Quant scaling matrixes	Y*	No
Number of temporal levels	8	5 temporal levels tested (reference list size adjustment may be necessary)
Number dependency Id	8	Ok
Max number decoded dependency Id (using inter-layer prediction)	3	Ok
Number quality levels	4	Ok
Color format	4:2:0	Ok
Color bit depth	8	Ok

Table 5: SVC decoder features

3.2. Software Architecture

SVC decoder interaction with VLC Client consists of three main parts: plugin_hhi_h264svc_dec, HHISVCLibW and svcdec as shown in Figure 5.

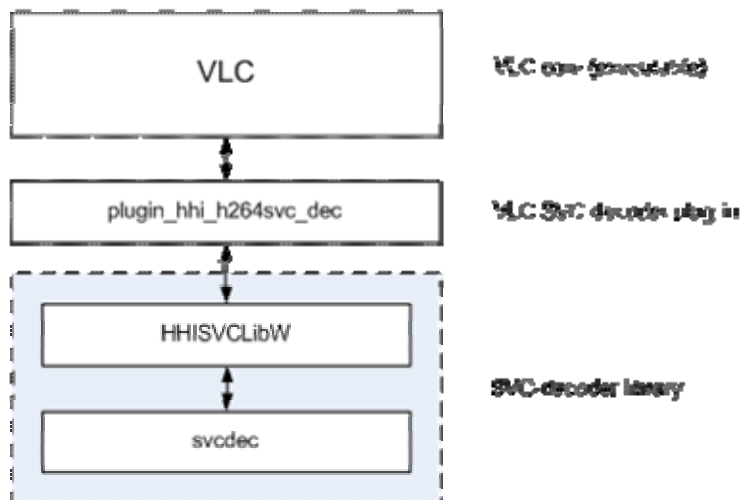


Figure 5: Architecture of SVC decoder plug-in for VLC

This architecture allows for the portability of the SVC decoder core, which is a crucial requirement regarding the exploitation of project results. At the same time, compatibility with the VLC interface is achieved. The SVC decoder core called “svcdec” is proprietary software developed and

* activation of the tool is subjected to levels definition



implemented by HHI. In ASTRALS it is used through a so-called wrapper module named “HHISVCLibW” which on the one hand controls the SVC decoder core “svcdec” and on the other hand exposes exactly the functionality requested by a VLC plug-in, which in this case is called “plugin_hhi_h264svc_dec”.

Interfaces and usage of the different modules are described in the following sections.

3.2.1. SVC Decoder Plug-in for VLC

General information about the implementation of VLC decoder plug-ins has been given in section 1.2 above. In addition to that, a special feature has been implemented for both the test and demonstration of scalability features. For that purpose, the SVC decoder plug-in defines some parameters that can be set resp. read by the user through VLC interface functions. By these parameters, the user can define a limit up to which operation point the SVC decoder will use the scalable bit stream data received by the VLC. The plug-in evaluates the scalability information carried in the SVC header extension of each SVC NAL unit and discards all NAL units exceeding at least one of the limitations given by these parameters (cf. Table 6 below).

<i>max-tl</i>	the maximum temporal level allowed for decoding
<i>max-did</i>	the maximum dependency ID allowed for decoding
<i>max-ql</i>	the maximum quality level allowed for decoding

Table 6: VLC parameters for SVC decoder

Depending on the macro CALC_DATA_RATES which can be set at compile time, three more parameters are available:

- "partial-rate": returns the number of its per second actually processed by the SVC decoder
- "total-rate" returns the number of its per second received by the data access module
- "smooth-rate" allows to apply an IIR filter on both partial and total rate values

For convenience, another parameter named *no-skip-pic-vout* has been defined which prevents the video output module of the VLC from skipping frames which might have been delayed to much either during transmission or by the decoder.

3.2.2. HHISVCLibW Wrapper Module

The HHISVCLibW wrapper module offers a tailored interface for the VLC decoder plug-in. The interface methods are described in Table 7 below.

Name	Comment
<i>h264DecOpen()</i>	open decoder instance and initialize
<i>h264DecClose()</i>	close decoder instance
<i>h264DecDecode()</i>	process i.e. (decode) one NAL unit
<i>h264DecGetPicInfo()</i>	return information about decoded frame, needed by the VLC main module to set-up video output
<i>h264DecGetCodecInfo()</i>	return information about used codec (non obligatory)

Table 7: HHISVCLibW Wrapper API



As mentioned above, the wrapper module interfaces between the host software (i.e. the VLC plug-in) and the decoder core library (i.e. svcdec). The host feeds the wrapper with single NAL units. Each NAL unit is forwarded to the decoder core. As soon as the decoded frame is ready for video output, the wrapper provides information about decoded frame to the host in the structure OutPicInfo shown in Table 8 below.

<i>Type</i>	<i>Field name</i>	<i>Description</i>
unsigned short	uiLHeight	Height of luminance component
unsigned short	uiLWidth	Width of luminance component
unsigned short	uiCHeight	Height of chrominance component
unsigned short	uiCWidth	Width of chrominance component
unsigned int	uiLStride	Stride of luminance component in buffer
unsigned int	uiCStride	Stride of chrominance component in buffer
unsigned int	uiLumOffset	Offset to first pixel of luminance component in buffer
unsigned int	uiCbOffset	Offset to first pixel of chrominance component Cb in buffer
unsigned int	uiCrOffset	Offset to first pixel of chrominance component Cr in buffer
unsigned char*	pucBufferLuma	The pointer to luminance component in frame buffer
unsigned char*	pucBufferChromaCb	The pointer to chrominance component Cb in frame buffer
unsigned char*	pucBufferChromaCr	The pointer to chrominance component Cr in frame buffer

Table 8: OutPicInfo structure

Figure 6 shows how the decoder core is controlled by the wrapper module. The red coloured functions are interface functions offered by “svcdec”. A short description of these functions is given in Table 9 below.

Name	Comment
<i>SVCDdec_Initialize()</i>	initialize decoder structure
<i>SVCDdec_Release()</i>	release decoder structure, release all used memory
<i>SVCDdec_ProcessNALU()</i>	process (decode) NAL unit
<i>SVCDdec_DPBCreate()</i>	create the decoded pictures buffer (DPB)
<i>SVCDdec_DPBDestroy()</i>	release the DPB
<i>SVCDdec_DPBPput()</i>	put one decoded frame into DPB
<i>SVCDdec_DPBPpush()</i>	get one decoded frame from DPB (next following)
<i>SVCDdec_DPBFfull()</i>	check if the DPB is full
<i>SVCDdec_DPBEempty()</i>	check if the DPB is empty

Table 9: Interface functions of svcdec

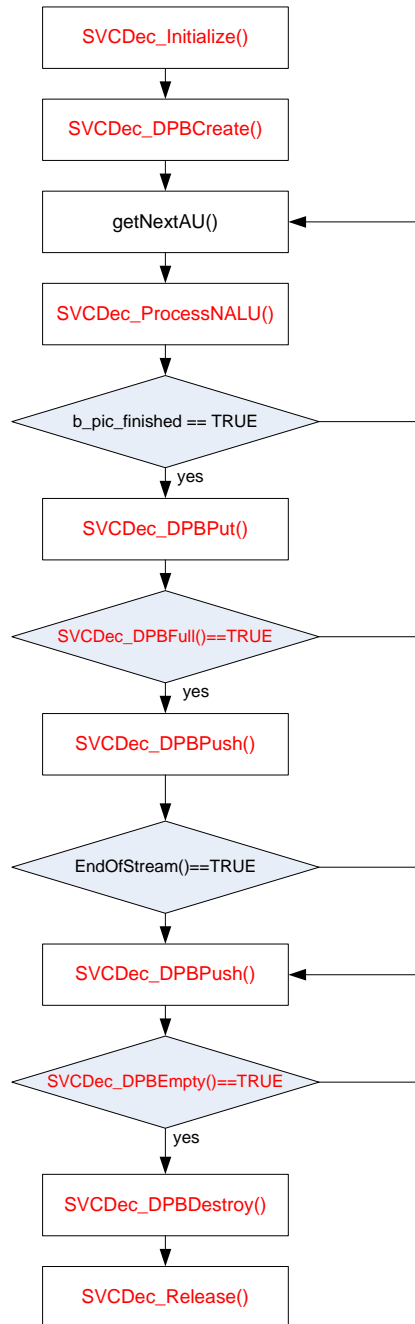


Figure 6: Typical use of SVC decoder, main flow chart

3.2.3. Decoder Core Library "svcdec"

In this section the decoding flow is shortly described. The main function of the decoder *SVCDec_ProcessNALU()* processes the incoming NAL units as it shown in Figure 7.

The incoming NAL units are added to the decoding list. The decoder analyses dependencies between NAL units. As soon as the first NAL unit of the next frame is received, the SVC decoder processes the NAL units of the current frame, starting from the base layer.

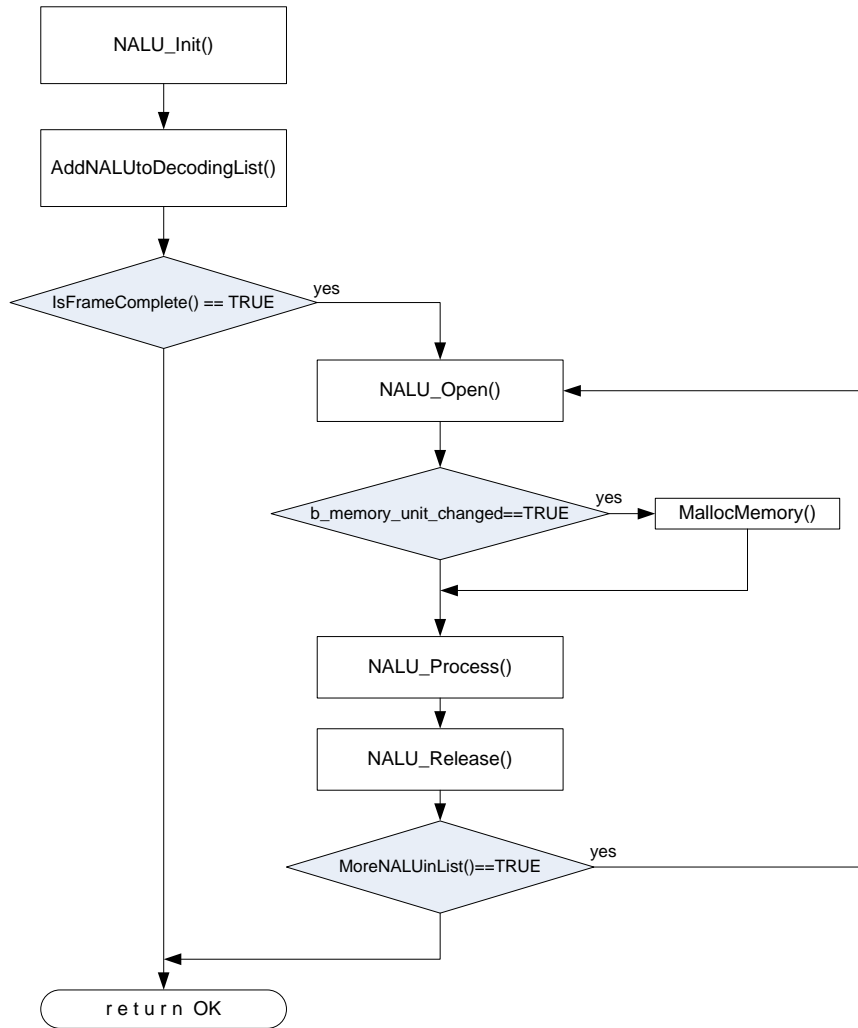


Figure 7: Basic flow chart of SVC decoder core

3.3. Profiling

A profiling of the SVC decoding complexity versus H.264/MPEG4-AVC baseline mode at the same bit rates has been done on an ARM-based test platform. Results are shown in Figure 8 for quality scalability in CGS (coarse grain scalability) and MGS (medium grain scalability) modes. At a total rate of 210 kbit/s, the SVC stream contains an AVC base layer coded at 102 kbit/s. The remaining data rate belongs to the SVC layer. Here, the SVC-decoder needs about 30% more cycles than the H.264/ MPEG4-AVC decoder for decoding. With each additional SVC layer the complexity increases.

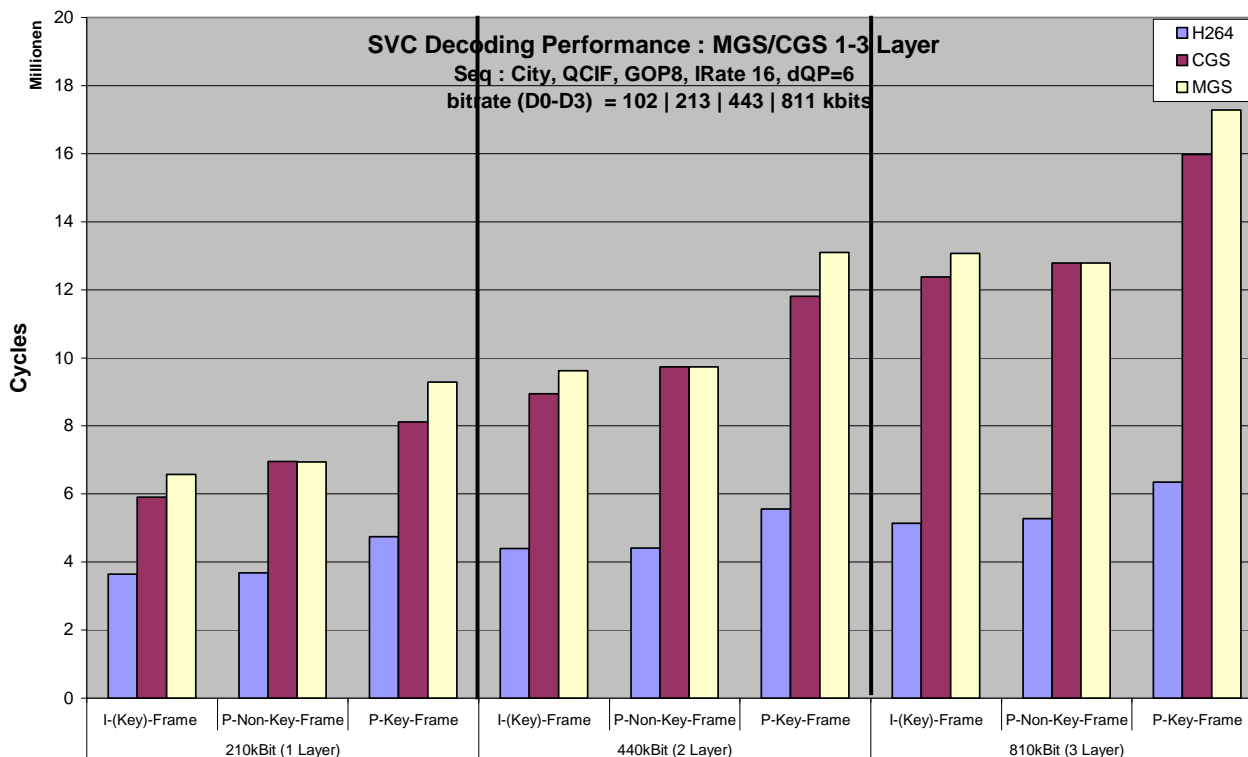


Figure 8: Complexity evaluation for SVC decoder

3.4. Optimization

The SVC decoder has been optimized regarding processing time on an Intel Xeon CPU, and its performance has been evaluated in comparison to the JSVM reference software. Results are shown for two well-known test sequences (foreman, soccer) using four different configurations for the SVC bit stream (cf. Table 10 below). Typically, code optimization has lead to a speed-up by a factor of eight with regard to the JSVM 8.7 reference software.

Dell Precision PWS690 Intel Xeon X5355 @2.66 GHz, 3 GB RAM Win XP Prof. SP2								
Sequence	scalability layers				max data rate kbit/s	JSVM8.7 frames / s	svcdec	speed gain
	base	enh 1	enh 2	enh 3				
foreman soccer	CIF	1 MGS			376 358	36,93 35,41	302,09 297,77	8,2 8,4
foreman soccer	4CIF	1 MGS			1276 1211	8,29 8,12	73,39 69,99	8,9 8,6
foreman soccer	QCIF	CIF	4CIF		1452 1540	8,17 8,08	56,53 55,86	6,9 6,9
foreman soccer	CIF	1 MGS	4CIF	1 MGS	2077 1886	6,2 6,19	54,74 53,93	8,8 8,7

Table 10: SVC decoder performance



Figure 9 shows the decoding speed using an AVC base layer and an MGS quality enhancement layer, both at CIF resolution. A similar two layers configuration, but at 4CIF resolution (corresponding to standard TV), is shown in Figure 10. SVC bit streams with two quality layers at standard TV resolution can be decoded at 60 fps.

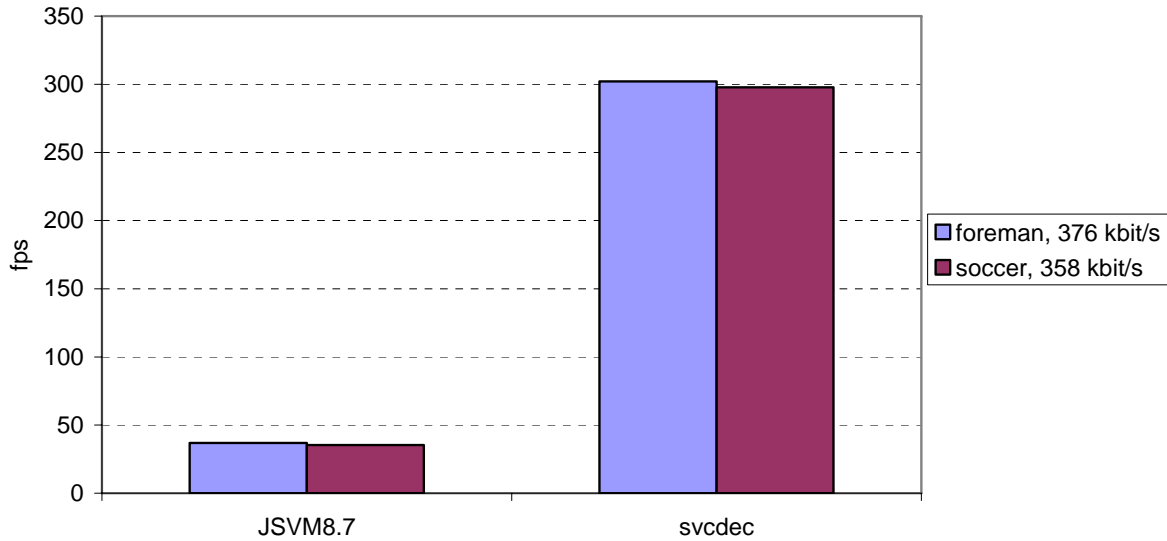


Figure 9: Speed optimization results, CIF + quality enhancement layer (MGS)

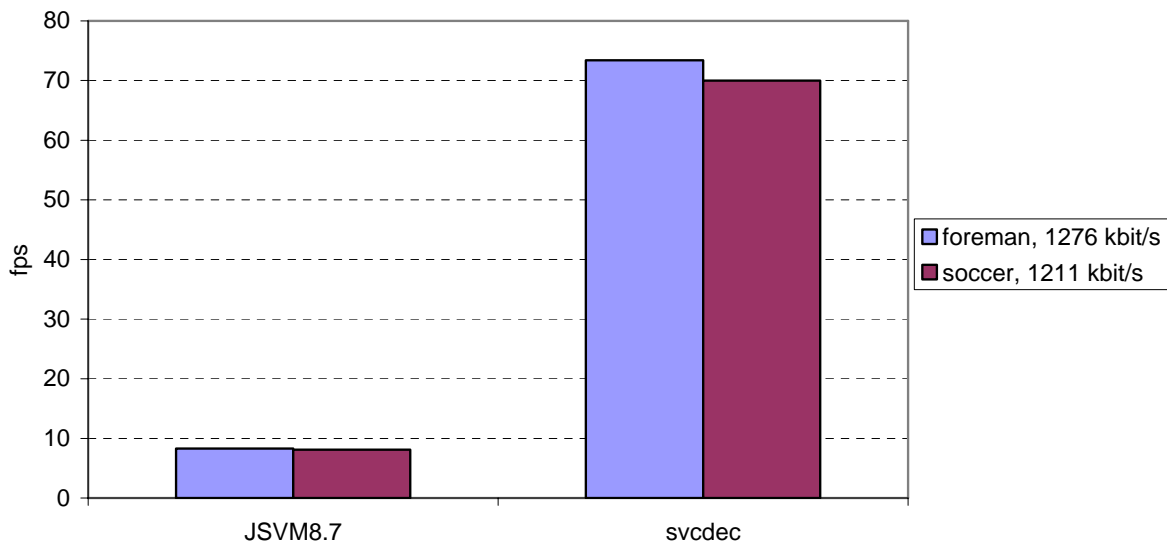


Figure 10: Speed optimization results, 4CIF + quality enhancement layer (MGS)



Next, some results are shown for spatial scalability. Figure 11 shows the SVC decoder performance for three spatial layers (QCIF, CIF, 4CIF) at a total data rate of about 1.5 Mbit/s.

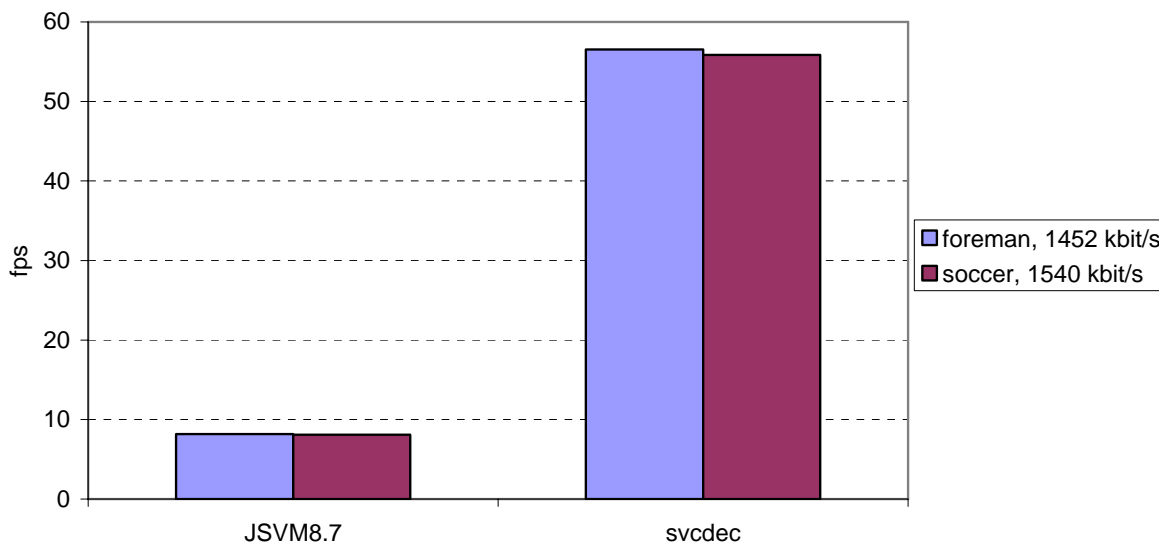


Figure 11: Speed optimization results, spatial scalability QCIF + CIF + 4CIF

The combination of spatial scalability with additional quality enhancement layers has also been tested. For a bit stream with four layers — two at CIF resolution (base layer and MGS quality enhancement layer) plus two layers at 4CIF resolution (MGS) — the results are demonstrated in Figure 12. Decoding of TV sequences with these scalability features is feasible at about 50 frames/s.

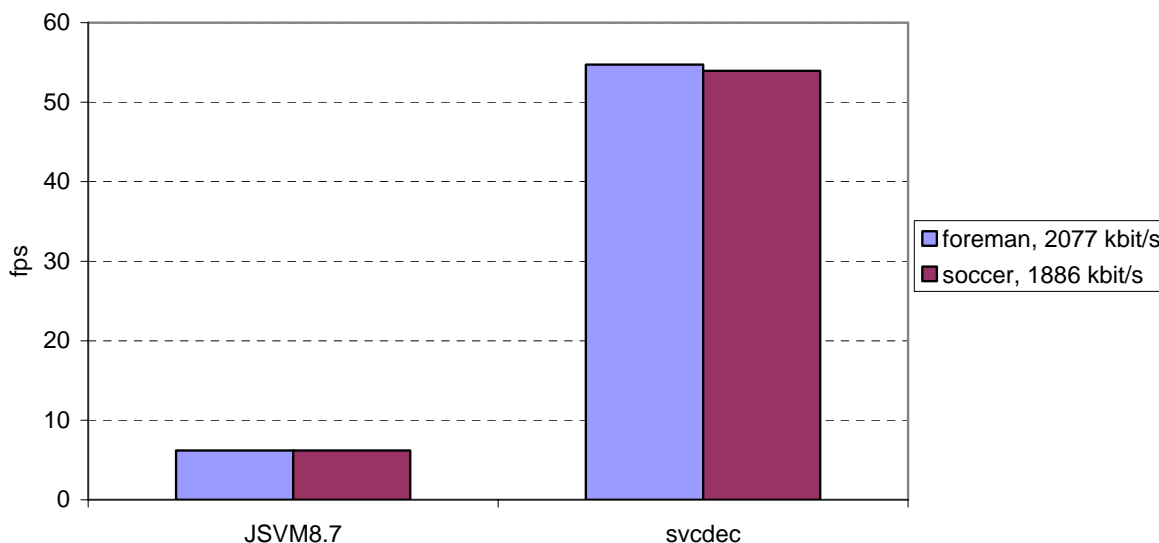


Figure 12: Speed optimization results, 4 layers: CIF + MGS + 4CIF + MGS



4. Conclusion

The ASTRALS Deliverable 4.4.2 software package described by this document is based on the VideoLAN Client (VLC) open source project in combination with proprietary libraries. It implements both the H.264/MPEG4-AVC and the SVC decoder client developed within Task 4.4 of WP4.

The H.264/MPEG-4 AVC decoder features an innovative approach for advanced error concealment (AEC), which is a major project result. Performance results are shown regarding error resilience as well as regarding decoding speed. For a wide range of error rates, AEC greatly improves the decoded picture PSNR.

RTSP session control and RTP/UDP data packet transmission are managed by a new transport layer plug-in for VLC provided by ASTRALS which is able to open several RTP sessions in parallel in order to support a Layered Multicast approach. This completely new feature can be used to choose the operation point at which the scalable video stream is decoded by connecting different clients to a different number of RTP sessions.

The performance optimized SVC decoder plug-in for VLC can decode sequences of TV resolution (resp. 4CIF, i.e. 640 x 576 pixels) with a total of four layers of spatial and quality scalability in real-time (more than 50 frames/s on a Xeon X5355 CPU at 2.6 GHz). It evidently supports partial reception of the scalable video stream. In this case the highest available layer is decoded. The data rate consumed by the decoder can be monitored through the VLC client interface, and optionally the number of layers to be decoded can be restricted. The software architecture chosen for this plug-in allows for easy portability to other decoder clients. Besides, the plug-in can be used on the ASTRALS platform based on Linux as well as under Microsoft Windows operating systems.



5. References

- [1] MPEG4 Part 10: Advanced Video Coding, ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC
- [2] RFC2326, Real-Time Streaming Protocol, <http://www.ietf.org/rfc/rfc2326.txt>
- [3] RFC2733, An RTP Payload Format for Generic Forward Error Correction, <http://www.ietf.org/rfc/rfc2733.txt>
- [4] Deliverable D4.3, “Adaptive Cross Layer (Trans)Coding”, ASTRALS project
- [5] T. Schierl, S. Wenger, *Signaling media decoding dependency in Session Description Protocol (SDP)*, doc ref <http://www.ietf.org/internet-drafts/draft-schierl-mmusic-layered-codec-04.txt>, June 22, 2007
- [6] S. Wenger, Y.-K. Wang, T. Schierl, "RTP Payload Format for SVC Video," AVT Working Group, <http://www.ietf.org/internet-drafts/draft-ietf-avt-rtp-svc-01.txt>, March 2007
- [7] Agrafiotis D., Bull D.R. and Canagarajah C.N., “Enhanced error concealment with mode selection,” IEEE TCSVT, vol. 16, no. 8, August 2006.
- [8] Joint Draft 9 of SVC Amendment, JVT-V201.doc
- [9] Thomas Wiegand, Gary J. Sullivan, Jens-Rainer Ohm, Ajay Luthra, „Meeting Report, Draft 6”, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, 22nd Meeting Marrakech Morocco, January 13-19, 2007

Annex A Software Package

The software package (Deliverable D.4.4.2) described by this document is provided to the ASTRALS project partners for the use within the ASTRALS project as an archive file named "ASTRALS_D4_4_2.tar.bz2". It contains a folder tree as shown in Figure 13 which comprises not only the parts developed in Task T4.4 but also some other software modules related to the VLC server and client framework.

Additionally, a "README_D4_4_2.txt" file is provided which explains the installation procedure for the complete package.

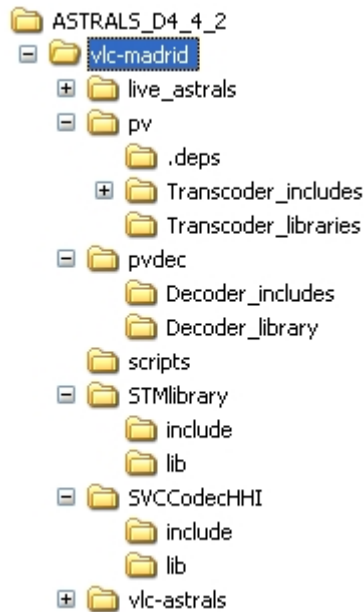


Figure 13: Structure of the software package D.4.4.2